Chip-Level and Reconfigurable Hardware for Data Mining Applications

by

Darshika Gimhani Perera

M.Sc., Royal Institute of Technology, 2003

B.Sc., University of Peradeniya, 1996

A Dissertation Submitted in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Electrical and Computer Engineering

© Darshika Gimhani Perera, 2012
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.

Supervisory Committee

Chip-Level and Reconfigurable Hardware for Data Mining Applications

by

Darshika Gimhani Perera

M.Sc., Royal Institute of Technology, 2003

B.Sc., University of Peradeniya, 1996

Supervisory Committee

Dr. Kin Fun Li, (Department of Electrical and Computer Engineering)
Supervisor

Dr. Fayez Gebali, (Department of Electrical and Computer Engineering)
Departmental Member

Dr. M. Watheq El-Kharashi, (Department of Electrical and Computer Engineering)
Departmental Member

Dr. Micaela Serra, (Department of Computer Science)
Outside Member

Abstract

Supervisory Committee

Dr. Kin Fun Li, (Department of Electrical and Computer Engineering)
Supervisor

Dr. Fayez Gebali, (Department of Electrical and Computer Engineering)
Departmental Member

Dr. M. Watheq El-Kharashi, (Department of Electrical and Computer Engineering)
Departmental Member

Dr. Micaela Serra, (Department of Computer Science)
Outside Member

From mid-2000s, the realm of portable and embedded computing has expanded to include a wide variety of applications. Data mining is one of the many applications that are becoming common on these devices. Many of today's data mining applications are compute and/or data intensive, requiring more processing power than ever before, thus speed performance is a major issue. In addition, embedded devices have stringent area and power requirements. At the same time manufacturing cost and time-to-market are decreasing rapidly. To satisfy the constraints associated with these devices, and also to improve the speed performance, it is imperative to incorporate some special-purpose hardware into embedded system design. In some cases, reconfigurable hardware support is desirable to provide the flexibility required in the ever-changing application environment.

Our main objective is to provide chip-level and reconfigurable hardware support for data mining applications in portable, handheld, and embedded devices.

We focus on the most widely used data mining tasks, clustering and classification. Our investigation on the hardware design and implementation of similarity computation (an important step in clustering/classification) illustrates that the chip-level hardware support for data mining operations is indeed a feasible and a worthwhile endeavour. Further performance gain is achieved with hardware optimizations such as parallel processing.

To address the issue of limited hardware foot-print on portable and embedded devices, we investigate reconfigurable computing systems. We introduce dynamic reconfigurable hardware solutions for similarity computation using a multiplexer-based approach, and for principal component analysis (another important step in clustering/classification) using partial reconfiguration method. Experimental results are encouraging and show great potential in implementing data mining applications using reconfigurable platform.

Finally, we formulate a design methodology for FPGA-based dynamic reconfigurable hardware, in order to select the most efficient FPGA-based reconfiguration method(s) for specific applications on portable and embedded devices. This design methodology can be generalized to other embedded applications and gives guidelines to the designer based on the computation model and characteristics of the application.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ALU	Arithmetic and Logic Unit
ASIC	Application Specific Integrated Circuit
ATR	Automatic Target Recognition
BRAM	Block Random Access Memory
CF	Compact Flash
CLB	Configurable Logic Block
CMOS	Complementary Metal Oxide Semiconductors
CPLD	Complex Programmable Logic Devices
CPU	Central Processing Unit
DCT	Discrete Cosine Transform
DDR-SDRAM	Double Data Rate Synchronous Random Access Memory
DMA	Direct Memory Access
DRH	Dynamic Reconfigurable Hardware
DSP	Digital Signal Processing
EEPROM	Electrically Erasable Programmable Read Only Memory
EVD	Eigenvalue Decomposition
FIR	Finite Impulse Filter
FMC	FPGA Mezzanine Connectors
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HDL	Hardware Description Language
I/O	Input/Output
IC	Integrated Circuits
ICAP	Internal Configuration Access Port
IPIC	Intellectual Property Interconnect
IPIF	Processor Local Bus IP Interface
KDD	Knowledge Discovery in Databases

KLT	Karhunen-Loeve Transform
kNN	k-Nearest Naighbour
LUT	Look Up Tables
MAC	Multiply-and-Accumulate
MIMD	Multiple Instruction Multiple Data
MSS	Multi-Spectral Scanner
OS	Operating System
PC	Principal Component
PCA	Principal Component Analysis
PCIe	Peripheral Component Interconnect Express
PDA	Personal Digital Assistant
PE	Processing Element
PLA	Programmable Logic Arrays
PLB	Processor Local Bus
PLD	Programmable Logic Devices
PROM	Programmable Read Only Memory
RAM	Random Access Memory
RC	Reconfigurable Cell
RISC	Reduced Instruction Set Computing
RM	Reconfigurable Module
RPU	Reconfigurable Processing Unit
SDRAM	Synchronous Random Access Memory
SIMD	Single Instruction Multiple Data
SRH	Static Reconfigurable Hardware
SVD	Singular Value Decomposition
VHDL	VHSIC Hardware Description Language
VLIW	Very Long Instruction Word

Glossary

Computation – is a process of performing a certain operation

Computation models – are various types of processing such as parallel (functional), pipeline, parallel (data), and computations with many identical functions

Operations or computation modules – are the functions (or tasks) in a computation model

Functional units – are the components where operations are executed

Stage – is a distinct part of a computation process with identifiable inputs and outputs

Processing Elements – are hardware circuitry used to execute operations autonomously

Reconfigurable module – is a functional unit or a PE that is reconfigurable

Acknowledgments

First and foremost, I wish to express my deepest gratitude to my supervisor, Dr. Kin Fun Li for giving me this opportunity to work with him and the valuable advice and the guidance given throughout my entire research work in numerous ways. This endeavour was successful because of him.

I would like to thank Dr. Fayez Gebali, Dr. Micaela Serra, and Dr. Watheq El-Kharashi for serving on my supervisory committee. I greatly appreciate their assistance and advice throughout my research work.

My heartfelt gratitude goes to my family including my parents, sister, and Johannes Menzel, who have endured my absence and helped me and encouraged me in numerous ways to pursue my research. I am grateful for them for helping me to achieve my goals.

Finally, I must thank all my colleagues for their friendship during my research at University of Victoria.

Chapter 1

1. Introduction and Motivation

There have been significant advances in mobile, handheld, and embedded devices since the mid-2000s. As a consequence, a wide variety of applications are becoming more and more common on these devices. This has led to research in sophisticated yet small-foot-print hardware and software solutions for embedded systems. However, portable and embedded devices have stringent area and power requirements. In addition, applications on embedded systems often must run with real-time constraints. Coupled with increasing pressure to decrease cost and shorten time-to-market, the design constraints of these devices pose a serious challenge to the embedded system designers.

According to a market research [56], the global market for embedded systems technologies was worth $92.0 billion in 2008, which will increase to $112.5 billion in 2013, for a compound annual growth rate of 4.1%. This research also shows that embedded hardware has the largest share of the market ($89.0 billion in 2008 to $109.0 billion in 2013) compared to embedded software ($2.2 billion in 2008 to $2.9 billion in 2013). Another study [43] done in 2005 demonstrated that annual growth rate of embedded systems market is 18%, whereas general-purpose computing is only 10%. This trend exhibits the embedded systems market is becoming larger than the general-purpose computing's. Embedded devices are starting to dominate in many aspects of our lives. These devices are used in various industries including automotive, avionics, telecom, aerospace, medical, and consumer electronics. For instance, embedded systems are incorporated in many consumer electronics such as mobile phones, personal digital assistants (PDAs), etc. All these illustrate that the embedded systems market will continue to flourish over the long term as new applications emerge [56].

One of the many applications that is becoming common on portable and embedded devices is data mining. Data mining is an important research area as many applications in various domains can make use of it to sieve through large volume of data to discover useful patterns and valuable knowledge. Examples of data mining applications that are appropriate for portable and embedded devices are: handwritten analysis, signature verification, palm-print or finger-print verification, iris verification, facial recognition, etc.

Data mining applications have numerous challenging issues of their own. One of the major issues of these applications is the speed performance. For instance, with the exponential growth of information on the Web and archived data sets in centralized and distributed database systems, effective and efficient information retrieval is becoming a major concern in many data mining applications. Unlike traditional data mining applications that target a bounded set of data, most of today's applications must continuously process a relatively large unbounded set of data. Also in many cases, the data need to be processed in real time to reap the actual benefits. These constraints have significant impact on the speed performance of these applications.

Consequently, new technologies and design methodologies are needed to improve the data mining process. In addition to algorithmic development, some kind of hardware support is imperative to enhance the speed performance of these applications. In some cases, reconfigurable hardware support is desirable to provide the flexibility required in the ever-changing application environment.

In order to provide hardware support for data mining applications, it is important to understand the intrinsic characteristics of these applications. First, data mining applications, for instance information retrieval, involve many operations at a higher level of abstraction such as clustering and classification, which often consist of multiple stages and lengthy processing. These operations are typically very large and complex. For example, both clustering and classification consist of several stages, including pattern representation and pattern proximity measure (clustering/classification), grouping (clustering), and labelling (classification). Second, there exist a large number of different algorithms for many data mining operations at a higher level of abstraction. For instance, there are a variety of algorithms for clustering and classification. These algorithms may use various methods to carry out an operation. In many cases, each operation itself is usually solvable by various methods, though having results of different quality. For example, there are many ways to measure similarity such as Cosine Similarity, Extended Jaccard, and Asymmetric Measure. They often produce different results in dissimilar application contexts. Third, in some cases, the operation to be performed next is not known in advance. Among several available operations, one must consider the current objective and other criteria such as recent results obtained and the external stimuli, in

order to determine the next actions without human intervention. Fourth, as in many other current-generation applications, new operations are being introduced in data mining, while some of the existing operations are being modified.

In order to address these characteristics, the hardware support for data mining applications should be capable of:

- Performing a variety of data mining operations.
- Selecting the next operation to process the data as needed on-the-fly.
- Changing the implemented operations dynamically.
- Adding new operations and modifying existing operations even after deployment.

Originally limited to a few applications such as scientific research and medical diagnosis, data mining has become vital to a variety of fields including biotechnology, multimedia, marketing, business intelligence, and network security [37]. This has dramatically increased the use of data mining not only in large corporations, but also among a growing number of individuals that typically use portable, handheld, and embedded devices. As mentioned earlier, one of the major constraints with these computing platforms is their limited hardware resources. Thus, it is worthwhile to investigate how these limited hardware resources can be used efficiently and effectively to provide sufficient support for data mining applications, while under the power, cost, time-to-market, and real-time constraints of the portable and embedded devices.

Throughout our research work, we aim to address primarily the area constraint and secondarily the power, cost, and time-to-market constraints of portable and embedded devices. These are done either experimentally or analytically, or both. The applications executed on these devices are typically performed in real-time. With real-time applications, the data are usually streamed in from different sources. Data mining operations must process the data that are continuously streamed in. Although we looked into the constraints associated with stream-in data, such as data transfer latency, we did not attempt to address these constraints in our present implementation research work. However, some techniques to address the data transfer latency are proposed. In addition, we did not attempt any hardware optimization that could potentially improve real-time speed performance.

To satisfy the requirements (or constraints) of the portable and embedded devices and also to improve the speed performance of data mining applications, it is imperative to incorporate some special-purpose hardware into embedded systems designs. These customized hardware algorithms should be executed in single-chip systems, since multi-chip solutions might not be suitable due to the limited foot-print on portable and embedded devices. The customized hardware is optimized for a specific application, and avoids the high execution overhead of fetching and decoding the instructions as in microprocessor-based (software-only) designs. As a result, customized hardware provides superior speed performance and often consumes less power [67],[156] than equivalent software running on microprocessor. Also, customized circuits are usually compact and occupy less hardware space on a chip compared to general-purpose circuits of a microprocessor. These advantages are especially important for portable and embedded devices. In addition, high performance processors are typically costly and consume high power [24],[135], making them infeasible and uneconomical for many portable, handheld, and embedded devices.

For more complex operations, it might not be possible to squeeze all the computation hardware into a single chip. An alternative is to take the advantage of reconfigurable computing systems. Reconfigurable hardware has similar advantages as special-purpose hardware, leading to low power [67],[156] and high performance. These advantages are:

- Provides customized circuits hence efficient to perform a specific application.
- Avoids the overhead of fetch/decode instructions as in microprocessor-based designs.

Furthermore, reconfigurable computing systems have added advantages:

- Utilizes a single chip to perform the required operations by reconfiguring the hardware on chip and reusing the same chip repeatedly.
- Provides a flexible computing platform to perform the required applications, similar to microprocessors. In this case, the on-chip hardware circuitry can be changed post fabrication to perform a variety of applications.
- Reduces the time to market, since it is pre-fabricated and hence immediately available (similar to off-the-shelf microprocessors).

This reconfigurable approach could address the constraints associated with portable and embedded devices, as well as the flexibility and performance issues in processing a large data set.

1.1. Our Research Objectives

The main objective of our research work is to provide chip-level and reconfigurable hardware support for data mining applications in portable, handheld, and embedded devices.

In order to achieve our main research objective, we divide our research work into three related major stages. The objectives for each progressive stage are:

- In the first stage, to investigate: the feasibility and potential performance gain of using hardware for data mining operations; and advantages of using parallel hardware.
- In the second stage, to investigate the feasibility of using reconfigurable hardware for data mining operations in portable, handheld, and embedded devices.
- In the third stage, to formulate a design methodology for FPGA-based dynamic reconfigurable hardware in order to select the most efficient reconfiguration method(s) to use in different scenarios considering computation models, application characteristics, size of the operations, etc. Guidelines and heuristics are based on theoretical analysis as well as from our experience (experimental and analytical) on reconfigurable computing.

1.2. Our Contributions

We make three major contributions in this dissertation corresponding to the three major stages mentioned above.

For the first stage, we introduce chip-level hardware solutions for three similarity measures and their corresponding similarity matrices, and FPGA-based processor array for parallel computation of similarity matrix. An algorithm is also developed to assign the computations efficiently to each processing elements (PE) of the processor array. This investigation illustrates that chip-level hardware support for data mining operations is indeed a feasible and worthwhile endeavour. Our hardware designs take advantage of the inherent parallelism and pipeline nature of the data mining operations. Even without performing any hardware optimization in the PEs, a substantial performance gain is

achieved using parallel processing architecture for similarity matrix computations. This contribution has led to publications [104],[105],[124],[125],[126].

To achieve the objective for the second stage, we introduce dynamic reconfigurable hardware solutions for two major data mining operations, similarity matrix computations (using multiplexer-based approach) and part of Principal Component Analysis (PCA) computation (using partial reconfiguration method). These two operations are used often in many applications such as handwriting analysis, finger-print verification, etc. Further experiments and analysis are also carried out on partial and dynamic reconfiguration. This investigation demonstrates that reconfigurable computing allows the required flexibility and performance to provide chip-level hardware support for data mining applications for portable and embedded computing, while satisfying the area, power, cost, and time-to-market requirements of these devices. A space-time cost analysis shows that a significant space saving is achieved using reconfigurable hardware, and the time penalty of the reconfiguration overhead is insignificant, especially for large volume of data. Our experimental results are encouraging and show great potential in implementing our target data mining applications using reconfigurable platform. Trading off speed as compared to parallel computation, complex processing can indeed be implemented in reconfigurable hardware for embedded and portable applications. Some parts of this contribution have been published in [105],[127],[128],[129].

Our third major contribution is the formulation of a design methodology for FPGA-based dynamic reconfigurable hardware. This design methodology offers the embedded hardware designers a guideline to select the most efficient reconfiguration method to use in different scenarios based on computation models, application characteristics, size of the operations, etc. It guides the designers to map computation models and application characteristics to the reconfiguration methods based on the associated advantages and disadvantages. This design methodology can be generalized to other embedded applications and is not limited to data mining applications.

1.3. Dissertation Organization

This dissertation is organized as follows.

In Chapter 2, background study is presented, which includes the various means of hardware support for application-specific operations. Data mining and its applications are introduced in this chapter. Existing research work on hardware support for data mining operations are also discussed.

In Chapter 3, we present our first major contribution, chip-level hardware support for data mining operations. This includes our initial investigation and proof-of-concept work using Cosine Similarity Measures and further investigation on other similarity measures and more complex operations using similarity matrix computations. The implemented hardware and software designs for similarity measure and similarity matrix computations are illustrated. Our investigation on parallel hardware approach is discussed and presented, which includes the FPGA-based processor array designed for parallel computation of similarity matrix and the algorithm developed to assign computations to the processing elements (PEs) of the array efficiently.

In Chapter 4, we present our second major contribution, reconfigurable hardware for data mining operations. This includes the investigation on state-of-the-art reconfigurable computing systems. In addition, the designed and implemented reconfigurable hardware solutions for similarity computations using multiplexer-based approach and for partial PCA using dynamic partial reconfiguration method are discussed and presented. Further investigation and analysis on dynamic partial reconfigurable hardware is also presented in this Chapter and Appendix C.

In Chapter 5, we present our third major contribution, the design methodology for FPGA-based dynamic reconfigurable hardware from our experience and analytical studies on reconfigurable computing. We present the computation models and applications that would benefit from FPGA-based reconfigurable hardware. Features, advantages, and disadvantages of different FPGA-based reconfiguration methods are also discussed. Finally, we provide guidelines on how to map an application's computation models and characteristics to the most efficient reconfiguration method(s).

In Chapter 6, we summarize our contributions, conclude, and discuss the future directions of our research work.

Chapter 2

2. Background Study

In this chapter, we present a background study of our research. In 2.1, we discuss and present various means of hardware support for application specific operations. In 2.2, we provide a general high-level framework of data mining algorithms, specifically clustering and classification, since these two are some of the most widely used tasks in data mining. In addition, we elaborate on their characteristics and also on computation models. Existing research work on hardware support for data mining operations is also discussed and presented in this section.

2.1. Hardware Support for Application Specific Operations

In this section, we discuss and present commonly used means of hardware support for application specific operations: application-specific integrated circuits, microprocessors, and reconfigurable computing systems. It should be noted that our investigation and discussion on microprocessor focuses on a single CPU system rather than multiple processors or multi-core systems.

2.1.1. Application Specific Integrated Circuits

Application-specific integrated circuits (ASICs) are designed to perform a specific computation or a set of computations, thus they can quickly and efficiently compute the specific task that it is customized for, leading to superior speed performance. ASICs can exploit parallelism in computations, since computations are implemented spatially distributed throughout the chip, rather than implementing them temporally on a single functional unit as in microprocessor-based designs [23],[49]. Unlike microprocessor-based designs, ASICs avoid the overhead associated with instruction fetch/decode/execute.

Each ASIC has fixed functionality, which cannot be altered post fabrication, impeding the flexibility of the architectures, preventing any post-design optimization and upgrades in applications [23].

Additionally, with ASICs, often only a specific application is implemented on a single chip; hence, in order to execute a variety of applications, we might have to implement

custom-designed hardware for each application on several chips, requiring more hardware space. This becomes an issue with portable, handheld, and embedded devices because of their limited hardware foot-print.

ASICs are often infeasible and uneconomical for many portable and embedded devices, because both the manufacturing cost as well as time to develop and market can be very high and unacceptable [67],[156]. For instance, if an ASIC-based design requires even minor modifications, the hardware designers are compelled to fabricate a new chip according to the modified design, because of its fixed functionality [39].

2.1.2. Microprocessors

Microprocessors provide an alternate solution that addresses the flexibility issues of ASICs. They provide a flexible computing platform for a large number of applications or operations [23]. An application is realized using a software program. The microprocessor interprets the program instructions and executes them to perform an operation. By changing the software instructions, microprocessors change the functionality of the system, without changing the underlying hardware [23],[39]. Therefore, unlike ASICs, a variety of applications can be executed on a single microprocessor.

Unfortunately, this flexibility comes with the penalty of relatively inferior performance than an ASIC. For instance, the set of instructions for a specific processor are usually determined during fabrication time. As a result, new operations can only be implemented out of existing instructions, whose underlying hardware might not be optimally designed with the new operations in mind. Unlike customized circuits of ASICs, a microprocessor typically uses general-purpose circuits for implementing instructions, leading to inferior performance. Additionally, each individual instruction has to be fetched, decoded, and then executed, resulting in high execution overhead [39].

Low-power operation is critical to many battery-dependent portable and embedded devices [67],[156]. It is important that the applications executed on these devices do not exceed the power constraints, since this will cause heating problems [143]. Power consumption of a microprocessor is much higher than customized hardware, mainly because of the general-purpose circuits and the overhead of instruction fetch/decode/execute [67]. Furthermore, the high “power consumption (100 watts or

more)" and high "cost (possibly thousands of dollars)" of the high-performance microprocessors place them out of reach for many portable and embedded applications [24],[156].

Unlike ASICs, in general, time-to-market is reduced by using an off-the-shelf microprocessor. The designer only has to write and verify the software for the application instead of an extensive design and test cycle.

2.1.3. Reconfigurable Computing Systems

Reconfigurable computing as a concept has been in existence since 1960, when Gerald Estrin proposed the idea of a "fixed plus variable structure computer" [109]. It consists of a standard processor and an array of "reconfigurable" hardware, the behaviour of which could be controlled by the main processor [59]. Similar to special-purpose hardware, reconfigurable hardware could also be customized to perform specific computations, resulting in high performance. In addition, after processing one computation, the hardware could be modified to perform another computation. Thus, reconfigurable computing system can be considered as a hybrid computer, which combines the flexibility of software with the speed performance of hardware [165].

After a two-decade gap, from around 1980s, research on modern reconfigurable computing systems started to revive [77]. Several research groups (both from industry and academia) proposed several reconfigurable architectures [165], such as: MATRIX [116], Garp [27], MorphoSys [144], RaPiD [44], PipeRench [70], PACT XAPP [18], REMARC [118], ADRES [114], etc. These designs were feasible only because of the advancement of the silicon technology, which lead to implementation of complex designs on a single chip [165]. Although the first commercial reconfigurable computer, Algotronix Cal-chip completed in 1991 [6], was not a commercial success, it was the stepping stone for the current commercially viable Field Programmable Gate Array (FPGA) based reconfigurable computing.

2.1.3.1. What is reconfigurable computing?

Reconfigurable computing bridges the gap between hardware and software, in order to achieve higher performance than software and a higher level of flexibility than hardware [39]. Reconfigurable computing system incorporates some form of hardware

programmability at run-time to provide the required flexibility without compromising performance [23]. The programmability is achieved using a number of physical control points, which can be changed periodically to perform different operations/applications using the same hardware [39]. These control points determine: the functionality of the computational units; the routing of the interconnection networks that connect these units; and the interface to the rest of the system. In this way, digital circuits can be mapped to the reconfigurable hardware, by mapping the functions to the computational units, and then by using the programmable routing to connect the units to form a necessary circuit [39]. Since the same hardware can be re/configured and reused as many times as needed, a single chip can be used to execute several different application requiring less hardware resources, which is important for portable and embedded devices with its stringent area requirements.

Because of the programmability, reconfigurable computing systems can exploit fine-grain and coarse-grain parallelism available in an application, which in turn provides significant performance advantages compared to microprocessors [23],[77]. Since the reconfigurability allows the hardware to adapt to a specific computation or set of computations in an application, reconfigurable computing systems typically achieve higher performance than software executed on microprocessors [23]. In addition, similar to ASICs, it avoids the high overhead of instruction fetch/decode/execute.

Similar to off-the-shelf microprocessor, reconfigurable computing systems in the form of programmable hardware are also immediately available, since they are pre-fabricated thus reducing the time-to-market.

2.2. Data Mining

In the above section, we discuss commonly used means of hardware support for application specific operations, and briefly discuss advantages and disadvantages of using them for portable, handheld, and embedded devices. In this section, we introduce data mining and its applications.

Choudhary, et al. [37] view data mining as a “powerful technique of transforming raw data into understandable and actionable form, which can then be used to predict future trends or to provide meaning to historical events”. It is a process of finding correlations or

patterns among various fields in large data sets; this is done by analyzing the data from many different perspectives, categorizing it, and summarizing the identified relationships [45].

As mentioned in [61],[76], "data mining is often set in the broader context of Knowledge Discovery in Databases (KDD)", which involves several stages: "selecting the target data, pre-processing the data, transforming them if necessary, performing the data mining to extract patterns and relationships, and then interpreting and assessing the discovered structures".

The second stage, data pre-processing, involves data cleaning, data verification, and defining variables [76]. The cleaned data is typically represented as feature vectors, one vector per object. A feature vector is an *m*-dimensional vector of numerical features that represent an object [134]. If the object is an image, the feature values might correspond to the pixel of an image, or if the object is a text document, then the feature values might be the frequency of occurrence of terms [110].

It is hard to explicitly distinguish the boundaries of the data mining part of the process. For many, data transformation, the third stage, is an essential part of the data mining processing [76]. Typically, raw data is difficult to comprehend, thus it might be beneficial to modify them prior to analysis. Data transformation may reveal structures that otherwise may not be obvious to the human eye [66]. However, user must be cautious of performing data transformation, since there is a possibility to go too far ahead in this direction, which might result in loss of important data that could be relevant to further studies.

The final stage, assessment and validation of the results, verifies the patterns and relationships produced by the data mining applications, since some of the patterns produced might not necessarily be valid [20],[76].

2.2.1. Main Tasks in Data Mining

As shown in Figure 1, data mining commonly involves any of the four main high-level tasks [76],[110]: Classification, Clustering, Regressions, and Association Rule Mining.

Classification is a process of assigning records or objects to one of several predefined classes or categories [91],[138]. In this case, once a set of predefined classes are given, we try to determine the class or the classes the given objects should be assigned [110].

Typically in classification, a set of example records, known as training set, is given. Each of these records consists of several dimensions or attributes, which are either continuous or categorical [91]. Continuous attributes are from an ordered domain, such as weight, speed, or age, whereas categorical attributes are from an unordered domain, such as gender, colour, or name. One of these dimensions or attributes is called the classifying attribute, which indicates the class to which each record belongs [138]. In classification, the goal is to "build a model of the classifying attribute based on the other attributes" [91],[138].

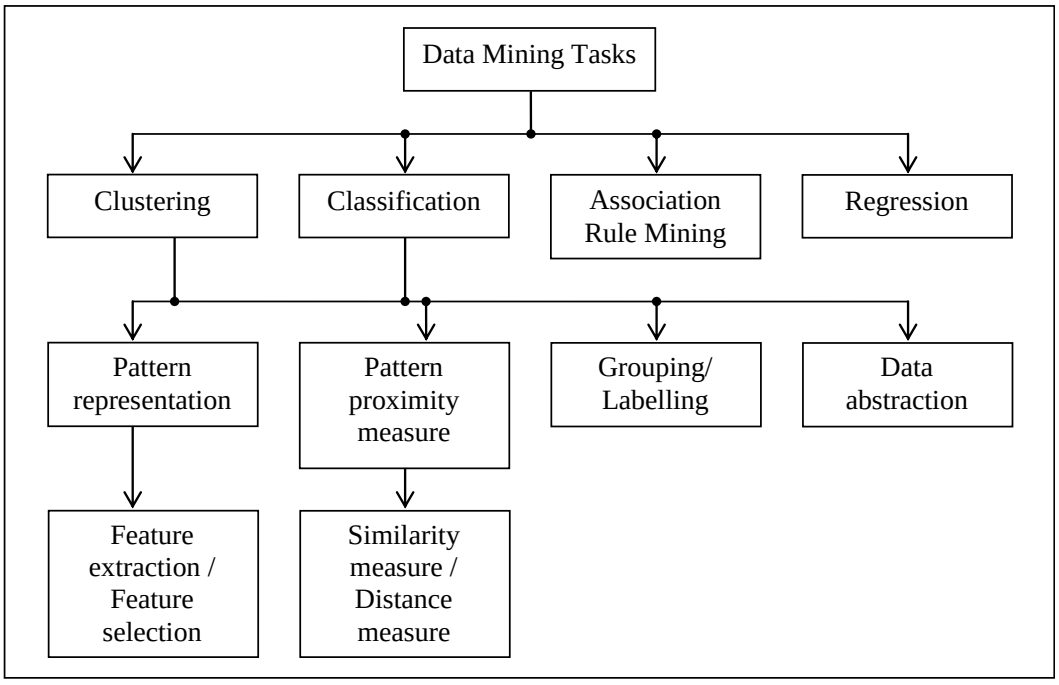


Figure 1 Data Mining Tasks

Clustering is quite similar to classification but the groups are not predefined, thus the algorithm tries to group similar objects together [88],[110]. As mentioned in [110], "clustering algorithms group a set of objects into subsets or clusters". The objective is to "create clusters that are coherent internally, but clearly different from each other", i.e., objects within a cluster should be as similar as possible, and objects among the clusters should be as dissimilar as possible [20],[110].

Regression Analysis is one of the oldest and most popular statistical techniques used in data mining for certain applications [32],[151]. Similar to classification, regression also

attempts to “build a model that permits the value of one variable to be predicted from the known values of other variables” [76]. Unlike classification, in which the variable being predicted can be categorical, in regression the variable is typically quantitative [76]. Regression develops a mathematical formula that fits a numerical data set. Linear regression is the simplest form of regression. This uses the formula of a straight line ($y = mx + b$), which has only one input variable. Alternatively, multiple regression uses more than one input variables and are used for more complex models such as sum-of-squared-error-function [76],[151].

Association rule mining is used to discover interesting relationships between variables (or patterns) in a large dataset in a relatively efficient manner [76]. The objective is to find the relationships between set of items, where the existence of some items suggests that others follow from them [76]. One of the popular applications of association rule mining is the market basket analysis, where the discovered rules could potentially lead to important marketing decisions [75],[81]. For instance, collecting information about customers’ buying habits and then applying association rule mining, supermarkets can determine the grocery products that are often purchased together. This type of information can be useful for marketing purpose, which might potentially increase in sales [75].

2.2.2. Clustering and Classification

Of these four main high-level data mining tasks, we are focusing on the widely used clustering and classification. Clustering and classification problems have been addressed in many different situations by many researchers in a variety of fields; thus, illustrating their demand and usefulness [88].

It is important to distinguish clustering from classification. Classification is a form of supervised learning, whereas clustering is unsupervised learning. In classification, a collection of labelled (or pre-classified) patterns are given and the “problem is to label a newly encountered yet unlabelled pattern”, whereas in clustering, the “problem is to group a given collection of unlabelled patterns into meaningful clusters” [88]. In classification, the labelled (or training) patterns are used initially to study the description of classes and then to label a new pattern. In clustering, the labels are associated with the clusters and are obtained exclusively from the given data set [88].

Clustering is a form of unsupervised learning, in the sense, that there is no human expert assigning objects to classes [110]. There exist several different clustering techniques and algorithms. For instance, flat (or partitional) clustering creates a flat set of clusters, which do not have any explicit structure that would relate clusters to each other [110]. K-means is one of the most commonly used flat clustering algorithms. Hierarchical clusters create a hierarchy of clusters, a structure that is more informative than those created by flat clustering [110]. It is represented in a tree structure called a dendrogram. Hierarchical clusters are either top-down (divisive – split) or bottom-up (agglomerate – merge). With bottom-up algorithms, initially each object is considered as a singleton cluster. Next, the algorithm determines which pairs of clusters are the best candidates to merge, and continues to merge pairs of clusters until all the clusters have been merged into a single cluster, which contains all the objects [76],[110]. Conversely, top-down algorithms initially start with a single cluster, which contains all the objects. Next, the algorithm determines the pairs of clusters to split, and proceeds to split the clusters recursively until individual objects are reached [76],[110]. Hierarchical clustering methods continue to merge or split clusters until a terminating criterion is met [88].

Classification is a form of supervised learning, mainly because there is a human expert, who serves as a teacher directing the learning process, defining the classes and labels of training objects [110]. It aims “to replicate a categorical distinction that a human expert imposes on the data” [110]. Several classification techniques and algorithms have been proposed over the years including decision tree, nearest neighbour, and naïve Bayesian. A large number of classifiers are typically linear classifiers, where the classification decisions are determined by the values of the linear combination of the attributes (or dimensions) [76],[110]. Naïve Bayesian and Support Vector Machines are instances of linear classifier. An example of non linear classifier is k-nearest neighbour (kNN).

2.2.3. Different Stages of Clustering and Classification

Typically, clustering and classification involves following steps [76],[88],[110], (Figure 1):

1. Pattern representation – feature selection and/or features extraction
2. Pattern proximity measure – similarity measure and/or distance measure

3. Grouping and/or labelling – clustering and/or classifying
4. Data abstraction (if needed)
5. Assessment of output (if needed)

2.2.3.1. Pattern Representation

Patterns (or records as mentioned in 2.2.1) are represented as multidimensional vectors, where each dimension (or attribute) is a single feature [134]. Pattern representation is the first step towards clustering or classification. By carefully studying the features of the vectors in the original data set and performing necessary transformations, the comprehensibility of the clustering/classification results could improve significantly. Pattern representation is used to extract the most descriptive and discriminatory features in the original data set; then these features can be used exclusively in subsequent analyses [88]. Feature selection and/or feature extraction techniques are commonly used for this purpose.

Feature selection is the process of identifying the most effective subset of the original features for subsequent use [88],[96]. Feature extraction is the use of one or more transformations of the input features to produce new prominent features, i.e., it computes new features from the original data set [88],[123]. These methods are typically used to obtain an appropriate set of features to use in clustering or classification. The goal is to improve the clustering/classification performance and computational efficiency [107].

The main idea of the former is to select a subset of input data by eliminating features with little or useless information [96]. Feature selection typically identifies the important features and the correlations among them, which in turn enlightens the users about the data. Feature selection in supervise learning aims to find a subset of features that produces a higher classification accuracy, whereas the goal of feature selection in unsupervised learning is to find a good subset of features that form high quality of clusters for a predefined number of clusters [72],[96].

2.2.3.2. Pattern Proximity Measure

Pattern proximity is typically measured by the distance function defined on pairs of patterns, i.e., pair of feature vectors [88]. Although a variety of distance functions are

available, they usually belong to two main categories [134],[199]: similarity measures and distance measures.

The term proximity is generally used to denote either a measure of similarity or dissimilarity [76]. A simple distance measure is used to "reflect dissimilarities between two patterns", by measuring the discrepancy between them [76],[88]. Commonly used distance measures are: Euclidean, Manhattan, and City-block. Similarity measures are used to "characterize the conceptual similarity between two patterns", thus reflecting the strength of the relationship between them [76],[88]. Commonly used similarity measures are [134],[199]: Cosine Similarity, Extended Jaccard, and Asymmetric measure.

This is an important step in any clustering and classification technique as well as many other data mining tasks. In clustering, similarity (or distance) measure is fundamental to the definition of a cluster [88],[150]. Similarity or distance measure between two patterns extracted from the same features space is imperative in clustering [88]. These measures often influence the shape of the cluster, since some of the objects might be close to one another according to one measure and further away according to another [161]. The distance or similarity measures should be chosen carefully, considering the feature types and scales [88],[150].

Our research work on hardware support for data mining operations starts with investigations on similarity measure computations. The proposed chip-level hardwares for similarity measure computations are discussed and presented in Chapter 3.

2.2.3.3. Grouping

This step can be performed in a number of ways. Some commonly used grouping schemes are [20],[60],[88]: hierarchical or partitional, and hard or fuzzy.

The distinction between hierarchical and partitional clustering methods is that hierarchical approaches produce a nested series of partitions, whereas partitional approaches produce only one [20],[88]. The nested series of partitions, produced by hierarchical clustering, is based on the criterion for merging or splitting clusters using similarity (or dissimilarity) [20],[88]. Partitional methods identify the partition that typically optimizes a clustering criterion locally [60],[88].

With hard clustering, each pattern is assigned to one and only one cluster during a grouping. However, with fuzzy clustering, each pattern is associated with every cluster

using some form of membership function [88]. Fuzzy clusters are converted to hard clusters by allocating each pattern to the cluster, which has the largest membership function [88].

2.2.3.4. Data Abstraction

Data abstraction is a process which involves in extracting a simple and compact representation of a data set [88]. For clustering such as K-means, data abstraction produces a compact representation of each cluster, known as cluster centriods or cluster prototypes [110].

As illustrated in [88], data abstraction is useful in decision making because of the following reasons: Firstly, it gives a simple description of the clusters, making it easier for a human to comprehend; secondly, it helps in achieving data compression so that further processing can be performed efficiently; thirdly, it increases the efficiency of the decision making tasks.

2.2.4. Clustering/Classifying High Dimensional Data

An important issue often arises while clustering or classifying is the problem of having too many attributes (or dimensions). There are four major problems associated with clustering or classifying high-dimensional data [20],[76],[99]:

- Curse of dimensionality – multiple dimensions are impossible to envision; also, since the amount of data often increases exponentially with dimensionality, multiple dimensions are becoming increasingly difficult to enumerate [20],[76],[99].
- As the number of dimensions increases, the concept of proximity or distance becomes less precise; this is especially true for spatial data [68],[99].
- Irrelevant features (or attributes) – Typically, clustering groups objects that are related based on the attribute's value. When there is large number of attributes, some of the attributes (or features) might not be meaningful for a given cluster [99]. Having irrelevant features negatively affects the proximity measures and the creation of clusters [20],[99].
- Correlations among subsets of features – When there is a large number of attributes, it is highly likely that some of the attributes are correlated [99].

A common way to overcome the above problems of high-dimensional data space is to represent the appropriate pattern before performing any other data mining tasks. For clustering, for instance, good pattern representation generates clusters that are simple and easy to understand, whereas poor pattern representation generates clusters that have complex structures that are difficult to understand [88]. Feature extraction and/or feature selection techniques are used for this purpose. For classification also, it has been shown [2] that the classification accuracy significantly improves after reducing the dimensionality of the data by using feature extraction and/or feature selection methods.

There are various dimensionality reduction techniques. Some of the feature selection methods that can be used are: Mutual Information [110], Chi Square [110], Sensitivity Analysis [2],[196], etc. Some of the feature extraction methods that can be used are: Singular Value Decomposition [71],[140], Principal Component Analysis [87],[90], Independent Component Analysis [84], Factor Analysis [47], etc.

2.2.4.1. Principal Component Analysis for Clustering/Classification

Among these techniques, Principal Component Analysis (PCA) is the most commonly [2],[90],[140] used dimensionality reduction technique in clustering and classification problems. In addition, due to the necessity of having a small memory foot-print of data, PCA is applied to many data mining applications that are appropriate for portable and embedded devices such as: handwritten analysis or signature verification, palm-print or finger-print verification, iris verification, facial recognition, etc.

PCA is a classical technique [162]: The main idea is to extract the prominent features of the data set and to perform data reduction (compression). PCA finds a linear transformation, known as Karhunen-Loeve Transform (KLT), which reduces the number of the dimensions of the feature vectors from m to d (where $d << m$) in such a way that the "information is maximally preserved in minimum mean squared error sense" [64],[139]. PCA reduces the dimensionality of the data by transforming the original data set to a new set of variables called Principal Components (PCs) to extract the prominent features of the data [90],[162]. According to Yeung and Ruzzo [197], "PCs are uncorrelated are ordered, such that the k^{th} PC has k^{th} largest variance among all PCs; and the k^{th} PC can be interpreted as the direction that maximizes the variation of the projection of the data points such that it is orthogonal to the first (k-1) PCs". Traditionally, the first few PCs are used in

data analysis, since they retain most of the variants amongst the data features (in the original data set), and eliminate by the projection those features that are highly correlated among themselves; whereas the last few PCs are often assumed to retain only the residual noise in the data [90],[197].

Advantages and disadvantages of PCA: Since PCA effectively reduces the dimensionality of the data, the main advantage of applying PCA on original data is to reduce the size of the computational problem [162]. Normally, when the number of attributes of a data set is large, it takes more time to process the data, since the number of attributes is directly proportional to processing time; thus, by reducing the number of attributes (dimensions), running time of the system can be minimized [162]. In addition, for clustering, it helps to identify the characteristics of the clusters [88]; and for classification, it improves classification accuracy [2][196]. The main disadvantage of applying PCA is the loss of information, since there is no guarantee that the sacrificed information is not relevant to the aims of further studies, and also there is no guarantee that the largest PCs obtained will contain good features for further analysis [87],[90].

PCA for K-means: In cluster analysis, PCA is initially applied to reduce the dimensionality of the data set before clustering, hoping that PCs may extract the cluster structure in the data set [197]. The first few PCs, containing the most variation in the original data set, are typically used in cluster analysis [90],[162]. PCA is widely used for K-means clustering [51],[52],[198]. K-means clustering is one of the most popular and effective clustering methods, which uses K prototypes (or centroids) to represent clusters by optimizing the squared-error function [51],[198]. Despite its popularity, K-means suffers from a well-known problem, i.e., its coordinate descend search method has the tendency to converge to local minima [25],[198]. These studies [51],[52],[198] illustrate that "PCs are actually the continuous (relaxed) solutions of the cluster membership indicators in the K-means clustering method". In this case, PCA facilitates K-means clustering to find a local optimum solution [51],[52].

2.2.4.2. Process of PCA

PCA computation consists of four stages [87],[140],[146]: compute mean, compute covariance matrix, compute eigenvalues and eigenvector, and compute PCs. In this case, we consider the original data set $\{X\}_{m\mathrm{X}n}$ as an mXn matrix, where m is the number of

dimensions and *n* is the number of vectors. Firstly, the data set is taken as an input and the mean is computed along the dimensions of the vectors. Secondly, covariance matrix is computed after computing the deviation from the mean. Subtracting the mean from each of the data dimensions produces a data set whose mean is zero. Covariance is always measured between two dimensions [87],[146]. With covariance, we can find out how much the dimensions vary from the mean with respect to each other [87],[146]. Covariance between one dimension and itself gives the variance.

Thirdly, eigenanalysis is performed on the covariance matrix to extract independent orthonormal eigenvalues and eigenvectors [3],[90],[140],[146]. Eigenvectors and eigenvalues tell us useful information about the data. Typically, eigenvectors are considered as "preferential directions" of a data set, or main patterns in the data; and eigenvalues are considered as quantitative assessment of how much a PC represents the data [3],[146]. Eigenvectors with the highest eigenvalues corresponds to the variables (dimensions) with the highest correlation in the data set; the higher the eigenvalues of a PC, the more representative it is of the data [3],[90]. Lastly, the set of PCs is computed and sorted by the eigenvalues in descending order, which gives the PCs in order of significance [90].

Different ways of performing PCA: There are different methods of performing PCA computation. These methods might depend on the applications, data sets used, etc. The most common algorithm for PCA involves the computation of the eigenvalue decomposition of a covariance matrix [90],[140],[146]. Another technique for dimensionality reduction is singular value decomposition (SVD) [71],[140]. This involves the singular value decomposition of data matrix. In fact, PCA is closely related to SVD; hence the names are often used interchangeably [87],[140],[146]. In addition, there are different ways of performing eigeanalysis or eigenvalue decomposition (EVD). One well known EVD method is Cyclic Jocabi method [71],[133]. However, this is good for small matrices, where number of dimensions are less than or equal to 10 (m=<10) [71],[140]. For larger matrices [121], where the number of dimensions are more than 10 (m>10), other algorithms such as QR [2],[196], Householder [121], or Hessenberg [121] methods should be employed. Among these methods, QR algorithm, first introduced in 1961, is one of the most efficient and accurate methods to compute eigenvalues and eigenvectors

during PCA analysis [121],[157]. It can simultaneously approximate all the eigenvalues of a matrix. For our work, we are using QR algorithm for EVD.

In summary, clustering and classifying high dimensional data presents many challenging problems. The computational cost of processing massive amount of data in real time is immense. PCA, with least effort, can reduce a complex high dimensional data set to a lower dimension, in order to unveil the simplified structures that are otherwise hidden, while reducing the size of the computational cost of analyzing the data [87],[140],[146]. In our research work, we propose partial and dynamic reconfigurable hardware support for the first two stages of PCA computation, which is presented in Chapter 4. Hardware support could further reduce the computational cost of processing data and enhance the speed performance of the PCA analysis.

2.2.5. Characteristics of Data Mining Operations

In the introduction, we mentioned some of the characteristics of data mining applications. Those characteristics are applicable to the high-level data mining operations presented in the previous section. In this section, we elaborate more on the intrinsic characteristics of and also the computation models used in data mining operations, specifically in clustering and classification, in terms of programmability and performance. In order to provide hardware support for data mining applications, it is important to understand the intrinsic characteristics and computation models of these applications. They play a significant role in determining whether certain type of hardware support is indeed a good match for the specific application on a portable and embedded device. Further investigation on these computation models and intrinsic characteristics as well as how and why they could potentially benefit from various reconfigurable hardwares is discussed and presented in Chapter 5.

2.2.5.1. Programmability

At the highest level, clustering and classification consist of several stages including feature selection, similarity measure, grouping, and data abstraction. These multi-stage operations are typically very large and complex to fit into a single chip, which becomes an issue with portable and embedded devices with their limited hardware foot-print. In this case, some form of reconfigurable computing approach might be required to perform these

operations on a single chip, by decomposing the operations and executing them at different times.

In addition, there exist a large number of different algorithms for clustering and classification. These algorithms may use various methods to solve the same operation of a stage. In many cases, each operation of a stage itself is usually solvable by various methods, though with results of different quality. For instance, there are many ways to measure similarity including, Cosine Similarity, Extended Jaccard, and Asymmetric Measure. They often produce different results in dissimilar application contexts. In case if all these methods and algorithms do not fit into the chip, we can use some form of reconfigurable computing approach, and execute these operations in a plug and play fashion as needed.

Many data mining application involve dynamic decision making and changing the operations dynamically. For instance, among several available operations, one must consider the current objective and other criteria such as recent results obtained and the external stimuli, in order to determine the next actions without human intervention. Also, as in many current applications, most data mining applications have evolving, new and emerging algorithms. As a result, today's data mining applications require some form of programmability and post-fabrication reprogrammability.

2.2.5.2. Performance

Most of today's data mining operations are compute-intensive, which is true for the operations mentioned in the preceding sections. Generally, each stage of clustering and classification consists of lengthy computation processes. It might be possible to decompose a computation process into several modules and employ different techniques to enhance speed performance. Data mining operations are usually amenable to pipelining and also exhibit a significant amount of functional parallelism at different levels of granularity, that is, from fine-grain to coarse-grain. The inherent functional parallelism and pipeline nature in data mining operations can be exploited to a great extent in hardware. For instance, using hardware, these operations can be implemented spatially, distributed throughout the chip, which allows harnessing the parallelism in computations [23],[49]. Having multiple independent operations executed on several processing elements (PEs) in parallel at the same time, leads to better speed performance. Also,

pipelining can be used in complex computation processes to maximize throughput of the data.

In addition, a large volume of data is being processed in each stage, requiring a long period of processing time. However, these data typically do not have any data dependencies, allowing the data to be processed concurrently in each stage. As a result, we can exploit data parallelism in each stage of clustering and classification.

Usually in portable, handheld, and embedded devices, the applications are executed in a real-time fashion. In this case, the data is typically streamed in from different sources. Data mining operations must process the continuously streamed-in data in real time for the users. With streaming data, it is possible to conceal the data transfer overhead and latencies by overlapping them with the computations, leading to higher performance. Also, having streaming data inputs and outputs can be an advantage, since the applications may not be constrained by the bandwidth of existing system buses [77].

2.2.6. Related Work on Hardware Support for Data Mining Operations

We have surveyed existing research work on hardware support for data mining. Some of the hardware platforms proposed for data mining, such as application specific processors [65],[86], only deal with specific aspects of the entire data mining process. In addition, they are not well suited for the enormous amount of ever-growing, unstructured data that need to be processed.

One of the typical hardware solutions to improve the speed performance of data mining applications is to use the massive power of parallel computers or processor clusters, including shared memory parallel machines [89], distributed memory machines [74], and in particular, Google's cluster architecture [17]. However, parallel processing of data mining applications utilizing powerful but resource-intensive multiprocessors is beyond the means of individuals. These users often have no other option but to rely on embedded portable devices to perform these applications.

There has been minimal research effort spent in the area of chip-level hardware support for data mining. Most of the research [15],[29],[58],[103],[167] targets focused operations in data mining. Chamberlain's group implemented the Mercury System [29] and a Field Programmable Gate Array (FPGA)-based search engine [167]. Their effort is focused on

pattern matching, which is only one of the many data mining tasks. Baker and Prasanna implemented a systolic array architecture on FPGA specifically for the Apriori algorithm [15], while Eslick et al. focused on the K-means algorithm using FPGA [58].

Choudhary's group [37] has carried out relatively the most extensive work in this area. They proposed a co-processor for the compute-intensive tasks of clustering [37],[130] and classification [119] with performance gain of 11 times and 5.58 times over software, respectively. However, their designs are ad hoc and have not considered the characteristics of the applications in general.

2.3. Chapter Summary and Conclusion

In this chapter, we surveyed various means of hardware support for application specific operations, and advantages and disadvantages of using them for portable and embedded devices. From these details, it is evident that ASICs provide superior speed performance and consume low power, whereas microprocessors provide a flexible computing platform. Reconfigurable computing systems provide higher level of flexibility than ASICs without compromising performance.

We also surveyed data mining and its applications, specifically clustering and classification, and elaborate on their characteristics. Clustering/classification have three important stages: pattern representation; pattern proximity measure; and grouping. PCA is the most commonly used method to reduce the dimensionality of the data in clustering and classification. Similarity measure is then applied to the reduced data set before grouping.

As illustrated in the previous section, many of today's data mining applications are compute-intensive, requiring more processing power than ever before. Also, these applications require some form of flexibility, because of the ever-changing application environments. Meeting these performance and flexibility requirements under the area, power, cost, and time-to-market constraints of portable and embedded devices are becoming increasingly challenging.

In addition, from our survey on existing research work for hardware support for data mining operations, it is evident that there has not been much progress in this important application area. Hence, it is worthwhile to investigate how to provide hardware support for data mining operations on portable and embedded devices, while considering the

various constraints associated with the computing platform as well as with the application domain.

In the next chapter, we analyze the advantages and disadvantages of using hardware over software. We discuss and present our investigation on the feasibility and performance gain of using hardware, and advantages of using parallel hardware.

Chapter 3

3. Hardware Support for Data Mining Operations

In this chapter, our objective is to investigate: the feasibility and potential performance gain of using hardware for data mining operations; and advantages of using parallel hardware.

First, we analyze the advantages and disadvantages of using hardware over software on microprocessors in 3.1. Note that our investigation and discussion on microprocessor focuses on a single CPU system rather than multiple processors or multi-core systems. Next, we investigate the feasibility and potential performance gain of using hardware for data mining. Our investigation [104],[124],[125] focuses on the hardware design and implementation of similarity measures and similarity matrix computation, discussed and presented in 3.2 and 3.3.

We exploit the inherent parallelism and pipeline nature in data mining operations to a great extent. Consequently, we introduce an SIMD-type (Single Instruction Multiple Data type) processor array [126] for parallel computation of similarity measures in 3.4. An algorithm is developed to assign the similarity measure computations to each processing element (PE) of the processor array efficiently.

3.1. Analysis – Hardware versus Software

To process a specific application in portable and embedded devices, one can use hardware such as ASICs and FPGAs, or use a processor to execute instructions. The advantages and disadvantages of using hardware versus software are discussed below.

Performance: With hardware, such as ASICs and FPGAs, customized circuits can be provided to perform specific computation(s), allowing them to quickly and efficiently compute the specific task that it is customized for, leading to superior speed performance. However, with processors, the underlying hardware circuits are general-purpose and designed for software; hence performance is inferior to ASICs or FPGAs.

Unlike processors, ASICs and FPGAs do not require an operating system (OS). Additionally, with the processor, the fetching/decoding/executing of each instruction,

leads to high execution overhead [39]. These instruction and OS overheads can be eliminated with hardware.

With ASICs and FPGAs, the computations are implemented spatially (in space), whereas with microprocessors the computations are implemented temporally (in time) [23],[49]. As illustrated in [49], with spatial computing, "the operators exist at a different point in space, which allows the computation to exploit parallelism to achieve high throughput and low computational latencies", whereas with temporal computing, "a small number of more general compute resources are reused in time, which allows the computation to be implemented compactly". Spatial (parallel) computing of these hardware-based designs is more efficient in comparison to the temporal (sequential) computing of microprocessor-based designs [111]. These hardware-based designs can leverage a variety of parallelism means to a greater degree than microprocessor-based (software-only) designs. They achieve better speed performance by exploiting: fine-grain and coarse-grain parallelism; data parallelism; and pipelining.

Flexibility: Typically, hardware (except reconfigurable hardware) has fixed functionality, which cannot be altered after the manufacturing process, whereas microprocessors provide a flexible computing platform to perform the required computations. The processor executes an application, which is realized using a software program, by decoding the program instructions. The functionalities of the system can be changed, merely by changing the program instructions, without changing the underlying hardware [23],[39]. Therefore, unlike fixed hardware such as ASICs, with microprocessors, numerous applications can be performed on a single processor. However, reconfigurable hardware such as FPGAs also provides a flexible computing platform to process a variety of applications, since on-chip hardware circuitry can be changed post fabrication.

Power: Low-power operation is critical to many portable, handheld and embedded devices "to improve battery life, to reduce costs of operation, and even to improve reliability" [67],[156]. It is crucial to maintain the power consumption at the desired level to ensure correct and useful operation of the system. Keeping the power dissipation low, will lead to lower device temperature and will reduce the effect of temperature-driven failures, thus enhancing the reliability of these systems [143]. Computations on hardware,

often consume less power than their software counterparts running on microprocessor, since they are typically implemented at lower clock rates and they also avoid the overhead of instruction cycles [67]. In hardware, the circuitry is optimized for specific application(s), such that the power consumption is usually much lower than that of a microprocessor [156]. Also, high-performance processors often consume high "power (100 watts or more)", making them unsuitable for many portable and embedded devices [24],[135].

Time-to-market: In general, time-to-market is reduced by using an off-the-shelf microprocessor. However, with fixed configuration hardware such as ASICs, we have to design the circuitry for an application, verify the designs, and fabricate the chip. Reconfigurable hardware, such as FPGAs, is also immediately available since it has already been fabricated. This would eliminate the fabrication time, potentially reducing the time to market.

Area: Another important issue in portable and embedded devices is their limited hardware foot-print. With ASICs and FPGAs, we can have customized and optimized circuits, which usually occupy less space on a chip, compared to general-purpose microprocessors.

In summary, portable and embedded computing systems often have stringent performance, area, and power requirements. In order to satisfy these requirements, we have to incorporate special-purpose hardware into embedded systems designs. From the above discussion, it is evident that hardware-based designs satisfy high-performance and low-power requirements, whereas microprocessor-based designs allow flexibility and reduced time-to-market. With reconfigurable hardware, we can further achieve the required flexibility and reduced time-to-market.

3.2. Initial Investigation and Proof-of-Concept

In this section, we investigate whether it is feasible and a worthwhile endeavour to use hardware for data mining operations. Since Cosine Similarity is one of the most widely used similarity measure functions in data mining, a hardware implementation [104] for Cosine Similarity is carried out as a proof-of-concept work towards our goal.

3.2.1. Design Approach and Development Platform

It is obvious that any hardware accelerator will outperform its software counterpart. Therefore, in addition to being a proof-of-concept investigation, we want to lay the ground work enabling further experiments for more complex and higher level design in the area of data mining.

We designed a simple Cosine Similarity function that makes use of lower level operators in a hierarchical manner. This is akin to a complex design at the system level (which is our eventual goal) using current practice of platform-based design approach incorporating components that can be abstracted and reused [50]. Similarly, we followed the software practice of information hiding and reuse. This parallel between the software and hardware implementations of the same operators allows us to make sound performance comparison, as well as building a library of components in both hardware and software for further experimentation.

There are several design issues that came up during the course of this investigation. The major ones being optimization during software compilation (we used C for our software modules), integer versus floating point computation, and the software constructs used, specifically in this case, function versus procedure calls and the use of For Loops. These issues are important when considering software performance as it is dependent on the skill and coding style of the programmer, the compiler used, and the targeted processor.

3.2.1.1. Experimental Platform

We used the AMIRIX AP-1000 [9] FPGA Development Board to carry out all our software and hardware experiments presented in this Chapter. This platform supports the Xilinx Virtex-II Pro FPGA [195], and has a large FPGA gate capacity and two embedded PowerPC (hard) processors and MicroBlaze (soft) processors for development purpose. The soft processor is built using the FPGA general-purpose logic. Unlike the hard processor, the soft processor must be synthesized and fit into the available gate arrays.

Our hardware modules are written in VHDL incorporating the IEEE Standard Logic Library 1164. For hardware experiments, we used the Xilinx Integrated Software Environment (ISE) 7.1i to design the circuits. As well, results obtained from simulations using Modelsim SE were compared to the results obtained from Xilinx ChipScope Pro

7.1i to verify and cross-check the correctness of the functionality and the reported execution time of our designs.

Our software modules are written in C and experiments were performed on the MicroBlaze soft core in order to make a fair comparison with the hardware designed and implemented on the same FPGA. Xilinx Embedded Development Kit (EDK) 7.1i is used for profiling the software modules and for verification purpose. Profiling data gives important details of the software execution behaviour including the average time spent on a function or procedure call, average time spent on a function or procedure and its descendents per call, etc. Additionally, assembly code generated by the compiler was examined to theoretically verify the results and behaviour of the software modules on the MicroBlaze for all the experiments performed in this chapter.

Software modules on the MicroBlaze soft core were compiled both with no optimization and with level II optimization in order to get a more accurate picture of performance. Level II was selected over other levels for optimization because it is the standard optimization level used for program deployment and because it activates nearly all optimizations that do not involve a speed-space tradeoff [194]. Therefore, the executable would not increase in size in a way that would impact performance because of the limited real estate on chip. For the hardware modules, no optimization was made.

The execution times, presented in this dissertation, for all the hardware and software designs were obtained while they were actually running on the chip.

The performance gain or speedup resulting from the use of hardware over software was computed using the following formula:

$$Speedup = \frac{BaselineExecutionTime(Software)}{ImprovedExecutionTime(Hardware)} \quad (1)$$

The following terminologies are used throughout this chapter:

- Software overhead: is the execution time overhead associated with the For Loops, function or procedure calls.
- Hardware overhead: is the initial setup time, i.e., the time taken for a circuit to produce the first result upon receiving the first set of inputs.

3.2.2. Fundamental Operators

3.2.2.1. Hardware Designs

The hardware modules for an adder and a multiplier were designed in VHDL. We decided to use the integer operators since the elements in the vectors are integers. We tested each individual design separately and found that both the adder and the multiplier are able to produce the desired output in every clock cycle (i.e., 12.5 ns with an 80-MHz system clock.) The maximum frequency for these 32-bit input hardware operators are 104 and 124 MHz, respectively.

We have experimented with several dividers and studied their characteristics. Since the Cosine Similarity function needs to generate a fractional remainder, we selected the pipelined divider v3.0 from the Xilinx IP core library. This 32-bit input divider requires a setup time in cycles equal to the number of bits of the dividend plus the number of bits of the remainder plus one. This divider is able to run at a maximum clock rate of 96 MHz and produces a result in every clock cycle after an initial setup time of 65 clock cycles.

3.2.2.2. Software Designs

```
//Sample code for adder function
unsigned int result, p, q, reg1;
unsigned int Adder(unsigned int x, unsigned int y);
int main(){
        p = 2; q = 3;
        reg1 = adder(p,q);
}
unsigned int Adder(unsigned int x, unsigned int y){
        result = x + y;
        return result;
}
```

```
//Sample code for adder procedure
unsigned int result, x, y, reg1;
void Adder();
int main(){
        x = 2; y = 3;
        adder();
        reg1 = result;
}
void Adder(){
        result = x + y;
}
```

Figure 2 Adder as a Function vs. a Procedure

We experimented with two different ways of configuring the fundamental operators, either as functions or procedures. Depends on one's preference and the particular requirement of the software modules, a programmer may decide to use either programming constructs or both. In general, a function call involves the passing of parameters and thus an increase in overhead of dealing with the stack frame, while a procedure call without parameter passing can use global variables though it may affect data memory allocation efficiency.

Figure 2 shows the sample codes for the ‘add’ operator implemented as a function versus a procedure.

The MicroBlaze can be customized to use a hardware floating-point unit to perform division, with the use of the assembly instruction ‘fdiv’.

3.2.2.3. Fundamental Operators Performance Comparison

In Table 1, the execution times of hardware and software versions of the fundamental operators are shown for one single operation with 32-bit inputs. The software implementations include the varieties of having the operators implemented as functions or procedures, and having no or level II optimization.

It can be observed that the hardware operators outperform their software equivalent in both the function and procedure cases, and with or without optimization. Note that we did not include the setup time for the hardware pipelined divider as the effective division time is only 12.5 ns when we process a continuous stream of data. The same cannot be said for the software division even though we were using a hardware floating-point unit for its computation.

Fundamental operators	**Hardware execution time (ns)**	**Software execution time (ns)**							
		Procedures				**Functions**			
		No optimization		**Level II optimization**		**No optimization**		**Level II optimization**	
		Execution time	**Gain**	**Execution time**	**Gain**	**Execution time**	**Gain**	**Execution time**	**Gain**
Adder	12.5	225.0	18	125.0	10	287.5	23	75.0	6
Multiplier	12.5	250.0	20	150.0	12	325.0	26	100.0	8
Divider	12.5	587.5	47	487.5	39	650.0	52	437.5	35

Table 1 Fundamental Operators Performance Comparison

3.2.3. Multiply and Accumulate (MAC)

3.2.3.1. Hardware MAC

The MAC hardware operator was designed as a sequence of multiplier, adder, and an accumulator register providing a feedback loop to the adder. The multiplier and adder presented in the previous section were re-used. Figure 3 shows this configuration.

This MAC module, running at 80 MHz, produces the first result after 2 clock cycles and consecutive results at every clock cycle thereafter. As a comparison, the Texas

Instrument C64x DSP [154] running at 1GHz requires a setup time of 15 cycles and produces four 16-bit results per cycle, while a PowerPC [85] 630 running at 200 MHz has a setup time of 4 clock cycles and produces a MAC result per cycle. Our MAC design, which we have not attempted to optimize, is comparable to the ones found in commercially available processors, and can run at a maximum clock speed of 140 MHz.

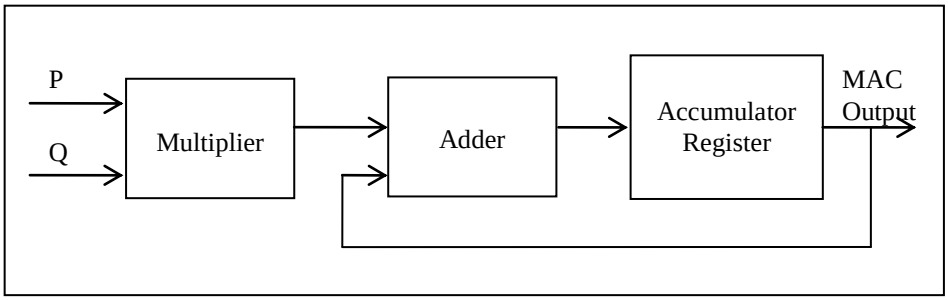


Figure 3 MAC Hardware Configuration

3.2.3.2. Software MAC

In the software implementation for the MAC operator, two versions were examined. With one version, a For Loop is implemented within the MAC module to perform all the required computation for *n* elements in the vectors. This would be the typical programming practice to process *n* items, especially with *n* being unknown in advance.

The other MAC version has the main program processed the multiply-and-accumulate computation by calling the MAC module *n* times, while the MAC module simply performs one multiplication and one addition by calling their respective modules. This software version was formulated to mimic the hardware equivalent of how *n* elements are being processed in the MAC module.

3.2.3.3. MAC Performance Comparison

The performance comparison of MAC hardware over software is shown in Table 2. It shows the operations for the dot product of two vectors of length 8. Note that the hardware execution time includes a 12.5 ns or one clock cycle setup time at 80 MHz. Unlike fundamental operators, the setup time is considered for MAC module, since it consists of several fundamental operators processing in a sequence; thus it takes time for the MAC module to produce the first result upon receiving the first set of inputs.

As expected, the performance gains are much higher than that of the fundamental operator cases, since the software overhead associated with the For Loop and function or procedure calls increases as the complexity or abstraction level of an operator (i.e., MAC) increases.

MAC implementation				**Total execution time (ns)**	**Performance gain**
Hardware				112.50	
Software	Procedures	MAC with For Loop	No optimization	9337.50	83.00
			Level II optimization	6481.25	57.61
		MAC without For Loop	No optimization	6562.50	58.33
			Level II optimization	4306.20	38.28
	Functions	MAC with For Loop	No optimization	10418.75	92.61
			Level II optimization	5212.50	46.33
		MAC without For Loop	No optimization	8862.50	78.78
			Level II optimization	3406.20	30.28

Table 2 MAC Performance Comparison

3.2.4. Cosine Similarity

The Cosine Similarity function is one of the most widely used similarity measures for clustering and classifying purpose. Given two vectors *p* and *q*, the Cosine Similarity is computed using a dot product and magnitude as follows:

$$\cos(p,q) = \frac{p \bullet q}{\|p\| . \|q\|} = \frac{\sum_{i=1}^{n} p_i q_i}{\sqrt{\sum_{i=1}^{n} p_i^2} \sqrt{\sum_{i=1}^{n} q_i^2}} \qquad (2)$$

Since we are working on a proof-of-concept basis, we have decided not to implement the square root in hardware though there are several efficient algorithms available. Instead, we implemented the square of *cos(p,q)* for the sake of simplicity without sacrificing generality and validity.

$$\cos(p,q)^2 = \left(\frac{p \bullet q}{\|p\| . \|q\|} \right)^2 = \frac{\left(\sum_{i=1}^{n} p_i q_i \right)^2}{\left(\sum_{i=1}^{n} p_i^2 \sum_{i=1}^{n} q_i^2 \right)} \qquad (3)$$

As a result, one multiply-and-accumulate (MAC) and one square (implemented as a multiplication) operations are needed in the numerator, and two MACs and one multiplication must be performed in the denominator, followed by a division. Thus, our hierarchical design consists of three fundamental operators: add, multiply, and divide at the lowest level; MAC at the next higher abstraction level; and the Cosine Similarity function at the highest level.

3.2.4.1. Hardware Cosine Similarity Module

Our hardware design for the Cosine Similarity takes advantage of its inherent parallelism, which adds another dimension of performance improvement especially in application-specific designs, in addition to the expected speedup using hardware over software.

As shown in Figure 4, the hardware module for Cosine Similarity consists of three sequential stages: three parallel MACs, two parallel multiplications, and a division. The inputs to the MACs are 8 bits, resulting in 16-bit outputs feeding into the multipliers. The divider takes 32-bit inputs and outputs a 32-bit fractional remainder and a 32-bit quotient, to produce a similarity measure ranging from 0 to 1.

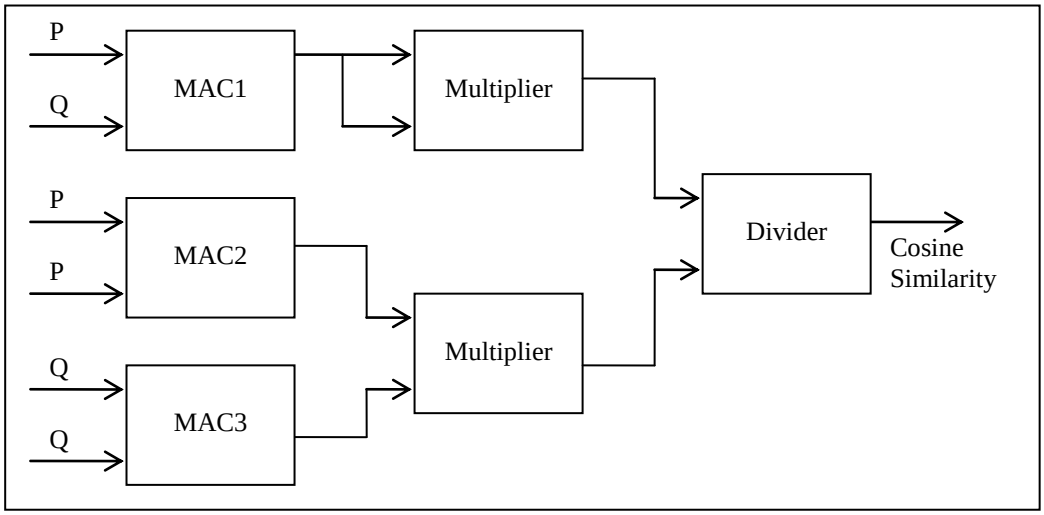


Figure 4 Cosine Similarity: Hardware Version

3.2.4.2. Software Cosine Similarity Module

In 3.2.3, it is observed that programming constructs play an important role in performance, especially with the For Loop. In order to take a more detailed examination of the effect of the For Loop, we have implemented three versions of Cosine Similarity. All lower level

modules are implemented as function calls instead of procedure calls to improve software execution.

The first version is shown in Figure 5, in which the Cosine Similarity calls the MAC module three times. The MAC in turn calls the multiplier and adder modules *n* times using a For Loop for the *n* elements in the vectors. In addition, the Cosine Similarity calls the multiplier twice and the divider once.

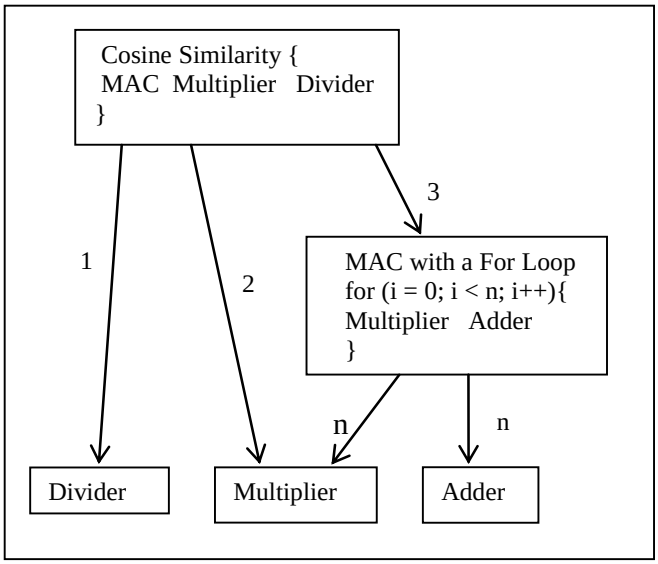


Figure 5 Cosine Similarity with For Loop in MAC

The second version of the Cosine Similarity has no For Loop within itself and in the MAC. The MAC simply performs a multiply and an addition, and is invoked *3n* times by the Cosine Similarity. The last version has three For Loops in the Cosine Similarity module itself. Each For Loop calls the MAC *n* times, and the MAC in turn calls the multiplier and the adder modules each once.

In all three Cosine Similarity versions, the divider uses floating-point operations, whereas other fundamental operations are performed using integer arithmetic. Thus, there is integer-to-floating-point conversion overheads involved as dictated by the compiler. This takes 424 cycles per conversion, thus a total of 848 cycles for two conversions in one Cosine Similarity software execution is required. This is a significant amount of overhead. For all the experiments in this chapter, we exclude this overhead in the software execution times to avoid exaggerating the hardware performance gain.

3.2.4.3. Cosine Similarity Performance Comparison

Table 3 shows the performance comparison of the three software Cosine Similarity modules with the hardware module. The execution times shown are for 2 vectors with 8 elements each. It should be noted that the hardware module takes 66 clock cycles or 825 ns of setup time at 80 MHz, and thereafter, produces a result in every 100 ns (in 8 clock cycles). The software overhead includes function calls, and For Loops, if any.

One might argue that the performance gains have decreased at this higher level of complexity as compared to previous lower levels, but this is largely due to the overhead associated with the hardware divider setup time. If a large volume of data streams through the Cosine Similarity hardware module, the setup time renders to nil while the overhead associated with the software module will increase with the amount of data processed. This point is further elaborated and discussed in the next section.

The three versions of software seem to have comparable performance; however, this deserves a deeper examination as the results shown in Table 3, are for vectors of eight elements only. Further analyses on this issue are discussed in the next section.

Description			**Cosine Similarity calculation**	**Overhead associated**	**Total execution time**	**Performance gain**
Hardware performance (ns)			100.0	825.0	925.0	
Software performance (ns)	Cosine Similarity with For Loop in MAC	No optimization	27992.4	5826.3	33818.8	36.56
		Level II optimization	10843.8	6387.5	17231.3	18.63
	Cosine Similarity calls the MAC sequentially with no For Loops	No optimization	28012.5	6556.3	34568.8	37.37
		Level II optimization	10843.8	3587.5	14431.3	15.60
	Cosine Similarity with For Loops to call MAC	No optimization	28012.5	13081.3	41093.8	44.43
		Level II optimization	10843.8	9031.3	19875.0	21.49

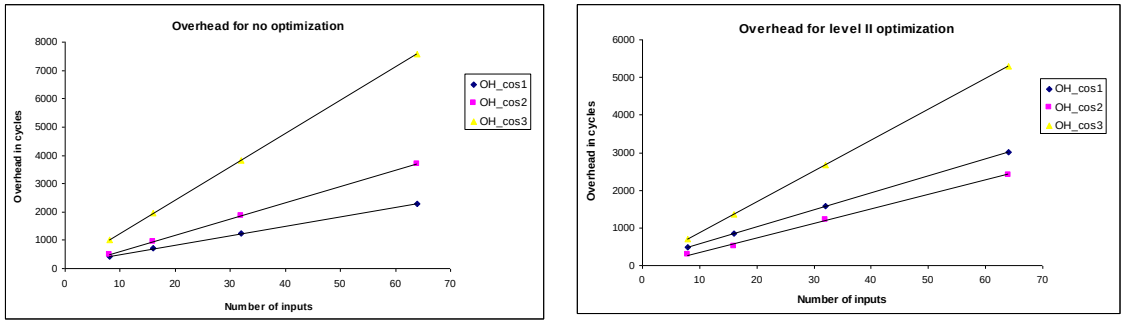
Table 3 Performance Comparison of Cosine Similarity

3.2.4.4. Analysis – Software Overhead versus Vector Size

We speculate that the seemingly lower performance gain at a higher abstraction level, as discussed in the previous section, is due to the low number of items being processed. Additional experiments were performed for the three Cosine Similarity software modules with vector size of 16, 32, and 64, using no and level II optimization.

Figure 6(a) and Figure 6(b) show the overhead with varying vector size for no and level II optimizations. In these figures, OH_cos1, OH_cos2, and OH_cos3 correspond to the three Cosine Similarity software versions respectively in 3.2.4.2. The overhead associated with each case indeed increases as the size of the vector increases. In contrast, the hardware overhead remains the same regardless of the number of items processed as the setup time is the only overhead and is constant.

It is interesting to note that OH_cos1 and OH_cos2 show different behaviour in the figures. Delving into more details concerning overhead, OH_cos1 has 3 For Loops, 6 function calls within Cosine Similarity, and 6*n* function calls within the MAC. OH_cos2 has (3 + 3*n*) function calls within Cosine Similarity, and 6*n* function calls within the MAC, while OH_cos3 has 3 For Loops in the Cosine Similarity resulting in 3*n* function calls within and 3 function calls outside the For Loops, and 6*n* function calls within the MAC.



(a) (b)

Figure 6 Vector Size vs. Software Overhead (a) None (b) Level II Optimization

From the above analysis, it is expected and shown in Figure 6(a), that OH_cos3 has the highest overhead, while OH_cos1 has the lowest overhead. In the case with level II optimization as shown in Figure 6(b), OH_cos3 still has the highest overhead as expected. However, in contrast to the no optimization case, OH_cos1 has higher overhead than OH_cos2. This is due to the fact that level II optimization does not perform loop unrolling. The foregoing analysis shows that the programming decisions such as software constructs used, level of optimization, etc., have a significant impact on software performance.

From these experiments, it can be concluded that hardware support for data mining operations is feasible and a worthwhile endeavour, provided that proper design strategies are used.

3.3. Further Investigation on More Complex Operations

We extended our investigation [104],[124],[125] to other similarity measures including Extended Jaccard and Asymmetric Measure [134],[199]. More complex designs at higher levels of abstraction, such as similarity matrix, were also investigated [104],[124],[125]. Experiments were performed on similarity matrices for all three similarity measures using a large number of vectors. Initial investigation on parallel hardware was also carried out. Performance gain of hardware over software was also examined.

3.3.1. Design Approach and Development Platform

Similar to our initial experiments, both software and hardware versions of the three similarity measures and the similarity matrix generation are implemented using a hierarchical platform-based design approach to facilitate component reuse at various levels of abstraction. As depicted in Figure 7, higher-level functions utilize lower-level sub-functions and operators.

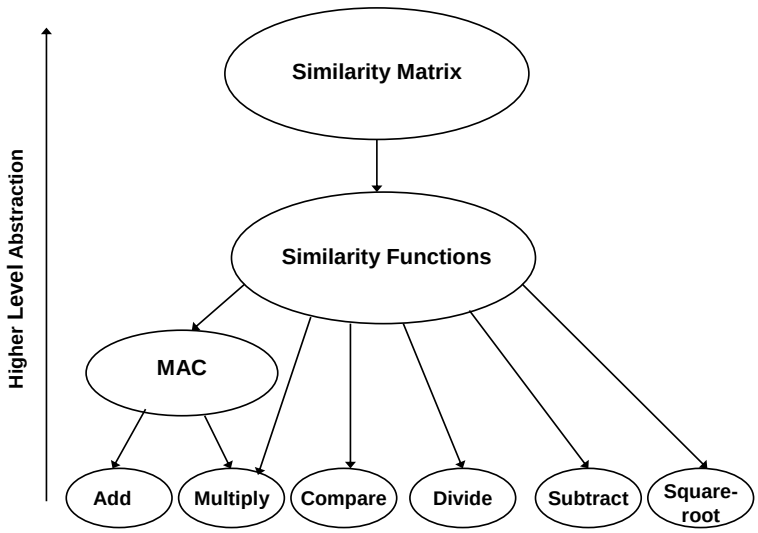


Figure 7 A Hierarchical Platform-Based Design Approach for Similarity Matrix

From our preliminary investigation, it is observed that programming decisions have a significant impact on software performance. Thus, we carried out experiments with a mix of various software design configurations. However, only the results for the most efficient software version are presented here. It should be noted that the same software configurations were used at all levels of abstraction to maintain consistency for the purpose of comparison.

The same development board, as well as the same tools and methods as in 3.2.1.1, are used to design, verify, and perform all our hardware and software modules and experiments. Additional software experiments were performed on a 502MHz UltraSparc IIe processor for cross platform comparison and to validate our designs. Execution times were measured using the clock function in C, which provides the CPU timing.

3.3.2. Similarity Measures

Since, hardware Cosine Similarity implementation for two vectors of size 8 executed up to 45 times faster than software [104] we extended our investigation to Extended Jaccard and Asymmetric Measure [134],[199]:

$$ExJaccard(p,q) = \frac{\sum_{i=1}^{n} p_i q_i}{\sum_{i=1}^{n} p_i^2 + \sum_{i=1}^{n} q_i^2 - \sum_{i=1}^{n} p_i q_i} \quad (4)$$

$$Asymmetric(p,q) = \frac{\sum_{i=1}^{n} \min(p_i, q_i)}{\sum_{i=1}^{n} p_i} \quad (5)$$

The hardware and software Cosine Similarity function using equation (3) in 3.2.4 are used for the experiments in this section.

3.3.2.1. Hardware Similarity Designs

The hardware design for Extended Jaccard is shown in Figure 8. It consists of three multiply-and-accumulate operators (MACs) (executing in parallel) and the fundamental

operators add, subtract, and divide. All the fundamental operators are in integers, except for the divider. The fractional divider produces a similarity measure ranging from 0 to 1.

The Asymmetric Measure module, as shown in Figure 9, consists of a comparator, two accumulators, and a divider. The hardware comparator compares the values of the two inputs and outputs the minimum. Each accumulator was designed as a sequence containing an adder and an accumulator register with a feedback loop to the adder.

For the hardware implementations, the same MAC operator (Figure 3) and Cosine Similarity function (Figure 4) are used from 3.2.3.1 and 3.2.4.1 respectively.

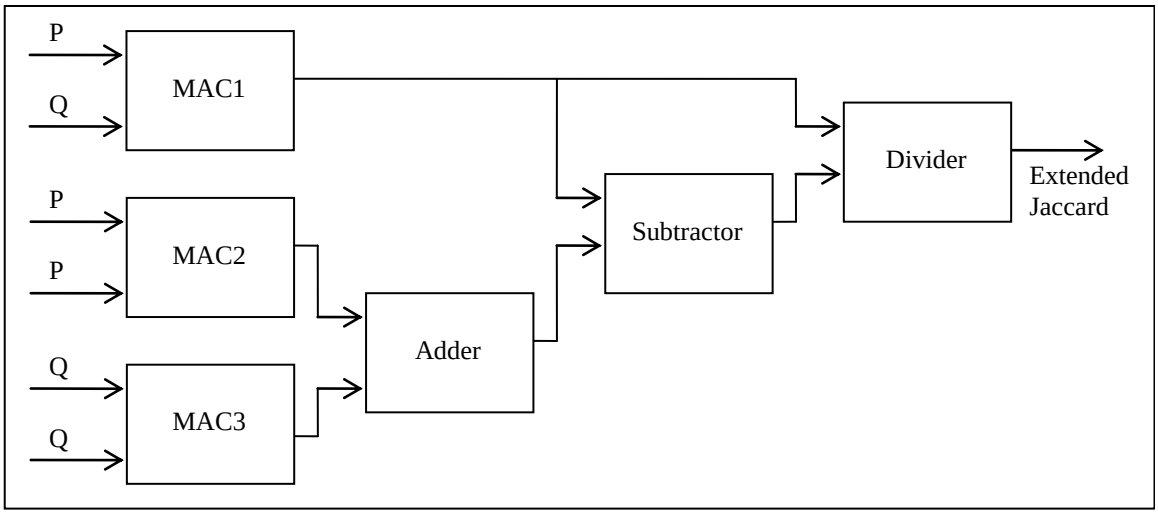


Figure 8 Extended Jaccard: Hardware Version

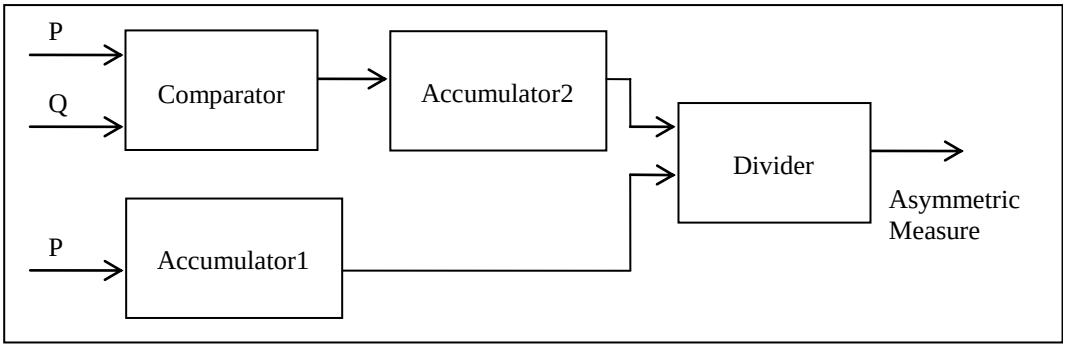


Figure 9 Asymmetirc Measure: Hardware Version

3.3.2.2. Software Similarity Designs

The Extended Jaccard software module, as shown in Figure 10(a), calls the MAC modules three times, and the MAC in turn calls the multiplier and adder modules *n* times each,

using a For Loop for the *n* elements in the vectors. In addition, Extended Jaccard calls the adder and subtractor one time each and the divider once.

The software version of Asymmetric Measure (Figure 10(b)) calls the accumulator module and the comparator-accumulator module one time each, and the divider once. The accumulator and comparator-accumulator in turn, using a For loop for the *n* elements in the vectors, call the adder modules *n* times, and the adder and the comparator modules *n* times, respectively. For the software version of Cosine Similarity, we used the one with the For Loop in MAC module (Figure 5) from 3.2.4.2.

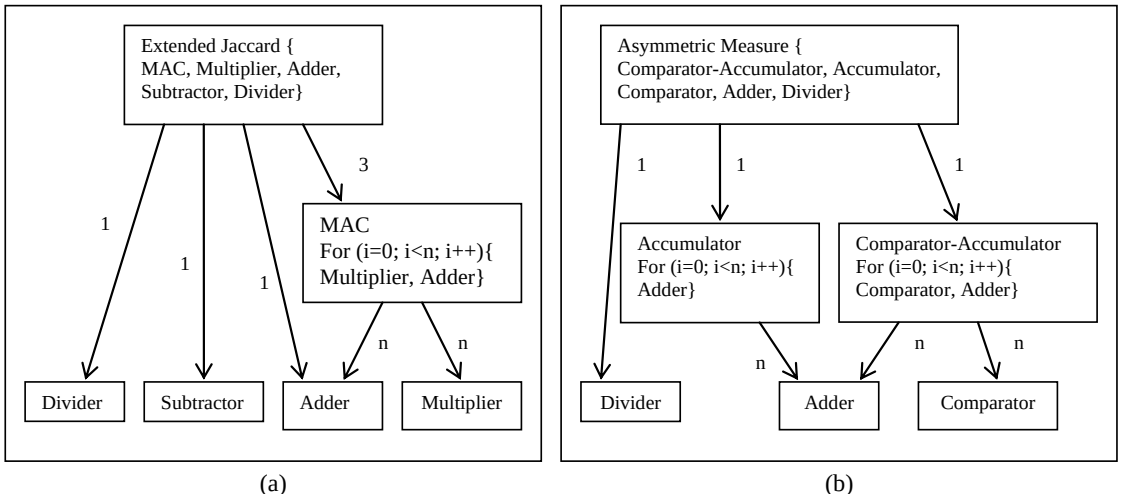


Figure 10 Software Versions: (a) Extended Jaccard (b) Asymmetric Measure

3.3.2.3. Performance Comparison: Similarity Measures

Table 4 shows the hardware (first row) and software (the last six rows, with no optimization and level II optimization) performance comparison of the three similarity functions. The similarity functions were computed for 2 vectors of 8 elements each. The execution times for the hardware modules are the same for all the similarity measures because of their similar degree of data dependency. After an initial setup time of 66 clock cycles, each hardware module produces a similarity result every 8 clock cycles at 80 MHz, whereas software overhead varies with the number of function calls, For Loops, and optimization levels.

As shown, the hardware speedups of the three similarity measures with no optimization in software are much higher than those with level II optimization, as level II optimization

increases software performance almost twofold. Furthermore, the performance gains for Cosine Similarity and Extended Jaccard are quite similar, while Asymmetric Measure is the lowest gain.

	Optimization level	Similarity calculation (ns)	Overhead (ns)	Total time (ns)	Speedup
Hardware for all three		100	825	925	
Cosine Similarity	None	27992	5826	33819	36.56
	Level II	10844	6387	17231	18.63
Extended Jaccard	None	27963	5806	33769	36.51
	Level II	10794	6656	17450	18.86
Asymmetric Measure	None	15285	3059	18344	19.83
	Level II	5669	4206	9875	10.68

Table 4 Performance Comparison for Three Similarity Measures

3.3.3. Similarity Matrix Computation on FPGA

We further our investigation to more complex designs, specifically, similarity matrix computation.

3.3.3.1. Similarity Matrix Design

We have designed hardware and software modules to generate a similarity matrix for the three similarity functions. Similarity matrices are a commonly used data structure to represent the similarity among a set of n document vectors; our similarity matrix is an $n \times n$ symmetric matrix with unit diagonal. Each element of the similarity matrix shows the similarity measure between two documents.

Both the hardware and software similarity matrix modules are designed to calculate the similarity measures between distinct vector pairs only. The similarity matrix function is implemented at a higher level than the similarity measure functions to maintain the hierarchical nature of our design. For software, the similarity matrix consists of two For Loops. Within these two For Loops, the respective similarity function is called to calculate the similarity measures.

3.3.3.2. Results and Analysis of Hardware Modules for Similarity Matrix

Each hardware design generating similarity matrix for the corresponding similarity measures, produces similarity result every 100 ns, after an initial setup time of 66 clock cycles; thus, one similarity calculation, i.e., a similarity measure between 2 vectors of size

8, takes 100 ns. The execution times for the hardware modules for all three similarity matrix functions are the same because of their similar degrees of data dependency.

Since a normal graph would not give us much insight into the execution characteristics, a logarithmic graph is used for the execution time for the similarity matrix hardware module among a varying number of vectors ranging from 2 to 8192 in increments of powers of two, as shown in Figure 11. It can be observed that the percentage of increase in execution times is linear for the similarity matrix among vectors ranging from 16 to 8192. By contrast, as can be seen at the beginning of the graph, the execution times for the similarity matrix among vectors of size 2, 4, and 8 have a different behaviour. This discrepancy is due mainly to the fact that a large percentage of the total execution time is spent on the overhead (i.e., the initial setup time).

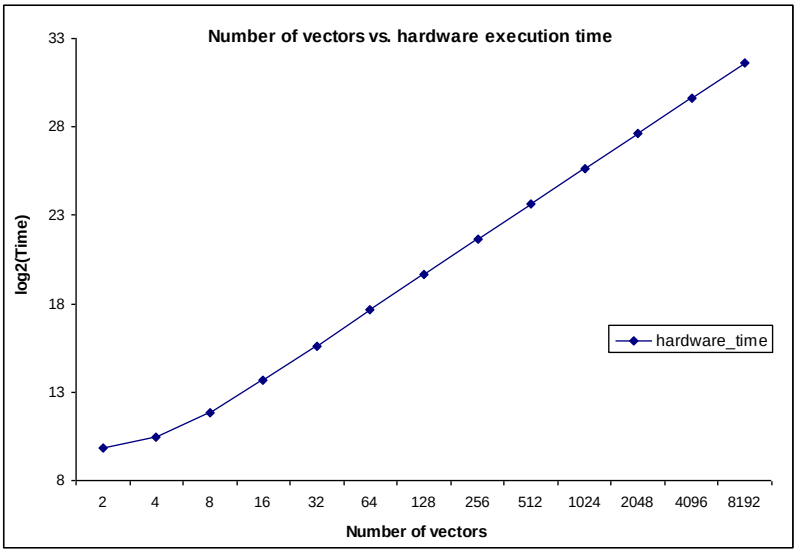


Figure 11 No. of Vectors vs. Hardware Execution Time for Similarity Matrix

Number of vectors	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192
Percent actual	11	42	77	94	98	100	100	100	100	100	100	100	100
Percent overhead	89	58	23	6	2	0	0	0	0	0	0	0	0

Table 5 No. of Vectors vs. Percentage of Time Spent on Overhead

Table 5 shows the percentage of time spent on actual similarity calculation (row 2), and the percentage of time spent on overhead (row 3). The percentage of time spent on overhead decreases with increasing number of vectors because the hardware overhead

consists only of setup time, which stays constant. For vector sets of 64 and beyond, the overhead is virtually non-existent.

Further, we have derived a theoretical formula to predict the hardware execution time for similarity matrix generation.

$$CPUClockCycles = N + R * Q * S \quad (6)$$

where N is the number of clock cycles for the initial setup time; R is the number of elements per vector; Q is the number of distinctive vector pairs; and S is the number of clock cycles required to produce one similarity-measure result after the initial setup time.

We can accurately predict the hardware execution time for similarity matrix generation among any number of vectors with any number of elements using formula (6), which we used to validated the hardware experimental results, by multiplying the theoretical result with the clock period of the design.

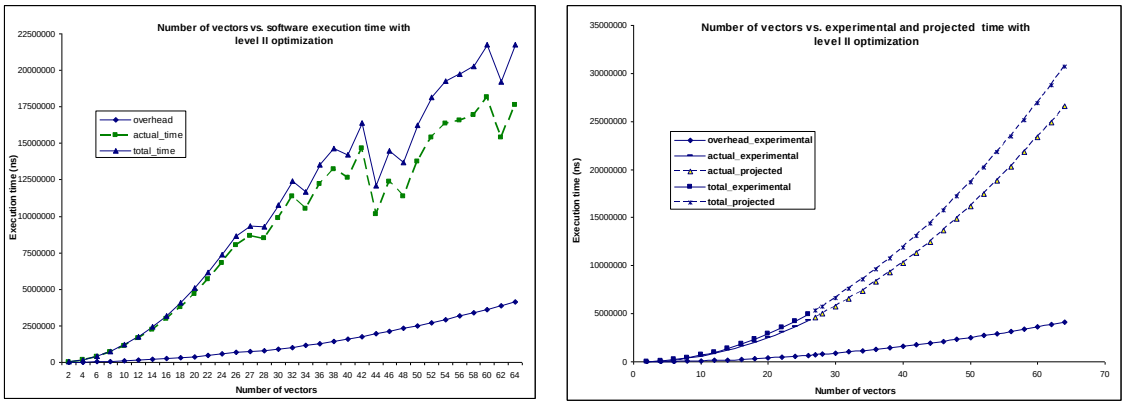
3.3.3.3. Results and Analysis of Software on MicroBlaze for Similarity Matrix

Software versions of the similarity matrix for all three similarity functions were executed on the MicroBlaze processor, using an even number of vectors ranging from 2 to 64 with an 80MHz system clock. Because of the limited on-chip memory of the FPGA used, experiments for higher numbers of vectors could not be performed.

Profiling data for the overhead, actual computation time, and the total time (overhead plus actual) were collected. Figure 12(a) shows the number of vectors versus execution time for the Extended Jaccard Matrix with level II optimization. Similar behaviour is observed for no optimization. In Figure 12(a), the top line indicates the total execution time, the middle line indicates the actual time spent on similarity matrix computation, and the bottom line indicates the overhead.

Graphs with similar behaviour are obtained for the similarity matrix function using Cosine Similarity and Asymmetric Measure. It has been observed that the profiler has certain limitations with respect to the execution times reported. The profiler is able to give correct results for a set of up to 26 vectors (i.e., 325 similarity-measure calculations) in similarity matrix computations at both optimization levels. This fact has also been verified

by examining the assembly code generated for the C code software. In addition, we were able to establish the execution times for each similarity measure from the assembly code.



(a) (b)

Figure 12 No. of Vectors vs. (a) Software Execution Time (b) Experimental and Projected Results

Further, it has been verified that the profiler gives correct results for the overhead of similarity matrix computation for a set of up to 56 vectors (i.e., 1540 similarity-measure calculations) with no optimization. For level II optimization, the profiler reports correct results for the overhead in computing the similarity matrix for up to 64 vectors (i.e., 2016 similarity-measure calculations). These details indicate that the profiler gives us confidence that measuring execution timing at the ns level is indeed accurate and can be used for scale-up projection.

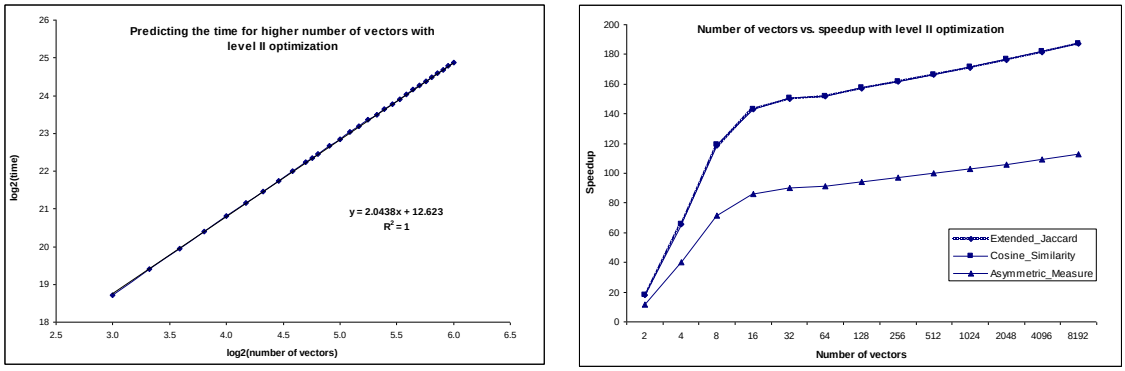
The overhead experimental results and the actual execution times for each similarity measure are used to project the expected execution times for similarity matrix computation beyond the 26-vector set, up to 56 vectors and 64 vectors for no optimization and level II optimization. Figure 12(b) shows the number of vectors versus execution time (experimental and projected results) for the Extended Jaccard matrix with level II optimization. No optimization shows similar behavior.

From Figure 12(b), it can be seen that the execution time increases proportionately with increasing number of vectors for the experimental results as well as for the projected results from 2 to 64 with level II optimization. Similar results are obtained for similarity matrix generation using Cosine Similarity and Asymmetric Measure.

3.3.3.4. Predicting MicroBlaze Performance

Experimental and projected results from the previous section were used to predict the execution times for numbers of vectors greater than 64, using regression technique, since higher number of vectors could not be processed, due to limited on-chip memory.

As shown in Figure 13(a), a logarithmic graph is plotted for the total execution time for similarity matrix computation among an even number of vectors in the range 8, 10, . . . , 64 for level II optimization, in order to predict the execution time for higher numbers of vectors.



(a) (b)

Figure 13 (a) Predicting Execution Time (b) Speedup: Hardware vs. Software Similarity Matrix

Number of vectors	Total time for similarity matrix computation on MicroBlaze (ns)					
	Extended Jaccard		Cosine Similarity		Asymmetric Measure	
	no optimization	level II optimization	no optimization	level II optimization	no optimization	level II optimization
2	43519	16625	43569	16663	28281	10556
4	256856	93769	257156	93994	165431	57400
8	1193738	430263	1195325	431313	766381	259606
16	5110113	1832350	5116150	1836888	3275238	1099988
32	21112576	7561826	21137051	7581149	13558399	4548473
64	87041495	30704671	87162244	30767442	55886636	18449133
128	360493998	127825462	360994096	128056035	231590366	76837114
256	1493034131	527062935	1495105351	527940462	959694510	316844559
512	6183600633	2173239457	6192178868	2176556010	3976907885	1306536244
1024	25610209436	8960921786	25645737346	8973352875	16480032083	5387616444
2048	106068109230	36948583356	106215252820	36994711578	68292116723	22216307502
4096	439295267131	152350153778	439904681999	152519209232	282997823249	91610886577
8192	1819400130011	628185636576	1821924103233	628795527594	1172723468045	377765501252

Table 6 Total Time for Similarity Matrix on MicroBlaze: None and Level II Optimization

Similar logarithmic graphs are drawn for similarity matrix computation using Cosine Similarity and Asymmetric Measure. Graph-fitting equations are generated for all three measures. The R-squared values indicate that the equations fit the data points very well. When the results from the prediction equations are compared to the actual experimental data, the margins of error are less than 1% for all three cases at both levels of optimization.

The predicted results among vectors ranging from 64 to 8192 together with experimental results for 2 to 32 vectors, for no optimization and level II optimization, are shown in Table 6. Both these results are useful for performance comparison with the hardware results from 3.3.3.2.

3.3.3.5. Performance Comparison: Hardware vs. Software on MicroBlaze

The speedups for similarity matrix computation with Extended Jaccard, Cosine Similarity, and Asymmetric Measure increase from 47 to 542, 47 to 543, and 31 to 350 respectively, with no optimization, and from 18 to 187, 18 to 187, and 11 to 113 with level II optimization, for a number of vectors ranging from 2 to 8192. Further, the performance gains using Extended Jaccard and Cosine Similarity measures are quite similar for both levels of optimization, whereas the performance gain using Asymmetric Measure is the lowest of the three results at both levels because of the less parallel nature of this measure.

As shown in Figure 13(b), the performance gain increases constantly for similarity matrix computation from 16 vectors upwards, but the performance gains among vectors of size 2, 4, and 8 display a different trend, due to the fact that hardware execution time does not increase proportionately with the number of vectors from 2 to 8. Similar trends were observed for no optimization. As discussed in 3.3.3.2, this is due mainly to the fact that a large percentage of the total hardware execution time is spent on overhead.

3.3.3.6. Performance Comparison: Similarity Matrix Using Four Hardware Modules in Parallel

To further examine hardware advantages, a parallel configuration was employed: four independent hardware modules in parallel on the FPGA were used to generate similarity matrix for the corresponding similarity measures. Each hardware module calculates an

element, i.e., one similarity measure between two vectors, of the similarity matrix. There is no data dependency among the hardware modules.

For the sake of simplicity we used a small 8-vector similarity matrix for this experiment. Since it is a small matrix we did not use a computation assignment algorithm to assign the vector pairs to the hardware modules. Since similarity matrix is symmetric with unit diagonal only the elements of the upper triangle need to be calculated. Therefore, with *n* number of vectors, the total number of similarity computations is *n**(*n*-1)/2; thus for an 8-vector similarity matrix only 28 vector pairs (of size eight each) need to be calculated. We separated 28 vector pairs into four, and specifically assigned them into four modules, so each module is executing 7 vector pairs in parallel. The results are written into the on-chip RAM. Table 7 shows the performance gain of non-parallel hardware versus hardware with four parallel modules.

Similarity matrix	Optimization level	Speedup (non-parallel)	Speedup (parallel)
Cosine Similarity	None	330	784
	Level II	119	283
Extended Jaccard	None	329	783
	Level II	119	282
Asymmetric Measure	None	211	503
	Level II	72	170

Table 7 Performance Comparison: Non-Parallel vs. Parallel Hardware

As shown in Table 7, the speedups in generating the similarity matrix using Cosine Similarity, Extended Jaccard, and Asymmetric Measure with no optimization are 784, 783, and 503 respectively for parallel hardware, whereas speedups are 330, 329, and 211 for non-parallel hardware. From these results, it is evident that the performance gain increases more than twofold when four hardware modules are used in parallel for all three similarity measures in generating the similarity matrix. Hence, it would be worthwhile to investigate parallel hardware when the tradeoffs between resources and performance gain are considered.

3.3.4. Similarity Matrix Computation on UltraSparc IIe

Additional software experiments were performed on a 502MHz UltraSparc IIe processor to validate our software designs as well as to compare performance using a different platform. CPU timing measurements were obtained for similarity matrix computation

among a varying number of vectors ranging from 2 to 8192 in increments of powers of 2. These results as well as the hardware execution times from 3.3.3.2 are used to carry out performance comparisons.

The speedups for similarity matrix computations with Extended Jaccard, Cosine Similarity, and Asymmetric Measure increase from 8 to 82, 8 to 81, and 5 to 44, respectively, for numbers of vectors ranging from 2 to 8192. The trends observed for performance gain are similar to and consistent with those found in Figure 13(b).

3.3.4.1. Performance Comparison: Hardware vs. Software on Different Platforms

From the above results and analysis, the highest rate of increase is observed for hardware speedup over software on the MicroBlaze with no optimization, whereas the lowest rate of increase is observed for hardware speedup over the UltraSparc IIe. Hence, although the hardware improves the overall performance, the incremental rate of improvement with the UltraSparc IIe is much less than that with the MicroBlaze. This result raises the issue of designing computation modules with a low-frequency general-purpose soft core on FPGAs, which involves tradeoffs between flexibility and performance.

With this investigation, we came up with several conclusions. First, from Table 4 and Figure 13(b), it is evident that the performance gains of the similarity matrix functions with higher abstraction levels are much better than the performance gains of the lower-level similarity functions. Second, the software execution overhead increases with the size or the number of vectors, whereas hardware overhead remains the same, since the setup time is the only overhead and is constant. Third, our hardware designs for similarity functions take advantage of the inherent parallelism, which is evident from Figure 4, Figure 8, and Figure 9 for Cosine similarity, Extended Jaccard, and Asymmetric Measure respectively. Fourth, from Table 7, parallel execution of the four hardware modules increases performance significantly. Hence, parallel hardware designs reduce execution time and increase performance gain and are worth investigating further.

3.4. Parallel Hardware Approach

In this section, we extend our investigation to more complex hardware architectures [126] and introduce an FPGA-based processor array for parallel computation of a similarity

matrix. An algorithm is developed to assign computation efficiently to the array of processing elements (PEs). Same design approach and development platform as in 3.3.1 are used. Theoretical performance metrics are derived and compared to the experimental results.

3.4.1. FPGA-Based Processor Array for Parallel Computation of Similarity Matrix

As shown in Figure 14, the processor array architecture consists of an array of *P* processing elements (PEs). The objective is to calculate the elements of the similarity matrix using parallel processing technique; hence, there is no communication and computation dependency among the PEs. Synchronized by a system clock, each PE performs the same computation (i.e., a similarity measure between two feature vectors) simultaneously. Due to the large volume of data to be processed, the PEs do not store the input vectors locally. Rather, a control unit is used to regulate and supply the appropriate vectors to the PEs. Consequently, an algorithm to assign computation at the level of a similarity measure between two vectors to each of the PEs is required.

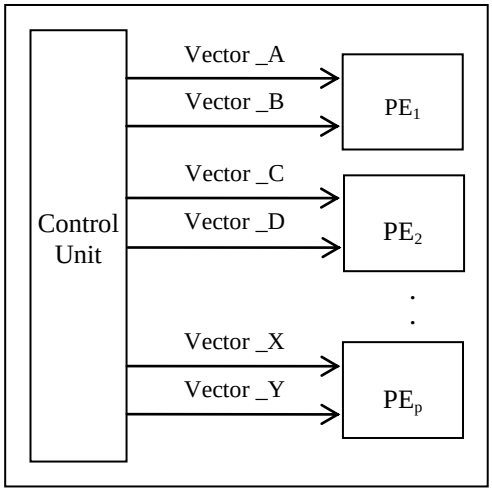


Figure 14 The Processor Array

Since the similarity matrix is symmetric with unit diagonal, only the elements of the upper triangle need to be calculated. The computation of a row is assigned to a PE. This has the advantage of minimizing circuit switching cost as one of the two input vectors to a PE only has to be fed once during the computation of one row. Intuitively, the *P* PEs are

assigned to compute the elements of *P* rows of the upper triangle, respectively and in parallel. A PE that finishes the computation of its assigned row will proceed to process the next unassigned row. This iterative process continues until all the elements of the upper triangle are computed.

3.4.2. Computation Assignment Algorithm

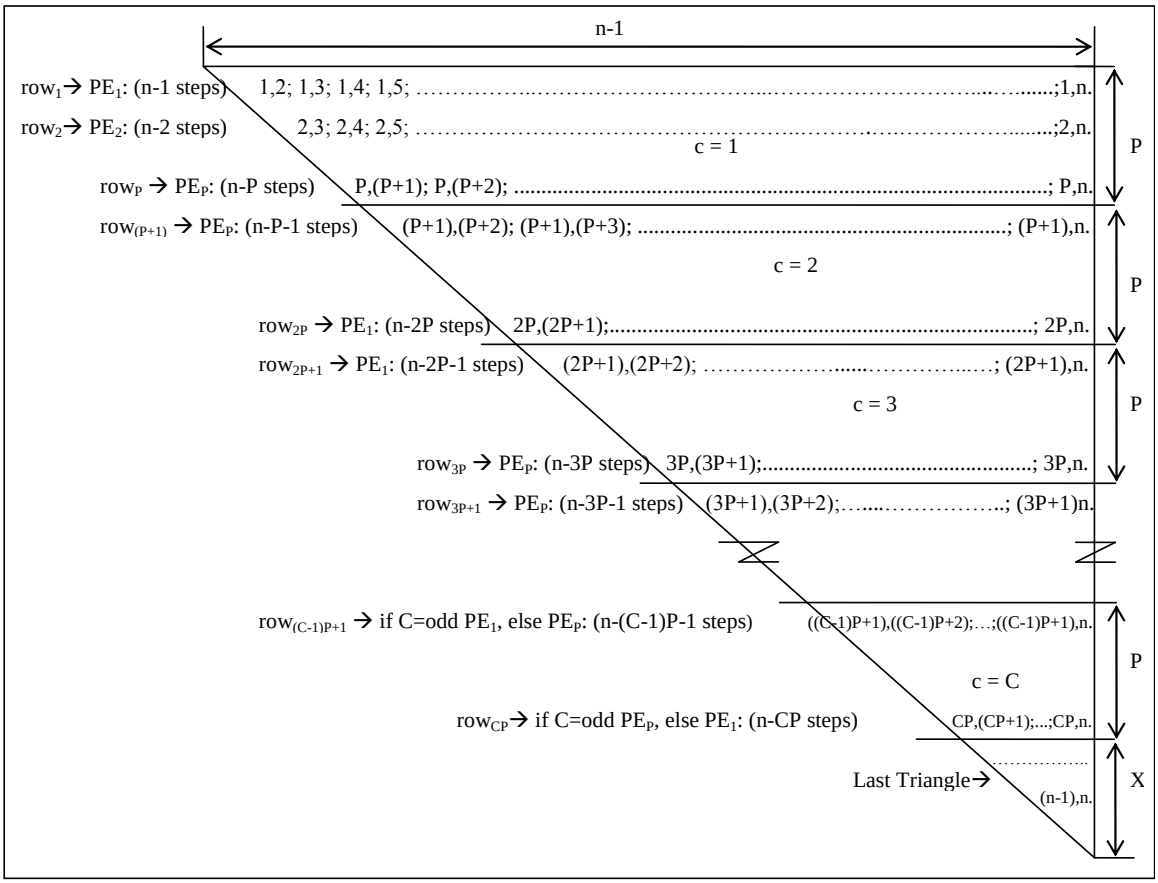


Figure 15 Assigning Similarity Matrix Computation to PEs

Let *n* (>> *P*) be the number of vectors. There are (*n*-1) rows and (*n*-1) columns in the upper triangle of the similarity matrix as shown in Figure 15. The first row consists of elements corresponding to vector pairs 1,2;1,3;…;1,(*n*-1);1,*n* and the last row consists of only one element, (*n*-1),*n*. (Thus, the terms *element* and *vector pair* are used interchangeable in this section.) Row computation assignments to the PEs are shown by the '→' notation, together with the number of steps (i.e., the number of similarity measure calculations) required to process the corresponding row.

Initially, row_1 to row_P are assigned to PE_1 to PE_P, respectively. The PEs thus start processing the first vector pair of their assigned rows in parallel, and their subsequent vector pairs in lock steps. Due to the triangular formation of the initial assigned rows, PE_1 needs to process (n-1) vector pairs in row_1 while PE_P processes (n-P) vector pairs in row_P. PE_P is the first PE to complete processing of its assigned row, and is then assigned row_{P+1}. PE_{P-1} is the next PE to complete its computation and is then assigned row_{P+2}. This pattern of assignment continues until PE_1 finishes processing row_1 and starts to process row_{2P}. At this point in time, all PEs are processing the vector pairs belonging to the same column in the matrix. Therefore, all the PEs would finish processing their assigned rows at the same time. The next computation assignment is similar to the initial assignment with row_{2P+1} to row_{3P} assigned to PE_1 to PE_P, respectively.

The above patterns of computation assignment would repeat until all the rows in the matrix are assigned and computed. In order to gain some insight on the details of computation assignment and computation complexity, the upper triangle of the similarity matrix can be viewed as consisting of a number of strips where the first strip is from row_1 to row_P, the second strip is from row_{P+1} to row_{2P}, etc. Each strip has P rows processed by the P PEs in parallel.

For n number of vectors, the total number of strips can be defined as

$$C = \left\lfloor \frac{n-1}{P} \right\rfloor \tag{7}$$

with an incomplete strip (less than P number of rows) labeled as the *Last Triangle* as shown in Figure 15. The number of strips varies from c = 1,2,…,(C-1),C. Let X be the remainder of (n-1) divided by P. X is the number of vector pairs in the first row of the last triangle, as well as its number of rows and columns due to symmetry. Since the last triangle is an incomplete strip, X ranges from 1 to (P-1).

3.4.2.1. Computation Complexity

In the processor array, several PEs are computing the same function in parallel. The time it takes for one similarity measure calculation is defined as a *similarity measure cycle*. Due to the parallel processing nature, y PEs are processing y vector pairs in y number of rows

simultaneously in one similarity measure cycle. In this section, the computation complexity to process *n* vectors with *P* PEs is examined.

3.4.2.1.1. Even and Odd Numbered Strips

As can be seen in Figure 15, PE_1 is the last PE to finish processing its assigned row_1 in the first strip c=1. There are (n-1) vector pairs in row_1. Therefore, it takes (n-1) similarity measure cycles to completely process all the vector pairs in the first strip. Note that during this (n-1) cycles, other PEs may have started processing rows in the second strip.

When PE_1 starts processing the first vector pair of the last row (row_{2P}) of the second strip, all the other PEs are processing the vector pairs of the same column as PE_1 in the second strip. Therefore, all the PEs will finish processing their assigned rows in the second strip simultaneously. PE_1 has to process (n-2P) vector pairs in row_{2P} in the second strip. Hence, it takes (n-2P) similarity measure cycles to finish processing the remaining vector pairs in the second strip after PE_1 starts its computation.

All the even numbered strips have the same computation complexity characteristic as the second strip. The computation time for even numbered strips in similarity measure cycles are:

c = 2: (n-2P)

c = 4: (n-4P)

c = 6: (n-6P)

c is even: (n-cP)

Similarly, all the odd numbered strips have the same computation characteristic as the first strip. For example, PE_1 has to process (n-2P-1) vector pairs in row_{2P+1} of the third strip. Therefore, it takes (n-2P-1) similarity measure cycles to completely process all the vector pairs in that strip. In summary, the computation time for the odd numbered strips in similarity cycles are:

c = 1: (n-1)

c = 3: (n-2P-1)

c = 5: (n-4P-1)

c is odd: (n-(c-1)P-1)

3.4.2.1.2. The Last Triangle

The number of similarity measure cycles to process all the vector pairs in the last triangle depends on whether the last strip c=C is odd or even, and the number of the vector pairs X on the first row of the last triangle.

If C is an odd number, then the last strip has the same characteristics as the first strip. The first PE to complete its assigned row is PE_P while PE_1 still has to process (P-1) vector pairs. Since $X \leq (P\text{-}1)$, in the worst case, PE_P has to process (P-1) vector pairs to complete the first row in the last strip. Therefore, the computation time for the last triangle in this case is considered and covered already in the computation time for strip C.

If C is an even number, then the last strip is similar to the second strip with all the PEs completing their assigned tasks at the same time. In the worst case, it takes PE_1 a total of (P-1) similarity measure cycles to process the (P-1) vector pairs in the first row of the last triangle. If X = 1, then only one similarity measure cycle is required to process the only vector pair in the last triangle.

Thus, the number of similarity measure cycles to process all the vector pairs in the last triangle is:

$$r((n\text{-}1) \bmod P), \text{ where } r = (C+1) \bmod 2$$

3.4.2.1.3. Theoretical Prediction

When the number of vectors to be examined is less than or equal to the number of deployed PEs, then the number of strips C is equal to 0. In this case, PE_1 needs (n-1) similarity measure cycles to process the first row. That is, if $n \leq P$, the number of similarity measure cycles to process the similarity matrix is (n-1).

When the number of vectors n is larger than the number of PEs, which usually is the case, the total computation time to generate the similarity matrix can be predicted using the results from the above analysis. The total computation time is simply the sum of similarity measure cycles for all the odd and even numbered strips, and the last triangle:

$$\left(\sum_{c=1}^{C} m\left(n - cP\right) + q\left(n - \left(c-1\right)P - 1\right) \right) + r((n-1) \bmod P) \qquad (8)$$

where, $m = (c+1) \bmod 2$, $q = c \bmod 2$, and $r = (C+1) \bmod 2$

The above equation is obtained from the algorithmic perspective. In the next section, a performance prediction equation from the hardware perspective is derived. Both of these equations are used to validate the correctness of the experimental results of the hardware designs.

3.4.3. Experimental Results and Analysis

Various hardware processor array configurations were implemented for the following three similarity measures: Cosine Similarity, Extended Jaccard, and Asymmetric Measure; using the same hardware implementation of equations (3), (4), and (5) as in 3.2.4.1 and 3.3.2.1.

The number of PEs was varied to illustrate the flexibility and reconfigurability of the computation assignment algorithm. Due to space limitation of the FPGA used, the maximum numbers of PEs deployable for Extended Jaccard, Cosine Similarity, and Asymmetric Measure were 8, 9, and 9, respectively.

Software modules were executed on MicroBlaze and UltraSparc-IIe processors for performance comparison. For both hardware and software similarity matrix implementations, varying numbers of 8-element vectors, in increments of power of two, ranging from 2 to 8192 were processed in our experiments.

3.4.3.1. Theoretical versus Experimental Results

For the hardware designs, the similarity matrix computations are assigned to the available PEs according to the algorithm described in 3.4.2. After an initial setup time of 66 clock cycles (at 80 MHz), each PE produces a similarity result in every 100 ns. Thus, one similarity measure cycle is defined as the 100 ns needed to compute the similarity between vector pairs in parallel.

The experimental execution time for different hardware designs are measured and compared with the similarity measure cycles obtained from equation (8), to validate the experimental results.

Similar to equation (6) in 3.3.3.2, in this case also, we derived a theoretical equation to predict execution time from a hardware perspective:

$$CPUClockCycles = N + \lfloor (R * Q * S) / P \rfloor \qquad (9)$$

where, N is the number of clock cycles for the initial setup time; R is the number of elements per vector; Q is the number of vector pairs; S is the number of clock cycles to produce one similarity measure result after the initial setup time; and P is the number of PEs.

The hardware execution time for similarity matrix generation for any number of vectors with any number of elements can be predicted accurately using equation (9), which is used to validate the hardware experimental results, by multiplying the theoretical result with the clock period of the design.

3.4.3.2. Analysis of Processor Array Results

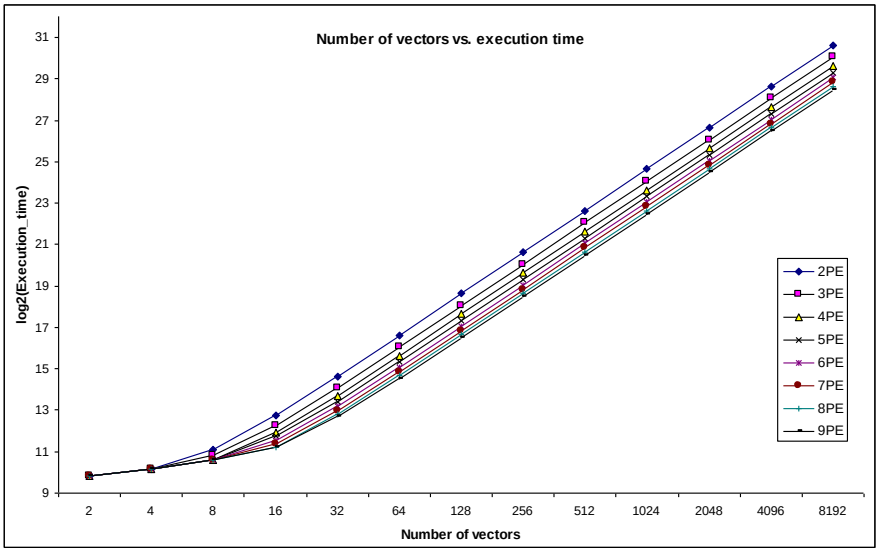


Figure 16 No. Vectors vs. Hardware Execution Time

Execution times for the processor array to generate the Cosine Similarity Matrix for varying number of vectors and varying number of PEs are shown in Figure 16. The top line of the graph indicates the execution times for 2 PEs, whereas the bottom line indicates the execution times for 9 PEs. Graphs for similarity matrix generation using Extended Jaccard and Asymmetric Measure show similar behaviour.

From Figure 16, it can be observed that the percentage of increase in execution times for similarity matrix is linear among vectors ranging from 16 to 8192 for varying number of PEs from 2 to 9. In contrast, as can be seen in the lower left of the graph, the execution

times for vectors 2, 4, and 8 are not consistent with the others. This discrepancy is mainly due to a large percentage of the total execution time is being spent on the overhead (i.e., the initial setup time) for small number of vectors (as discussed in Table 5 in 3.3.3.2).

Furthermore, the execution times for similarity matrix among 2 vectors are the same regardless of the number of PEs. The execution times for similarity matrix among 4 vectors are the same for 2 to 9 PEs. The execution times for similarity matrix among 8 vectors are the same for 4 to 9 PEs. The reasons for these phenomena are explained in the following.

For 2 vectors: Only one PE is needed to process two vectors, therefore, regardless of the number of available PEs, it takes (n-1) = (2-1) = 1 similarity measure cycle as discussed in 3.4.2.1.3.

For 4 vectors: According to the computation assignment algorithm and equation (7), the number of strip C is 1 for 2 and 3 PEs, with the number of similarity cycles being 3 according to equation (8). With 4 or more PEs, it takes (n-1) = (4-1) = 3 similarity measures cycles as described in 3.4.2.1.3.

For 8 vectors: For 4 to 7 PEs, equation (7) gives C =1 and therefore it takes 7 similarity measure cycles to process the 8-vector similarity matrix according to equation (8). With the number of vectors less than or equal to the number of PEs, 8 and 9 PEs would also take 7 cycles as discussed in 3.4.2.1.3.

3.4.3.2.1. Varying Number of PEs with Constant Number of Vectors

Number of PEs	2PE	3PE	4PE	5PE	6PE	7PE	8PE	9PE
Percent of work	50%	33%	25%	20%	16.7%	14%	12.5%	11%

Table 8 No. PEs vs. Percentage of Work for Constant No. Vectors

From Table 8, it can be observed that the percentage of work done, i.e., the amount of work load carried out relative to the total work load, by a single PE in each hardware configuration decreases, but not linearly. This explains why there is a bigger gap between the results of 2 PEs and 3 PEs in Figure 16, why the lines get denser with increasing number of PEs, and why there is a very small gap between the results of 8 PEs and 9 PEs. Thus, for a constant number of vectors, the performance improves but the incremental rate of improvement decreases with increasing number of PEs.

3.4.3.2.2. Varying Number of Vectors with Constant Number of PEs

As shown in Table 9, the percentage of work done by each PE in each of the hardware configurations does not change with the increasing number of vectors, thus the incremental rate of improvement is zero. For a constant number of PEs, the execution time increases with increasing number of vectors but the rate of increase in execution time of 2 PEs is much higher than that for 8 PEs. As can be seen in Figure 16, this leads to divergent slopes for various number of PEs with respect to the number of vectors.

Number of vectors		100	200	400	800
Percent of work	2PE	50%	50%	50%	50%
	4PE	25%	25%	25%	25%
	8PE	12.5%	12.5%	12.5%	12.5%

Table 9 No. PEs vs. Percentage of Work for Varying No. Vectors

3.4.3.3. Performance Comparison: Hardware, MicroBlaze, & UltraSparc-IIe

Software versions to generate the similarity matrix for all three similarity functions are executed on the 80-MHz MicroBlaze on the same FPGA as the processor array. Experimental speedup results for the Cosine Similarity Matrix having vectors ranging from 2 to 8192, in increments of power of two, with level II optimization are shown in Figure 17. Similar results are obtained for no optimization. The top line of the graphs indicates the performance gain for 9 PEs while the bottom line indicates the gain for 2 PEs.

As observed from the graphs, for small numbers of vectors, the performance gains are the same. This is due to the fact that the execution time remains constant when n is less than or equal to P as discussed in 3.4.3.2. Furthermore, the rate of increase of speedups for vectors 8 to 64 has a different trend than the speedups for vectors ranging from 128 to 8192. This is due to the diminishing overhead, and the effect of the number of PEs and the number of vectors on the incremental rate of performance improvement as discussed in 3.4.3.2.

The speedup of generating the Cosine Similarity matrix using 502MHz UltraSparc-IIe exhibits similar patterns as in the MicroBlaze case (similar to Figure 17). With MicroBlaze, speedup of 4887 and 1687 for no optimization and level II optimization are attained using a processor array of 9 PEs for computing the similarity among 8192

vectors, while speedup of 732 is achieved with UltraSparc. The lower performance gain in the latter case can be attributed to the clock frequency of UltraSparc being more than 5 times higher than that of the processor array, and the fact that no optimization has been performed in the PE design.

Hardware and software comparison for similarity matrix generation using Extended Jaccard and Asymmetric Measure for both MicroBlaze and UltraSparc also show similar behaviour as that of the Cosine Similarity Matrix. It is expected that further performance gain can be achieved with hardware optimizations.

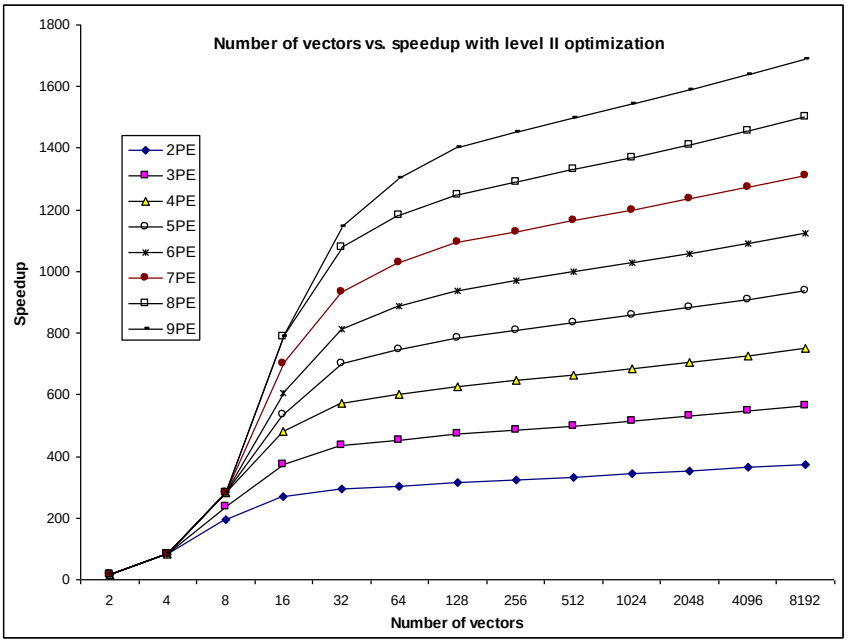


Figure 17 Cosine Similarity Speedup over Software on MicroBlaze (Level II Optimization)

3.5. Chapter Conclusion and Discussion

From the experiments presented in this chapter, it can be concluded that hardware support for data mining operations is feasible and a worthwhile endeavour, when proper design strategies are used. For software implementations, the designers should be wary of certain programming decisions such as software constructs employed, optimization during software compilation, etc., since these decisions have a significant impact on software performance.

We introduced hardware support for three similarity measures: Cosine Similarity, Extended Jaccard, and Asymmetric Measure; and their corresponding similarity matrices. We also introduced an FPGA-based processor array for parallel computation of a similarity matrix, and an algorithm to assign computation efficiently to the array of PEs. From the results and analysis, it is observed:

- Performance gain of hardware over software increases with the complexity or abstraction level of operations.
- Software overhead on MicroBlaze increases with the size/number of vectors (unlike hardware overhead).
- Performance gain is further enhanced with our hardware designs
    - By exploiting the functional parallelism in operations (three similarity measures).
    - By exploiting the data parallelism in operations (similarity matrix computation) – using many PEs executing different data in parallel.

Data mining is a high-level concept, involving complex and compute-intensive operations at higher-levels of abstraction. Many of these operations are usually amenable to pipelining and exhibit significant amount of functional parallelism, which can be exploited to a great extent in hardware. In addition, data mining commonly involves processing large amount of data. Usually these data do not have any data dependencies, which allow exploiting data parallelism. For instance, performance gain, using our processor array with 9 PEs, is 9 times (4887/543) higher than using a single PE, leading to 100 percent parallelism. These investigations demonstrate that hardware is indeed a good approach for data mining operations.

There are additional advantages associated with our processor array and assignment algorithm. Our algorithm is independent of the number of PEs used or the number of vectors being processed. Our processor array is easily scalable in size and performance, since the complexity of the PEs is not an issue either in the algorithm or in the architecture. The PEs are reconfigurable to perform different functionalities regardless of their complexity, thus the processor array can have PEs with different operations. Hence, our processor array can be used not only in data mining but also in other applications which demand parallelism to achieve better performance.

The work in this chapter has been published in the following proceedings and journal:

- IEEE International Symposium on Data Mining and Information Retrieval, (DMIR'07) [104];
- IEEE Pacific Rim International Conference on Communication, Computers and Signal Processing (PacRim'07) [125];
- IEEE Canadian Journal for Electrical and Computer Engineering (CJECE), Volume 33, Winter 2008 [124];
- IEEE International Conference on Advanced Information Networking and Application (AINA'08) [126];
- Springer LNCS Transactions on Computational Collective Intelligence, 2012, [105].

In the next chapter, we investigate the feasibility of using reconfigurable computing for data mining operations and introduce reconfigurable hardware solutions for two key data mining operations: similarity measure and principal component analysis (PCA). To facilitate this investigation, we have analyzed the advantages and disadvantages of using reconfigurable hardware over non-reconfigurable hardware, as presented below.

3.5.1. Reconfigurable versus Non-Reconfigurable Hardware

Application-specific hardware can be implemented as non-reconfigurable computing systems (full custom-designed hardware) such as ASICs and reconfigurable computing systems such as FPGAs.

Flexibility: Non-reconfigurable computing systems, such as ASICs, have fixed functionality, hindering any optimizations and upgrades in applications, after fabrication [23]. With reconfigurable systems, on-chip hardware circuitry can be changed (re/configured) to perform a variety of applications post fabrication, and also allows post-design optimizations and upgrades of applications.

Area: Non-reconfigurable computing systems are often designed and fabricated for specific applications, typically implementing one application on a single chip, thus requiring several chips to execute several applications. This becomes an issue with portable and embedded devices because of their limited hardware foot-print. Post-fabrication re/programmability of reconfigurable systems allows numerous applications to be performed on a single chip, requiring less hardware space on a chip than non-

reconfigurable systems. However, in comparison to full custom designed hardware, per circuit, reconfigurable systems require more silicon area to implement the same functionality due to their reprogrammability [77],[100].

Power: Reconfigurable computing systems typically consume more power than non-reconfigurable systems such as ASICs, since re/programmability requires more transistors than a customized integrated circuit (IC) [77],[100]. A study [100] done in 2007 shows that FPGA consumed 7.1 to 14 times more dynamic power than an equivalent ASIC on average.

Cost: In general, reconfigurable computing systems such as FPGAs have higher per-unit costs but lower non-recurring costs than non-reconfigurable systems such as ASICs [67],[77]. As a result, using FPGAs as a substitute for ASICs is only cost-effective in low volumes. For high volumes, FPGA device cost could eventually exceed ASIC device cost, because FPGA requires much more silicon area to implement equivalent functionality due to its reprogrammability [77]. However, if an application requires any modifications post fabrication, the cost of replacing the non-reconfigurable ASICs is high. Conversely, reconfigurable systems are upgradeable post fabrication, thus prolonging the useful life of the system, while minimizing the long term cost.

Design time: Designing and implementing reconfigurable computing systems can take longer time than that of non-reconfigurable systems. Apart from having an expertise on full custom-based designs, the reconfigurable hardware designers have to be knowledgeable on different reconfiguration methods, familiarized with specialized tools for reconfiguration, etc. Therefore, the total design time for a reconfigurable design is often higher than that of a non-reconfigurable (full custom-based) design.

Performance: Reconfigurable computing systems are typically slower than full custom-based designs. In addition, the reconfiguration time overhead has a considerable impact on the overall speed performance of the application [129]. However, in some cases, there is a possibility to overlap the reconfiguration with computation, reducing the effect of reconfiguration time overhead on speed performance.

In summary, reconfigurable computing systems provide a flexible and area efficient computing platform for portable and embedded devices. However, non-reconfigurable computing systems (full custom designed) consume less power, have slightly higher

performance, and have lower design time compared to reconfigurable systems. Also, reconfigurable systems have higher per-unit costs and lower non-recurring costs than non-reconfigurable computing systems. One of the most challenging issues of portable and embedded devices is their limited hardware foot-print; hence reconfigurable computing approach is worth pursuing although there are tradeoffs associated, as discussed. The reconfigurable hardware approach is investigated extensively in the next chapter.

Chapter 4

4. Reconfigurable Hardware for Data Mining Operations

In this chapter, our objective is to investigate the feasibility of using reconfigurable hardware for data mining operations in portable, handheld, and embedded devices.

As concluded in the previous chapter, chip-level hardware is indeed a good approach for data mining operations. Our analysis on reconfigurable versus non-reconfigurable hardware illustrates that the former is promising to provide hardware support for specific applications on portable and embedded devices. Especially, with the limited hardware foot-print on portable and embedded devices, this approach is worth pursuing. Although there is a penalty of the time spent on reconfiguring the hardware, a single chip can be utilized to perform numerous applications. Reconfigurable hardware could provide a flexible and efficient platform for satisfying the area, performance, cost, and power requirements of many portable and embedded devices.

We present and discuss state-of-the-art reconfigurable computing systems in 4.1. We introduce reconfigurable hardware solutions for similarity matrix computations with three similarity measures using a multiplexer-based approach, and for Principal Component Analysis (PCA) using partial reconfiguration method, which are presented in 4.2 and 4.3 respectively. These two operations are often used for applications that are found in portable and embedded devices such as handwritten analysis, finger-print verification, etc. Further investigation and analysis on dynamic partial reconfiguration are presented in 4.4.

4.1. State-of-the-art Reconfigurable Computing Systems

Reconfigurable computing is a technology that “promises an intermediate trade-off between flexibility and performance” [23]. It provides significant flexibility advantage over conventional ASICs and significant performance advantage over conventional microprocessors. In order to gain insight into this technology, we studied different facets of reconfigurable computing systems.

4.1.1. Standard Interface – RPU and Host System

Reconfigurable computing systems are usually formed with a combination of a reconfigurable processing unit (reconfigurable fabric), a general-purpose processor, memory and possibly other structures [77],[156]. As illustrated in Figure 18, there are multiple ways in which we can integrate the reconfigurable processing unit (RPU) to the host processor [23],[39],[156].

Firstly, the RPU can be used exclusively to provide a "reconfigurable functional unit" within the host processor, thus creating a very tightly coupled structure [39],[78]. In this case, the RPU behaves as a functional unit of the datapath of the processor, which allows the creation of custom instruction that can be modified overtime [39],[156].

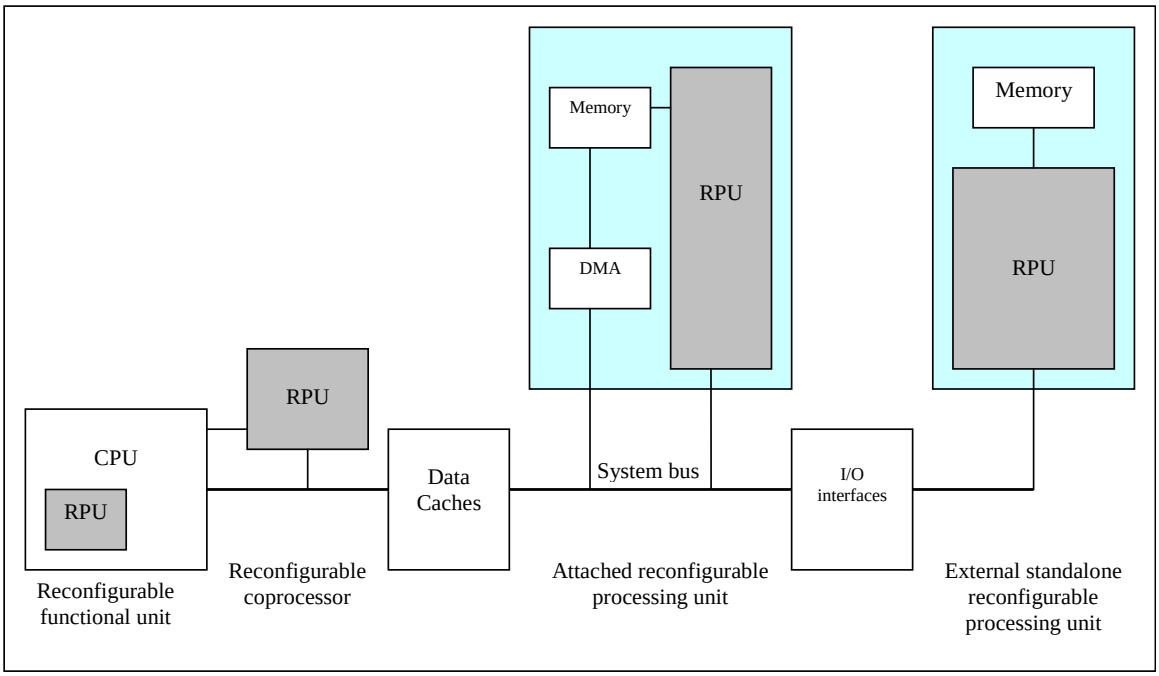


Figure 18 Standard Interface Between RPU and Host System [14],[39]

Secondly, the RPU can be used as a "reconfigurable coprocessor", creating an intermediate coupling structure [39],[117]. Unlike the previous structure, the coprocessor performs the actual computations independently of the host processor, and sends the results after completion, whereas the processor generally initializes the RPU and provides information about the location of data [39]. This type of coupling structure enables both the reconfigurable coprocessor and the host processor to execute simultaneously. In addition, reconfigurable coprocessor can operate for a longer period of time without processor intervention.

Thirdly, the RPU can be used as an “attached reconfigurable processing unit”, creating a loosely coupled structure than the above two [10],[39]. In this case, the RPU performs as an additional processor, creating a multiprocessor system [39]. A Direct Memory Access (DMA) unit can be used to transfer data from the host memory to the local memory of the reconfigurable computing system [14][39]. Since this type of coupling incurs high communication overhead than the previous structures, the communication between the processor and the reconfigurable unit should be kept minimum. Also, by shifting significant portions of a computation to the attached RPU, we can achieve more computation independence [39].

Finally, the RPU can be used as an “external stand-alone processing unit”, which is the most loosely coupled form [39],[132]. With this type of coupling the reconfigurable processing unit rarely communicates with the host processor, since the communication between them is relatively slow compared to any of the previous structures [156]. Hence, this structure is suitable for computations that are autonomous and computations that are processed for a long period of time with the least communication with the host processor.

Each of these structures has their own advantages and disadvantages [39]. For instance, tightly coupled structures have lower communication overhead and have fewer independent computations. Conversely, loosely coupled structures have higher communication overhead and have more independent computations. Hence, applications that require frequent communication with the processor can benefit from a tightly coupled structure, and applications with autonomous computations, which require significant processing time can benefit from loosely coupled structure.

Apart from the above coupling structures, current RPUs can have processors embedded into the reconfigurable units [156]. These processors can either be hard core processors such as PowerPC [188], or soft core processors such as MicroBlaze [179] and NIOS II [8], which are built from the logic within the reconfigurable unit. Especially, in a loosely coupled structure, where the RPU functions as an external standalone processing unit, having the option of using an embedded processor within the reconfigurable unit might be beneficial. For instance, some of the control functions executed on the external host processor can instead be executed on the internal embedded processor. This allows quick decision making and keeps external interactions minimal.

4.1.2. Analysis – FPGA-Based vs. Non FPGA-Based Reconfigurable Hardware

Some reconfigurable processing units are composed of standard FPGAs [4],[41],[78], [131],[169], while others are composed of custom-designed configurable hardware [18] [27],[44],[70],[94],[95],[101],[114],[116],[118],[144].

A vast majority of reconfigurable computing systems use FPGAs, which consist of a matrix of logic blocks and an interconnection network that are both programmable [23]. The logic blocks are based on look-up-tables (LUTs), each having few inputs (three to six) and a single output. Each LUT is considered as a very fine-grained computational unit. Typically, LUTs are combined into clusters, which are called configurable logic blocks (CLBs) or logic elements (LE) [156]. The logic blocks as well as the interconnection network are both very flexible and can be used to implement any digital circuit.

Some reconfigurable systems are based on custom-reconfigurable silicon devices [18] [27],[44],[70],[94],[95],[101],[114],[116],[118],[144]. The reconfigurable processing unit of these systems comprise of an array of processing elements (PEs) called reconfigurable cell (RC) array and an interconnection network.

Typically, RCs have identical functionalities. A RC usually consists of an ALU-like (Arithmetic Logic Unit) structure, multiplexers, registers, shifters, register files, etc. Some RCs might have a RAM, multiplier, and additional control logic. The ALUs can perform standard arithmetic and logic operations and in some cases multiplication and division operations. The functional model of the RC is quite similar to the data-path of a conventional microprocessor [144].

The configuration is usually done using configuration instructions, which are stored in the configuration memory. These instructions are loaded into the instruction register of each RC, and are used to configure the RCs and program the interconnection network. A RC can be configured to implement one of several ALU operations. According to the configuration instruction, the required ALU operation is selected and the RCs are connected to form the necessary circuit. The instructions are executed through multiplexers, tri-state-drivers and various other control logics, thus controlling the routing and the internal structure of the design. Since the configuration instructions are stored in the memory, there is an overhead of fetching these instructions.

In custom-designed reconfigurable systems, having "ALU-like structures resembles multi-processors or very long instruction word (VLIW) processors than they do FPGAs" [77]. In addition, the way routing is performed within these systems further differentiate them from FPGA-based reconfigurable systems [77]. For example, PADDI [33] has a distinct VLIW nature, whereas MorphoSys [144] exhibits single instruction multiple data (SIMD) functionality.

Flexibility: With most custom-designed reconfigurable systems, FPGA's fine-grained look-up-tables are replaced with coarse-grained reconfigurable cells, and they commonly use coarse-grained routing architectures, making them less flexible than FPGA-based reconfigurable systems. In reconfigurable hardware, granularity refers to the width of the data path, i.e., fine-grained bit-level manipulation and coarse-grained word-level manipulation [77],[80]. Most of the custom-designed reconfigurable systems are designed to operate on fixed data widths such as 8, 16 or 32-bit data. If there is a need to operate on data elements with varying number of bits (e.g., 12-bit), we are forced to use a RC with 16-bit functional unit and routing architecture. FPGAs are often configured and reconfigured, not only to take advantage of small bit widths, but also to take advantage of large data widths [39]. Computation structures of arbitrary bit widths can be created with fine-grained architectures, whereas coarse-grained architectures have fixed bit widths [39],[156]. In addition, in fine-grained routing architecture each wire can be switched independently, whereas in the coarse-grained routing architecture the entire bus is switched as a unit [156]. Clearly, such coarse-grained RCs and routing architectures are far less flexible than the fine-grained ones.

For different custom-designed reconfigurable systems, the reconfigurable cell array and the interconnections networks of the reconfigurable processing units are organized differently, usually according to the requirements of the applications. For instance, the reconfigurable cell array is organized in: SIMD fashion in MorphoSys [144] (targets multimedia applications – graphics and image processing, data encryption, etc.); pipelined fashion in PipeRench [70] (targets stream-based media applications – Automatic Target Recognition (ATR), Discrete Cosine Transform (DCT), etc.); linear array fashion in RaPiD [44] (targets DSP applications – Finite Impulse Response (FIR), DCT, etc.); to name a few. This is mainly because the structure of the custom-designed reconfigurable

systems is determined during the manufacturing process, and cannot be altered, similar to ASIC designs. Additionally, most of these systems are designed and fabricated for specific application(s). In contrast, FPGA-based reconfigurable hardware can be configured and reconfigured to execute a number of different applications even after the manufacturing process. Because the hardware on chip can be changed post fabrication, we can configure the FPGA to have different organizations, such as SIMD, MIMD (Multiple Instruction Multiple Data), pipelined, etc. For instance, if there are multiple independent computations that can be executed in parallel, then we can reconfigure the FPGA to perform these computations in parallel using multiple processing elements (PEs); and if there is one computation (or several dependent computations) that can be executed in a pipelined fashion, then we can reconfigure the chip to do so. This provides more evidence that FPGA-based systems are more flexible than custom-designed reconfigurable systems.

Area: In custom-designed reconfigurable systems, although only one operation per RC is selected and used during a computation, the rest of the ALU operations reside on chip for each RC, occupying valuable area on the reconfigurable device. Hence, the overall chip area for the reconfigurable processing unit with the custom-designed reconfigurable systems might be higher than that of one using FPGA-based reconfigurable systems, making them less area efficient.

Since the custom-designed reconfigurable systems are designed and fabricated for specific applications, these systems typically require an entire application to fit into the chip, making it difficult to use these systems for large applications. However, FPGAs can be used to execute a large application that does not fit into the chip by partitioning the whole circuit into small sub-circuits that will fit into the chip and executing them at different times via dynamic reconfiguration.

In addition, post-fabrication re/programmability of FPGA-based reconfigurable hardware gives its ability to reuse the same chip to process a variety of applications, by changing the on-chip hardware circuitry from one application to another as needed.

A reconfigurable system, however, custom-designed for a specific circuit, might take less area than that with FPGA-based reconfigurable system for that specific circuit [77],[100]. Some studies (done in 2005 [156] and 2007 [100]) showed that a circuit implemented on an FPGA-based reconfigurable systems occupies significantly more area

compared to the same circuit implemented on a custom-designed reconfigurable systems, because of FPGA's fine-grained architecture.

Performance: In FPGA-based reconfigurable systems, flexible routing at bit level comes with performance overhead, in comparison to the more rigid routing of custom-designed reconfigurable systems [23],[156]. Lower granularity of FPGA-based reconfigurable systems have a performance penalty due to larger delays when building computation modules of a larger size using small functional units [23],[100].

Power: Fine-grained architecture of FPGA-based reconfigurable systems often consumes more power than coarse-grained architecture of custom-designed reconfigurable systems [156]. It has been shown [67],[100] that LUT-based reconfigurable hardware such as FPGAs are less power efficient than RC-based reconfigurable hardware.

In summary, FPGA-based reconfigurable systems provide a relatively flexible computing platform to satisfy the stringent area requirements of portable and embedded devices. Because of its flexibility numerous applications can be designed and implemented on a single chip, regardless of them fitting into the chip. However, custom-designed reconfigurable systems are more efficient in terms of power and performance. In Appendix B we present a discussion on FPGA performance from various perspectives as well as the historical development of FPGA.

4.1.3. Programmable Logic Devices – CPLD versus FPGA

Reconfigurable hardware can be implemented using programmable logic devices (PLDs). There are two major types of PLDs [26],[34],[40]: CPLDs (Complex Programmable Logic Devices) and FPGAs. FPGAs usually have small programmable logic cells based on look-up-tables (LUTs) with few inputs and a single output, whereas CPLDs have much larger logic cells based on programmable logic arrays (PLA) with large number of inputs and outputs [26],[40]. Since FPGAs use small programmable cells, they can offer higher logic density than CPLDs [34],[40]. However, CPLDs are much faster than FPGAs and their timing delays are more predictable because PLAs are much larger than LUTs, so that CPLD implementation results in fewer levels of logic [26],[34]. Another difference between CPLDs and FPGAs is that FPGAs have advanced features like built-in hardwired processors (e.g., IBM PowerPC), substantial amounts of embedded memory, clock

management systems, and higher-level embedded functions (adders, multipliers). Some FPGAs allow partial reconfiguration, where some parts of the device can be reprogrammed (reconfigured) while the remaining parts are in operation. In conclusion, FPGA is a better medium for implementing the reconfigurable logic devices compared to CPLD.

4.1.4. Standard Reconfiguration Process in FPGAs

Reconfigurable hardware designs, such as FPGA-based designs, are typically written in a hardware description language (HDL) such as Verilog or VHDL [39],[77]. This abstract design has to undergo a series of steps to fit into FPGA's available logic [77]: The first step is logic synthesis, which converts high-level logic constructs and behavioural code into logic gates; the second step is technology mapping, which separates the gates into groupings that match the FPGA's logic resources (generates net list); the next two consecutive steps are placement and routing, where placement allocates the logic groupings to the specific logic blocks, and routing determines the interconnect resources that will carry the signals [77]. The final step is bitstream generation, which creates a "configuration bitstream" for programming the FPGA.

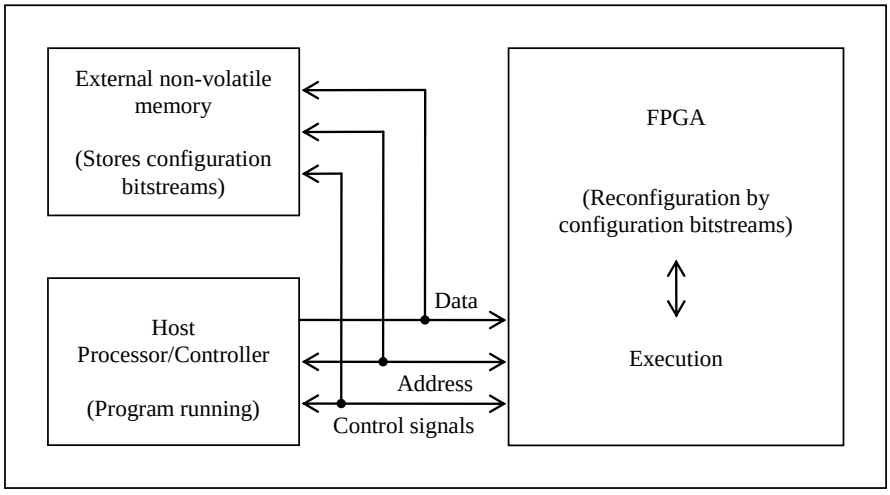


Figure 19 Standard Reconfiguration Process in FPGAs

Configuration bitstream is a binary file that sets all of the FPGA's programmable bit locations to configure the logic blocks and routing resources appropriately [39],[77]. Typically, the configuration bitstream is saved in an external non-volatile memory such as

EEPROM (Electrically Erasable Programmable Read Only Memory). As shown in Figure 19, by using either a host processor or a host controller, the configuration bitstream is downloaded to the programmable bit locations of the FPGA and used to program (configure) the FPGA to its appropriate hardware circuitry to perform a specific computation or set of computations [120]. The configuration bitstream must be downloaded every time the system is powered up, as well as anytime the user wants to change the circuitry during system operation [77].

Depending on the interface supported by the device, there are two possible ways of downloading the bitstream, either “bit-serial” or “bit-parallel” [23],[120],[171]. With the serial programming mode, the FPGA is configured by loading one bit per configuration clock cycle. With the parallel programming mode, the FPGA is configured by loading 8, 16, 32 bits per configuration clock cycle. It is 8-32 times faster than the serial mode and useful for applications where reprogramming is common and speed is important [77].

4.1.5. Static versus Dynamic Reconfiguration

We can distinguish reconfigurable hardware into two types: static and dynamic.

4.1.5.1. Static Reconfiguration

With static reconfiguration, the full configuration bitstream of an application/computation is downloaded (at system start-up) and the chip is reconfigured only once and never changed throughout the run-time-life of the application/computation. The same application or computation is running continuously, without changing (reconfiguring) the on-chip hardware. In order to execute a different application, a full configuration bitstream of that application has to be downloaded again and the entire chip has to be reconfigured. System has to be interrupted for every download and reconfiguration process.

Static reconfiguration is appropriate for applications that do not require any modifications during the run-time life of the application, and applications that are typically run for a long period of time (months or years).

4.1.5.2. Dynamic Reconfiguration

With dynamic reconfiguration, the full configuration bitstream of an application/computation is downloaded (at system start-up), and the on-chip hardware is

configured but is often changed during the run-time-life of the application/computation. This kind of reconfiguration allows changing either parts of the chip or the whole chip, (or either parts of an application or the whole application) as needed on-the-fly, to perform several different computations, for example, due to dynamic changes in the operating environment. New full or partial configuration bitstreams are downloaded and the chip is reconfigured as needed to process new applications, without human intervention and in certain circumstances without interrupting the system operations.

Dynamic reconfiguration is appropriate for applications that require modification during the run-time life of the application, and applications that are executed for short period of time (second/hours) and changed frequently.

In summary, dynamic reconfiguration has the ability to perform hardware optimization based upon present results or external stimuli determined at run-time. In addition, with dynamic reconfiguration, we can run a large application on a smaller chip by partitioning the application into sub-circuits and executing the sub-circuits on chip at different times.

In this chapter following terminologies are used:

- Reconfiguration time overhead for multiplexer-based approach is the time taken to change the hardware from one configuration to another.
- Reconfiguration time overhead for partial reconfiguration is the time taken to download and change the on-chip hardware circuitry from one configuration to another.

4.2. Reconfigurable Hardware Solution for Similarity Matrix Computation

There are several different reconfigurable computing approaches. Most of these reconfigurable approaches are typically based on single FPGA (e.g., OneChip [169], Chimaera [78], Chameleon [30])), multi-FPGAs (e.g., PAM [164], VCC [28], Splash [69]), processors with dynamic instruction set (such as DISC [168], PADDI [33], and RAW [13]), and custom-designed reconfigurable hardware [18],[27], [44],[70],[94],[95],[101],[114],[116],[118],[144]. A conventional method of reconfiguring a design is to use a multiplexer-based approach, which is somewhat similar to custom-designed reconfigurable hardware.

In this section, we introduce a reconfigurable hardware solution [127] using a multiplexer-based approach for similarity matrix computation. Our hardware design can be dynamically reconfigured to accommodate three different similarity measures. The architecture of the reconfigurable hardware design, and the experimental results and analysis on the design are discussed and presented.

4.2.1. Design Approach and Development Platform

Both software and reconfigurable hardware designs are implemented using a hierarchical platform-based design approach as in 3.3.1 (Figure 7). Unlike previous experiments, for these experiments, we incorporated the square-root function for the Cosine Similarity measure.

The same AMIRIX AP-1000 Development Platform was used to perform all our software and reconfigurable hardware experiments. Reconfigurable hardware modules were designed in VHDL and executed on FPGA (running at 80 MHz), and software modules were written in C and executed on the PowerPC hard processor (running at 240 MHz). Both the hardware modules and software modules were designed and verified using available tools (Modelsim SE, Xilinx ISE 8.2i, EDK 8.2i, etc.), and using the same methods as in 3.3.1. Additional software experiments were performed on a 502MHz UltraSparc-IIe processor for cross-platform performance comparison purpose.

4.2.1.1. Benchmark Data Sets

Our initial experiments (in Chapter 3) were performed using synthetic data sets. Although our results from these experiments show that a substantial performance gain can be achieved using on-chip hardware support for data mining, it is crucial to verify that a large volume of real data can be processed efficiently. Hence, we decided to use real benchmark data sets for our remaining experiments. After investigating several database archives including [46],[148],[152],[159],[160], for our first reconfigurable hardware experiments [127], we used two benchmark data sets [12],[163] that are quite different from each other to illustrate the versatility of our design. The first one is SatLog Landsat Satellite benchmark, which is a dense data set, generated from Landsat Multi-Spectral Scanner (MSS) image data [12]. The data set has 4435 records, where each record (vector) has 36 attributes. The second one is Caravan Insurance benchmark, which is a sparse data set and

contains custom data for an insurance company [163]. It consists of 5822 records, where each record had 86 attributes.

4.2.1.2. Development Platform

Xilinx Virtex-II Pro has large banks of external memory which can be accessed by the FPGA hardware modules and the PowerPC embedded processor using memory controllers. The FPGA itself contains 64KB of on-chip memory [185], which is not sufficient to store the large volume of data commonly found in many data mining applications. Hence, a 64MB external memory, the DDR-SDRAM (Double-Data-Rate Synchronous Dynamic Random Access Memory), has been integrated into the system. There are 4 banks of DDR-SDRAMs available with the development platform [186], which would give a workable 256MB of memory. As shown in Figure 20, the processor's local bus (PLB) running at 80MHz acts as the glue logic for the system.

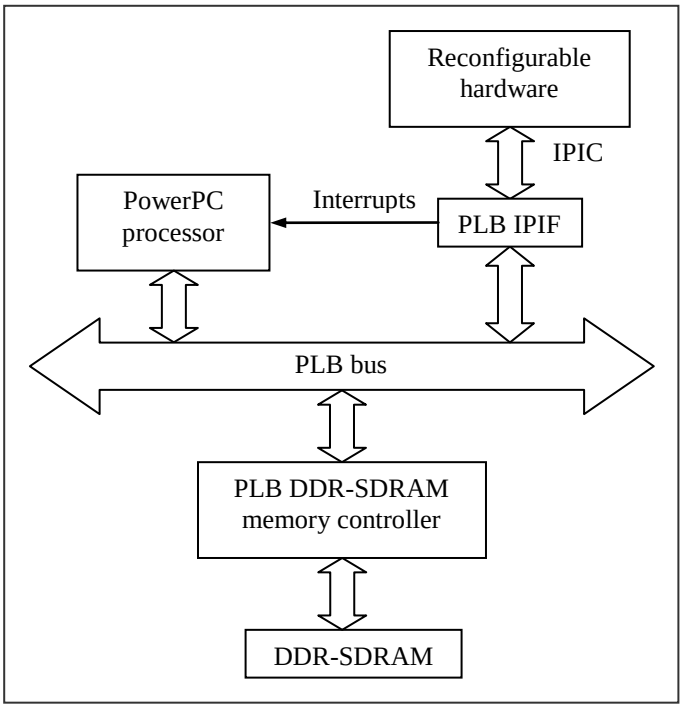


Figure 20 Development Platform Block Diagram

In order for the reconfigurable hardware to communicate with the PowerPC and the DDR-SDRAM, it is connected to the PLB bus [185],[187] through the PLB Intellectual Property Interface (IPIF) module, using a set of ports called the Intellectual Property

Interconnect (IPIC). Through the IPIF module, the reconfigurable hardware can be enhanced with stream-in data from the DDR-SDRAM.

With this hardware-software interface, the reconfigurable hardware can receive a signal from the PowerPC via the PLB bus and starts processing, reads/writes data/results from/to DDR-SDRAM, and send a signal to the PowerPC when execution is over. When the PowerPC sends a signal to the reconfigurable hardware, it can then continue to execute other tasks until the reconfigurable hardware writes back the result to the DDR-SDRAM and sends a signal to notify the processor. Execution time of the reconfigurable hardware can be obtained using the PowerPC's internal timer.

Data mining applications typically involve large volume of data, which must be stored in external memory and streamed in to the reconfigurable hardware for processing. This can incur significant delay. It is therefore necessary to address the memory access latency issue in FPGA reconfigurable hardware. It should be noted that software programs reside in on-chip memory and only data are stored in off-chip memory.

4.2.2. Multiplexer-Based Reconfigurable Hardware Design

Our reconfigurable hardware solution using a multiplexer-based approach for similarity matrix computation is presented here. The architecture of a single processing element (PE) is shown in Figure 21. This single PE consists of all the modules required for the three similarity measures. The modules represented by A, C, and E are used for Asymmetric Measure, Cosine Similarity and Extended Jaccard, respectively. Most of the modules are being reused for other similarity measures. A PE can be dynamically configured to compute any one of the three similarity measures. The multiplexers are used to control the internal structure and routing of the PE in order to select the necessary configuration. This selection depends on the signals (Sel1, Sel2, and Sel3 in Figure 21) received from the processor.

The data path of the reconfigurable hardware design consists of an array of eight PEs. Therefore, several computation configurations are possible:

1. All PEs computing Cosine Similarity simultaneously.
2. All PEs computing Extended Jaccard simultaneously.
3. All PEs computing Asymmetric Measure simultaneously.

4. All PEs computing two or three similarity measures simultaneously.

A typical scenario in computing the similarity among a set of objects is to determine which operation to perform next, based on the distribution of the computed similarities. The multiplexer configuration gives the flexibility to process the data using two or three different similarity functions simultaneously (configuration 4) or only one of the three similarity measures (configuration 1, 2 or 3). If the data are distinguishable using one or more of the similarity measures, then one may proceed to the clustering stage; otherwise, the data may have to be refined further by the feature selection or feature extraction process.

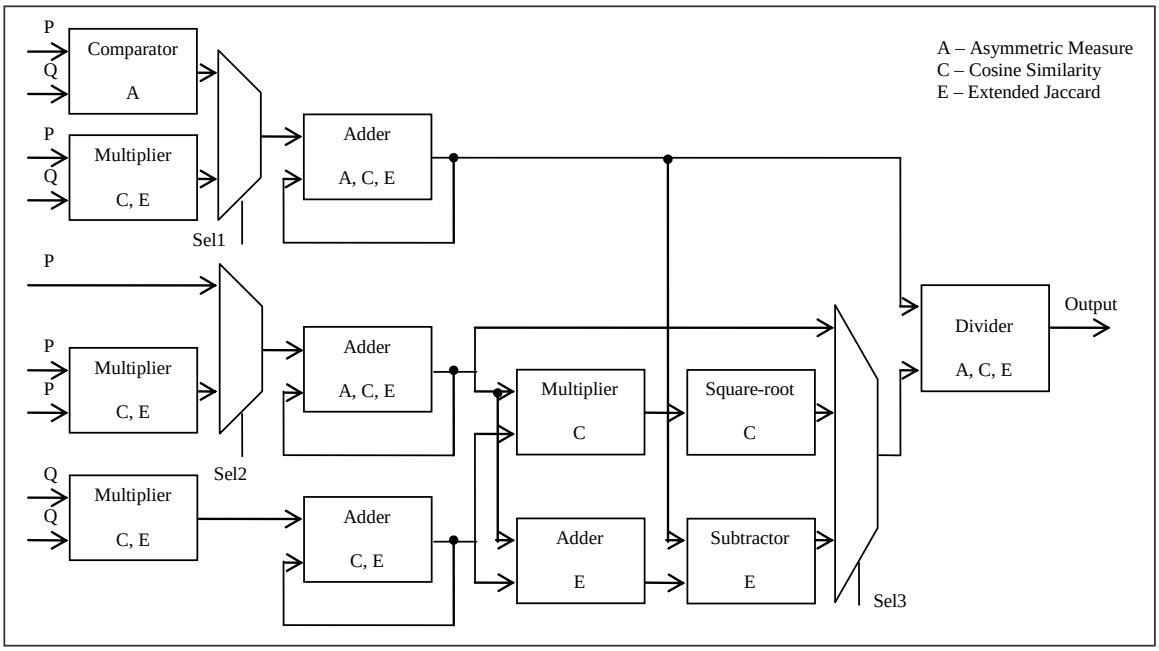


Figure 21 Multiplexer-Based Similarity Measure Computation Modules (3) in a Single PE

Reconfigurable hardware using the multiplexer-based approach enables one to employ various algorithms to process the same set of data and compare the results in order to make a sound decision. It also allows the hardware circuitry to be reconfigured dynamically, in this case, to accommodate three different similarity measures.

4.2.3. Multiplexer Experimental Results and Analysis

4.2.3.1. Space and Time Analysis

In order to examine the viability of the multiplexer-based reconfigurable design, a cost analysis on space and time is carried out. The space and time aspects of the configuration where all three similarity measures are implemented in one single module with multiplexers are compared to configurations where the three similarity measures are implemented as separate entities.

4.2.3.1.1. Space Requirement

The number of occupied slices (configurable logic blocks in FPGAs) for various configurations as shown in Table 10 shows that the total number of slices required for all three similarity measures as separate entities is 12258, whereas the number of slices required for all three similarities housing in a single module with multiplexers is 4535. The space saving using the multiplexer approach is about 63%. Figure 22 compares the cost of space for various configurations.

Configuration on a single PE	Number of slices	Total gate count	Configuration modules	Initial setup (clock cycles)
Cosine Similarity	4334	131255	4Mult+3Add+1Sqrt+1Div	77
Extended Jaccard	3987	118273	3Mult+4Add+1Sub+1Div	68
Asymmetric Measure	3937	104928	1Comp+2Add+1Div	66
Multiplexer version for all 3 measures	4535	135031	4Mult+1Comp+4Add+1Sqrt+1Sub+3Mux+1Div	77

Table 10 Space and Time Statistics for Various Configurations

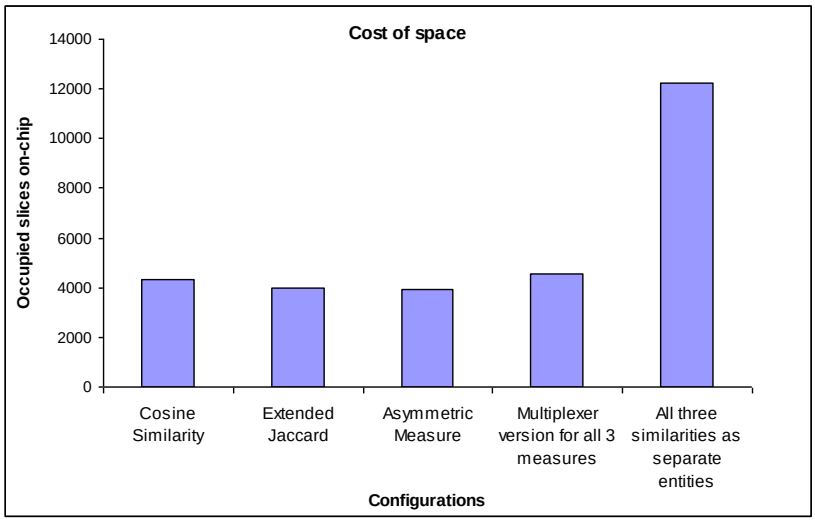


Figure 22 Cost of Space for Various Configurations

As shown in Figure 21, the multiplexer-based design reuses most of the modules for all three similarity measures. Table 11 shows the gate count for each operator. It is evident that when the highest gate count operator, the divider, is reused, one can achieve considerable space savings.

In the multiplexer-based design, the additional hardware required for reconfiguration is the multiplexers. The gate count for one multiplexer is 649, giving 1947 as the total gate count for three multiplexers. From Table 10, the total gate count for the entire multiplexer-based design is 135031. Thus, the additional hardware required for reconfiguration is a relatively insignificant 1.44% of the whole design. Furthermore, the additional overhead cost for reconfiguration is not significant considering the 63% space savings obtained in this case.

Hardware operators	Gate count
Multiplier	4003
Square-root	7300
Adder	637
Subtractor	640
Divider	83994
Comparator	739
Multiplexer	649

Table 11 Gate Count of Individual Operators

4.2.3.1.2. Time Overhead

After the initial setup, each of the four configurations produces a result in every clock cycle. As shown in Table 10, all four configurations take similar number of clock cycles for their initial setup. For the multiplexer design, the initial setup time includes the time for circuit switching and routing. Our analysis shows that the time penalty of reconfiguration overhead, i.e., time taken to reconfigure the hardware from one configuration to another, is practically zero.

4.2.3.2. Computation and Memory Access Time Analysis

Since memory access maybe a bottleneck in hardware designs, experiments were performed (on SatLog data set) using the multiplexer-based design to compute similarity matrices, to study this effect. All eight PEs computed SatLog's similarity matrix using Cosine Similarity, Extended Jaccard, and Asymmetric Measure, separately and in

sequence. The total execution time measured as shown in Table 12 (row 1) includes the time to read/write the data/results from/to the memory, the actual execution time to process the data with the corresponding similarity measure, and the time for the processor to send a signal to the reconfigurable hardware to select the necessary configuration.

In our design, the data is stored in an external memory and streamed into the FPGA to be processed. Therefore, it is important to distinguish the actual time spent on processing the data and the time spent on accessing data from the external memory.

Using the theoretical equation (9) derived in 3.4.3.1, we can accurately predict the actual time spent on processing data on the hardware design:

$$CPUClockCycles = N + \lfloor (R * Q * S) / P \rfloor \quad (9)$$

The actual time taken to execute the three similarity measures separately and in sequence using the above equation is presented in Table 12 (row 3). Since computation is performed using three different similarity measures, the processor has to send a signal twice to the multiplexers in the reconfigurable hardware to select the next configuration to compute the similarity matrix. Hence, the total time taken to select the next configuration is 1200*2 or 2400 ns.

Activity	Time (ns)
Total execution time (experimental) with reconfigurable hardware using multiplexer-based approach	91553869662
Time for the processor to select the necessary configuration	2400
Time (using equation 9) to process the data	1659219543
Time to read/write data/results from/to DDR-SDRAM	89894647719

Table 12 Breakdown of Total Execution Time on Reconfigurable Hardware

From Table 12, it can be observed that 98% of the total execution time is spent on data transfer from/to the external memory, whereas only 2% is spent on actual computation. This significant amount of time spent on data transfer to/from memory is a major performance bottleneck; thus, it is worthwhile to investigate techniques to reduce memory access time.

4.2.3.3. Hardware and Software Performance Comparison

Since software designs also have to deal with the memory access latency issue, additional experiments were carried out to examine this effect on general-purpose processor and to compare speed performance with the reconfigurable multiplexer-based hardware design. Similar to the hardware experiments, a software program using the three similarity measures in sequence to calculate the similarity matrix of SatLog was executed on UltraSparc-IIe. These results are presented in Table 13.

Considering the total execution time from Table 12 and Table 13, it can be observed that the reconfigurable hardware design is about 10 times faster than the equivalent software on the UltraSparc. This speedup, however, is not as impressive compared to our experiments in Chapter 3. The lower performance gain is due to the time spent by the reconfigurable hardware on accessing external memory.

From Table 13, it can be derived that the UltraSparc, unlike the reconfigurable hardware, spent 13% of the total execution time on data transfer from/to memory and 87% on actual computation. Given the fact that a processor has the additional overhead of fetching each instruction from the memory, it can be concluded that the actual computation (87%) is not efficient relative to the frequent memory access (13%). Further, from Table 12 and Table 13, the actual execution times show that the reconfigurable hardware can be executed up to 469 times faster than the software on the UltraSparc processor. If the memory access latency can be reduced, reconfigurable hardware designs should be able to achieve much better speed performance over software implementations.

Activity	Time (ns)
Total execution time as measured by software running on UltraSparc processor	892890000000
Actual time to process the data	778320000000
Time to read/write data/results from/to memory	114570000000

Table 13 Software Execution Time on UltraSparc

4.3. Reconfigurable Hardware Solution for Principal Component Analysis

From the above experiments, it is observed that the multiplexer-based approach is a cost-effective and flexible solution for similarity matrix computation using the three similarity measures. Most of the functional units corresponding to the sub-functions of the operations are being reused, thus saving critical space in a resource-scarce environment.

Further, the additional space and time overheads of the additional multiplexers are insignificant.

It should be emphasized that the multiplexer-based approach is quite appropriate for the similarity matrix computation in this case due to the many identical sub-functions exist among the three similarity measures. However, in other cases where the individual sub-functions among the different operations/algorithms are highly distinct, a multiplexer-based design may not be feasible. Hence, we extended our investigation to other FPGA-based reconfiguration methods, which include single context, multi context, partial reconfiguration and MultiBoot. Using any of these methods, the on-chip hardware can be dynamically reconfigured to execute any number of operations during the run-time life of an application without human intervention. These reconfiguration methods and the associated advantages and disadvantages are analyzed, discussed, and presented in detail in Chapter 5.

In this section, we introduce a dynamic reconfigurable hardware solution [128],[129] for Principal Component Analysis (PCA) using partial reconfiguration method. Signature verification and handwritten analysis are two prime example applications in portable and embedded devices. For these applications, initially PCA is applied, to reduce the dimensionality of the data, then followed by similarity measure. The first two stages of PCA, Mean and Covariance Matrix computations, were designed and implemented as hardware on FPGA to be reconfigured dynamically during execution. Using part of a handwriting analysis application together with a benchmark dataset, experiments were performed to evaluate the feasibility, efficiency, and flexibility of reconfigurable hardware.

4.3.1. Design Approach and Development Platform

Similar to our previous experiments, for these experiments also, both software and hardware versions (static reconfigurable hardware (SRH) and dynamic reconfigurable hardware (DRH)) of the various computations were implemented using a hierarchical platform-based design approach. Our hierarchical design consists of four fundamental operators, add, multiply, subtract, and divide at the lowest level, Mean and Covariance Matrix computations at the next higher abstraction level, and the PCA at the highest level.

All our hardware and software experiments were carried out on the new (circa 2011) ML605 FPGA development board [180], which is built on a 40nm CMOS process technology. The ML605 board utilizes a Xilinx Virtex 6 XC6VLX240T-FF1156 device. The development platform includes large on-chip logic resources (37680 slices), two MicroBlaze soft processors, and onboard configuration circuitry for development purpose. It also includes 15MB on-chip BRAM (Block Random Access Memory) and 512MB DDR3-SDRAM external memory to hold large volume of data. To hold the configuration bitstreams, ML605 board has several external non-volatile memories including 128MB of Platform Flash XL, 32MB BPI Linear Flash, and 2GB Compact Flash. Additional user desired features could be added through daughter cards attached to the two on-board FMC (FPGA Mezzanine Connectors) expansion connectors.

Both the static and dynamic reconfigurable hardware modules were designed in VHDL and executed on the FPGA (running at 100MHz) to verify their correctness and performance. Xilinx ISE 12.2i and EDK 12.2i were used for the SRH designs. Xilinx ISE 12.2i, EDK 12.2i, and PlanAhead12.2i (with partial reconfiguration features) were used for the DRH designs. ModelSim SE and Xilinx ChipscopePro 12.2i were used to verify the results and functionalities of the designs. No hardware optimization was attempted. Software modules were written in C and executed on the MicroBlaze processor (running at 100MHz) on the same FPGA with level-II optimization. Xilinx EDK 12.2i was used to verify the software modules.

4.3.1.1. Benchmark Data Sets

Since our intention is to provide a reconfigurable hardware solution for data mining operations, on portable, handheld, and embedded devices, we decided to utilize a data set that is appropriate for applications on these devices. After exploring many options, we decided on a real benchmark data set, the “Optdigit” [7], for recognizing handwritten characters. The database consists of 200 handwritten characters from 43 people. The data set has 3823 records (vectors), where each record has 64 attributes (elements).

We found several papers that used this data set for PCA computations. We obtained source codes written in MatLab for PCA analysis from some of the authors [98],[155]. Results from the MatLab code [155] on the optdigit data set were used to verify our results using reconfigurable hardware designs as well as software designs.

4.3.1.2. Development Platform

Xilinx Virtex-6 has large banks of external memory which can be accessed by the FPGA hardware modules and the MicroBlaze embedded processor using memory controllers. As the development platform in 4.2.1.2, in this case also, we integrated a 512MB external memory, the DDR3-SDRAM [182], into the system, since on-chip memory is not sufficient to store the large volume of data commonly found in many data mining applications

Similar hardware-software interface, as in Figure 20, was designed, in order for the hardware (both SRH and DRH) to communicate with the MicroBlaze and the DDR3-SDRAM, using the PLB bus [78]. Execution times of the hardware as well as MicroBlaze were obtained using the hardware XPS_Timer [79], which was running at 100MHz.

4.3.1.3. Reconfiguration on Virtex 6

There are two different reconfiguration methods that can be used with Virtex-6 FPGAs: MultiBoot and Partial Reconfiguration. MultiBoot [82] is a reconfiguration method that allows full bitstream reconfiguration, whereas partial reconfiguration [183] allows partial bitstream reconfiguration. We used partial reconfiguration method for our dynamic reconfigurable hardware design.

4.3.1.3.1. Partial Reconfiguration

Dynamic partial reconfiguration allows reconfiguring parts of the chip that requires modification, while interfacing with the other parts that remain operational [53],[54],[183]. Figure 23, which is modified from [183], depicts the basic premise of partial reconfiguration. During the design and implementation processes, the chip is divided into two different parts: reconfigurable and static. Initially, the FPGA is configured by downloading a full bitstream for the entire chip upon power-up. Next, the function implemented in the Reconfigurable Modules (RM), i.e., reconfigurable parts, can be modified by downloading one of the several partial bitstreams, while the static parts remain operational [183]. The features, advantages, and disadvantages of partial reconfiguration are discussed in detail in 5.2.3.

Internal Configuration Access Port (ICAP) is the fundamental module used to perform in-circuit reconfiguration [55],[191]. As indicated by its name, ICAP is an internally

accessed resource and not intended for full chip configuration. As stated in [80], this module "provides the user logic access to the FPGA configuration interface, allowing the user to access configuration registers, readback configuration data, and partially reconfigure the FPGA" after initial configuration is done [82]. The protocol used to communicate with ICAP is a subset of the SelectMAP protocol [53].

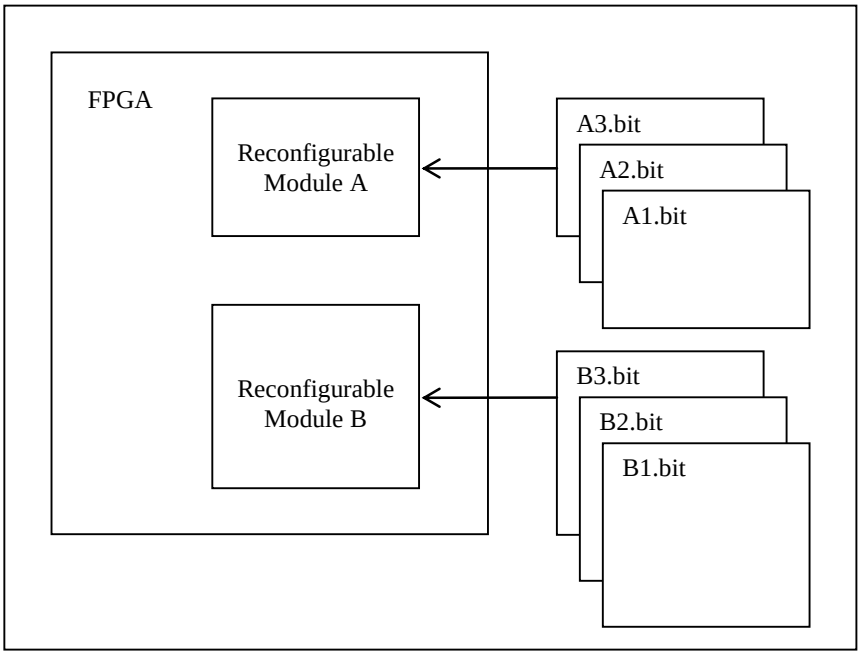


Figure 23 Basic Premise of Partial Reconfiguration [183]

Virtex-6 FPGAs support reconfiguration via internal and external configuration ports [53],[54],[191]. Full and partial bitstreams are typically stored in external non-volatile memory, and the configuration controller manages the loading of the bitstreams to the chip and reconfigures the chip when necessary. Configuration controller can be either a microprocessor or routines (small state machine) programmed into the FPGA. The reconfiguration can be done using a wide variety of techniques, one of which is shown in Figure 24. In our design, full and partial bitstreams are stored in the Compact Flash (CF), and ICAP is used to load the partial bitstreams. In this design, the ICAP module is instantiated and controlled through software running on the MicroBlaze processor. During run-time, the MicroBlaze processor transmits the partial bitstreams from the non-volatile memory to ICAP to accomplish the reconfiguration processes.

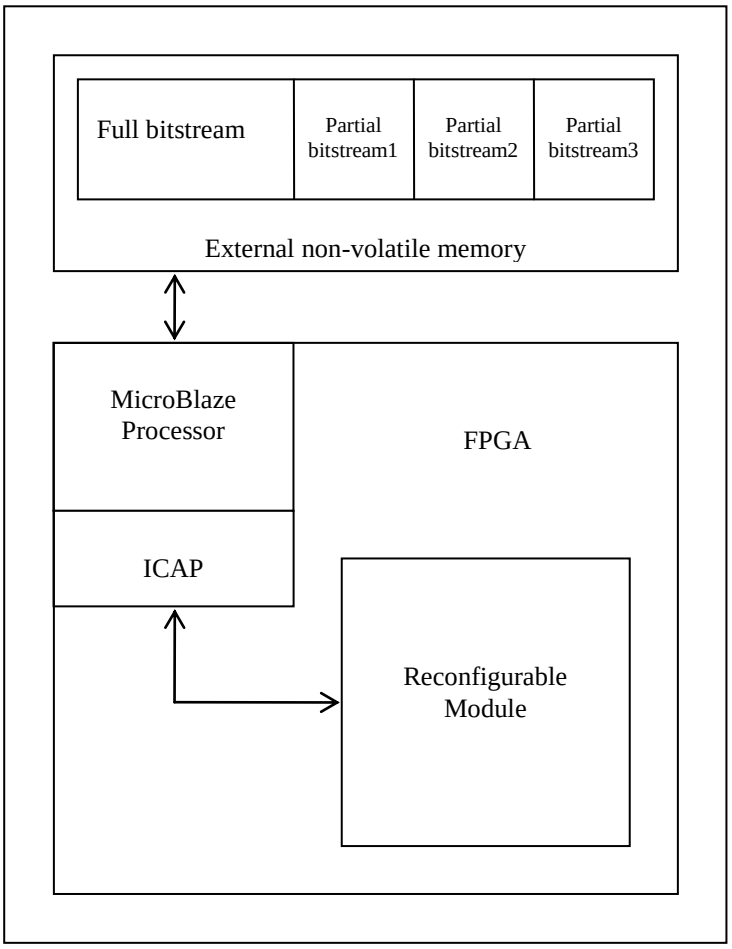


Figure 24 Partial Reconfiguration by MicroBlaze and ICAP [53][191]

4.3.1.3.2. MultiBoot

The MultiBoot feature in Virtex-6 chips enables reconfiguring the FPGA with different full configuration bitstreams stored in a PROM, either Platform Flash XL or BPI Flash PROM [11],[82],[83]. After the FPGA is configured by the initial full bitstream from the PROM, the FPGA application can trigger a MultiBoot event and reconfigures itself from different full bitstreams in the PROM [82]. The number of bitstreams supported by the Virtex-6 MultiBoot feature depends upon the density of the target FPGA, and is typically to a maximum of four [82].

A MultiBoot control module (Figure 25), which includes both a state machine and the ICAP, must be incorporated into the FPGA application to send the necessary commands (called IPROG) to trigger a MultiBoot reconfiguration [11],[82],[166]. Although the MulitBoot feature can be triggered by different methods, a simple and flexible way is to

create a small state machine (internal to the FPGA) to send an internal command sequence (called IPROG) to the FPGA configuration control via the ICAP. Once a MultiBoot operation is triggered, (i.e., when the configuration logic receives the IPROG command), the FPGA restarts its configuration process as usual: the FPGA performs a complete reset (except the dedicated reconfiguration logic) and reconfigures from the PROM the new full configuration bitstream [82],[83]. The features, advantages, and disadvantages of MultiBoot are discussed in detail in 5.2.4.

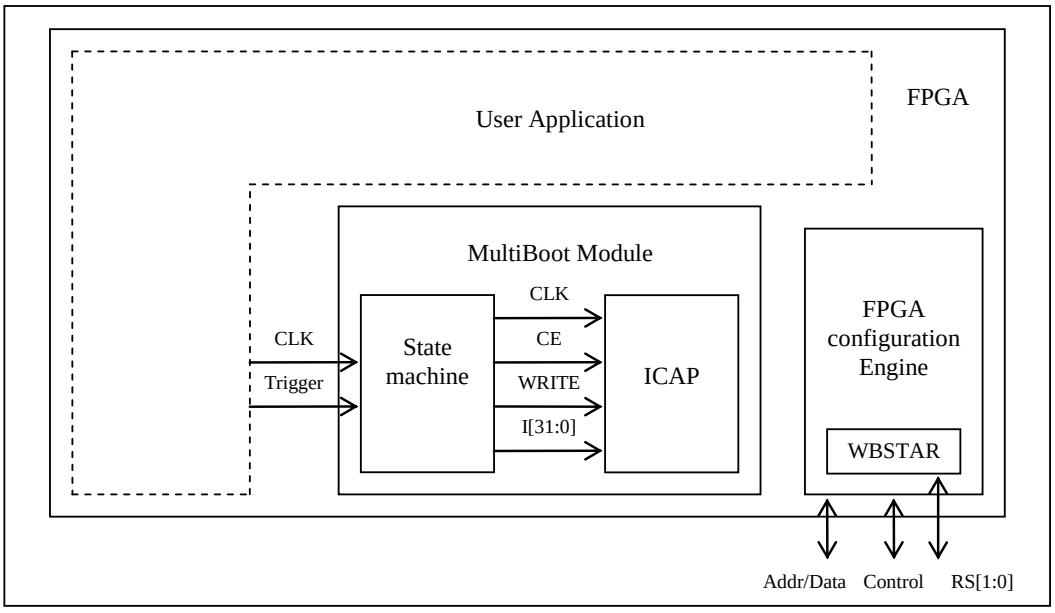


Figure 25 MultiBoot Design Block Diagram [82]

4.3.1.4. Our Interface – FPGA and Host System

Our reconfigurable computing system is implemented as a standalone processing unit, which consists of a reconfigurable processing unit (RPU), a non-volatile memory to store the configuration bitstreams, and the SDRAM to store data and results.

We could either use an embedded processor or a small finite-state-machine (FSM) on chip to control the configuration flow as well as to control other functions. We integrated an embedded MicroBlaze processor to our RPU to control the configuration flow. Some of the control functions were executed on the processor and other control functions were implemented using FSMs.

Our reconfigurable computing system is connected to the external host system through the PCIe (Peripheral Component Interconnect Express) bus. However, some portable and handheld devices might not have a microprocessor. In this case, we can use simple control logic to integrate our reconfigurable computing system to the portable device. In both cases, we can load and initiate the application on the reconfigurable system (on chip) from the host system by pressing a switch and the applications are processed on chip independent of the host system. The interactions between our reconfigurable system and the host system are kept to a minimal.

4.3.2. Dynamic Partial Reconfigurable Hardware for PCA

Initially, we investigated different stages of PCA [52],[87],[140], and then considered each stage as individual operations, and provided hardware support for each stage separately. We then focused on a reconfigurable hardware solution for the first two stages of the PCA computation: Mean and Covariance Matrix computations. Our hardware design can be reconfigured partially and dynamically from one stage to another, in order to perform these two operations on the same area of the chip.

The original equations [87],[140] for Mean and Covariance Matrix for PCA computation are as follows:

Original equation for Mean:

$$\overline{X_j} = \frac{\sum_{i=1}^{n} X_{ji}}{n} \tag{10}$$

Original equation for Covariance Matrix:

$$Cov(X_j, X_{j+1}) = \frac{\sum_{i=1}^{n} (X_{ji} - \overline{X}_j)(X_{(j+1)i} - \overline{X_{j+1}})}{(n-1)} \tag{11}$$

However, we modified the equations slightly, so that the division of Mean is delayed, and done at the final Covariance output. This way, we are able to use integer operations for both the Mean and Covariance reconfigurable hardware designs. It should be noted we are planning to use floating-point operations for the last two stages of the PCA computations, which include QR algorithm for eigenanalysis. Accordingly, we might extend the use of floating-point operations to Mean and Covariance computations as well.

The modified equation for Mean is:

$$\overline{X_j} = \sum_{i=1}^{n} X_{ji} \tag{12}$$

The modified equation for Covariance Matrix is:

$$Cov(X_j, X_{j+1}) = \frac{\sum_{i=1}^{n} (nX_{ji} - \overline{X}_j)(nX_{(j+1)i} - \overline{X_{j+1}})}{n(n-1)} \tag{13}$$

Reconfigurable hardware design for the Mean computation consists of a data path and a control path. As shown in Figure 26, the data path of the Mean was designed as a sequence of an adder and an accumulator register providing a feedback loop to the adder. Mean is measured along the dimensions; hence the total number of Mean results is equal to the number of dimensions (*m*) in the data set.

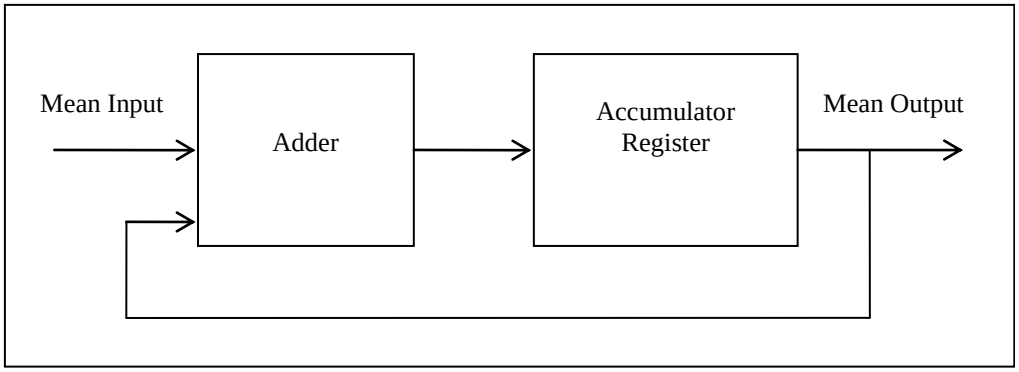


Figure 26 Data Path for the Mean Module

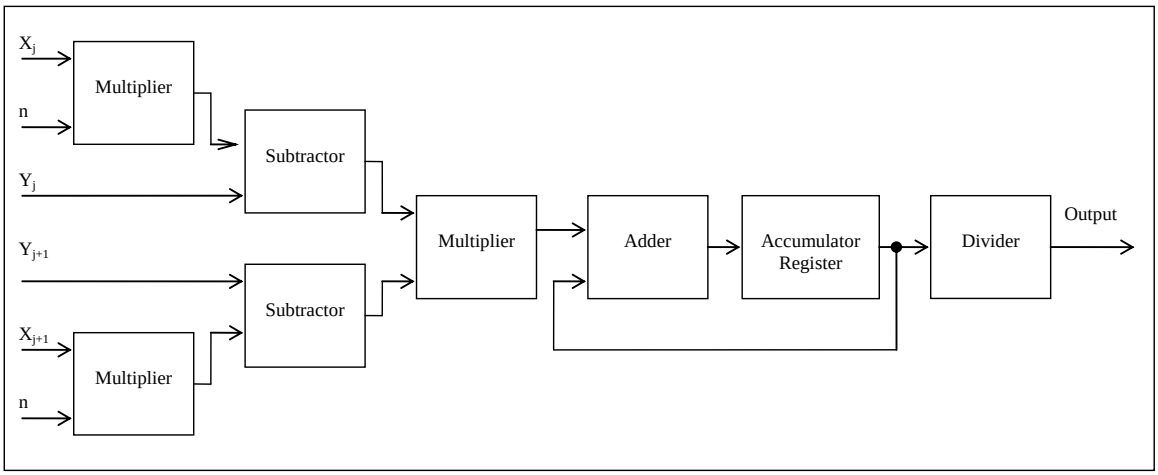


Figure 27 Data Path for the Covariance Matrix Module

Reconfigurable hardware design for the Covariance computation also consists of a data path and a control path. Data path of the Covariance Matrix design, as depicted in Figure 27, consists of 3 multipliers, 2 subtractors, an adder, an accumulator register, and a divider. Since division is not done in the Mean operation, the inputs (X_j and X_{j+1}) (from the data set) to the Covariance are multiplied by the number of vectors (n) before computing the deviation from Mean. The inputs, Y_j and Y_{j+1}, are the Mean results from the above Mean module.

The Covariance Matrix is a square symmetric matrix, hence only the diagonal elements and the elements of the upper triangle have to be computed. Thus, the total number of Covariance results is equal to $m*(m+1)/2$, where m is the number of dimensions. The upper triangle elements of the Covariance Matrix are measured between two dimensions, and the diagonal elements are measured between one dimension and itself.

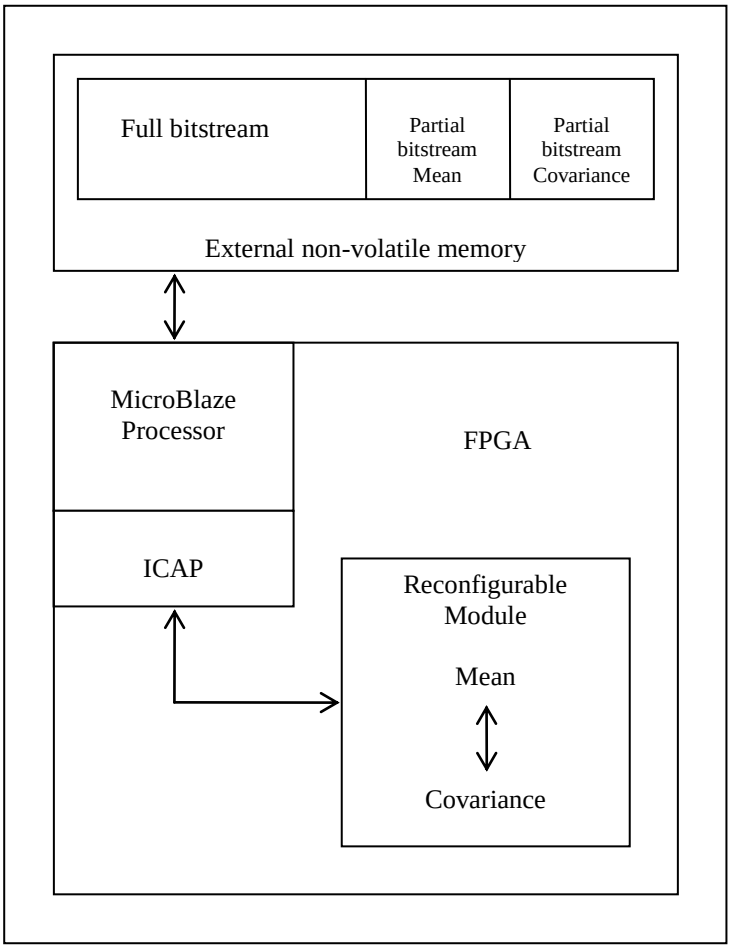


Figure 28 Partial Reconfiguration of Mean and Covariance

In our design, the numerator of the Covariance is computed for an element of the Covariance Matrix and only the final Covariance result goes through the divider. The divider used is a Xilinx IP core, Divider Generator v3.0 [176].

Initially the full bitstream that includes the reconfigurable module (RM) of the Mean design is downloaded to the FPGA and the Mean computation is performed. After execution of the Mean, the RM for Mean sends a signal to the processor. Then the processor downloads the partial bitstream for the RM for Covariance Matrix and the Covariance computation is performed. Loading of the partial bitstream is done without interrupting the operations of the remaining parts of the chip. After the execution of the Covariance, the RM for Covariance sends a signal to the processor. As shown in Figure 28, partial bitstreams for Mean and Covariance Matrix modules are stored in external non-volatile memory and downloaded to the region, of the RM, when necessary.

After processing one set of data for both the Mean and the Covariance. The processor can dynamically and partially reconfigure the chip again to the Mean, without downloading the full bitstream. Thus, any number of Mean and Covariance computations can be performed for any number of data sets, without interrupting the operation of the system.

4.3.3. PCA Experimental Results and Analysis

4.3.3.1. Space and Time Analysis

4.3.3.1.1. Space Requirement

In order to examine the viability of the dynamic reconfigurable hardware design using partial reconfiguration, cost analysis on space and time is carried out for hw_v1 (static reconfigurable hardware (SRH)) and hw_v2 (dynamic reconfigurable hardware (DRH)).

As shown in Table 14, the total number of slices required for Mean and Covariance as separate entities in hw_v1 is 6031, whereas, the number of slices required for the DRH (hw_v2) is 4279, which is simply the Covariance module with a larger RM than the Mean. Hence, space saving using partial reconfiguration (in order to reconfigure the hardware on-chip from one computation to another, i.e., from Mean to Covariance) is about 30%

since the same area of the chip is being reused for both the Mean and Covariance computation.

Configuration	Number of occupied slices
hw_v1a – SRH (Mean as a separate entity)	2597
hw_v1b – SRH (Covariance as a separate entity)	3434
hw_v2 – DRH with the larger RM (Covariance module)	4279

Table 14 Space Statistics for Various Configurations – hw_v1 vs. hw_v2

4.3.3.1.2. Reconfiguration Space Overhead – Extra Hardware On Chip for Reconfiguration

Hardware ICAP (Internal Configuration Access Port) for reconfiguration enables an embedded processor, such as the MicroBlaze, to read and write the FPGA configuration memory through the ICAP at run time. In our design, using MicroBlaze and ICAP, configuration bitstreams are fetched from the compact flash (CF) and the chip is reconfigured at run time. SystemACE Interface Controller on chip is the interface between the PLB and the SystemACE CF peripheral.

Since we are storing the full and partial bitstreams in external CF, the only hardware required on chip for reconfiguration is the ICAP and the SystemAce Interface Controller. On Virtex 6, resource utilizations for ICAP [177] and the SystemAce Interface Controller [193] (required for CF) are about 460 and 78 slices respectively, resulting in a total of 538 slices. The above utilization numbers should be regarded as estimates since these numbers might vary slightly when they are combined with other designs in the system.

4.3.3.1.3. Reconfiguration Time Overhead

The time taken to reconfigure from Mean to Covariance is 51542205 PLB clock cycles, which is 515 milliseconds with the MicroBlaze running at 100 MHz. When implementing the design using PlanAhead, the partial bitstream for the reconfigurable module is 169776 bytes, or 1358208 bits. According to the partial reconfiguration user guide [183], using ICAP at 100MHz and 3.2Gbps, a partial bit file can be loaded in about: 1358208 bits / 3200000000 bps = 424 microseconds. This is significantly less than the measured 515 milliseconds.

After further investigations, it is found that this big difference is quite normal due to the partial bitstreams being stored in the CF and also the sequential access nature of the MicroBlaze processor.

The above calculation is correct, provided that ICAP is continuously enabled. That is, the ICAP should meet the following requirements at the input of ICAP: Clk is 100MHz and is applied continuously; CE of ICAP is asserted continuously; and write ICAP is asserted continuously and input data are given in every input Clk.

In the above scenario, the configuration uses the full bandwidth of 3.2Gbps (100MHz x 32 bits), and the reconfiguration can be completed within 424 microseconds. However, MicroBlaze executes instructions sequentially and the partial reconfiguration sequence is as follows:

- MicorBlaze requests SystemACE controller to retrieve data from the CF.
- SystemACE controller reads data from the CF (since CF is external to the chip, there is access delay).
- MicroBlaze requests this data from SystemACE controller and stores it in an internal register.
- MicroBlaze writes the data to ICAP.

Because of this sequential execution, partial reconfiguration takes about 515ms. Partial reconfiguration time is usually in the range of milliseconds for the bit files of size similar to this case (around 1358208 bits).

4.3.3.2. Results and Analysis for hw_v1 (SRH) and hw_v2 (DRH) for Mean and Covariance Computations

Experiments were performed (on opdigit data set) to evaluate the dynamic reconfigurable hardware designs. For all the experiments, data were read from the DDR3-SDRAM, processed, and the final results were written back to the SDRAM. Unlike our similarity matrix experiments in 4.2.1.2, data were not streamed-in.

In our reconfigurable hardware designs, data size, number of vectors, number of elements, and hence the number of Covariance results are variables that can be changed externally. These experiments were performed using various data sizes in order to

examine scalability. The number of elements was kept the same and only the data size and the number of vectors varied.

4.3.3.2.1. Execution Time for hw_v1 (SRH)

In order to evaluate the dynamic reconfigurable hardware (DRH) designs for Mean and Covariance, we designed and implemented static reconfigurable hardware (SRH) for Mean and Covariance computations as separate entities. With SRH design, a full bitstream of the Mean was downloaded to the chip and the chip was reconfigured only once. In order to run the Covariance design, a full bitstream of that design had to be downloaded again and the entire chip had to be reconfigured. System had to be interrupted for every download and reconfiguration process.

Experiments were performed separately on Mean and Covariance SRH designs with varying data sizes and the execution times are obtained as shown in Table 15. Similar to the Figure 29(a), the execution time for Mean increases linearly with the size of the data set. Covariance shows similar linear behaviour.

Data size	No. of vectors	Execution time (plb_clk_cycles)		
		Mean	Covariance	Total
24448	382	1005737	56892238	57897975
48960	765	2012722	113786562	115799284
73408	1147	3017061	170532385	173549446
97856	1529	4021429	227278148	231299577
122368	1912	5028399	284172486	289200885
244672	3823	10052778	568049998	578102776
489344	7646	20104163	1135953559	1156057722
734016	11469	30155561	1703857078	1734012639
978688	15292	40206963	2271760678	2311967641
1223360	19115	50258372	2839664247	2889922619
1468032	22938	60309756	3407567780	3467877536
1712704	26761	70361149	3975471460	4045832609

Table 15 Execution Times for Mean and Covariance – hw_v1

4.3.3.2.2. Execution Time for hw_v2 (DRH)

Using the DRH design, a full bitstream with Mean was downloaded. Mean operation was performed. Then the hardware was reconfigured to Covariance and the Covariance operation was performed. In order to process different data sets, the hardware was again

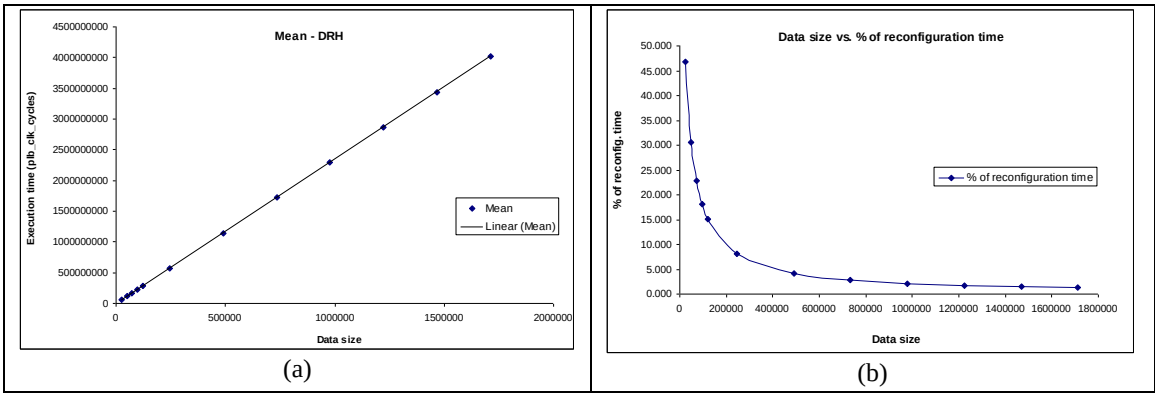
reconfigured to Mean and so on without downloading the full bitstream or without interrupting the system operation.

Unlike hw_v1 (SRH), for hw_v2 (DRH), execution times were measured sequentially: execution of Mean, reconfiguration from Mean to Covariance, execution of Covariance. These execution times for different data sizes are presented in Table 16.

It is evident from Table 16 (column 4) that the reconfiguration time does not vary with the size of the data set. There is a slight variation but it is less than 100 clock cycles, i.e., 1000 ns. This is expected, since reconfiguration time depends on the size (i.e., area) of the reconfiguration module but not on the number of data being processed.

Data size	No. of vectors	Execution time (plb_clk_cycles)				
		Mean	Reconfiguration	Covariance	Total Time	% of time on reconfiguration from total
24448	382	955523	51542209	57471300	109969032	46.870
48960	765	1912491	51542183	114881143	168335817	30.619
73408	1147	2867011	51542220	172170642	226579873	22.748
97856	1529	3821520	51542128	229467001	284830649	18.096
122368	1912	4778408	51542188	286872341	343192937	15.018
244672	3823	9553384	51542205	573449293	634544882	8.123
489344	7646	19105721	51542149	1146710103	1217357973	4.234
734016	11469	28657878	51542190	1719981813	1800181881	2.863
978688	15292	38210042	51542145	2293247773	2382999960	2.163
1223360	19115	47762370	51542175	2866515344	2965819889	1.738
1468032	22938	57314615	51542161	3439785664	3548642440	1.452
1712704	26761	66866888	51542182	4013047309	4131456379	1.248

Table 16 Execution Times for Mean, Reconfiguration, and Covariance – hw_v2



(a) (b)

Figure 29 (a) Mean – hw_v2, (b) Percentage of Reconfiguration Time from Total

As shown in Figure 29(a), execution time for Mean increases linearly with the size of the data set. Covariance shows similar behaviour. Also the total execution time of the

whole process increases linearly with the size of the data set. This ascertains our expectation that reconfigurable hardware scales linearly with data size.

In hw_v2 (DRH), for smaller data sets, a significant percentage of total time is spent on reconfiguration. From Table 16 (last column) and Figure 29(b), it can be seen that reconfiguration time is amortized and decreases as the size of the data increases.

4.3.3.3. Comparison – Total Time for hw_v1 (SRH) and hw_v2 (DRH)

Data size	Execution time (plb_clk_cycles)		Hw_v2/Hw_v1
	Total time for hw_v1 (SRH)	Total time for hw_v2 (DRH)	
24448	57897975	109969032	1.899
48960	115799284	168335817	1.454
73408	173549446	226579873	1.306
97856	231299577	284830649	1.231
122368	289200885	343192937	1.187
244672	578102776	634544882	1.098
489344	1156057722	1217357973	1.053
734016	1734012639	1800181881	1.038
978688	2311967641	2382999960	1.031
1223360	2889922619	2965819889	1.026
1468032	3467877536	3548642440	1.023
1712704	4045832609	4131456379	1.021

Table 17 Total Time for hw_v1 vs. hw_v2

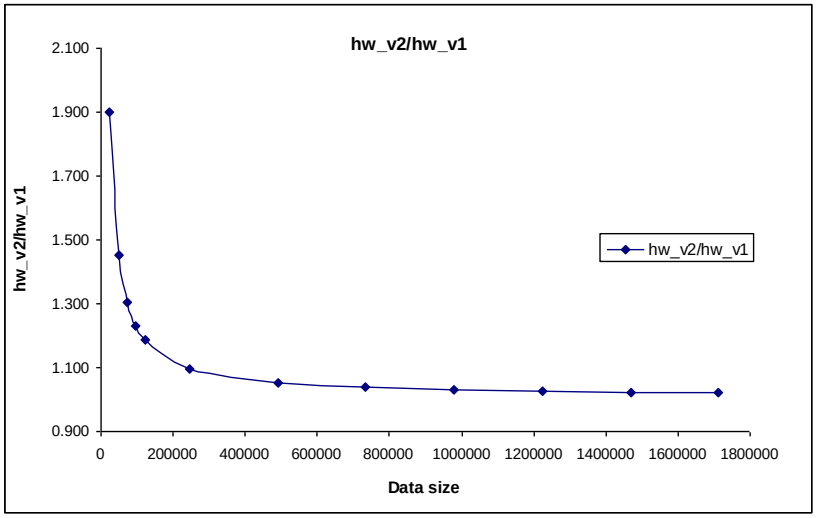


Figure 30 hw_v1 vs. hw_v2 in terms of Total Time

As can be seen in Table 17, there is a significant difference between the total times of hw_v1 (SRH) and hw_v2 (DRH) for smaller data sizes but this decreases as the size of the data increases. As mentioned above, for smaller data sets in hw_v2 (DRH), a significant

percentage of the total time is spent on reconfiguration though the actual reconfiguration time remains the same with varying data set size. The configuration time becomes negligible as the size of data increases, as shown in Figure 30.

4.3.3.4. Performance Comparison: hw_v1 (SRH), hw_v2 (DRH), vs. Software on MicroBlaze

Software experiments were performed using the MicroBlaze soft processor on the same platform, in order to evaluate the DRH as well as the SRH. Similar to hw_v2 (DRH), execution times for software designs were also measured in a sequence, first for Mean and then for Covariance. Although execution times were obtained using different data sizes, for the performance comparison, only the data set of size 244672 is used. The execution times are presented in Table 18.

Considering the total execution times from Table 18, hw_v2 (DRH) is 4.3 times faster while the hw_v1 (SRH) is 4.7 times faster than the equivalent software on the MicroBlaze. The speedups, however, are not as impressive compared to our experiments in Chapter 3. The lower performance gain is due to the additional time incurred on accessing external memory by the reconfigurable hardware [127]. It should also be noted that no optimization was attempted on hardware designs.

	Execution time (plb_clk_cycles)			Speedup	
	Hw_v1 – SRH	**Hw_v2 - DRH**	**Sw on MicroBlaze**	**DRH vs. Sw**	**SRH vs. Sw**
Mean	10052778	9553384	55504920	5.8	5.5
Reconfiguration		51542205			
Covariance	568049998	573449293	2641989233	4.6	4.7
Total	578102776	634544882	2697494153	4.3	4.7

Table 18 Performance Comparison – hw_v1 and hw_v2 vs. Software on MicroBlaze

From Table 18, execution times for the Mean and Covariance modules in hw_v1 (SRH) and hw_v2 (DRH) configurations are slightly different. Additional experiments were carried out to investigate this discrepancy, which are presented in 4.4.

4.3.3.5. Computation and Memory Access Time Analysis: hw_v2 (DRH) vs. Software on MicroBlaze

Since reconfigurable hardware spent considerable time in accessing memory, it is important to distinguish the actual time spent on processing the data and the time spent on

accessing data from the external memory. This may give us clues to improve efficiency of the hardware.

We made theoretical estimates on the computation time for the Mean and the Covariance Matrix separately. One Mean computation theoretically takes 2 clock cycles. Since Mean is measured along the dimension of the vectors, the number of Mean computations equals to the number of dimensions (*m*). Hence, the total number of cycles to compute Mean for (*n*) number of vectors and (*m*) number of dimensions is:

$$No.ofCyclesforMeanComputation = 2 * n * m \quad (14)$$

For Covariance, each computation takes 5 clock cycles and followed by a division, which takes 34 clock cycles. The number of Covariance results equals to the number of (upper triangle and diagonal) elements of a square symmetric matrix, which is equal to *m**(*m*+1)/2. Hence, the total number of cycles to compute the Covariance for (*n*) number of vectors and (*m*) number of dimensions is:

$$No.ofCyclesforCovarianceComputation = (5 * n + 34) * m * (m + 1)/2 \quad (15)$$

Activity	Time (plb_clk_cycles)	
	Mean	Covariance
Execution time (experimental)	9553384	573449293
Time (theoretically) to process the data	489344	39829920
Time to read/write data/results to/from ddr3-sdram	9064040	533619373

Table 19 Breakdown of Execution Time for Mean and Covariance – hw_v2

Using equations 14 and 15, the execution times for Mean and Covariance for hw_v2 are estimated and presented in Table 19. The percentage of time spent on actual Mean computation is 5.12% and on data transfer to/from external memory is 94.88%. The percentage of time spent on actual Covariance computation is 6.95% and on data transfer to/from external memory is 93.05%. Significant processing time is spent on reading and writing data/results from/to the external memory; hence, it is the major bottleneck in the system and worthy of further improvement.

Since software designs also have to deal with the memory access latency issues, additional experiments were carried out to examine memory access latency using the

MicroBlaze. These results are present in Table 20. The percentage of time spent on actual Mean computation in software is 83.47%, whereas data transfer from external memory amounts to 16.53%. Similarly, actual Covariance computation takes 88.16% and data transfer from external memory amounts to 11.84%.

Activity	Time (plb_clk_cycles)	
	Mean	Covariance
Execution time running equivalent sw on MicroBlaze processor	55504920	2641989233
Actual time to process the data	46328331	2329263115
Time to read/write data/results to/from ddr3-sdram	9176589	312726118

Table 20 Breakdown of Execution Time for Mean and Covariance – Software on MicroBlaze

Considering the actual execution times for Mean and Covariance modules from Table 19 and Table 20, the Mean module of hw_v2 can be executed up to 95 times faster than its software counterpart on the MicroBlaze. The Covariance module of hw_v2 can be executed up to 59 times faster than software on the MicroBlaze. It should be noted, that for the reconfigurable hardware designs, no optimization was done and data were not streamed in.

4.4. Further Investigation and Analysis on Dynamic Partial Reconfigurable Hardware

When we performed the above experiments, execution times of the Mean and the Covariance modules in the hw_v1 (SRH) and hw_v2 (DRH) were slightly different. Additional experiments and analysis were carried out to investigate this discrepancy. Our objective was to ascertain that the results obtained from the above (partial reconfigurable) experiments are correct and to find out why there is a discrepancy in the execution times between the two hardware versions and what causes this discrepancy.

For these experiments, we used a simple Adder and a Multiplier, which takes one clock cycle each to execute. For both the static reconfigurable hardware (SRH) and the dynamic reconfigurable hardware (DRH), the Adder and Multiplier were designed and implemented using the same design approach and the development platform as in 4.3.1.

The dynamic reconfigurable hardware solutions used partial reconfiguration to reconfigure from one function to another.

Experiments were performed on the same "optdigit" benchmark data set for handwritten analysis. For all the experiments, data was read from the DDR3-SDRAM, processed, and final results were written back to the SDRAM.

These experiments and results are presented and discussed in details in Appendix C.

4.4.1. Results, Analysis, and Proposed Solutions

From the experimental results and analysis in Appendix C, it is clear that addition or multiplication time is only one clock cycle, regardless of the number of consecutive additions or multiplications. This is expected, since addition or multiplication operations are designed to execute in only one clock cycle, and it is synchronous with the system clock.

It is also observed that per read/write time varies with different number of consecutive reads/writes within the same hardware versions. Read/write operations are performed through PLB bus interfacing with the external memory. Factors such as the asynchronous nature of the read/write operation, completing read/write operation just after the rising-edge of the clock which delays the state change, refresh operations, traffic patterns, all have an impact on the data transfer time. Hence, per read/write time varies for different number of consecutive reads/writes. These reasons are explained in detail in 4.4.1.1 below.

In addition, per read/write time varies with different number of consecutive reads/writes between the two hardware versions. For instance, these two hardware versions were designed and implemented using different design flows and the interfacing of dynamic reconfigurable hardware with the rest of the system is different from the interfacing of the static reconfigurable hardware with the rest of the system. These might cause the difference in execution times between hw_v1 and hw_v2. These issues are further elaborated in 4.4.1.1 below.

4.4.1.1. Factors that Impact Read/Write Times

There is a difference between the two hardware versions in terms of read/write, and also the range of read/write time within the same hardware version. In this section, we discuss

the factors that may impact the read/write operations and how these factors affect these operations.

4.4.1.1.1. SDRAM

We present a summary of the factors that impact read/write time with regard to SDRAM and then discuss how and why they impact read/write time.

- Activate time for new banks/rows
- Precharge time for changing rows within the same bank
- Write Recovery time to change from read access
- Bus Turnaround time to change from read to write
- Refresh time

Data transfer time (read and write) is affected by the overhead associated with accessing the SDRAM. Based on the traffic pattern, the amount of overhead can vary significantly. Hence, it is important to analyze both the command and address patterns. As illustrated in [175], "for the command pattern, grouping reads together and writes together result in least amount of overhead, whereas requesting alternate write and read commands result in high overhead to account for Write Recovery time and Bus Turnaround time". For example, if the memory controller receives a read command, when it is in the write state, the controller has to wait for the write_to_read time before issuing the read command [42]. Similarly, if the controller receives a write command from the command block, when it is in the read state, the controller has to wait for the read_to_write time before issuing the write command [42]. Address patterns can also have a significant impact on the overhead. As mentioned in [175], "sequentially bursting across a row has little or no overhead, whereas, a random address pattern results in high overhead due to Activate and Pre-charge times".

Refresh operations also have an impact on the data transfer time. From the simulation and timing diagrams, one read operation and one write operation take 31 and 17 clock cycles respectively. However, these values do not take into account the time taken up by the refresh operations of the SDRAM. Our 512 MB DDR3-SDRAM device needs to be refreshed every 7.8 micro seconds. Hence, the SDRAM memory controller sends a refresh commands every 7.8 micro seconds [42]. In a typical commercial SDRAM, usually, a

period for refreshing all rows is 64 ms [115]. For an SDRAM consisting of several banks, whose cells are arranged in 8K (=8192) rows, 8192 refresh commands must be issued every 64 ms to ensure all rows of all banks are properly refreshed. Hence, the average refresh rate is 7.8125 micro seconds.

As illustrated in [73], assuming that it takes 4 clock cycles to access (read) each row, then it takes 8192*5 = 32,768 cycles to refresh all rows. At a clock rate of 100 MHz, the time needs to refresh all rows is = 32768/100*10^6 = 0.328 ms. At a clock rate of 100 MHz, the time needed to refresh one row is = 4/100*10^6 =0.04 micro seconds. Thus, refresh overhead is 0.328/64 = 0.0051, which is 0.51% of total time available for accessing memory.

4.4.1.1.2. Difference Between Two Hardware Versions

We present a summary of the factors that impact read/write time due to the difference between two hardware versions and then discuss how and why they impact read/write time.

- Interfacing of dynamic reconfigurable hardware with the rest of the system is different from the interfacing of the static reconfigurable hardware with the rest of the system.
- Two hardware versions are designed and implemented using different design flows.

Our two hardware versions, hw_v1 and hw_v2, were implemented with different design flows. Static reconfigurable hardware, hw_v1, was implemented using EDK design flow, whereas dynamic reconfigurable hardware, hw_v2 was implemented using PlanAhead design flow.

Also, in hw_v2, the Adder and the Multiplier were implemented as reconfigurable modules (RMs), and a specific portion of the chip was partitioned for the reconfigurable modules using PlanAhead design flow. However, for hw_v1, we did not select a specific portion of the chip for Adder or Multiplier modules. The compiler decided the placing and routing of these modules.

Further, with hw_v2, the interfacing of the reconfigurable modules with the rest of the system is different due to the ICAP interfacing with the reconfigurable modules. This is not the case with the Adder and Multiplier modules in hw_v1.

The above factors might have an impact on the execution times of the reconfigurable modules. For example, consider the operation count (op_cnt) for the two hardware versions in Table 21, which entries are taken from Table 33 and Table 34 from Appendix C. From our experiments and analysis, we know that op_cnt only takes one clock cycle. The additional time taken is the time for the processor to send a signal to the hardware to start processing the data and for the hardware to send a signal to inform the processor once the data is processed, which occurs at the beginning of the process and at the end of the process respectively. From Table 21, there is a time difference (last column) in the additional time between hw_v1 and hw_v2.

Data size		Execution time for the first state (plb_clk_cycles)		Additional time taken (plb_clk_cycles)		Time difference (plb_clk_cycles)
		hw_v1	hw_v2	hw_v1	hw_v2	
0.1x	24467	24540	24590	73	123	50
1x	244672	244749	244784	77	112	35
10x	2446720	2446795	2446854	75	134	59

Table 21 Time for Operations Count (op_cnt) for hw_v1 and hw_v2

Actual addition and multiplication operations take only one clock cycle in both hw_v1 and hw_v2. In hw_v2, the addition and multiplication operations are done within the reconfigurable hardware module, without any interaction with the rest of the system. Unlike addition and multiplication operations, read and write operations are performed through PLB bus interfacing with the rest of the system. This creates an additional overhead for the read/write operations in hw_v2.

Hence, part of the difference in execution times between hw_v1 and hw_v2 is not due to the actual computations within the dynamic reconfigurable modules but from the read/write operations that need interfacing with the rest of the system.

4.4.1.1.3. Asynchronous Nature of Read/Write Operations

We present a summary of the factors that impact read/write time due to the asynchronous nature of these operations and then discuss how and why they impact read/write time.

- Handshake protocol.
- State changes happening just after the rising edge of the clock.

In our reconfigurable hardware designs, a read operation from the external memory is performed in two states of the FSM. The details of this FSM and its states can be found in

Appendix C. In the first state, user_ip requests a read operation and puts the address on the address bus, and waits for the master command acknowledgment signal from the PLB bus. Once it gets the command acknowledgment signal, it goes to the next state. In the second state, user_ip waits for the data to be ready on the read data bus. When the data is ready and once it gets the master transfer completion signal from the PLB bus, it goes to the next state to perform the next required operation and so on.

Similar to the read operation, a write operation to the external memory is also performed in two states of the FSM. In the first state, user_ip requests a write operation and puts the address on the address bus and the data on the write data bus, and waits for the master command acknowledgment signal from the PLB bus. Once it gets the command acknowledgment signal, it goes to the next state. In the second state, user_ip waits for the master transfer completion signal from the PLB bus to go to the next state to perform the next required operation and so on. Unlike read operation, with write operation, user_ip does not have to wait for the data to be written to the SDRAM. The SDRAM memory controller allows the write data to be pushed in before or after the address request.

In the read and write operations as described, state changes at the rising edge of the system clock. However, within the states, reading/writing involves handshake protocol, which is asynchronous. In some situations, completion of the read/write operations might happen just after the rising edge of the clock. In this case, it has to wait for the next rising edge to change the state. This might have a significant impact on the per read/write time.

Handshake protocol can also affect the data transfer time. For example, in case of read operation, the slave performs the required read operation by placing the data on the data bus. At the same time, data ready signal is asserted. If an extra delay is introduced by the interface circuitry before it places the data on the bus, the slave must delay the data ready signal accordingly, thus, affecting the data transfer time.

4.4.1.2. Techniques to Address Memory Access Latency

Throughout this chapter, it is observed that a significant amount of time, (93% - 98%) of the total execution time, is spent on data transfer to/from external memory. Hence, it is the major bottleneck in the system and worthy of further improvement.

Since data mining applications typically involve large volume of data, this considerable long data transfer time to/from external memory is highly significant and has an adverse effect on the overall performance. It is therefore necessary to address the memory access latency issue in FPGA reconfigurable hardware. We investigate techniques to reduce the time latency in accessing memory external to the FPGA. Several techniques can be used to ratify this problem and are detailed below.

The reconfigurable hardware implementation for similarity matrix computation has eight PEs working in parallel. Although they produce results simultaneously, reading the data from the DDR-SDRAM and writing the results back, however, are done sequentially. There is a possibility to integrate a multi-port DDR-SDRAM memory controller [181],[182] to read/write data/results from/to DDR-SDRAM simultaneously.

In addition, an external hard drive can be integrated into the development platform through the Rocket I/O interface available in Xilinx Virtex-6 and/or Virtex-II Pro. The Rocket I/O multi-gigabit interface can be used to transfer data at a full duplex rate of 2.488 Gb/s to 6.6 Gb/s [170],[190]. This would allow much faster data access than using the DDR-SDRAM through the PLB bus. Also, this would make it possible to store and execute very large volume of data (hundreds of gigabytes) that typically exist in many data mining applications. The current setup of the development platform can only accommodate data sets up to 512 MB.

Applications are typically either I/O bound or compute bound. An application is I/O bound, if the number of I/O operations is greater than or equal to the number of computations in the application [77]. In this case, in order to enhance the speed performance, we have to increase the bandwidth to the memory instead of performing more computations in parallel.

There are typically three levels of memory hierarchy on FPGAs [77]:

- On-chip memory (BlockRAM) – fast and wide data-width
- On-board memory (SDRAM) – fast but limited data-width
- On-host system memory – slow memory located in the host system (PC)

An enormous amount of data needs to be processed in data mining applications. Since on-chip memories are not sufficient to store the entire data set, it is usually stored in off-chip memory. Memory access latency can be reduced by buffering data between off-chip

memory and on-chip memory during execution. Also, by having separate input/output ports in the on-board memory, we can simultaneously read/write to the external off-chip memory.

Each data item transferred between the memory and the chip directly impacts the overall speed performance of the application. Bandwidth between the FPGA and the off-chip memory is limited. Hence, by carefully organizing the memory resources and designing the memory interface, we can fully utilize the available memory bandwidth in order to enhance the speed performance of an application.

4.5. Chapter Conclusion and Discussion

We studied different facets of reconfigurable computing. This study illustrates that the advantages of reconfigurable computing systems, specifically, of FPGA-based reconfigurable hardware, including flexibility, upgradeability, compact circuits and area efficiency, shorter time-to-market, and relatively low cost, have opened up new avenues for reconfigurable computing. These advantages are especially important for providing chip-level hardware support for applications, such as data mining, on portable, handheld, and embedded computing.

We introduced a reconfigurable hardware solution using multiplexer-based approach for similarity matrix computation, which can be dynamically reconfigured to accommodate three similarity measures. This design shows a significant space saving, since most of the modules for all three similarity measures are reused. Also, extra hardware required for reconfiguration is relatively insignificant, and the reconfiguration time overhead is practically zero.

We also introduced a reconfigurable hardware solution for PCA using partial reconfiguration method, which can be partially and dynamically reconfigured from Mean to Covariance and vice versa. This design also shows a significant space saving, since the same area of the chip is being reused for both the Mean and Covariance computation. The extra hardware required for reconfiguration is relatively low compared to the whole chip and remains constant regardless of the size of the reconfiguration module. Considering the reconfiguration time overhead, there is a difference between the theoretical estimate and the experimental value. This is mainly because we used a MicroBlaze processor as a

configuration controller, which executes instruction sequentially. We might be able to get similar values as the theoretical ones, by using a FSM as the configuration controller and downloading the configuration bitstream using “bit-parallel” mode.

From the results and analysis, it is observed that a significant amount of time (93% - 98%) is spent on data transfer to/from memory. Since data mining applications typically have a large amount of data, this considerably long memory access time is highly significant and certainly has an adverse effect on the overall performance. We proposed techniques to reduce the time latency in accessing memory external to the FPGA as in 4.4.1.2.

It is also observed that microprocessors, unlike reconfigurable hardware, spend significant amount of time (83% - 88%) on actual computation. Since processor has the additional overhead of fetching each instruction from the memory, the actual computation is not efficient relative to the frequent memory access. Considering the actual computation time, speedup for reconfigurable hardware designs for similarity matrix is 469, and for Mean and Covariance are 95 and 59 respectively. If the memory access latency can be reduced, reconfigurable hardware designs should be able to achieve much better speed performance over software implementations. It should be noted that for the partial and dynamic reconfigurable hardware designs (for Mean and Covariance), no optimization was done and data were not streamed in. Better performance is expected if these design aspects are refined.

From the experiments, for all the partial and dynamic reconfigurable hardware designs, including Mean/Covariance, and Addition/ Multiplication, it is observed:

- Our reconfigurable hardware scales linearly with data size.
- Reconfiguration time does not vary with the size of the data set.
- Percentage of reconfiguration time is amortized and decreases as the size of the data increases.

These experimental results are promising, since data mining typically involves processing large data sets, which renders the reconfiguration time insignificant.

From the additional experiments carried out to investigate the discrepancy of the execution times between hw_v1 (SRH) and hw_v2 (DRH), it is observed:

- Per read/write time varies with different number of consecutive reads/writes within the same hardware versions. This is due to asynchronous nature of these operations, state changing before the operation is complete, refresh operations, and traffic patterns.
- Per read/write time varies with different number of consecutive reads/writes between two hardware versions. This is due to different design flows, and interfacings to the rest of the system are different.
- Per additions/multiplications time does not vary with different number of consecutive additions/multiplications within the same hardware versions or between the two versions. This is expected since these operations are designed to execute in one clock cycle, and synchronous with the system clock.

It can be concluded that the time discrepancy between hw_v1 and hw_v2 are not due to the actual computations within the reconfigurable module but is from the read/write operations that need interfacing with the rest of the system. We could minimize the impact of additional overhead associated with read/write operations in reconfigurable hardware, by minimizing the interface to the rest of the system, and keeping intermediate results and frequently used data in registers within the same hardware module, if the input data is being reused several times keeping the data once read on the same hardware module for subsequent use. In addition, since read/write usually takes one clock cycle if data is on chip, read/write overhead is typically lower if data is kept on chip instead of in off-chip memory.

The results shown in our experiments are encouraging and show great potential in implementing data mining applications using reconfigurable platform. Complex processing can indeed be implemented in reconfigurable hardware for embedded and portable applications. In the next chapter, we propose a design methodology for FPGA-based dynamic reconfigurable hardware.

Parts of the work in this chapter have been published in:

- IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC'09) [127];
- IEEE Pacific Rim International Conference on Communication, Computers and Signal Processing (PacRim'11) [128];

- IEEE International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC'11) [129] **(Best Paper Award)**.
- Springer LNCS Transactions on Computational Collective Intelligence, 2012, [105].

Chapter 5

5. Design Methodology for FPGA-Based Dynamic Reconfigurable Hardware

In previous chapters, we progressively analyzed, discussed and presented the advantages and disadvantages of designing and implementing specific applications with respect to hardware versus software, reconfigurable versus non-reconfigurable hardware, and FPGA-based versus non FPGA-based reconfigurable hardware. These analyses illustrated that FPGA-based reconfigurable hardware provides a flexible and area efficient computing platform under the constraints associated with portable and embedded devices. Multiple applications can be processed on a single chip, by dynamically reconfiguring the chip from one application to another as needed. In this Chapter, our objective is to formulate a design methodology for FPGA-based dynamic reconfigurable hardware in order to select the most appropriate reconfiguration method(s) to use in different scenarios. The computation models, application characteristics, and size of the operations are considered. This methodology is based on theoretical analysis as well as our experience from experimental work on FPGA-based dynamic reconfigurable hardware.

In 5.1, we discuss and present how and why certain computation models and applications with certain characteristics could potentially benefit from FPGA-based dynamic reconfigurable hardware. In 5.2, we analyze, discuss, and present the features, advantages and disadvantages of existing FPGA-based reconfiguration methods. In 5.3, we present how we map the computation models and characteristics of applications to the most efficient reconfiguration method(s). The following definitions are used throughout this Chapter.

- Computation – is a process of performing a certain operation.
- Computation models – are various types of processing such as: parallel (functional), pipeline, parallel (data), and computations with many identical functions.
- Operations or computation modules – are the functions (or tasks) in a computation model.
- Functional units – are the components, where operations are executed.

- Stage – is a distinct part of a computation process with identifiable inputs and outputs.
- Processing Elements – are hardware circuitry used to execute operations autonomously.
- Reconfigurable module – is a functional unit or a PE that is reconfigurable.

5.1. Computation Models and Application Characteristics

From the discussions and analyses in the previous Chapter, it is evident that FPGA-based reconfigurable hardware has numerous advantages that are important for portable and embedded devices. However, there are still some challenging questions that need to be answered in the design process. Some of these questions are: what are the tradeoffs associated, particularly in terms of design flexibility, speed performance, and chip area usage; and whether FPGA-based reconfigurable hardware is indeed a good match for a specific application on portable and embedded devices. Throughout this section, we aim to answer the latter by considering the computation models and characteristics of the applications.

5.1.1. Computation Models Suitable for Reconfigurable FPGAs

There exist many computation models. In an application, operations are executed either sequentially or in parallel [142]. In general, parallel execution is accomplished with one of the following two orthogonal concepts: by using multiple functional units in a non-pipelined or pipelined fashion [142].

The following computation models are presented as they are deemed to be suitable for and could potentially benefit from FPGA-based dynamic reconfigurable hardware:

- Parallel (functional) computations
- Parallel (data) computations
- Pipelined computations
- Computations with many identical sub-functions or sub-tasks

5.1.1.1. Parallel (Functional)

Parallel execution involves multiple independent operations using multiple processing elements (PEs) or functional units at the same time. In this case, we consider functional parallelism, where multiple independent operations are executed simultaneously.

Functional parallelism can be utilized at different levels of granularity, that is, from fine-grained to coarse-grained. This is similar to the notion of Multiple Instruction Multiple Data (MIMD), where multiple autonomous PEs simultaneously execute different operations on different data [142].

Functional parallel computation model can greatly benefit from dynamic reconfigurability of FPGAs. Assuming a large number of multiple independent operations that can be executed in parallel, a single chip might only be able to accommodate a limited number of these independent operations running in parallel at a time. After these operations are executed, then the chip can be reconfigured on-the-fly to the next required independent operations running in parallel, and so on. Dynamic reconfigurability of FPGA allows executing a large number of multiple independent operations on a single chip in parallel, regardless of them fitting into the chip at a time. In addition, with current state-of-the-art FPGAs, we can reconfigure parts of the chip that require modification while others parts are still operational [39],[77],[112]. Functional parallel computation model can further benefit from this feature. Assuming there are several independent operations executing on the chip in parallel at the same time. If one or more independent operations finish processing, those can be reconfigured on-the-fly to the next required operations, while others are still being executed in parallel, thus overlapping computations with reconfiguration. This conceals the reconfiguration time overhead. Also, more parallel operations can be executed on a single chip in less time, since the processed operations do not have to wait for the other operations to finish. This is an important feature for applications where speed performance is an issue.

5.1.1.2. Parallel (Data)

In data parallelism, the same operation is executed on several processing elements with different data in parallel. Data parallelism arises in a variety of application fields. As a result, both the data structures and the operations performed may differ from one application to another [142]. In all cases, the data structures (such as vectors and matrices) are processed in parallel.

Data parallel computation model can benefit from the dynamic reconfigurability of FPGAs. For instance, multiple applications or a single application might require different data parallel architectures such as Single Instruction Multiple Data (SIMD), Systolic,

Vectorizing, Associative Process, etc. The chip can be reconfigured from one data parallel architecture to another as needed on-the-fly according to the requirement of the application. As a result, different applications or a single application with different data parallel architectures can be executed on a single chip, regardless of them fitting into the chip at a time. Another important aspect of data parallel model is the connectivity between the PEs. With FPGA-based reconfigurable hardware, the interconnection network can be reprogrammed to have different connectivity such as near-neighbour, tree, pyramid, hypercube, etc.

5.1.1.3. Pipelined

"The term 'pipeline' refers to the temporal overlapping of processing" [142]. Pipelining is often used in complex computations to maximize throughput. In pipelining, several sub-functions of a single operation (or several dependent operations) are executed in a pipelined fashion. In this case, a number of successive functional units are utilized to perform a single operation or multiple dependent operations [142]. Typically, a single operation is divided into a number of successive stages (or sub-tasks), where each stage (or sub-task) is typically executed on a single functional unit. For multiple dependent operations, each dependent operation is also executed on a single functional unit. Each input data item usually goes through the whole pipeline stage by stage**.** The pipeline stages perform like an assembly line, i.e., inputs are received from the previous stage and outputs are transferred to the subsequent stage [79],[142]. The significant features of pipelined execution are: each stage can begin processing a new data item before the successive stages finish processing prior data items, and the execution time for each pipeline stage is designed to be the same in order to achieve maximum throughput with a synchronous clock. Pipeline improves throughput of the data rather than the execution time of individual data [79].

Pipeline computation model can significantly benefit from the partial dynamic reconfigurability of FPGAs, which allows reconfiguring part of the chip that requires modification while others parts are still operational and interfacing with the rest of the system. Assuming a large number of multiple dependent operations that can be executed in pipelined fashion, a single chip might only be able to accommodate a few dependent operations at a time. Considering a scenario where a chip can accommodate only three

pipeline stages at a time. In this case, after the first stage (with the first dependent operation) finishes processing data set1, the second stage (with the second dependent operation) starts processing data set1. The first and second stages are processing data set2 and set1 respectively. Next, the first stage is being reconfigured on-the-fly to the next required pipeline stage (in this case the fourth stage with the fourth dependent operation), while the second stage and third stage (with the third dependent operation) are processing data set2 and set1 respectively. The reconfiguration is overlapped with computation. Partial dynamic reconfigurability of FPGA allows executing a large number of multiple dependent operations on a single chip in a pipelined fashion, regardless of them fitting into the chip, by reconfiguring some pipeline stages while interfacing with the remaining stages that are operational.

In pipelining, if the chip can only accommodate two pipeline stages at a time, we have to execute these pipeline stages sequentially (pipelined example in 5.3.1.2.1). As a result, we could not increase throughput using pipelining. However, if the chip is able to accommodate three or more pipeline stages, then we can execute them in a pipelined fashion (pipelined example in 5.3.1.2.2) to increase throughput. Consider a scenario where 3 out of 4 pipeline stages fit into the chip at a time (in 5.3.1.2.2). In order to process 4 sets of data, the total processing time using partial reconfiguration is 9T, where T is the time taken to process one set of data by each pipeline stage. If the reconfiguration time is less than or equal to T, maximum benefit is harnessed with a pipeline design on a single chip. In this case, the reconfiguration is completely overlapped with the computation, concealing the reconfiguration time overhead [92]. If the reconfiguration time is more than T, the data processing is delayed till the next required pipeline stage is available. In case of the normal pipeline, the total processing time to process 4 sets of data would be 7T. If the 4 sets of data are processed sequentially, the total processing time would be 16T. Although partial reconfiguration increases throughput by executing dependent operations in pipelined fashion, the increase in throughput is less than that of the normal pipeline.

5.1.1.4. Computations with Many Identical Sub-Functions or Sub-Tasks

Some computations have many identical sub-functions among them. For instance, many identical sub-functions exist among the three similarity measures introduced in 3.3: Cosine Similarity, Extended Jaccard, and Asymmetric Measure.

This computation model can greatly benefit from the partial dynamic reconfigurability of the FPGAs, which allows reconfiguring parts of the chip that requires modification while others parts remain intact or are still operational and interfacing with the rest of the system. Assuming there are some operations that have a large number of identical sub-functions among each other. Also assuming only one operation as a separate entity (i.e., as a single PE) can fit into the chip at a time. Initially, the first operation is downloaded to the chip. After processing the first operation, the chip is partially and dynamically reconfigured to the next operation. During the reconfiguration from the first operation to the second operation, the identical sub-functions that exist among these two operations remain intact and only the rest of the sub-functions are reconfigured to their appropriate hardware circuitry. Reconfiguring only parts of the chip as opposed to reconfiguring the entire chip, reduces reconfiguration time overhead. This is important to applications that are sensitive to the reconfiguration time overhead, even though this time is relatively short. Also in some cases, with partial and dynamic reconfigurability, earlier stages of the first operation can be reconfigured to the sub-functions of the second operation while the later stages are still working on the first operation. This overlapping of reconfiguration and computation conceals the reconfiguration time overhead.

5.1.1.5. Benefits to Computation Models

The computation models presented indeed benefit from FPGA-based reconfigurable hardware. The dynamic reconfigurability of FPGA allows the execution of: a large number of independent operations (functional parallel model); large number of dependent operations (pipeline model); different architectures (data parallel model); regardless of them fitting into the chip at the same time. These computation models as well as the computation model with many identical sub-functions among each other could further benefit from the partial dynamic reconfigurability of FPGA, which allows reconfiguring some parts of the chip while other parts are operational.

5.1.2. Application Characteristics Suitable for Reconfigurable FPGAs

In this section, we discuss and present how and why applications with the following characteristics could potentially benefit from FPGA-based dynamic reconfigurable hardware:

- Multi-stage and lengthy computation process
- Various methods to carry out an operation
- Dynamic decision making and changing the operations dynamically
- Evolving algorithms and new and emerging algorithms
- Adapt to different standards and operating conditions

5.1.2.1. Multi-Stage and Lengthy Processing

Many applications often consist of multi-stage and lengthy computation process. For example, clustering and classification in data mining applications consist of several stages: feature selection, similarity measure, grouping, and data abstraction. When the multi-stage applications have large and complex circuitry, it might not be possible to squeeze all the stages into a single chip. In this case, the application has to be decomposed into several sub-components (perhaps stage by stage), so that it can fit into the chip at different times, by reconfiguring the chip from one stage to another, using FPGA-based reconfigurable hardware. Flexibility and reusability features of FPGAs allow multi-stage, large, and complex applications to be executed on a single chip, regardless of them fitting into the chip. Moreover, the on-chip programmable hardware can be reconfigured dynamically post fabrication in real-time fashion, which enables the reuse of the same silicon multiple times. This is the preferred way of executing such large and lengthy applications, particularly in portable and embedded devices with their limited hardware foot-print.

5.1.2.2. Various Methods to Carry Out an Operation

Some applications use different algorithms in different situations. For instance, there exist a large number of algorithms to process clustering and classification. These algorithms consist of several stages, where each operation of a stage itself is usually solvable by various methods, though with results of different quality. For instance, there are many ways to measure similarity such as Cosine Similarity, Extended Jaccard, and Asymmetric Measure. They often produce different results in dissimilar application contexts. When applications use different algorithms for different situations, a single chip might not be able to accommodate all the algorithms. Also, when there are various methods to carry out an operation of a stage, all these methods might not be able to fit into the chip simultaneously. One may try a different algorithm to get better results. The chip can be

reconfigured as needed on-the-fly from one algorithm to another using FPGA-based reconfigurable hardware. Consequently, we can use different operations in a plug and play fashion. By combining different operations in various combinations, we can build many algorithms with different properties, and process them on a single FPGA.

5.1.2.3. Dynamic Decision Making and Changing Operations Dynamically

Many applications involve dynamic decision making and changing the operations dynamically. For instance, the operation to be performed next is not known in advance, so one must consider the current objective and other criteria such as recent results obtained and the external stimuli, in order to determine the next actions without human intervention. In some cases, selecting the next operation to process the data as well as changing the operations as needed on-the-fly, require some parts of the chip being modified while the rest of the system is operational. This can be facilitated by current FPGA-based reconfigurable hardware, which allows full or partial dynamic reconfiguration of the chip.

5.1.2.4. Evolving Algorithms and New and Emerging Algorithms

Many applications have evolving algorithms, and new and emerging algorithms are continuously developed for them. In addition, some applications might require post-design optimizations and upgrades. The programmable and reprogrammable nature of FPGA-based reconfigurable hardware allows modifying the existing operations and adding new operations dynamically, even after deployment. The flexibility (re/programmability) of FPGAs also allows incremental design flow, even after the manufacturing process. As a result, applications can be modified with recent optimizations and in-field upgrades as desired, later in their life. All these modifications and upgrades can be done by reprogramming the chip at anytime, i.e., dynamically during the run-time of the applications or after a period of time (for instance, every three to four months).

5.1.2.5. Adapt to Different Standards and Operating Conditions

Some computationally intensive applications [111], such as wireless communications require adaptation to different and evolving standards and operating conditions. For instance, consider the software-defined radio application in [14],[38], which supports

multiple communication standards, but mostly one at a time. This application uses FPGA-based reconfigurable hardware to dynamically change the communication standard, so that it can operate in a different frequency range using a different protocol. The on-chip hardware circuitry is automatically reconfigured either at the beginning of a call or a data transfer session (to select the most efficient protocol at the moment), or during a call or a data transfer session (to transfer from one protocol to another) [38]. This is one of the first commercially available applications that used partial dynamic reconfigurability of FPGA. Post-fabrication re/programmability of FPGA-based reconfigurable hardware allows adapting to different standards and operating conditions.

5.1.2.6. Benefits to Applications

In summary, applications that require some form of flexibility, reusability, extensibility, upgradeability, etc., can indeed benefit from FPGA-based reconfigurable hardware. Its post-fabrication re/programmability features are beneficial for the applications that require some form of reprogrammability (either dynamically in real-time fashion or in periods of time such as tri-monthly), which could lead to increase in reuse, and reducing device cost or gate count.

5.2. Features, Advantages, and Disadvantages of FPGA-Based Reconfiguration Methods

In this section, we analyze, discuss, and present the features, advantage, and disadvantages of FPGA-based reconfiguration methods. Existing FPGA-based reconfiguration methods include single context, multi context, partial reconfiguration, and MultiBoot. These methods allow the dynamic re/configuration of the hardware on chip to perform a variety of operations during the run-time life of an application without human intervention. The following terminologies are used within the context of reconfiguration methods.

- **Configuration bitstream:** Also known as a context, is a binary file that sets all of the FPGA's programmable bit locations to configure the logic blocks and routing resources. Initially the configuration bitstreams are stored in an external non-volatile memory, and then downloaded to the chip and used to configure the chip's hardware circuitry.

- **Inactive context:** After a configuration bitstream is downloaded and stored in on-chip memory, it is called an inactive context. These inactive contexts can be kept on-chip and used repeatedly to reconfigure the chip as needed.
- **Active context:** After a configuration bitstream is downloaded to the chip and used to re/configure the chip, it is called an active context. Active context does not necessarily mean that it is running.

5.2.1. Single Context

Traditionally, most of the applications running on an FPGA use single context, which allows only one full configuration bitstream for the entire chip to be downloaded to the chip at a time [39],[77]. With single context, a full configuration bitstream is downloaded to all the programmable bit locations of the chip at a time and the entire chip is configured to the appropriate hardware circuitry. As a result, the entire chip has to be reprogrammed even for the smallest changes in the design. This method does not require extra hardware on chip for reconfiguration, thus all the resources on chip can be utilized for logic and routing of the design. Although this simplifies the reconfiguration hardware, it incurs higher reconfiguration time overhead when only a small part of the design needs to be changed [39]. The reconfiguration time overhead is the time required to download and change the configuration.

Single context requires an external controller [120] to control the configuration flow, since the controller performing the re/configuration (deciding which configuration bitstream to be downloaded next) has to be active during the whole process. An external controller, whose primary purpose might be to perform other tasks, can be used to coordinate the downloading of the configuration bitstream into the programmable bit locations on the chip [120]. After the execution of one application with one configuration bitstream, the external controller determines which of several configuration bitstreams (stored in the external memory) should be downloaded next and downloads the next configuration bitstream and reconfigures the chip. A disadvantage of external downloading and reconfiguring is that it incurs higher reconfiguration time overhead, which can be milliseconds [171] or more, compared to having configuration bitstreams stored on chip.

5.2.2. Multi Context

Unlike single context, multi context allows multiple independent configuration bitstreams, some as active contexts and some as inactive contexts, to be downloaded initially to the chip simultaneously [158]. Hence, multi context allows more than one active context to be executed in parallel on the same chip, and also allows several inactive contexts to be stored in on-chip configuration memory, which are the major advantages of this method.

In this case, the on-chip configuration memory (e.g., with *n* inactive contexts) is distributed around the chip, such that each configuration memory cell, associated with a programmable bit location, has *n* memory bits of storage for each of the *n* inactive context [36],[158]. If the next required inactive context(s) is (are) in configuration memory, the entire chip can be reconfigured in a single cycle [77],[136]. Therefore, in contrast to the conventional single context method, multi context can hold several different inactive contexts in on-chip configuration memory and can switch between the contexts on demand quickly, since no access to external memory is required. Hence, another compelling advantage of multi context is that it allows extremely fast reconfiguration within an order of nanoseconds [57],[136].

However, the ability to perform a reconfiguration of the full chip in a single cycle can be a disadvantage. In this case, all the programmable bit locations are loaded from the configuration memory simultaneously, and potentially the majority of the programmable bit locations may be changed from 0 to 1 or vice versa [77]. Switching the configuration bits in many locations in a single cycle, increases dynamic power, which could violate system power constraints [77],[158]. This is a sensitive issue for embedded systems.

Another drawback of multi context is that it requires extra hardware on-chip for reconfiguration and for on-chip memory to hold inactive contexts, which occupy valuable area of the chip that could otherwise be used for logic and routing [158]. Therefore, the active area (for logic and routing of the design) of a chip using multi context is less than that of a chip using single context of the same area [48],[77]. For instance, a chip using multi context holding four inactive contexts (one per entire chip) in configuration memory has only 80 percent of the active area (for logic and routing) of that using single context [48].

Similar to single context, multi context also requires an external controller to control the configuration flow [120]. However, if there is a possibility to download and hold all the required contexts (active and inactive) on chip with the initial downloading, we can eliminate the need to interface with an external controller. In this case, when computation of one active context is near completion, the active context itself can trigger the on-chip configuration memory to reconfigure that part of the chip, which occupied by the active context, with the next required context [141]. This allows self-reconfiguration, which is another advantage of the multi context method.

In multi context, the external controller can also be used to pre-fetch configuration bitstreams to the chip from external memory [57]. Hence, another benefit of the multi context method is that it allows background loading of configuration bitstreams while the active contexts are still operational [48],[136],[158]. In addition, multi context allows reconfiguring parts of the chip, which require modification, while the rest of the active contexts are still operational, overlapping computation with reconfiguration [39],[77]. Pre-fetching and reconfiguring parts of the chip, potentially has the benefit of hiding some of the reconfiguration time overhead.

With multi context, it is possible to communicate signals between the current context and the next context (after reconfiguration), using registers [31],[108]. However, proper connections to the registers are not guaranteed after reconfiguration, since electrical signals of the registers might be affected during reconfiguration, which can cause glitches or loss of information [113].

5.2.3. Partial Reconfiguration

In some cases, only some parts of the design require modification. Partial reconfiguration, as indicated by its name, allows reconfiguring a portion of a chip that requires modification [39],[77],[112]. This has two advantages. One is that, similar to multi context, some parts of the chip are reconfigured while the remaining parts are in operation, thus overlapping computation with reconfiguration. Another is that, the reconfiguration can be carried out while interfacing with the remaining parts (of the design) that are operational.

With partial reconfiguration, first, the FPGA is configured by loading an initial full configuration bitstream for the entire chip upon power-up [53],[183]. After the FPGA is fully configured and operational, multiple partial bitstreams can be downloaded simultaneously to the chip, and specific regions of the chip can be reprogrammed with new functionality "without compromising the integrity of the applications" running in the remainder of the chip [53],[183]. Partial bitstreams are used to reconfigure only selective parts of the chip. During the design and implementation process, the logic in the reconfigurable hardware design is divided into two different parts: reconfigurable and static. The reconfigurable parts are replaced by the contents of the partial bitstreams, while the current static parts remain operational, completely unaffected by the reconfiguration [183].

In the late 2010s, partial reconfiguration tools used Bus Macros [106],[137], which ensures fixed routing resources for signals used as communication paths for reconfigurable parts, and when the parts are reconfigured [137]. With the new PlanAhead [184] tools for partial reconfiguration, Bus Macros become obsolete. Current FPGAs (such as Virtex-6) have an important feature: a "non-glitching" (or "glitchless") technology [53],[191]. Due to this feature, some static parts of the design could be in the reconfigurable regions without being affected by the act of reconfiguration itself, while the functionality of reconfigurable parts of the design is reconfigured [53]. For instance, when we partition a specific region and consider it as a reconfigurable part, some static interfacing might go through the reconfigurable part or some static logic (e.g., control logic) might exist in the partitioned region. These are overwritten with the exact program information, without affecting their functionalities [53],[54].

Since partial reconfiguration allows us to change some parts of the design without reconfiguring the entire chip, another benefit of partial reconfiguration is that it reduces the total reconfiguration time. The reconfiguration time is directly proportional to the configuration bitstream length (size of the configuration data) [93],[112]. Hence, by changing only portion of the configuration bitstream, as opposed to reconfiguring the entire chip, the total reconfiguration time is reduced to microseconds or more [92].

Another advantage of partial reconfiguration is that, after the initial downloading and configuration of the chip, it is possible to use an internal controller to download the next

required configuration bitstreams [53],[183],[184]. Internal controller is either a microprocessor or routines (small state machine) programmed into the FPGA [53],[183]. Furthermore, the internal controller is often used to control the configuration flow, allowing self-reconfiguration. This eliminates the need to interface with an external controller. Apart from self-reconfiguration, having an internal controller allows quick decision making, and less time (2-3 cycles) to communicate with other peripherals as opposed to using an external controller (20-30 cycles). However, the internal controller must be kept as a static part of the design and communicate with the reconfigurable parts. This will guarantee the integrity of the control circuits during and after reconfiguration [31].

Similar to multi context, with partial reconfiguration it is also possible to pre-fetch the configuration bitstreams and to store them in on-chip Block Random Access Memory (BRAM), while the circuits are operational [21]. Hence, another benefit of partial reconfiguration is that it allows background loading of configuration bitstreams from the external memory while the circuits are still operational. This also has the benefit of hiding some of the reconfiguration time overhead, which includes downloading time and reconfiguration time. However, partial reconfiguration requires extra hardware for reconfiguration and for on-chip BRAM to hold the inactive contexts, which can be a drawback [31],[53],[183].

In the late 2010s, for partial reconfiguration, the extra hardware required on chip for reconfiguration was quite significant, due to Bus Macros. However, since Bus Macros have become obsolete, partial reconfiguration requires less extra hardware on chip for reconfiguration. Especially, if the full and partial bitstreams are stored in an external non-volatile memory, the only hardware required on chip for reconfiguration is the Internal Configuration Access Port (ICAP) and the Interface controller to the external memory. For instance, if the memory is a Compact Flash (CF), then we can use the SystemAce Interface Controller [177]. On Virtex 6 chip, resource utilizations for ICAP [193] and the SystemAce Interface Controller are about 460 and 78 slices respectively, resulting in a total of 538 slices. In this case, the internal controller can be a state machine, which is typically quite small and takes very small space on chip. Considering the total slices,

37680, on the Virtex 6 (XC6VLX240T-1FFG1156) chip, extra hardware for partial reconfiguration occupies 1.43% of the whole chip.

5.2.4. MultiBoot

Similar to single context, MultiBoot is another reconfiguration method that requires full reconfiguration of the chip [82]. MultiBoot enables reconfiguration of the chip with different full configuration bitstreams stored in an external non-volatile memory [82],[83]. First, the chip is configured by the initial full bitstream from the memory, then the application running on the chip can trigger a MultiBoot event and reconfigures itself from different full bitstreams housed in the external memory [82].

Unlike single context, with MultiBoot, reconfiguration is typically done using an internal configuration controller. In this case, the user application on the chip incorporates a MultiBoot control module, which includes a small state machine (internal to the FPGA) and the ICAP [11],[166]. One advantage of having an internal controller is that it allows self-reconfiguration. Another advantage is that it helps quick decision making, which takes less time (2-3 cycles) to communicate with other peripherals as opposed to using an external controller (20-30 cycles). For instance, when deciding on the configuration bitstream to be loaded next, the decision is done on-chip and then the internal MultiBoot controller fetches the bitstream from external memory.

During reconfiguration, the control logic on chip, i.e., MultiBoot control module and the interface controller to the external memory, remains intact while the rest of chip is being reconfigured [82],[166]. Unlike partial reconfiguration, with MultiBoot, parts of the chip can not be selectively reconfigured. As a result, the entire chip (except control logic) has to be reprogrammed even for the smallest changes in the design. Also, external loading and reconfiguring the entire chip incurs higher reconfiguration time overhead, which can be milliseconds or more [82], as compared to reconfiguring only parts of the chip.

With MultiBoot, the only extra hardware required on chip for reconfiguration is the MultiBoot controller (state machine and ICAP) and the interface controller to the external memory. For example, if the memory is a Platform Flash PROM, then we can use the XPS multi-channel external memory controller [192]. On Virtex 6 chip, resource utilizations

for ICAP [177] and the memory controller are about 460 and 701 slices respectively, resulting in a total of 1161 slices. In this case, the state machine is quite small and takes very small space on chip. Considering the total 37680 slices on the Virtex 6 (XC6VLX240T-1FFG1156) chip, the extra hardware for MultiBoot reconfiguration occupies 3.08% of the chip.

5.2.5. Analysis on Reconfiguration Time and Space Overhead

We analyze and discuss the reconfiguration time and space overheads for different FPGA-based reconfiguration methods. Reconfiguration time and space overheads are the extra time required for reconfiguration and extra hardware required on chip for reconfiguration, respectively.

5.2.5.1. Reconfiguration Time Overhead

The reconfiguration time overhead is the time required to load and change the configuration. This has to be done whenever we want to change the application or the functionality of the hardware. Especially for dynamic reconfigurable hardware, it is necessary to ensure that the advantages of hardware acceleration are not overshadowed by the reconfiguration time overhead [39].

Reconfiguration takes a specific time, determined by the FPGA device type, bitstream size, and clock speed, and independent of the pattern (corresponding to a specific design) of the configuration bitstream [5].

Usually, single context designs take from tens to hundreds of milliseconds to reconfigure the hardware. Single context typically uses serial interface for downloading and reconfiguration. In this case, the reconfiguration time is estimated using the full configuration bitstream and the configuration clock frequency [171], as in equation (16) below. With multi context, the reconfiguration from one context to another can be done in a single cycle [77], if the next required context is in the on-chip configuration memory. Hence, the reconfiguration time is equal to the configuration clock period. Partial reconfiguration is done through the ICAP; hence the reconfiguration time depends on the ICAP throughput and the size of the partial configuration bitstream, as in equation (17) below. According to the partial reconfiguration user guide [183], for Virtex chips, using an ICAP (100MHz and 3.2Gbps) a partial bitstream of 1358208 bits can be loaded and

reconfigured in about 1358208 bits / 3200000000 bps = 424 microseconds [129]. However, depending on the reconfiguration area, partial reconfiguration might take several milliseconds to reconfigure when the bitstream is in the order of ten-millions. With MultiBoot, the reconfiguration time [11] is estimated using the full configuration bitstream, configuration clock frequency, and the bus width of the external non-volatile memory interface, as in equation (18) below. If the external memory is Platform Flash PROM, then the reconfiguration is done using a parallel programming mode such as the SelectMap protocol [82]. The following terms are used for the three equations below.

- *TotalNoOfConfigurationBits*: is the total number of configuration bits (corresponding to a full bitstream for single context and MultiBoot) for the entire chip, which is a constant for a specific chip.
- *NoOfConfigurationBits*: is the number of configuration bits for a part of the chip (corresponding to a partial bitstream for partial configuration), which varies with the size of the reconfigurable part.

$$ReconfigurationTime(serial) = TotalNoOfConfigurationBits * ConfigurationClkPeriod \quad (16)$$

$$ReconfigurationTime(ICAP) = NoOfConfigurationBits/ICAPThroughput \quad (17)$$

$$ReconfigurationTime(Parallel) = TotalNoOfConfigurationBits/(ConfigurationClkFrequency * BusWidth) \quad (18)$$

Xilinx Virtex-6 (XC6VLX240T) has 73859552 configuration bits [191]. With a configuration clock frequency of 100MHz, the reconfiguration time overhead to reconfigure (downloading and changing) the entire chip for single context, partial reconfiguration, and MultiBoot using the above equations are 739 ms, 23 ms, and 23 ms, respectively. In this case, for comparison purpose, we consider the worst case scenario for partial reconfiguration, where the entire chip is being reconfigured. These times are constant for single context and MultiBoot for a specific chip, whereas for partial reconfiguration, this time varies with the size of the reconfigurable module. For multi context, the reconfiguration can be done in a single cycle. For instance, with multi context

on Xilinx XC4000E, the entire configuration takes 30ns [158]. In this case, the reconfiguration time remains the same (typically a single cycle) regardless of the size of the area being reconfigured. The reconfiguration for partial and MultiBoot are done using a parallel interface (both 32 bit), whereas single context is done using serial interface. Clearly, the reconfiguration time overhead using parallel mode is much lower (1/32) than that using the serial mode. With some FPGAs, it might be possible to use parallel mode for single context; thus, the reconfiguration time overhead for these three reconfiguration methods would be the same (23 ms), when reconfiguring the entire chip. However, for the applications that process large volume of data, where processing time is significant, the reconfiguration time overhead of few milliseconds might not be an issue.

5.2.5.2. Reconfiguration Space Overhead

The reconfiguration space overhead is the extra hardware required on chip for reconfiguration and the on-chip memory required to hold inactive contexts. For some reconfiguration methods, the reconfiguration space overhead is unavoidable.

Single context designs do not require extra hardware on chip for reconfiguration, utilizing all the resources on chip (100 percent) for logic and routing of the design. With multi context, it requires extra hardware [158] on chip for reconfiguration and for on-chip memory to hold inactive contexts. In [48], a chip using multi context storing four inactive contexts in configuration memory has only 80 percent of the active area (for logic and routing of the design) of that using single context. Hence, in order to hold one inactive context (that would take up the space for the entire chip) in the configuration memory, the extra hardware required is 5% of the chip [48]. Although, the percentage of area occupied by an inactive context might change from one chip to another, in general, the extra hardware required for on-chip memory increases with the number of inactive contexts. For both partial reconfiguration and MultiBoot, the extra hardware required on chip for reconfiguration is ICAP [177] and the interface controller [193] to the external non-volatile memory. Considering the total slices, 37680, on the Virtex 6 (XC6VLX240T) chip, the extra hardware for partial reconfiguration, and MultiBoot are 1.43%, and 3.08% of the chip, respectively.

Although, at a glance, reconfiguration space overhead seems like a major drawback, since it occupies valuable real estate of the chip that could otherwise be used for logic and

routing of the design, the benefits can far outweigh the drawbacks. In many cases, having reconfiguration circuitry on chip and having on-chip memory to hold the inactive contexts are essential to improve the reconfiguration process, particularly when the speed is an issue as in dynamic reconfiguration. These systems trade chip area for higher speed performance.

Features	**Reconfiguration Methods**			
	Single context	**Multi context**	**Partial reconfiguration**	**MultiBoot**
When downloading and storing configuration bitstreams:				
1. Download multiple configuration bitstreams to the chip simultaneously	Not possible [39],[77]	Possible [158]	Possible [53],[183]	Not possible [82]
2. Storage of configuration bitstream in on-chip memory	Not possible [39],[77]	Possible [36],[158]	Possible [31],[53],[183]	Not possible [82],[83]
3. Background loading (pre-fetching) of configuration bitstreams	Not possible [39],[77]	Possible [48],[57],[136]	Possible [21]	Not possible [82],[83]
4. Internal controller to download the configuration bitstream and to control the configuration flow	Not possible [120]	Not possible [120]	Possible [31],[53],[183]	Possible [11],[166]
When Reconfiguring the chip:				
5. Self-reconfiguration	Not possible [120]	Possible [141]	Possible [53],[183], [184]	Possible [11],[166]
6. Reconfigure parts of the chip while the rest of the system on chip, contributing to useful computation, is operational	Not possible [39],[77]	Possible [39],[77]	Possible [39],[77],[112]	Not possible [82],[166]
7. Reconfigure parts of the chip while interfacing with the rest of the system on chip, contributing to useful computation, is operational	Not possible [39],[77]	Not possible [31],[108], [113]	Possible [31],[53],[137] ,[191]	Not possible [82],[166]
8. Reconfigure in a single cycle	Not possible [171]	Possible [39],[77]	Not possible [92],[93],[112]	Not possible [82]

Table 22 Features of Different Reconfiguration Methods

5.2.6. Summary of Features, Advantages, and Disadvantages of Reconfiguration Methods

In this section, we summarize the features of the FPGA-based reconfiguration methods in Table 22. In addition, the requirements and effects of each feature and their associated advantages and disadvantages (marked as A and DA respectively) are presented in the Tables 23-30.

In the next section, we map the computation models and application characteristics to the most suitable or efficient reconfiguration methods. The information in Tables 23-30 which includes the requirements, effects, and their associated advantages and disadvantages of each feature, is useful in selecting the most suitable reconfiguration method. Thus, during the mapping process, the designer should carefully consider the advantages and disadvantages associated with each feature from Tables 23-30 in order to select the most suitable reconfiguration method.

Feature 1 – downloading multiple configuration bitstreams simultaneously	
Requirements and Effects	**Advantages (A) and Disadvantages (DA)**
Can have multiple contexts on chip as active contexts and as inactive contexts simultaneously (for both multi context [36],[158] and partial reconfiguration [53],[183])	A - allows switching between contexts quickly
Can execute more than one active context in parallel (for both multi context [158] and partial reconfiguration [53],[183])	A - allows executing multiple operations in parallel on chip at the same time
No need to download configuration bitstreams from external memory every time when it is necessary to reconfigure the chip, as long as the next required inactive context is in on-chip memory	A - hides download time (for both multi context and partial reconfiguration)

Table 23 Requirements, Effects, Advantages, and Disadvantages of Downloading Multiple Bitstreams Simultaenously

Feature 2 – storage of configuration bitstreams in on-chip memory	
Requirements and Effects	**Advantages (A) and Disadvantages (DA)**
Need extra hardware for on-chip memory to store the inactive contexts (for both multi context [39],[48] and partial reconfiguration [21])	DA - reduces the active area for logic and routing of the design
Since the configuration bitstreams are on chip, no need to download them from the external memory, whenever it is necessary to reconfigure the chip	A - hides download time (for both multi context and partial reconfiguration) A - allows switching between contexts quickly (for both multi context [57],[136] and partial reconfiguration)

Table 24 Requirements, Effects, Advantages, and Disadvantages of Storing Bitstreams in On-Chip Memory

Feature 3 – background loading (pre-fetching) of configuration bitstreams	
Requirements and Effects	**Advantages (A) and Disadvantages (DA)**
Need extra hardware for on-chip memory to hold the downloaded configuration bitstream until the bitstream is used for reconfiguration after the execution of the current circuit is completed (for both multi context [39],[48] and partial reconfiguration [31],[53],[183])	DA - reduces the active area for logic and routing of the design
Need an external controller to pre-fetch the bitstreams from the external memory (for multi context [57][136] only)	DA – requires complex circuitry and design for interfacing
No need to disrupt the system operation to download the configuration bitstream (for both multi context [57],[136],[158] and partial reconfiguration [21])	A – overlaps download with computation
Can have more than one context on chip simultaneously (at least one active and one inactive) (for both multi context [36],[158] and partial reconfiguration [53],[183])	A - allows switching between contexts quickly

Table 25 Requirements, Effects, Advantages, and Disadvantages of Background Loading of Bitstreams

Feature 4 – internal controller to download configuration bitstreams and to control configuration flow	
Requirements and Effects	**Advantages (A) and Disadvantages (DA)**
Need extra hardware on chip for the internal controller (for both partial reconfiguration [53],[183] and MultiBoot [11],[166])	DA - reduces the active area for logic and routing of the design
Have to keep the internal controller as a static part of the design (for both partial reconfiguration [31] and MultiBoot [11],[82])	A - guarantees the integrity of the control circuitry during and after reconfiguration DA - requires extra design work
Have to ensure proper and continuous communication among the controller and the other (on-chip and off-chip) peripherals (during reconfiguration) (both on-chip and off-chip peripherals in partial reconfiguration [31][53],[137],[191], but only for off-chip peripherals in MultiBoot [11],[82])	A - ensures that the interfacing between the reconfigurable parts and the static parts are not compromised due to glitches during and after reconfiguration DA - requires extra design work with some chips but not with chips with glitchless technology
Eliminate the need to interface with an external controller (for both partial reconfiguration [53],[183],[184] and MultiBoot [11],[82])	A - allows self-reconfiguration A - requires less complicated design and circuitry A - eliminates external communication and enables quick decision making, since it takes less time to communicate with the other peripherals on chip (2-3 cycles) as opposed to using an external controller (20-30 cycles)
Can download and store the next required configuration bitstreams from external memory whenever it is needed (for partial reconfiguration [21])	A - hides download time

Table 26 Requirements, Effects, Advantages, and Disadvantages of Using an Internal Controller to Control Configuration Flow

Feature 5 – self reconfiguration	
Requirements and Effects	**Advantages (A) and Disadvantages (DA)**
Need extra hardware on chip for the control circuitry (for both partial reconfiguration [53],[183],[184] and MultiBoot [11],[166])	DA - reduces the active area for logic and routing of the design
Have to download all the configuration bitstreams to the chip with initial downloading (for multi context only [120],[136])	DA - needs extra hardware for on-chip memory to hold all the inactive contexts DA - reduces the active area for logic and routing of the design
Eliminate the need to interface with an external controller (for multi context [136], partial reconfiguration [53],[183],[184] and MultiBoot [11],[82])	A - requires less complicated design and circuitry A - eliminates external communication and enables quick decision making, since the decision for reconfiguration takes place on chip, it takes less time to communicate for reconfiguration as opposed to an external controller making the decision (for multi context, partial reconfiguration and MultiBoot)

Table 27 Requirements, Effects, Advantages, and Disadvantages of Self Reconfiguration

Feature 6 – reconfigure parts of the chip while the remainder is operational	
Requirements and Effects	**Advantages (A) and Disadvantages (DA)**
Have to make sure that active parts do not rely on the reconfigurable parts during and after reconfiguration (for multi context)	A - guarantees the integrity of the operations running in the remainder of the chip DA - requires extra design work
No need to disrupt the whole system during reconfiguration (for multi context [39],[77] and partial reconfiguration [39],[77],[112])	A - overlaps computation with reconfiguration
Can reconfigure parts of the chip that require modification without reconfiguring the whole chip (for multi context [39],[77] and partial reconfiguration [39],[77],[112])	A - avoids reconfiguring the entire chip for small changes in the design A - reduces reconfiguration time – as reconfiguring part of the chip as opposed to reconfiguring the entire chip (for multi context and partial reconfiguration [92],[93],[112])

Table 28 Requirements, Effects, Advantages, and Disadvantages of Reconfiguring Parts of the Chip while the Remainder of Chip is Operational

Feature 7 – reconfigure parts of the chip while interfacing with the operational remainder of the chip	
Requirements and Effects	**Advantages (A) and Disadvantages (DA)**
Have to make sure that active parts do not rely on the reconfigurable parts during reconfiguration (for partial reconfiguration)	A - guarantees the integrity of the operations running in the remainder of the chip DA - requires extra design work
Have to ensure proper and continuous communication among the reconfigurable parts and active parts, before, during and after reconfiguration (for partial reconfiguration [53],[106],[137],[191])	A - ensures that interfacing between the reconfigurable parts and the static parts are not compromised due to glitches during and after reconfiguration DA - requires extra design work with some chips but not the chips with glitchless technology
No need to disrupt the whole system during reconfiguration	A - overlaps computation with reconfiguration (for partial reconfiguration [39],[77],[112])
Can reconfigure parts of the chip that require modification without reconfiguring the whole chip (for partial reconfiguration [39],[77],[112])	A - avoids reconfiguring the entire chip for small changes in the design A - reduces reconfiguration time – as reconfiguring part of the chip as opposed to reconfiguring the entire chip (for multi context and partial reconfiguration [92],[93],[112])
Allows hardware reuse - can reserve part(s) of the chip allowing hardware modules corresponding to different function(s) to be reused among several operations/applications (for partial reconfiguration)	A - reduces reconfiguration time A. - overlaps computation with reconfiguration

Table 29 Requirements, Effects, Advantages, and Disadvantages of Reconfiguring Parts of the Chip while Interfacing with the Operational Remainder of Chip

Feature 8 – reconfigure in a single cycle	
Requirements and Effects	**Advantages (A) and Disadvantages (DA)**
Can be done only if the next required configuration bitstream (inactive context) is in on-chip configuration memory (for multi context [77])	A - reduces reconfiguration time drastically (for multi context [57],[136])
Switching configuration bits in many locations in a single cycle can increase dynamic power consumption (if major part of the chip is being reconfigured)	DA - violates system power constraints (for multi context [36],[158])

Table 30 Requirements, Effects, Advantages, and Disadvantages of Reconfiguring in Single Cycle

5.3. Mapping Computation Models and Application Characteristics to Reconfiguration Methods

In this section, our intention is to illustrate how the most appropriate reconfiguration method to use for a certain application can be identified. Each of the FPGA-based reconfiguration methods has its own special features, advantages, and disadvantages. Selecting a specific reconfiguration method and designing the corresponding hardware for an application are important and challenging tasks in FPGA-based reconfigurable computing systems. Different applications can have different computation models and intrinsic characteristics. According to these characteristics and computation models, some reconfiguration methods might not be suitable or desirable. If selecting a reconfiguration method and designing the corresponding hardware are done in an ad hoc manner, we might not be able to provide an efficient and effective reconfigurable hardware solution for a given application. Hence, one has to consider and match the intrinsic characteristics and computation models of an application to the features of the reconfiguration method(s) in order to identify the most appropriate reconfiguration method(s).

To illustrate the mapping of computation models and application characteristics to the reconfiguration methods, we use several scenarios based on the size of the hardware circuitry of an operation, and hence the number of computation modules (or operations) that would potentially fit into the chip. In this section, computation module and operation are used interchangeably. These scenarios, presented below, are selected for their ease of visualization for illustration purpose and they are by no means exhaustive. We assume that initially there exist four computation modules (A,B,C,D).

- Case 1 – One computation module would fit into the chip at a time. That is, only one computation module (A or B or C or D) out of four fits into the chip at a time. For instance, the chip can be reconfigured as A→B, or B→C, or C→D, etc.
- Case 2 – Two computation modules would fit into the chip at a time. That is, two computation modules (A+B or C+B or A+D or C+D or A+C or B+D) out of four fit into the chip at a time, such that the chip can be reconfigured to any one of these configurations, either by changing one module at a time (while the other module is operational) or both the modules at the same time. For instance, it can be reconfigured as (A+B)→(C+B), or (C+B)→(C+D), or (A+B)→(C+D), etc.

- Case 3 – Three computation modules would fit into the chip at a time. That is, three computation modules (A+B+C or D+B+C or D+A+C or D+A+B) out of four fit into the chip at a time, such that the chip can be reconfigured to any of these configurations, either by changing one module at a time (while the other modules are operational) or all the modules at a time. For instance, it can be reconfigured as (A+B+C)→(D+B+C), or (D+B+C)→(D+A+C), or (D+A+C)→(D+A+B), or (A+B+C)→(D+A+B), or (A+B+C)→(D), etc.

It should be noted that we can repeat the reconfiguration of a module, which has been executed before, if necessary. For instance, in Case 1, after reconfiguring A→B, B→C, and C→D, we can reconfigure D→A.

We present suitable examples using the above cases to illustrate how the reconfiguration works for each computation model and application characteristic. These examples are also used to emphasize how we can select the most efficient reconfiguration method(s). Cost analysis on space and time is also carried out for each example. For all the examples presented, we do not consider the initial (or the first) download and reconfiguration time, when measuring the total processing time.

5.3.1. Mapping Computation Models

In this section, we analyze, discuss, and present how we map the computation models to the most appropriate reconfiguration method(s) and why.

5.3.1.1. Parallel (Functional)

Two examples are presented to illustrate an application that has multiple independent computation modules that can be executed at the same time in parallel. We assume that there are four independent computation modules (A,B,C,D), and these four modules together are too large to fit into the chip at the same time.

5.3.1.1.1. First Scenario for Parallel (Functional)

The first example is based on Case 2, where two computation modules out of four would fit into the chip at a time. The chip is being reconfigured from (A+B)→(C+B)→(C+D), or (A+B)→(C+D).

In this case, multi context or partial reconfiguration might be the most appropriate reconfiguration methods to use, whereas single context and MultiBoot might not be efficient. The rationales are discussed below.

A. Multi Context for Parallel (Functional) First Scenario

Multi context allows multiple configurations to be downloaded to the chip simultaneously [158], thus, a chip can have several active contexts as well as several inactive contexts. With multi context, A and B can be downloaded and reconfigured simultaneously. With the initial downloading of A and B, if there is enough space on chip, C and D can also be downloaded and stored in the on-chip configuration memory. Multi context allows A and B (active contexts) to be executed in parallel on the same chip and C and D (inactive contexts) to be stored in on-chip configuration memory. Also multi context allows reconfiguring parts of the chip, while the remaining parts are still operational [39],[77]. Since these four computation modules are independent from each other, the reconfiguration of A→C or B→D can be done separately, after processing A and B respectively. For example, after processing A, reconfiguration A→C is done while B is still processing. Or, after processing B, reconfiguration B→D is done while A is still processing. In this way, if A and B finish processing at different times, they can be reconfigured at different times, thus overlapping reconfiguration with computation. Also with multi context, the reconfiguration can be done in a single cycle [57],[136], which is important for applications where speed performance is a concern. If A and B both finish processing at the same time, the reconfigurations A→C and B→D are done at the same time. In this case, the entire chip is being reconfigured in a single cycle [136]. As illustrated in [36],[77], switching the configuration bits in many locations (changing from 0 to 1 and vice versa) in a single cycle would increase dynamic power and may violate system power constraints. As a result, multi context might be suitable only when some parts of the chip are being reconfigured but not when the entire (or a large portion of the) chip is being reconfigured. This issue is especially critical for power-aware embedded devices.

B. Partial Reconfiguration for Parallel (Functional) First Scenario

With partial reconfiguration, we can have A and B as two partial reconfigurable modules, which can be executed in parallel. Since partial reconfiguration also allows reconfiguring portions of the chip that require modification [39],[77],[112], similar to multi context, we can reconfigure these modules separately. That is, reconfigurations A→C and B→D can be done separately, after processing A and B respectively, similar to the example presented in multi context. In this way, if A and B finish processing at different times, they can be reconfigured at different times, thus overlapping reconfiguration with computation. Also, by changing only parts of the chip that require modification, as opposed to reconfiguring the entire chip (as in single context), reconfiguration time overhead is reduced as shown in [92],[93]. If A and B both finish processing at the same time, reconfigurations A→C and B→D are done at the same time. In this case, the total processing time would be higher than that when the two reconfigurations are done at separate times, because a large portion of the chip is being reconfigured and the reconfiguration time is proportionate to the size of reconfigurable module. In addition, reconfiguration is not overlapped with computation.

C. Single Context and MultiBoot for Parallel (Functional) First Scenario

Single context or MultiBoot might not be suitable because both of these methods require reconfiguring the entire chip [77],[82] even for the smallest changes of a design. If A and B finish processing at different times, each would have to wait for the other to finish, before being reconfigured. Also, reconfiguring the entire chip incurs higher reconfiguration time overhead compared to reconfiguring only part of a chip as with partial reconfiguration [82],[171].

D. Time and Space Complexity for Partial Reconfiguration and Multi Context Parallel (Functional) First Scenario

Execution times: A is T_a; B is T_b; C is T_c; D is T_d.

Reconfiguration time: A→C is T_{ac}; B→D is T_{bd}.

Space occupied: A is S_a; B is S_b; C is S_c; D is S_d.

Time Complexity for Parallel (Functional) First Scenario:

When two computation modules finish processing at different times: For both multi context and partial reconfiguration the maximum total processing time (also the worst case scenario) is as follows:

$$(TotalProcessingTime)_{max} = max(T_a, T_b) + max(T_{ac}, T_{bd}) + max(T_c, T_d) \quad (19)$$

The two best case scenarios would be,

If $T_a < T_b$; and $T_c > T_d$; and if $T_a + T_{ac} \leq T_b$; and if $T_c \geq T_{bd} + T_d$; then the reconfiguration time is completely overlapped with the computation time. The total processing time is:

$$TotalProcessingTime \leq T_b + T_c \quad (20)$$

Also, if $T_a > T_b$; and $T_c < T_d$; and if $T_a >= T_b + T_{bd}$; and if $T_c + T_{ac} =< T_d$; then the reconfiguration time is completely overlapped with the computation time. The total processing time is:

$$TotalProcessingTime \leq T_a + T_d \quad (21)$$

In this case, the designer should arrange the operations, to be executed and reconfigured, in such a way that all of the "if" conditions stated in the above examples are satisfied. If these conditions are fulfilled, the designer should be able to execute these operations with the least amount of time, since the reconfiguration time overhead is completely concealed by the computation time. To accomplish this, the designer should check the computation times of each operation as well as the reconfiguration times from one operation to another, and schedule the processing of each operation to get the minimum execution time. Since these operations are independent, the designer has the flexibility to arrange the operations in any order as desired.

In this case, multi context would be the more efficient reconfiguration method to use, since the reconfiguration time overhead is much less than that of partial reconfiguration. In multi context, the next required inactive context is typically in on-chip configuration

memory, which eliminates the download time completely. However, in partial reconfiguration, the partial bitstreams usually have to be downloaded from the external memory, incurring higher reconfiguration time overhead.

When two computation modules finish processing at the same times: In this case, the entire chip has to be reconfigured. Unlike other reconfiguration methods, with multi context, the reconfiguration time overhead does not vary with the size of the reconfigurable module if the next required inactive context is in on-chip configuration memory. Also, in this case, the whole chip can be reconfigured in a single cycle [136]. Therefore, with multi context, the total processing time of this scenario is the same as in equation (19). However for partial reconfiguration the total processing time is as follows:

$$(TotalProcessingTime)_{max} = T_a + T_{ac} + T_{bd} + max(T_c, T_d) \quad (22)$$

where, $T_a = T_b$, since the first two computation modules finish processing at the same time.

In the above, reconfiguration time overhead for multi context is $T_{ac} = T_{bd} = T_{ac}+T_{bd} = 1$ cycle, whereas for partial reconfiguration is $(T_{ac}+T_{bd})$. When measuring the total processing time (in equations (19) and (22)), the reconfiguration time overheads for partial reconfiguration are measured according to equation (17) (from 5.2.5.1), which depends on the size of the reconfigurable module. The reconfiguration time overhead for multi context remains the same regardless of the size of the reconfigurable module, and can be done in a single cycle [136].

At a glance, multi context seems to be the more efficient reconfiguration method to use since partial reconfiguration incurs higher reconfiguration overhead, when reconfiguring the entire chip. However, multi context has the issue of spike in dynamic power consumption [48], especially when the entire chip is being reconfigured in a single cycle. The designer should consider these issues when selecting the most suitable reconfiguration method.

In this scenario, considering the issue of dynamic power increase in multi context, partial reconfiguration would be the more appropriate reconfiguration method to use, even

though partial reconfiguration incurs higher reconfiguration time overhead when reconfiguring the whole chip.

Space Complexity for Parallel (Functional) First Scenario:

For both multi context and partial reconfiguration, the occupying area is as follows:

With multi context, in order to hold one inactive context (that would take up the space of the entire chip) in the on-chip configuration memory, the maximum space required is 5% of the chip [48] (from 5.2.5.2). With partial reconfiguration, the extra hardware required on chip for reconfiguration is a constant and quite close to 1.5% of the chip (from 5.2.5.2). Therefore, for both methods, in order to have:

- A and B running in parallel:
- C and D running in parallel;
- C and B running in parallel, if B is running, while reconfiguring A→C;
- A and D running in parallel, if A is running while reconfiguring B→D;

The occupying areas of the operations must satisfy the following conditions:

$(S_a + S_b) =< 95\%$ and $(S_c + S_d) =< 95\%$ and $(S_c + S_b) =< 95\%$ and $(S_a + S_d) =< 95\%$ of the chip for multi context in general.

$(S_a + S_b) =< 98.5\%$ and $(S_c + S_d) =< 98.5\%$ and $(S_c + S_b) =< 98.5\%$ and $(S_a + S_d) =< 98.5\%$ of the chip for partial reconfiguration on Virtex 6.

In order to select the most suitable reconfiguration method, the designer should measure the area (S) occupied on chip for each operation and should verify that area of the four operations indeed satisfy the above two conditions for the corresponding reconfiguration method(s). If only one condition is satisfied, then the most likely scenario would be for partial reconfiguration, since it requires less extra hardware on chip for reconfiguration than that for multi context. Undoubtedly, partial reconfiguration would be the more appropriate method to use in this case. If the areas of the four operations satisfy the conditions in both reconfiguration methods, then the designer should further refer to Tables 23-30 and carefully consider the advantages/disadvantages of each reconfiguration method to select the most suitable one. If the combinations of the operations do not satisfy

the given conditions, the designer should aim to arrange the operations in such a way that different combinations would satisfy these conditions. Since these operations are independent, the designer has the flexibility to arrange the operations in any order as desired. If the areas of the 4 operations do not satisfy any of the conditions with any combinations, then single context should be used as an alternative, since it does not require any extra hardware on chip for reconfiguration and the whole chip can be used for logic and routing of the design.

5.3.1.1.2. Second Scenario for Parallel (Functional)

The second example is based on Case 1, where only one computation module out of four, i.e., (A or B or C or D), would fit into the chip at a time, such that chip can be reconfigured as A→B→C→D. Although, these four computation modules are independent, in this case, each module has to be executed separately one after another.

In this case, single context and MultiBoot might be the most appropriate reconfiguration methods to use, whereas partial reconfiguration might be suitable in certain situations and multi context might not be desirable. The rationales are discussed below.

A. Single Context, MultiBoot or Partial Reconfiguration for Parallel (Functional) Second Scenario

Since only one operation fits into the chip at a time, after an operation is processed, the entire chip is being reconfigured to the next operation, and so on. Single context and MultiBoot allow reconfiguring the entire chip from one operation to another [77],[82]. In addition, single context might be the simplest method to use since it does not require any extra hardware on chip for reconfiguration, and the entire chip is utilized for logic and routing of the design. Both MultiBoot and partial reconfiguration require extra hardware on chip for reconfiguration; thus, these two methods can be used if there is enough space on chip for extra hardware required for reconfiguration. Example of this would be the partial reconfiguration designs for Adder and Multiplier (from 4.4) and the Mean and Covariance [129].

B. Multi Context for Parallel (Functional) Second Scenario

With multi context, typically the entire chip is being reconfigured in a single cycle [136]. Switching the configuration bits in many locations in a single cycle, increases dynamic power, which may violate system power constraints [36],[77]. Moreover, in order to hold three inactive contexts (one per whole chip) in the on-chip configuration memory, it requires 3*5% = 15% of the chip area [48]. Hence, multi context might not be efficient when reconfiguring a large portion of the chip or the entire chip.

C. Time and Space Complexity for Single Context, MultiBoot, and Partial Reconfiguration for Parallel (Functional) Second Scenario

Execution times: A is T_a; B is T_b; C is T_c; D is T_d.

Reconfiguration time: A→B is T_{ab}; B→C is T_{bc}; C→D is T_{cd}.

Space occupied: A is S_a; B is S_b; C is S_c; D is S_d.

Time Complexity for Parallel (Functional) Second Scenario:

For singe context, MultiBoot and partial reconfiguration the total processing time is as follows:

$$TotalProcessingTime = T_a + T_{ab} + T_b + T_{bc} + T_c + T_{cd} + T_d \quad (23)$$

It should be noted that when measuring the total processing time in equation (23), the reconfiguration time overheads for single context, partial reconfiguration, and MultiBoot are measured according to equations (16), (17), and (18) respectively (from 5.2.5.1). These equations are reproduced below.

$$ReconfigurationTime(serial) = TotalNoOfConfigurationBits * ConfigurationClkPeriod \quad (16)$$

$$ReconfigurationTime(ICAP) = NoOfConfigurationBits/ICAPThroughput \quad (17)$$

$$ReconfigurationTime(Parallel) = TotalNoOfConfigurationBits/(ConfigurationClkFrequency * BusWidth) \quad (18)$$

The Xilinx Virtex-6 chip has 73859552 configuration bits [191]. As mentioned in 5.2.5.1, time to reconfigure the entire chip for single context, partial reconfiguration, and MultiBoot (using the above 3 equations) are 739 ms, 23 ms, and 23 ms, respectively. In this case, the reconfiguration for partial and MultiBoot are done using parallel interface (both 32 bits), whereas single context is done using serial interface. As a result, partial reconfiguration and MultiBoot take the least amount of time to reconfigure compared to single context, since the parallel mode is undoubtedly faster. However, with some FPGAs, it might be possible to use parallel mode for single context; thus, the reconfiguration time overhead would be par with partial reconfiguration and MultiBoot, when reconfiguring the entire chip.

The designer should be well aware of the type of interfacing utilized for each reconfiguration method, since it can have a significant impact on the overall speed performance. In the above case, either MultiBoot or partial reconfiguration would be the most efficient reconfiguration methods to use as they incur the least reconfiguration time overhead. If all three reconfiguration methods can use parallel mode, then single context would be the most appropriate to use because of its simplicity. However, when there are more than one option available, the designer should always consider the advantages and disadvantages associated with each reconfiguration method (from Tables 23-30), when selecting the most suitable one, especially when the entire chip is being reconfigured.

Space Complexity for Parallel (Functional) Second Scenario:

For single context, partial reconfiguration, and MultiBoot the occupying area is as follows:

With single context, it does not require extra hardware on chip for reconfiguration. For partial reconfiguration and MultiBoot, the extra hardware required on chip for reconfiguration is constant and quite close to 1.5% and 3.1% of the chip respectively (from 5.2.5.2). Therefore, in order to execute these four operations one after another, the occupying areas of these operations must satisfy the following conditions:

S_a =< 100% and S_b =< 100% and S_c =< 100% and S_d =< 100% of the chip for single context on Virtex 6.

S_a =< 98.5% and S_b =< 98.5% and S_c =< 98.5% and S_d =< 98.5% of the chip for partial reconfiguration on Virtex 6.

S_a =< 96.9% and S_b =< 96.9% and S_c =< 96.9% and S_d =< 96.9% of the chip for MultiBoot on Virtex 6.

With single context, the designer does not have to measure the chip area occupied by the computation modules; however, designer should verify that the computation modules indeed fit into the chip. Also, single context designs are simpler, since the designer does not have to deal with partitioning the chip so that only specific regions of the chip are being reconfigured at a time. As a result, single context is the least restrictive; hence, it would be the more appropriate one to use. However, it is always a good practice to consider the advantages and disadvantages given in Tables 23-30.

If the designer decides to use either MultiBoot or partial reconfiguration instead, then the designer should measure the areas (S) occupied on chip for each module to verify that areas of the four modules indeed satisfy the above area conditions for the corresponding reconfiguration methods, and also consider the advantages/disadvantages in Tables 23-30.

5.3.1.2. Pipeline

Two examples are presented to illustrate an application that has multiple dependent computation modules or a single computation module with a number of sub-tasks that can be executed in a pipelined fashion. We assume that there are four dependent computation modules (A,B,C,D), (or a single computation with four pipeline stages (A,B,C,D)), and these four modules (or stages) together are too large to fit into the chip at the same time.

5.3.1.2.1. First Scenario for Pipeline

The first example is based on Case 2, where two computation modules or only two out of four pipeline stages, would fit into the chip at a time. In pipelining, if the chip can only accommodate two pipeline stages at a time, we have to execute these two 2-stage modules one after another, sequentially. Initially, A and B, both fit into the chip. Then the chip is reconfigured from (A+B)→(C+B)→(C+D)→(A+D), or (A+B)→(C+D), etc.

In this case, partial reconfiguration might be the most appropriate reconfiguration method to use, whereas single context and MultiBoot might be suitable in certain situations, and multi context might not be advisable. The rationales are discussed below.

A. Partial Reconfiguration for Pipeline First Scenario

Partial reconfiguration allows reconfiguring portion of the chip that requires modification, while interfacing with the remaining parts that are operational [39],[77],[112]. In this case, the pipeline stages are designed as partial reconfigurable modules. With pipelining, it is important to have proper and continuous communication among pipelined stages during and after reconfiguration. Also, we should be able to reconfigure some pipeline stages while others are still operational and interfacing with the rest of the system.

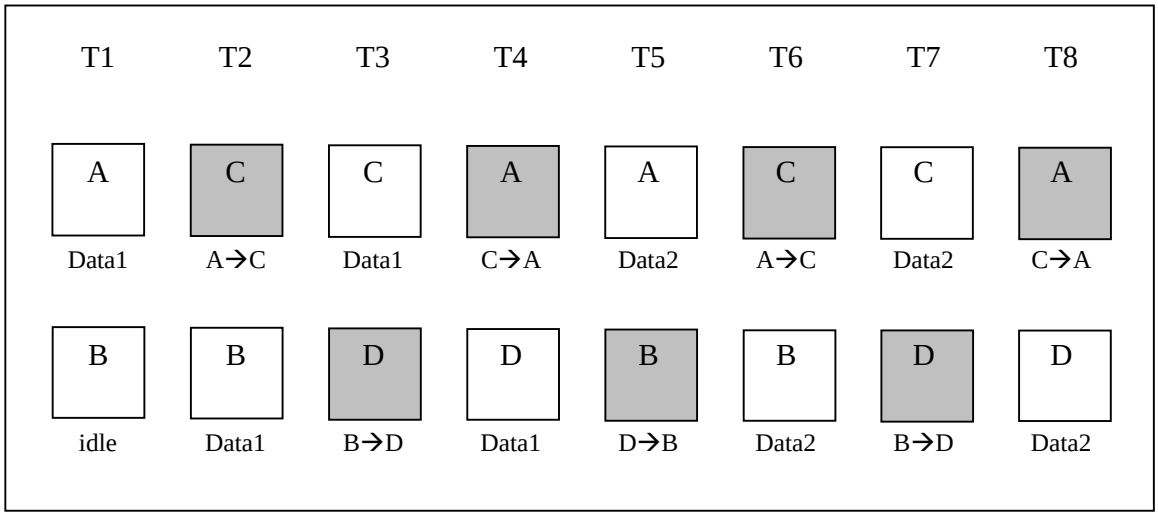


Figure 31 Pipelining (2 Stages on Chip at a Time) with Partial Reconfiguration

As shown in Figure 31, with partial reconfiguration, the chip will initially consist of the first two pipeline stages, A and B, as two reconfigurable modules. Computation modules being reconfigured (e.g., A→C) at a specific time are in grey. Computation modules processing a specific data set (e.g., Data1) at a time are in white. The time taken to execute one pipeline stage is T. Initially (at time T1), A is processing the first data set (Data1). After A processes Data1, B starts processing it. At time T2, A is being reconfigured from A→C, while B is processing Data1. In order to harness the maximum throughput of the pipeline design on a single chip, A is being reconfigured to C, instead of processing the second data set. Once B processes Data1, i.e., at time T3, B is being reconfigured from

B→D, while C is processing Data1. After C processes Data1, D starts processing, and so on. As illustrated in Figure 31, this process continues until each data set is processed by all the pipeline stages. If the reconfiguration time is $\leq$ T, we can harness the maximum benefit of pipeline design on a single chip, since the reconfiguration is completely overlapped with the computation, concealing the reconfiguration time overhead [92]. Also, by changing only portion of the chip, as opposed to reconfiguring the entire chip, the reconfiguration time overhead is reduced [92],[93].

B. Single Context and MultiBoot for Pipeline First Scenario

Single context or MultiBoot might be suitable in certain situations, although the pipeline stages can not be reconfigured separately, since both of these methods require reconfiguring the entire chip even for the smallest changes of a design [77],[82]. With these two methods, A and B have to be downloaded as one full configuration. In this case, unlike partial reconfiguration, the first two pipeline stages finish processing a few data sets (e.g., 2 data sets) at a time, before being reconfigured to the next two pipeline stages, depending on the situation. For instance, in real-time situation, it might be necessary to process smaller number of (e.g., 2 or 3) data sets at a time, in order to have a continuous flow of data and hence the results through the pipeline. To illustrate one of the many options, we consider a scenario, where two pipeline stages process 2 data sets at a time. In this case, after A finishes processing the first data set (Data1), B starts processing it. While B is processing Data1, A processes the second data set (Data2). Next, A has to wait till B finishes processing Data2. After B finishes processing Data2, then C and D have to be downloaded as one full configuration and the chip is reconfigured from (A+B)→(C+D). Then C and D process the two data sets consecutively. After D finishes processing Data2, reconfiguration from (C+D)→(A+B) is done. The same procedure continues for the next two data sets, and so on. In this case, unlike partial reconfiguration, the reconfiguration is not overlapped with computation. Since the entire chip has to be reconfigured, it incurs higher reconfiguration time overhead as opposed to reconfiguring part of the chip using partial reconfiguration [82],[171]. However, with these two reconfiguration methods, we still have the advantage of being able to execute dependent operations in pipelined fashion on a single chip.

C. Multi Context for Pipeline First Scenario

Multi context might not be suitable. The four computation modules are designed and implemented as separate configurations. With multi context, A and B are downloaded and reconfigured simultaneously, as active contexts. With the initial downloading of A and B, if there is enough space on chip, C and D can also be downloaded and stored, in the on-chip configuration memory, as inactive contexts. In this case, after A processes Data1, B starts processing it. Then we can reconfigure A→C, while B is processing Data1. With multi context, proper communication between computation modules cannot be guaranteed during and after reconfiguration, which can cause glitches and loss of information [113]. To use this method, it is imperative to ensure that the interfaces between pipeline stages, in this case, between B and C after reconfiguration A→C, are not compromised due to glitches during and after reconfiguration.

D. Time and Space Complexity for Partial Reconfiguration, Single Context, and MultiBoot for Pipeline First Scenario

Execution times: A is T_a; B is T_b; C is T_c; D is T_d.

Since each pipeline stage takes same amount of time: $T_a = T_b = T_c = T_d = T$.

Reconfiguration time: A→C is T_{ac}; B→D is T_{bd}; C→A is T_{ca}; D→B is T_{db}.

Reconfiguration time: A+B→C+D is $T_{ab_{\rightarrow}cd}$; C+D→A+B is $T_{cd_{\rightarrow}ab}$.

Space occupied: A is S_a; B is S_b; C is S_c; D is S_d.

Time Complexity for Pipeline First Scenario:

For partial reconfiguration, the best case scenario would be:

If $T_{ac} \leq T$; $T_{bd} \leq T$; $T_{ca} \leq T$; $T_{db} \leq T$; then the reconfiguration time is completely overlapped with computation, hiding the reconfiguration time overheads (measured according to equation (16) from 5.2.5.1). In this case, only one module is reconfigured at a time.

Times taken to process 1 and 2 sets of data are 4T and 8T, respectively. Time taken to process X sets of data is:

$$TotalProcessingTime = 4XT \quad (24)$$

For single context and MultiBoot the total processing time to process certain number of data sets might vary.

If the reconfiguration time for half of the chip is T, then the reconfiguration time for the entire chip is: $T_{ab \rightarrow cd} = 2T$; $T_{cd \rightarrow ab} = 2T$. The reconfiguration time overheads for single context and MultiBoot are measured according to equations (16) and (18) respectively (from 5.2.5.1).

In the following, the total number of data sets is X, and the number of data sets processed at a time is Y, where $X \geq Y$.

When X = Y: If the first two stages process all the data sets, before being reconfigured to the next two stages, then the total processing times to process: one set of data = 2T+2T+2T = 6T; two sets of data = 3T+2T+3T = 8T; three sets of data = 4T+2T+4T = 10T; etc. Then the total processing time to process X sets of data is:

$$TotalProcessingTime = 2(X+2)T \tag{25}$$

When X > Y: If the first two stages finish processing a few data sets (e.g., 2, 3, or 4 data sets) at a time, before being reconfigured to the next two stages, then the total processing time varies, according to the number of data sets Y processed at a time. For instance, if two stages process 2 data sets at a time, then the total processing time: for the first 2 data sets = 3T+2T+3T=8T; for the next 2 data sets = 2T+3T+2T+3T=10T. If two stages process 3 data sets at a time, then the total processing time: for the first 3 data sets = 4T+2T+4T=10T; for the next 3 data sets = 2T+4T+2T+4T=12T. Then the total processing time to process X sets of data, by processing Y number of data sets at a time is:

$$TotalProcessingTime = 2(Y+2)T + 2(Y+3)T\left\lfloor \frac{(X-1)}{Y} \right\rfloor \tag{26}$$

In this case, the designer should measure the computation time for each operation as well as the reconfiguration time from one operation to another, and check whether these

times satisfy the "if" conditions stated above. If these conditions are satisfied, the designer should be able to execute the four dependent operations in pipelined fashion for any number of data sets using partial reconfiguration, since the reconfiguration time overhead is completely hidden by the computation time.

With single context and MultiBoot, time to process X sets of data can be either less than or more than that of partial reconfiguration, depending on the number of data sets Y processed at a time. With partial reconfiguration, there is a continuous flow of data through the pipeline and outputs results continuously. However, with these other two reconfiguration methods, depending on the value of Y, the data might or might not be processed continuously through the pipeline. In addition, unlike partial reconfiguration, with single context and MultiBoot, it is necessary to store the intermediate results from the first two pipeline stages in temporary storage, which could be a disadvantage. In this case, the amount of storage required depends on the value of Y. The designer should be aware of these issues when selecting the most appropriate reconfiguration method and also consider the advantages and disadvantages given in Tables 23-30.

- Considering equation (24), when X = 4, X = 20, and X = 400, the total processing times for partial reconfiguration are 16T, 80T, and 1600T respectively.
- Considering equation (25), when X = 4, X = 20, and X = 400, the total processing times for single context and MultiBoot are 12T, 44T, and 804T respectively.
- Considering equation (26), for smaller values of Y (e.g., when Y = 2), when X = 4, and X = 400, the total processing times for single context and MultiBoot are 18T, and 1998T respectively.
- Considering equation (26), for larger values of Y (e.g., when Y = 10), when X = 20, and X = 400, the total processing times for single context and MultiBoot are 50T, and 1038T respectively.

From the above numbers, for the cases with equations (24) and (25), when X increases, the incremental rate of the total processing time of partial reconfiguration is higher than the other two methods. For the cases with equations (24) and (26), when X increases, the incremental rate of the total processing time of partial reconfiguration is either lower for smaller values of Y or higher for larger values of Y, compared to the other two methods. The total processing time difference between partial reconfiguration and the other two

methods increases with Y. Also, the required temporary storage increases with the increasing Y. These tradeoffs should be taken into consideration when selecting the most suitable reconfiguration method. Especially for larger values of Y, single context and MultiBoot might seem more efficient with respect to processing time; however, partial reconfiguration provides features that allow the data to flow smoothly through the pipeline without requiring any temporary storage for intermediate results.

Space Complexity for Pipeline First Scenario:

For partial reconfiguration the occupying area is as follows:

The extra hardware required on chip for reconfiguration is a constant and is quite close to 1.5% of the chip (from 5.2.5.2). Therefore, in order to have:

- A and B on chip at the same time (T1);
- B and C on chip at the same time (T2);
- C and D on chip at the same time (T3);
- A and D on chip at the same time (T4);

The occupying areas of the operations must satisfy the following conditions:

$(S_a + S_b) =< 98.5\%$ and $(S_b + S_c) =< 98.5\%$ and $(S_c + S_d) =< 98.5\%$ and $(S_a + S_d) =< 98.5\%$ of the chip for partial reconfiguration on Virtex 6.

For single context and MultiBoot the occupying area is as follows:

With single context, it does not require extra hardware on chip for reconfiguration. With MultiBoot, the extra hardware required on chip for reconfiguration is a constant and is quite close to 3.1% of the chip (from 5.2.5.2). Therefore, in order to have:

- A and B on chip at the same time;
- C and D on chip at the same time;

The occupying areas of the operations must satisfy the following conditions:

$(S_a + S_b) =< 100\%$ and $(S_c + S_d) =< 100\%$ of the chip for single context on Virtex 6.

$(S_a + S_b) =< 96.9\%$ and $(S_c + S_d) =< 96.9\%$ of the chip for MultiBoot on Virtex 6.

In this case, the designer should measure the area (S) occupied on chip for each operation to verify that the areas of the four operations indeed satisfy the above conditions for the corresponding reconfiguration methods. If the areas of the four operations satisfy the conditions in all three reconfiguration methods, then the designer should carefully consider the advantages and disadvantages associated with each reconfiguration (by referring to Tables 23-30) to select the most suitable one. In this case, single context might be the least restrictive, since the designer does not have to deal with partitioning the chip so that only specific areas of the chip are being reconfigured at a time. In addition, with single context, the whole chip can be used for logic and routing of the design. On the other hand, with single context, for large values of Y (e.g., Y=10), the data do not flow continuously through the pipeline as in partial reconfiguration, and require extra memory to store intermediate results; however, the processing time is lower than that of partial reconfiguration. For smaller values of Y (e.g., Y=2), for single context, the data continuously flow through the pipeline, and the extra memory required for intermediate results is insignificant; however, the processing time is quite close to (when Y=3) or a bit higher (when Y=2) than that of partial reconfiguration. Designer should be well aware of these issues when selecting the most suitable reconfiguration method for pipelined operations. The designer should ensure that both the time and space conditions are satisfied, in order for the pipelined operations to work well on a single chip. In order to have a workable solution, the designer should also ensure that the interfacings between pipeline stages are not compromised due to glitches during and after reconfiguration.

5.3.1.2.2. Second Scenario for Pipeline

The second example is based on Case 3, where three computation modules, i.e., three out of four pipeline stage, would fit into the chip at a time. In pipelining, if the chip is able to accommodate three pipeline stages, we could execute them in pipelined fashion. Initially, A, B, and C fit into the chip. Then the chip is reconfigured from (A+B+C)→(D+B+C)→(D+A+C)→(D+A+B), or (A+B+C)→D.

In this case, partial reconfiguration might be the most appropriate reconfiguration method to use, whereas single context and MultiBoot might be suitable in certain situations, and multi context might not be workable. The rationales are discussed below.

A. Partial Reconfiguration for Pipeline Second Scenario

Similar to the previous example, in this case also, the pipeline stages are designed as partial reconfigurable modules. With partial reconfiguration, we should be able to reconfigure some pipeline stages while the others are still operational and interfacing with the rest of the system [39],[77],[112]. This allows proper and continuous communication among pipeline stages during and after reconfiguration, which is important for pipeline operations.

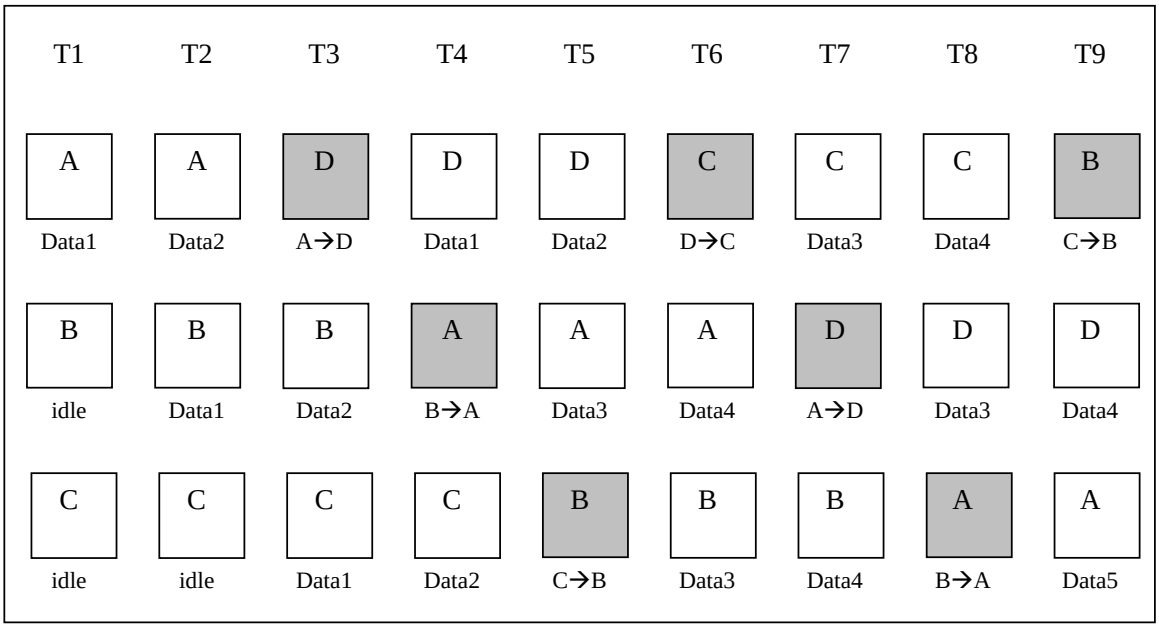


Figure 32 Pipelining (3 Stages on Chip at a Time) with Partial Reconfiguration

With partial reconfiguration, as illustreated in Figure 32, the chip will initially consist of the first three pipeline stages, A, B, and C, as three reconfigurable modules. Similar to the previous example, the computation modules being reconfigured (e.g., A→D) at a specific time are in grey. Computation modules processing a specific data set (e.g., Data1) at a time are in white. The time taken to execute one pipeline stage is T. Initially (at time T1), A is processing the first data set (Data1). After A processes Data1, B starts processing it. At time T2, A and B are processing Data2 and Data1 respectively. At time T3, A is being reconfigured from A→D, while B and C are processing Data2 and Data1 respectively. In order to harness the maximum throughput of the pipeline design on a single chip, A is being reconfigured to D, instead of processing the third data set. At time

T4, B is being reconfigured from B→A, while C and D are processing Data2 and Data1 respectively, and so on. As shown in Figure 32, this process continues until each data set is processed by all the pipeline stages. If the reconfiguration time is $\leq$ T, we can harness the maximum benefit of pipeline design on a single chip, since reconfiguration is overlapped with computation, concealing the reconfiguration time overhead [92]. Since only part of the chip is being reconfigured, as opposed to the entire chip, the reconfiguration time overhead is reduced [92],[93].

B. Single Context and MultiBoot for Pipeline Second Scenario

Similar to the previous pipeline example, in this case also, single context or MultiBoot might be efficient in certain situations. However, both of these methods require reconfiguring the entire chip even for the smallest changes of a design [77],[82]. As a result, the pipeline stages can not be reconfigured separately. With these two methods, A, B, and C have to be downloaded as one full configuration. This can be done in several ways. The most efficient way is to allow the first three stages to process a few data sets (e.g., 2 data sets) at a time, before being reconfigured to the next pipeline stage, depending on the situation, for instance, when processing data in real-time. In order to illustrate one of the many options, we consider a scenario where three pipeline stages process 2 data sets at a time. In this case, after A finish processing the first data set (Data1), B starts processing it. While B is processing Data1, A processes second data set (Data2). Unlike partial reconfiguration, A has to wait while B and C finish processing Data2 and Data1 respectively. Next A and B have to wait till C finishes processing Data2. After C finishes processing Data2, D has to be downloaded as one full configuration, and the chip is reconfigured from (A+B+C)→D. Then D will process both data sets consecutively. After D finishes processing Data2, reconfiguration from D→(A+B+C) is next. The same procedure continues for the next two data sets, and so on. In this case also, unlike partial reconfiguration, the reconfiguration is not overlapped with computation, and the entire chip has to be reconfigured, incurring higher reconfiguration time overhead [82],[171]. However, with these two reconfiguration methods, we still have the advantage of being able to execute dependent operations in pipelined fashion in a single chip.

C. Multi Context for Pipeline Second Scenario

Multi context might not be suitable. The four computation modules are designed and implemented as separate configurations. With multi context, A, B, and C are downloaded and reconfigured simultaneously, as active contexts. With the initial downloading of A, B, and C, if there is enough space on chip, D is also downloaded and stored in the on-chip configuration memory as inactive context. In this case, after A processes Data1, B starts processing it. Next, A and B process Data2 and Data1 respectively. After A processes Data2, we can reconfigure A→D, while B and C are processing Data2 and Data1 respectively. With multi context, proper communication between computation modules cannot be guaranteed during and after reconfiguration, which can cause glitches and loss of information [113]. In order to use this method, it is imperative to ensure that the interfacing between pipeline stages, in this case, the interface between C and D after reconfiguration from A→D, is not compromised due to glitches during and after reconfiguration.

D. Time and Space Complexity for Partial Reconfiguration, Single Context, and MultiBoot for Pipeline Second Scenario

Execution times: A is T_a; B is T_b; C is T_c; D is T_d.

Since each pipeline stage takes same amount of time: $T_a = T_b = T_c = T_d = T$.

Reconfiguration time: A→D is T_{ad}; B→A is T_{ba}; C→B is T_{cb}; D→C is T_{dc}.

Reconfiguration time: A+B+C→D is $T_{abc\rightarrow d}$; D→A+B+C is $T_{d\rightarrow abc}$.

Space occupied: A is S_a; B is S_b; C is S_c; D is S_d.

Time Complexity for Pipeline Second Scenario:

For partial reconfiguration the total processing time is as follows:

The best case scenario would be,

If $T_{ad} \leq T$; $T_{ba} \leq T$; $T_{cb} \leq T$; $T_{dc} \leq T$; then the reconfiguration time is completely overlapped with the computation, concealing the reconfiguration time overheads (measured according to equations (17) from 5.2.5.1). In this case, only one module is reconfigured at a time.

Times taken to process 1 and 2 sets of data are 4T and 5T, respectively. Times taken to process 3 and 4 sets of data are 8T and 9T, respectively. Times taken to process 5 and 6

sets of data are 12T and 13T, respectively. Hence, time taken to process X sets of data is as follows.

If X is an odd number, total processing time is (2X+2)T, and if X is an even number, total processing time is (2X+1)T.

$$TotalProcessingTime = m(2X+2)T + n(2X+1)T \quad (27)$$

where, m = X mod 2 and n = (X+1) mod 2.

For single context and MultiBoot the total processing time is as follows:

The total processing time to process certain number of data sets might vary. With these two methods, when reconfiguring from (A+B+C)→D or D→(A+B+C), the reconfiguration time would be the same, since the entire chip is being reconfigured by downloading a full bitstream. In case of partial reconfiguration, the reconfiguration time for one module (occupying one third of the chip) is T, hence for comparison purpose, for single context and MultiBoot, we consider the reconfiguration time for the entire chip as: $T_{abc \rightarrow d}$ = 3T; $T_{d \rightarrow abc}$ = 3T. The reconfiguration time overheads for single context and MultiBoot are measured according to equations (16) and (18) respectively (from 5.2.5.1).

In the following, the total number of data sets is X, and the number of data sets processed at a time is Y, where X $\geq$ Y.

When X = Y: If the first three stages process all the data sets, before being reconfigured to the next stage, then the total processing times to process: one set of data = 3T+3T+T = 7T; two sets of data = 4T+3T+2T = 9T; three sets of data = 5T+3T+3T = 11T; etc. The total processing time to process X sets of data is:

$$TotalProcessingTime = (2X+5)T \quad (28)$$

When X > Y: If the first three stages finish processing a few data sets (e.g., 2, 3, or 4 data sets) at a time, before being reconfigured to the next stage, then the total processing times varies, according to the number of data sets Y processed at a time. For instance, if

three stages process 2 data sets at a time, then the total processing time: for the first 2 data sets = 4T+3T+2T=9T; for the next 2 data sets = 3T+4T+3T+2T=12T. If two stages process 3 data sets at a time, then the total processing time: for the first 3 data sets = 5T+3T+3T=11T; for the next 3 data sets = 3T+5T+3T+3T=14T. Therefore the total processing time to process X sets of data, by processing Y number of data sets at a time is:

$$TotalProcessingTime = (2Y+5)T + 2(Y+4)T\left\lfloor \frac{(X-1)}{Y} \right\rfloor \quad (29)$$

In this case, the designer should measure the computation time for each operation as well as the reconfiguration time from one operation to another, and check whether these times satisfy the "if" conditions stated above for partial reconfiguration. If these conditions are satisfied, the reconfiguration time is completely concealed by the computation time, enabling the designer to efficiently execute these four dependent operations in pipelined fashion on a single chip. Since more than two pipeline stages fit into the chip, execution time is significantly less by processing these operations in pipelined fashion than processing them sequentially, for any number of data sets. For instance, in a pipelined fashion, processing time for X sets of data is (2X+2)T (if X is odd) or (2X+1)T (if X is even), whereas sequentially it is 4XT.

- Considering equation (27), when X = 4, X = 20, and X = 400, the total processing times for partial reconfiguration are 9T, 41T, and 801T respectively.
- Considering equation (28), when X = 4, X = 20, and X = 400, the total processing times for single context and MultiBoot are 13T, 45T, and 805T respectively.
- Considering equation (29), for smaller values of Y (e.g., when Y = 2), when X = 4, and X = 400, the total processing times for single context and MultiBoot are 21T, and 2397T respectively.
- Considering equation (29), for larger values of Y (e.g., when Y = 10), when X = 20, and X = 400, the total processing times for single context and MultiBoot are 53T, and 1117T respectively.

From the above numbers, for the cases with equations (27) and (28), when X increases, the incremental rate of the total processing time of partial reconfiguration is almost the

same as the other two methods. For the cases with equations (27) and (29), when X increases, the incremental rate of the total processing time of partial reconfiguration is lower than the other two methods, for smaller values of Y as well as for larger values of Y.

Unlike the 2-stage pipeline example, in this case, partial reconfiguration takes less time to process X sets of data than the other two methods, regardless of the number of data sets Y processed at a time. Also, with single context and MultiBoot, for large values of Y, the data are not processed continuously through the pipeline; and also temporary storage for intermediate results is required. Undoubtedly, partial reconfiguration is the most efficient method to use in this scenario, since it allows continuous flow of data through the pipeline with less processing time, outputting results continuously, without requiring any temporary storage for intermediate results.

Space Complexity for Pipeline Second Scenario:

For partial reconfiguration the occupying area is as follows:

The extra hardware required on chip for reconfiguration is a constant and is quite close to 1.5% of the chip (from 5.2.5.2). Therefore, in order to have:

- A, B, and C on chip at the same time (T1)
- B, C, and D on chip at the same time (T2)
- A, C, and D on chip at the same time (T3)
- A, B, and D on chip at the same time (T4)

The occupying area must satisfy the following conditions:

$(S_a + S_b + S_c) =< 98.5\%$ and $(S_b + S_c + S_d) =< 98.5\%$ and $(S_a + S_c + S_d) =< 98.5\%$ and $(S_a + S_b + S_d) =< 98.5\%$ of the chip for partial reconfiguration on Virtex 6.

For single context and MultiBoot the occupying area is as follows:

With single context, it does not require extra hardware on chip for reconfiguration. With MultiBoot, the extra hardware required on chip for reconfiguration is a constant and is quite close to 3.1% of the chip (from 5.2.5.2). Therefore, in order to have:

- A, B, and C on chip at the same time;
- Only D on chip;

The occupying areas of the operations must satisfy the following conditions:

$(S_a + S_b + S_c) =< 100\%$ and $(S_d) =< 100\%$ of the chip for single context on Virtex 6.
$(S_a + S_b + S_c) =< 96.9\%$ and $(S_d) =< 96.9\%$ of the chip for MultiBoot on Virtex 6.

The area (S) occupied on chip by each operation should be measured to verify that the areas of the four operations indeed satisfy the above conditions for the corresponding reconfiguration methods. If these conditions are satisfied for all three reconfiguration methods, as the next step, designer should refer to Tables 23-30 and weigh in the advantages and disadvantages associated with each reconfiguration to select the most efficient one. Although, at a glance, single context might seem simpler and least restrictive, hence more efficient: however, the designer should also consider other factors, such as processing time, the value of Y and its effect on the data flow and the extra memory required to store intermediate results. Designer should be aware of these issues when selecting the most suitable reconfiguration method for pipelined operations. The designer should ensure that both the time and space conditions are satisfied, in order for the pipelined operations to work well on a single chip. Also the interfacings between pipeline stages should not be compromised due to glitches during and after reconfiguration, in order to have a workable solution. With a 3-stage pipeline solution, partial reconfiguration enables the designer to execute this 4-stage computation efficiently on a single chip, with less processing time regardless of the value of Y.

If the above combination of operations (three at a time) do not satisfy the given area and time conditions, another alternative for the designer is to have two operations (instead of three) on the chip at a time. Similar to the example given in 5.3.1.2.1 (with 2 pipeline stages on chip at a time), the designer could execute these four operations in pipelined fashion using any of the above three reconfiguration methods on a single chip.

5.3.1.3. Computations with Many Identical Sub-Functions or Sub-Tasks

One example, which is based on Case 1, is presented to illustrate an application that has operations with many identical sub-functions among them. We assume that there are four operations (A,B,C,D), and these four operations together are too large to fit into the chip at the same time. Also, for Case 1, only one operation (A or B or C or D), as a separate

entity, would fit into the chip at a time, such that the chip can be reconfigured from A→B→C→D. Since these four operations have a large number of identical sub-functions among each other, the chip is reconfigured to take any one of these configurations by changing the whole operation or by changing only the non-identical sub-functions among the operations.

In this case, partial reconfiguration might be the most effective reconfiguration method to use, whereas multi context, single context and MultiBoot might not be appropriate. The rationales are discussed below.

A. Partial Reconfiguration for Computations with Many Identical Sub-Functions

Partial reconfiguration might be most suitable. Since partial reconfiguration allows reconfiguring parts of the chip that require modification while other parts remain intact or are still operational and interfacing with the rest of the system [39],[77],[112], proper and continuous communication among the sub-functions can be guaranteed during and after reconfiguration. With partial reconfiguration, initially A is downloaded and processed. During the reconfiguration from A→B, the functional units corresponding to the identical sub-functions between A and B remain intact, while the rest of the functional units of B is downloaded as partial bitstreams and reconfigured to its appropriate hardware circuitry. This procedure is continued for reconfigurations from B→C, and C→D. This reduces the reconfiguration time overhead, since only part of the chip is being reconfigured as oppose to the whole chip [92],[93]. Also in some cases, the early stages (with different sub-functions) of A can be reconfigured to the next required sub-functions (of B) while the later stages of A are still operating, thus overlapping reconfiguration with computation.

B. Single Context and MultiBoot for Computations with Many Identical Sub-Functions

Single context or MultiBoot might not be proper to use because both of them require reconfiguring the entire chip even for the smallest changes of a design [77],[82]. With these two methods, A has to be downloaded as one full configuration. After A is processed, the entire chip has to be reconfigured from A→B. Similarly, the chip has to be reconfigured from B→C and C→D, after processing B and C respectively. In this way, the entire chip has to be reconfigured three times to execute these four operations as separate

entities. Even though, these operations have a large number of identical sub-functions among each other, they can not be selectively reused with these two reconfiguration methods. Also, reconfiguring the entire chip incurs higher reconfiguration time overhead [82],[171], compared to reconfiguring parts of the chip as with partial reconfiguration.

C. Multi Context for Computations with Many Identical Sub-Functions

Multi context also might not be suitable. With multi context, we can download and reconfigure the chip with A (as the active context), and simultaneously download and store B, C, and D (as inactive contexts) in on-chip configuration memory, if there is enough space on chip [158]. When we reconfigure the chip from A→B, we can let the functional units (corresponding to the sub-functions) shared by A and B remain intact and reconfigure the rest of the chip. With multi context, proper communication between functional units cannot be guaranteed during and after reconfiguration, which can cause glitches and loss of information [113]. To use this method, it is important to ensure that the interfacings among the functional units are not compromised due to glitches during and after reconfiguration. Also, with multi context, in order to hold three inactive contexts (one per whole chip) in the on-chip configuration memory, it requires 3*5% = 15% of the chip [48], which is not space efficient.

D. Time and Space Complexity for Partial Reconfiguration for Computations with Many Identical Sub-Functions

Execution times: A is T_a; B is T_b; C is T_c; D is T_d.

Reconfiguration time: A→B is T_{ab}; B→C is T_{bc}; C→D is T_{cd}.

Space occupied: A is S_a; B is S_b; C is S_c; D is S_d.

Time Complexity for Computation with Many Identical Sub-Functions:

For partial reconfiguration the total processing time is as follows:

$$TotalProcessingTime = T_a + T_{ab} + T_b + T_{bc} + T_c + T_{cd} + T_d \quad (30)$$

In this case, partial reconfiguration would be the most efficient method to use since it allows reconfiguring the functional units corresponding to the different sub-functions that

exist among the operations, while identical sub-functions among the operations remain intact. This allows reuse of functional units among several different operations that have identical sub-functions among each other. Also, partial reconfiguration has the least reconfiguration time overhead (measured according to the equation (17) from 5.2.5.1), compared to single context or MultiBoot.

Space Complexity for Computation with Many Identical Sub-Functions:

For partial reconfiguration the occupying area is as follows:

The extra hardware required on chip for reconfiguration is a constant and quite close to 1.5% of the chip (from 5.2.5.2). Therefore, in order to execute these four operations using partial reconfiguration, the occupying area must satisfy the following condition:

S_a =< 98.5% and S_b =< 98.5% and S_c =< 98.5% and S_d =< 98.5% of the chip for partial reconfiguration on Virtex 6.

In this case, the designer should measure the area (S) occupied on chip for the four operations to verify that the areas of these four operations indeed satisfy the above condition for partial reconfiguration. If these conditions are not satisfied, i.e., if each of the operations occupies more than 98.5% of the chip, then the next best alternative would be to use single context. As mentioned in 5.2.5.2, except for single context, all the other reconfiguration methods require more than 1.5% of the chip for extra hardware for reconfiguration. However, with single context, the total processing time would be higher than that with partial reconfiguration, because the entire chip is being reconfigured. Designer should be aware of these issues when selecting the most appropriate reconfiguration method for computations with many identical functions. In any case, it is always a good practice to consider the advantages and disadvantages given in Tables 23-30 when considering alternative approaches.

5.3.1.4. Parallel (Data)

For applications with data parallelism, we execute the same operation on several processing elements (PEs) with different data in parallel. In data parallelism, the operations executed on the PEs might change with different applications or even within a

single application. We present an example, Case 1, where only one operation out of four, i.e., A or B or C or D, would fit into the chip at a time, such that the chip is reconfigured from A→B→C→D. In this case, these four operations have to be executed (on several PEs with different data) separately one after another, which is similar to the example in 5.3.1.1.2. Applying the analysis used in that example, similar results are obtained here: single context and MultiBoot might be the most efficient reconfiguration methods to use, whereas partial reconfiguration might be appropriate in certain situations and multi context might not be suitable.

5.3.2. Mapping Application Characteristics

In this section, we analyze, discuss, and present what would be the most efficient reconfiguration method(s) to use for each application characteristic and why.

5.3.2.1. Multi-Stage and Lengthy Processing

Two examples are presented to illustrate an application that has multi-stage and lengthy computation process. We assume that there are four stages corresponding to four operations (A,B,C,D), and these four stages together are too large to fit into the chip at the same time.

5.3.2.1.1. First Scenario for Multi-Stage and Lengthy Processing

The first example is based on Case 2, where two operations out of four stages, would fit into the chip at a time. Initially, A and B, both fit into the chip. Then the chip is reconfigured from (A+B)→(C+B)→(C+D), or (A+B)→(C+D), etc. This is similar to the example in 5.3.1.2.1. Applying the analysis used in that example, similar results are obtained: partial reconfiguration might be the most efficient reconfiguration method to use, whereas single context and MultiBoot might be suitable in certain situation, and multi context might not be appropriate.

5.3.2.1.2. Second Scenario for Multi-Stage and Lengthy Processing

The second example is based on Case 1, where only one operation (or stage) out of four, i.e., A or B or C or D, would fit into the chip at a time, such that the chip is reconfigured from A→B→C→D. In this case, these four operations have to be executed sequentially, one after another, which is similar to the example in 5.3.1.1.2. Applying the analysis used

in that example, similar results are obtained: single context and MultiBoot might be the most efficient reconfiguration methods to use, whereas partial reconfiguration might be proper in certain situations and multi context might not be suitable.

5.3.2.2. Various Methods to Carry Out an Operation

An example based on Case 1 is presented to illustrate an application that uses different algorithms in different scenarios, or an application that uses various methods to carry out an operation. We assume an application has four stages corresponding to four operations (A,B,C,D), where each stage is solvable by four different methods (operations). For instance, A is solvable by either A_1, A_2, A_3, or A_4, similarly for B, C, and D.

For Case 1, only one operation or only one method of one stage (A_i or B_i or C_i or D_i) out of many would fit into the chip at a time, such that the chip is reconfigured from $A_i \rightarrow B_i \rightarrow C_i \rightarrow D_i$, where i = 1, 2, 3, or 4. In this case, after the first stage A is processed using any one of the operations (A_1,A_2,A_3,A_4), it decides which operation to use for the second stage B from any one of the operations (B_1,B_2,B_3,B_4). Then that operation is downloaded and the chip is reconfigured from $A_i \rightarrow B_i$. This process continues until all the stages are executed. In this case, these operations are executed sequentially one after another, which is similar to the example in 5.3.1.1.2. Applying the analysis used in that example, similar results are obtained: single context and MultiBoot might be the most efficient reconfiguration methods to use, whereas partial reconfiguration might be suitable in certain situations and multi context might not be appropriate.

5.3.2.3. Dynamic Decision Making and Changing Operations Dynamically

Two examples are presented to illustrate an application that requires dynamic decision making and changing the operations dynamically. We assume that there are four operations (A,B,C,D), and these four operations together are too large to fit into the chip at the same time.

In this case, once an operation (e.g., A) or operations are processed, the configuration controller, according to certain criteria, decides which operation is next (B, or C, or D) and downloads the configuration bitstream of that operation and reconfigures the chip. The decision making, downloading, and reconfiguring are done on-the-fly. With single context [120], the decision-making is done by an external configuration controller, whereas with

MultBoot [83],[166] and partial reconfiguration [53],[183],[184], decision is made by an internal configuration controller. For multi context [136] also, the decision making can be done internally, only if the next required context is stored as an inactive context in the configuration memory on chip. In this case, when an active operation is near completion, the active context itself can trigger the on-chip reconfiguration memory to reconfigure the chip, with the next required context.

5.3.2.3.1. First Scenario for Dynamic Decision Making and Changing Operations Dynamically

The first example is based on Case 2, where two operations would fit into the chip at a time. The chip is being reconfigured from (A+B)→(C+B)→(C+D)→(A+D), or (A+B)→(C+D).

Independent operations: If these four operations are independent from each other, then they can be executed in parallel, which is similar to the example in 5.3.1.1.1. Applying the analysis used in that example, similar results are obtained: multi context or partial reconfiguration might be the most suitable reconfiguration methods to use, whereas single context and MultiBoot might not be workable.

Dependent operations: If these four operations are dependent, then they have to be executed one after another, which is similar to the example 5.3.1.2.1. Applying the analysis used in that example, similar results are obtained: partial reconfiguration might be the most efficient reconfiguration method to use, whereas single context and MultiBoot might be suitable in certain situations, and multi context might not be appropriate.

5.3.2.3.2. Second Scenario for Dynamic Decision Making and Changing Operations Dynamically

The second example is based on Case 1, where only one operation out of four, i.e., A or B or C or D, would fit into the chip at a time, such that the chip is reconfigured from A→B→C→D. In this case, regardless of the four operations being dependent or independent, these operations have to be executed separately one after another, which is similar to the example in 5.3.1.1.2. Applying the analysis used in that example, similar results are obtained: single context and MultiBoot might be the most efficient

reconfiguration methods to use, whereas partial reconfiguration might be efficient in certain situations and multi context might not be suitable.

5.3.2.4. Evolving Algorithms

Considering Case 1, an example is presented to illustrate an application that has evolving algorithms or operations. We assume an application initially has four operations (A,B,C,D) and each operations is being evolved into four different operations. For instance, A is being evolved into (A_1,A_2,A_3,A_4), similarly for B, C, and D.

For Case 1, only one operation (A or B or C or D) or one evolved operation (A_i or B_i or C_i or D_i) out of many would fit into the chip at a time, such that the chip is reconfigured from A→B→C→D, or A_i→B_i→C_i→D_i, where i = 1, 2, 3, or 4. In this case, after A is processed, if we have to process A_1, then the chip is reconfigured from A→A_1, and A_1 is processed. If we have to process B_1, then the chip is reconfigured from A→B_1, and B_1 is processed, and so on. This is similar to the example in 5.3.1.1.2. Applying the analysis used in that example, similar results are obtained: single context and MultiBoot might be the most efficient reconfiguration methods to use, whereas partial reconfiguration might be efficient in certain situations and multi context might not be suitable.

It should be noted that the newly evolved operations have to be stored in the external non-volatile memory. Hence, the designer should ensure that external non-volatile memory is large enough to store these additional evolved operations.

5.3.2.5. New and Emerging Algorithms

Another example considering Case 1 is presented to illustrate an application that has new and emerging algorithms or operations. We assume an application initially has four operations (A,B,C,D), and new operations (E,F,G,H) are being integrated into the application. For Case 1, only one operation (A or B or C or D or E or F or G or H) out of many would fit into the chip at a time, such that the chip is reconfigured from A→B→C→D→E→F→G→H. In this case, these operations have to be executed separately one after another, which is similar to the example in 5.3.1.1.2. Applying the analysis used in that example, similar results are obtained: single context and MultiBoot might be the best reconfiguration methods to use, whereas partial reconfiguration might be efficient in certain situations and multi context might not be suitable.

Similar to the previous example, the additional new operations have to be stored in an external non-volatile memory. Hence, the designer should ensure that external non-volatile memory is large enough.

5.3.2.6. Adapt to Different Standards and Operating Conditions

An example is presented based on Case 1 to illustrate an application that requires adapting to different standards and operating conditions. We assume an application has four protocols, whose functional units correspond to four operations (A,B,C,D). Other assumptions are that each protocol itself occupies only a part of the chip; however, the whole system (which includes a single protocol) occupies the entire chip. Hence, only one protocol can be active on a single chip at a time.

For Case 1, only one protocol (A or B or C or D) out of four (together with the rest of the system) would fit into the chip at a time, such that the chip is reconfigured from A→B→C→D. This example is applicable when using the chip for different operating conditions. In this case, partial reconfiguration might be the most efficient reconfiguration method to use, whereas single context and MultiBoot might not be suitable, and multi context might not be workable. The rationales are discussed below.

A. Partial Reconfiguration for Applications that Require Adaptation to Different Standards and Operating Conditions

In this case, the functional units for each protocol are designed as partial reconfigurable modules. With partial reconfiguration, we should be able to reconfigure these functional units to have different protocols, without compromising the integrity of the application running in the remainder of the chip. Also, we can ensure proper and continuous communication between the functional units and the rest of the system during and after reconfiguration [39],[77],[112].

With partial reconfiguration, the chip will initially consist of the functional unit with the first protocol A. Whenever it is necessary to change the protocol, the partial configuration bitstream for the next protocol is downloaded to that area of the chip and the chip is reconfigured to the next required protocol, without interrupting the system operation as well as interfacing with the rest of the system. By changing only portion of

the chip, as opposed to reconfiguring the entire chip, reconfiguration time overhead is reduced [92],[93].

B. Single Context and MultiBoot for Applications that Require Adaptation to Different Standards and Operating Conditions

Single context or MultiBoot might not be appropriate because both of them require reconfiguring the entire chip even for the smallest changes of a design [77],[82]. This will interrupt the system operation, since the rest of the system is running on the remainder of the chip. As the entire chip has to be reconfigured, it incurs higher reconfiguration time overhead [82],[171].

C. Multi Context for Applications that Require Adaptation to Different Standards and Operating Conditions

Multi context also might not be workable. With multi context, proper communication between operations cannot be guaranteed during and after reconfiguration, which can cause glitches and loss of information [113]. In this case, after reconfiguration from one protocol to another, the interface between the functional units with the protocols and the rest of the system might be compromised.

D. Time and Space Complexity for Partial Reconfiguration for Applications that Require Adaptation to Different Standards and Operating Conditions

Execution times: A is T_a; B is T_b; C is T_c; D is T_d.

Reconfiguration time: A→B is T_{ab}; B→C is T_{bc}; C→D is T_{cd}.

Space occupied: A is S_a; B is S_b; C is S_c; D is S_d.

Space occupied by the rest of the system is: S.

Time Complexity for applications that require adaptation to different standards and operating conditions:

For the partial reconfiguration method the total processing time is as follows:

$$TotalProcessingTime = T_a + T_{ab} + T_b + T_{bc} + T_c + T_{cd} + T_d \quad (31)$$

In this case, partial reconfiguration would be the most efficient method to use since it only reconfigures the area occupied by the protocol, without compromising the integrity of the application running in the remainder of the chip. This method ensures proper and continuous communication between the functional units (occupied by protocols) and the rest of the system during and after reconfiguration. Also, partial reconfiguration allows these protocols to be executed and reconfigured with the least amount of reconfiguration time overhead (measured according to equation (17) from 5.2.5.1), compared to single context or MultiBoot.

Space Complexity for applications that require adaptation to different standards and operating conditions:

For partial reconfiguration the occupying area is as follows:

The extra hardware required on chip for reconfiguration is a constant and is quite close to 1.5% of the chip (from 5.2.5.2). Therefore, in order to have:

- A with the rest of the system on chip at the same time;
- B with the rest of the system on chip at the same time;
- C with the rest of the system on chip at the same time;
- D with the rest of the system on chip at the same time;

The occupying areas of the operations together with the rest of the system must satisfy the following conditions:

$(S_a + S) =< 98.5\%$ and $(S_b + S) =< 98.5\%$ and $(S_c + S) =< 98.5\%$ and $(S_d + S) =< 98.5\%$ of the chip for partial reconfiguration on Virtex 6.

The on-chip area occupied by each operation should be measured to verify that the four operations indeed satisfy the above conditions for partial reconfiguration. If the given area conditions are not satisfied, i.e., if each operation occupies more than 98.5% of the chip, the designer should consider using single context as the next best alternative, since it does not require any extra hardware on chip for reconfiguration. However, unlike partial reconfiguration, with single context, the whole system has to be interrupted when reconfiguring from one protocol to another, since the entire chip has to be reconfigured.

This can be a problem for certain applications, which requires some parts of the system to remain operational. This also leads to higher reconfiguration time overhead compared to partial reconfiguration. Designer should be aware of these issues when selecting the best reconfiguration method for an application that requires adaptation to different standards and operating conditions during its life time.

5.4. Chapter Conclusion and Discussion

In this chapter, we propose a design methodology for FPGA-based dynamic reconfigurable hardware. This design methodology would allow the designers to select the most suitable or efficient reconfiguration method(s) to use in different scenarios, considering computation models, application characteristics, size of the operations, etc. These guidelines are based on the theoretical analysis as well as from our experience (experimental and analytical) on working with FPGA-based dynamic reconfigurable hardware.

Computation models and the intrinsic characteristics of applications play significant roles in determining whether FPGA-based reconfigurable hardware is indeed a good match for the specific application on a portable or embedded platform. Consequently, we investigate computation models and application characteristics that could potentially benefit from this hardware and discuss the how's and why's.

From the investigation on computation models, it is found that certain computation models indeed benefit from FPGA-based reconfigurable hardware. FPGA's dynamic reconfigurability enables processing any number of independent (functional parallel) or dependent (pipeline) operations as well as different architectures (data parallel), regardless of them fitting into the chip at a time. All the computation models presented in this chapter also benefit from partial dynamic reconfigurability of FPGA.

From the investigation on application characteristics, it is shown that applications that require some form of flexibility, reusability, extensibility, upgradeability, etc. can indeed benefit from FPGA-based reconfigurable hardware. Applications that require some form of reprogrammability (either dynamically in real-time fashion or in periods of time such as tri-monthly) can benefit from FPGA's post-fabrication re/programmability, which could lead to increase in reuse and decrease in device cost and gate count.

Next we investigate different FPGA-based reconfiguration methods: single context, multi context, partial reconfiguration, and MultiBoot. We study their features, advantages and disadvantages. Analysis on reconfiguration time and space overhead is also carried out for each reconfiguration method.

From the analysis on reconfiguration time overhead, it is clear that single context (with serial interface) takes the longest time to reconfigure the chip compared to other reconfiguration methods; whereas with multi context, the reconfiguration can be done in a single cycle. The reconfiguration time overhead becomes insignificant when processing large amount of data.

From the analysis on reconfiguration space overhead, it is evident that no extra hardware is required in single context reconfiguration. For MultiBoot and partial reconfiguration, the extra hardware required for reconfiguration remains the same regardless of the size of the reconfigurable parts. Conversely, for multi context, the extra hardware required for on-chip memory increases with the number of inactive contexts. However, having configuration circuitry on chip and having on-chip memory to hold the inactive contexts can significantly enhance the dynamic reconfiguration process; thus trading off chip area for higher speed performance.

Finally, we discuss and present how we map computation models and application characteristics to reconfiguration methods. We discuss the advantages and disadvantages of using different reconfiguration methods and present the most efficient or suitable reconfiguration method(s) to use in different illustrated scenarios considering computation model, application characteristics, size of the operations, etc. Time and space analysis are also presented for each scenario. The most suitable or efficient reconfiguration methods are selected based on their associated advantages and disadvantages, as well as estimating the total processing time, and estimating the areas occupied by operations on chip.

Typically, the device cost of FPGA depends on the density of the gates. One can use a smaller and lower-cost FPGA with less number of gate counts to execute different applications on a single chip regardless of them fitting into the chip at a time. This is a cost-effective solution, if the speed performance requirement of the application(s) is not compromised. The dynamic reconfigurability of FPGAs provides us with acceptable tradeoffs.

Chapter 6

6. Conclusions and Future Work

6.1. Conclusions

With the advancement of mobile and embedded computing, a wide variety of applications are becoming common on these devices. When designing and implementing specific applications on these devices, embedded designers encounter numerous challenges, which include stringent area and power limitations, and reduced cost and time-to-market requirements. Data mining is one of many applications that are becoming common on these devices. Many of today's data mining applications are compute and/or data intensive, requiring more processing power, thus speed performance becomes another design and implementation issue. In order to overcome these challenges it is imperative to incorporate some special-purpose hardware into embedded systems designs.

Firstly, we investigated the feasibility and potential performance gain of using chip-level hardware for data mining operations, and hardware advantages using parallel hardware. After analyzing the advantages and disadvantages of designing and implementing specific applications on hardware over software, and also after an initial proof-of-concept work, we introduced chip-level hardware solutions for three similarity measures and their corresponding similarity matrices. We also introduced a SIMD-type processor array for parallel computation of similarity matrix, and developed an algorithm to assign the computations efficiently to each processing elements (PEs) of the array. From these investigations, we concluded that chip-level hardware support for data mining operations is indeed a feasible and worthwhile endeavour, and speed performance improved significantly. Performance gain is further enhanced by employing hardware optimizations such as parallel processing.

Secondly, we investigated the feasibility of using reconfigurable hardware for data mining operations in portable and embedded devices. Initially, we analyzed the advantages and disadvantages of designing and implementing specific applications on reconfigurable over non-reconfigurable hardware. Although there are tradeoffs associated, reconfigurable computing approach is worth pursuing, mainly because of the limited

hardware foot-print of portable and embedded devices. As a result, we extended our investigation to reconfigurable computing systems. An analysis on FPGA-based over non FPGA-based reconfigurable hardware was also carried out. Next, we introduced and implemented dynamic reconfigurable hardware solutions for two major data mining operations, similarity matrix computation using multiplexer-based approach, and part of the principal component analysis (PCA) using partial reconfiguration method. Additional experiments and analysis were carried out on dynamic partial reconfiguration to ascertain certain issues arisen during our experiments on PCA and to gain more insight into partial reconfiguration method.

With our reconfigurable hardware designs, a significant space saving is achieved; and specifically for large volume of data, the reconfiguration time overhead is insignificant. From the experimental results, it is evident that there is a considerable advantage in implementing our target data mining applications using reconfigurable platform. Although, memory access has an adverse effect on overall performance, there are various techniques to reduce memory access latency as discussed. Further, using various hardware optimization techniques such as parallel processing, complex data mining applications can indeed be implemented in reconfigurable hardware for portable and embedded devices. We concluded that reconfigurable hardware indeed allows the required flexibility and performance to provide chip-level hardware support for data mining applications for portable and embedded computing, while satisfying the area, cost, and time-to-market requirements of these devices.

Thirdly, we formulated a design methodology for FPGA-based dynamic reconfigurable hardware in order to select the most appropriate FPGA-based reconfiguration method(s) to use in different scenarios considering computation models, application characteristics, size of the operations, etc. This is a significant contribution to reconfigurable computing as FPGA-based reconfigurable hardware has many advantages including flexibility, upgradeability, compact circuits and area-efficiency, low-power consumption, shorter time-to-market, and relatively low cost, which are especially important for portable, handheld, and embedded devices. The design methodology is based on theoretical analysis as well as from our experience designing and implementing applications on FPGA-based dynamic reconfigurable hardware. Guidelines are provided to the designers in the

mapping of an application's computation models and characteristics to the most suitable reconfiguration method(s) based on the associated advantages and disadvantages. This design methodology can easily be generalized to other embedded application than data mining applications.

6.2. Future Work

This dissertation has made three significant contributions to process data mining operations efficiently on mobile and embedded devices by providing chip-level and reconfigurable hardware support. However, there are still more work to be done and some challenging issues that need to be addressed. Some of this foreseen work, challenging problems, and potential solutions are discussed and presented below.

6.2.1. Validate Design Methodology

In Chapter 5, we formulated a design methodology for FPGA-based reconfigurable hardware in order to select the most suitable reconfiguration methods to use in different scenarios. In 5.3, we presented how we map the computation models and applications characteristics to the reconfiguration methods. These reconfiguration methods are selected based on their associated advantages and disadvantages, as well as an estimation of the total processing time for each reconfiguration method and area occupied by the operations on chip.

For future work, we intend is to setup experiments to validate our design methodology that the selected reconfiguration methods are indeed the most suitable ones for each and every scenario.

6.2.2. Implement Proposed Techniques to Address Data Transfer Latency

In 4.4.1.2, we proposed several techniques to address the memory access latency issues. Our intention is to design the hardware and set up experiments to implement and incorporate these techniques to reduce data transfer latency.

6.2.3. Reconfigurable Hardware Solution for Last Two Stages of PCA

In Chapter 4, we proposed a reconfigurable hardware solution for the first two stages of PCA. Our intention is to introduce a reconfigurable hardware solution for the whole PCA process. The next step is to design and implement a reconfigurable hardware for the last two stages of the PCA computation: performing eigenvalue/eigenvector analysis, and identifying the set of principal components (PCs) that have the maximum variance on the data set.

As mentioned in Chapter 2, there are different ways of performing eigeanalysis or eigenvalue decomposition (EVD). After studying several different algorithms [2],[71],[121],[133],[157], we selected QR algorithm [121],[157] to be designed and implemented in reconfigurable hardware. It is one of the most efficient and accurate methods to perform eigenanalysis for data sets with a large number of dimensions. We have already designed and implemented the software modules, in C programming language, for the last two stages of the PCA computation.

As future work, we intend to design and implement the last two stages of PCA computation in reconfigurable hardware, to be integrated with the first two stages of this multi-stage computation. Finally, we will propose a reconfigurable hardware solution for the full PCA computation, by incorporating all the stages of PCA, in order to perform them on the same chip by reconfiguring the hardware from one stage to another as needed.

6.2.4. Hardware Optimization

As mentioned in Chapter 4, we did not attempt any hardware optimization for our reconfigurable hardware designs. For the Mean and Covariance Matrix computations, we only used one Mean module and one Covariance Matrix module. Reconfigurable hardware designs for Mean and Covariance modules do not take significant portion of the chip. Hence, we could have multiple modules for Mean and Covariance to perform these operations.

As mentioned earlier, Mean is measured along the dimensions; thus the total number of Mean computations is equal to the number of dimensions (m) in the data set. Covariance Matrix is a square symmetric matrix; hence the total number of Covariance computations are $m*(m+1)/2$, which are the diagonal elements and the elements of the upper triangle. In this case, computing Mean can be done in parallel, along each dimension of the vectors

using *m* number of PEs. Also, each element of the Covariance Matrix can be computed independently from each other using multiple PEs.

In our design, Mean and Covariance Matrix computations are done sequentially. That is: first, data is read from the external memory; second, Mean or Covariance is performed; third, results are written back to the external memory. We could pipeline these stages, in order to overlap read/compute/write operations for better performance.

6.2.5. Library of Components for Handheld Applications

Our intention is to provide a library of components for data mining applications on portable, handheld and embedded computing devices.

As mentioned in Chapter 2, PCA is applied to many data mining applications that are appropriate for portable and embedded devices such as handwritten analysis, signature verification, palm-print or finger-print verification, iris verification, facial recognition, etc. Each of these applications might use different algorithms to perform PCA, as well as different methods to perform other data mining operations such as pattern proximity measure, etc.

By having a library of components for these applications, we can use these components in a plug and play fashion to be processed on a single chip by reconfiguring the hardware on chip from one component to another or one application to another as needed on-the-fly.

Bibliography

[1] Actel Corporation, "Flash FPGAs in the Value-based Market White Paper", Technical Report 55900021-0, Actel, 2005, http://www.actel.com.

[2] Addison, J.F.D, S. Wermter, and G.Z. Arevian, "A Comparison of Feature Extraction and Selection Techniques" In Proc. of Int. Conf. on Artificial Neural Networks (ICANN), pp.212-215, 2003.

[3] Agilent Technologies, Inc., "Principal Component Analysis", 2005.

[4] Alam, S.R., P.K. Agarwal, M.C. Smith, J.S. Vetter, and D. Caliga, "Using FPGA Devices to Accelerate Biomolecular Simulations", IEEE Transaction on Computers, vol.40, no.3, pp.66-73, Mar. 2007.

[5] Alfke, P., "Dynamic Reconfiguration - XAPP093", (Version 1.1), Nov. 1997. http://www.xilinx.com/support/documentation/application_notes/xapp093.pdf.

[6] Algotronix, Inc., www.algotronix.com

[7] Alpaydin, E., and C. Kaynak "Optical Recognition of Handwritten Digits Data Set", UCI Machine Learning Repository.

[8] Altera, Inc., "NOIS II Processor Reference Handbook" May 2011, http://www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf.

[9] Amirix, Inc., www.amirix.com/products.

[10] Annapolis Microsystems, Inc, "Wildfire Reference Manual", Annapolis, MD, 1998.

[11] Arshak, K. and C.S. Ibala, "Using MultiBooting Approach to Reduce the FPGA Device Utilization and to Create a Redundant System", University of Limerick, Ireland.

[12] Asuncion, A. and D.J. Newman, UCI Machine Learning Repository, University of California, Irvine, School of Information and Computer Sciences, 2007, www.ics.uci.edu/~mlearn/MLRepository.html.

[13] Babb, J., M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agrawal, "The RAW Benchmark Suite: Computation Structures for General-Purpose Computing", In Proc. of IEEE Symp. on Field-Programmable Custom Computing Machines, pp.134-143, Apr. 1997.

[14] Bailey, D.G., "Design for Embedded Image Processing on FPGAs", Wiley, 2011

[15] Baker, Z., and V.K. Prasanna, "Efficient Hardware Data Mining with the Apriori Algorithm on FPGAs", in Proc. of 13th Annual IEEE Symp. on Field Programmable Custom Computing Machines, pp. 3-12, Apr. 2005.

[16] Barr, M., "Reconfigurable Computing Primer", Multimedia Systems Design, pp.44-47, Sep. 1998, http://www.netrino.com/Embedded-Systems/How-To/Reconfigurable-Computing, Retrieved in Dec. 2011.

[17] Barroso, L.A, J. Dean, and U. Holzle, "Web Search for a Planet: the Google Strip Architecture", IEEE Micro, pp.22-28, Mar./Apr. 2003.

[18] Baumgarte, V., G. Ehlers, F. May, A. Nuckel, M. Vorbach, and M. Weinhardt, "PACT XPP – A Self Reconfigurable Data Processing Architecture", The Journal of Super Computing, vol.26, no.2, Sep. 2003.

[19] Benkrid, A., K. Benkrid, and D. Crookes, "Design and Implementation of Generic 2D Biorthogonal Discrete Wavelent Transform on an FPGA", in Proc. of IEEE

11th Annual Symp. on Field Programmable Custom Computing Machines, Apr. 2000.

[20] Berkhin, P., "Survey of Clustering Data Mining Techniques", Technical Report, Accrue Software, 2002.

[21] Berthelot, F., F. Nouvel, and D. Houzet, "Partial and Dynamic Reconfiguration of FPGAs: A Top Down Design Methodology for An Automatic Implementation", in Proc. of IEEE Computer Society Annual Sym. on Emerging VLSI Technologies and Architectures, Mar. 2006.

[22] Bodha, C., "Introduction to Reconfigurable Computing: Architectures", Springer, 2007.

[23] Bondalapathi, K. and V. Prasanna, "Reconfigurable Computing Systems", in Proc. of IEEE, vol.90, no.7, pp.1201-1217, July 2002.

[24] Borkar, S., "Design Challenges of Technology Scaling", IEEE Micro, vol.19, no.4, pp.23-29, 1999.

[25] Bradley, P., and U. Fayyad, "Refining Initial Points for k-means Clustering". In Proc. of 15th Int. Conf. on Machine Learning, 1998.

[26] Brown, S., and J. Rose, "Architecture of FPGAs and CPLDs: A Tutorial," IEEE Design and Test of Computers, vol. 13, no. 2, pp. 42-57, 1996.

[27] Callahan, T.J., J.R. Hauser, and J. Wawrzynek, "The Garp Architecture and C Compiler", IEEE Transaction on Computers, vol.33, no.4, pp.62-69, Apr. 2000.

[28] Casselman, S., "Virtual Computing and the Virtual Computer", IEEE Workshop on FPGAs on Custom Computing Machines, Apr. 1993.

[29] Chamberlain, R.D., R.K. Cytron, M.A. Franklin and R.S. Indeck, "The Mercury System: Exploiting Truly Fast Hardware for Data Search", in Proc. of Int. Workshop on Storage Network Architecture and Parallel I/Os, pp.65-72, Sep. 2003.

[30] Chameleon Systems, Inc.

[31] Chang, D., and M. Marek-Sadowska, "Partitioning Sequential Circuits on Dynamically Reconfigurable FPGAs", IEEE Transactions on Computers, vol.48, no.6, pp.565-578, June 1999.

[32] Chapple, M., "Data Mining: An Introduction", http://databases.about.com/od/datamining/a/datamining.htm, Retrieved in Dec. 2011.

[33] Chen, D., and J.M. Rabaey, "A Reconfigurable Multiprocessor IC for Rapid Prototyping of Algorithmic-Specific High-Speed DSP DataPaths", IEEE Journal on Solid-State Circuits, vol.27, no.12, Dec. 1992.

[34] Chen, D., J. Cong, M. Ercegovac, and Z. Huang, "Performance Driven Mapping for CPLD Architectures", in Proc. 9th ACM/SIGDA Int. Symp. Field Programmable Gate Arrays, vol.22, pp.39-47, 2003.

[35] Chen, W., and M. Leeser, "Finite Difference Time Domain: A Case Study using FPGAs", Reconfigurable Computing: The Theory and Practice of FPGA-Based Computing, Chapter 32, Morgan Kaufmann, 2008.

[36] Chong, W., S. Ogata, M. Hariyama, and M. Kameyama, "Architecture of a Multi-Context FPGA Using Reconfigurable Context Memory", in Proc. of 19th IEEE Int. Symp. on Parallel and Distributed Processing, Apr. 2005.

[37] Choudhary, A., R. Narayanan, B. Ozisikyilmaz, and G. Memik, "Optimizing Data Mining Workloads Using Hardware Accelerators", in Proc. of 10th Workshop on Computer Architecture Evaluation using Commercial Workloads, Feb. 2007.

[38] Clarke, P., "Xilinx, ASIC Vendor Talk Licensing" EE Times, June 2001, http://eetimes.com/electronics-news/4102293/Xilinx-ASIC-vendors-talk-licensing, Retrieved Dec. 2011.

[39] Compton, K. and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software", ACM Computing Surveys, (CSUR), vol.34, no.2, pp.171-210, June 2002.

[40] Cong, J., and H. Huang, "Technology Mapping and Architecture Evaluation for k/m-Macrocell-Based FPGAs", ACM Transactions on Design Automation of Electronic Systems, vol.10, no.1, pp.3-23, Jan. 2005.

[41] Corrillo, J.E., and P. Chow, "The Effect of Reconfigurable Units in Superscalar Processors", In Proc. of ACM/SIGDA 9th Int. Symp. on Field Programmable Gate Arrays, pp.141-150, 2001.

[42] Cosoroaba, A., "High-Performance DDR3 SDRAM Interface in Virtex-5 Devices", XAPP867 (v1.2.1), July 2009, http://www.xilinx.com/support/documentation/application_notes/xapp867.pdf, Retrieved Dec. 2011.

[43] Craigen, D. "Embedded Systems", Chapter 2 from "Validation, Verification and Certification of Embedded Systems", NATO Research and Technology Organization, Oct. 2005.

[44] Cronquist, D.C., C. Fisher, M. Figueroa, P. Franklin, and C. Ebeling, "Architectural Design of Reconfigurable Pipelined Datapaths - RaPiD", In Proc. of 20th Conf. On Advanced Research in VLSI, pp.23-40, Mar. 1999.

[45] Data Mining: What is Data Mining? http://www.anderson.ucla.edu/faculty/jason.frand/teacher/technologies/palace/datamining.htm, Retrieved in Dece. 2011.

[46] Data Sets for Data Mining, available at www.inf.ed.ac.uk/teaching/courses/dme/html/datasets0405.html.

[47] DeCoster, J., "Overview of Factor Analysis", 1998, http://www.stat-help.com/notes.html.

[48] Dehon, A., "DPGA Utilization and Application", in Proc. of 4th ACM Int. Symp. on Field Programmable Gate Arrays, pp.115-121, 1996.

[49] DeHon, A., and J. Wawrzynek, "Reconfigurable Computing: What, Why, and Implications for Design Automation", In Proc. of 36th Annual IEEE/ACM Conf. on Design Automation, (DAC'99), pp.610-615, June 1999.

[50] Densmore, D., R. Passerone, and A. Sangiovani-Vincentelli, "A Platform-Based Taxonomy for ESL Design," IEEE Design and Test, pp.359-374, Sept./Oct. 2006.

[51] Ding, C.H.Q, and X. He, "K-means Clustering via Principal Component Analysis" in Proc. of 21st Int. Conf. on Machine Learning, July 2004.

[52] Ding, C.H.Q., and X. He, "Principal Component Analysis and Effective k-means Clustering", in Proc. of SDM, 2004.

[53] Dye, D., "Partial Reconfiguration of Virtex FPGAs in ISE 12", Xilinx Inc., WP374 (v1.0), July, 2010.

[54] Dye, D., "Partial Reconfiguration of Xilinx FPGAs Using ISE Design Suite", WP374 (v1.1), July, 2011.

[55] Eck, V., P. Kalra, R. LeBlanc, and J. McManus, "In-Circuit Partial Reconfiguration of RocketIO Attributes" XAPP662 (v2.4), May, 2004.

[56] Embedded Systems: Technologies and Markets, "Global Market for Embedded Systems worth $112.5 Billion in 2013", Electronic.ca Publications, Apr. 2009, www.pr.com/press-release/148626, Retrieved in Nov. 2011.

[57] Enzler, R., C. Plessl, and M. Platzner, "Co-simulation of a Hybrid Multi-Context Architecture", in Proc. of 3rd Int. Conf. on Engineering of Reconfigurable Systems and Algorithms, (ERSA), pp.174-180, 2003.

[58] Estlick, M., M. Lesser, and J. Theiler, "Algorithmic Transformations in the Implementation of k-means Striping on Reconfigurable Hardware", in Proc. of 9th Annual IEEE Symp. on Field Programmable Custom Computing Machines, pp.103-110, Feb. 2001.

[59] Estrin, G., "Reconfigurable Computer Origins: The UCLA Fixed-Plus-Variable (F+V) Structure Computer", IEEE Annals of the History of Computing, 2002

[60] Everitt, B.S., S. Landau, M. Leese, and D. Stahl, "Cluster Analysis", Wiley, 5th edition, 2011.

[61] Fayyad, U., G.P. Shapiro, and P. Smyth, "From Data Mining to Knowledge Discovery in Databases", In Proc. of American Association for Artificial Intelligence, pp.37-54, July 1997.

[62] Freeman, M., M. Weeks, J. Austin, "Hardware Implementation of Similarity Functions," IADIS Int. Conf. on Applied Computing, 2005.

[63] Fry, T.W, and S. Hauck, "SPIHT image compression on FPGAs" IEEE Transactions on Circuits and Systems for Video Technology, pp.1138-1147, 2005.

[64] Fukunaga, K., "Introduction to Statistical Pattern Recognition", 2nd ed. Academic Press, New York, 1990.

[65] Fushimi, S., and M. Kitsuregawa, "GREO: a Commercial Database Processor Based on a Pipelined Hardware Sorter", in Proc. of ACM SIGMOD Int. Conf. on Management of Data, pp.449-452, May 1993.

[66] Gabrys, B., R.J. Howlett, L.C. Jain, "Knowledge-Based Intelligent Information and Engineering Systems", Springer, 2006.

[67] Garcia, P., K. Compton, M. Schulte, E. Blem and W. Fu, "An Overview of Reconfigurable Hardware in Embedded Systems", EURASIP Journal on Embedded Systems, pp.1-19, 2006.

[68] Gnanabadkaran, A, and K. Duraiswamy, "An Efficient Approach to Cluster High Dimensional Spatial Data Using K-Mediods Algorithm", European Journal of Scientific Research, ISSN 1450-216X, vol.49, no.4, pp.617-624, 2011.

[69] Gokhale, M., W. Holmes, A. Kposer, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti, "Building and Using Highly Parallel Programmable Logic Arrays', Computer, pp.81-89, Jan. 1991.

[70] Goldstein, S.C., H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R.R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler", IEEE Transaction on Computers, vol.33, no.4, pp.70-77, Apr. 2000.

[71] Golub, G.H., and C.F. van Loan, "Matrix Computations", 3rd Edition, John Hopkins University Press, Baltimore, 1996.

[72] Guyon, I., and A. Elisseeff, "An Introduction to Variable and Feature Selection", Journal of Machine Learning Research 3, pp.1157-1182, 2003.

[73] Hamacher, C., Z. Vranesic, and S. Zaky, "Computer Organization", 5th edition, McGraw Hill, 2002.

[74] Han, G.E-H., Karypis, and V. Kumar, "Scalable Parallel Data Mining for Association Rules," IEEE Transactions on Data and Knowledge Engineering, vol.12, no.3, pp.337-352, May/June 2000.

[75] Han, J., H. Cheng, D. Xin, and X. Yan, "Frequent Pattern Mining: Current Status and Future Directions", Data Mining and Knowledge Discovery, Springer, pp.55-86, 2007.

[76] Hand, D.J., H. Mannila, and P. Smyth, "Principles of Data Mining", The MIT Press, Cambridge, 2001

[77] Hauck, S., and A. Dehon, "Reconfigurable Computing: The Theory and Practice of FPGA-Based Computing", Morgan Kaufmann Publishers, 2008.

[78] Hauck, S., T.W. Fry, M.M. Hosler, and J.P. Kao, "The Chimaera Reconfigurable Functional Unit", IEEE Transaction on Very Large Scale Integration (VLSI) Systems, vol.12, no.2, pp.206-217, Feb. 2004.

[79] Hennessay, J.L., and D.A. Patterson, "Computer Architecture – A Quantitative Approach", Morgan Kaufmann, 4th edition, 2007.

[80] Herbordt, M.C., T. VanCourt, Y. Gu, B. Sukhwani, A. Conti. J. Model and D. DiSabello "Achieving High Performance with FPGA-Based Computing", IEEE Transaction on Computers, vol.40, no.3, pp.50-57, Mar. 2007.

[81] Hipp, J., U. Güntzer, and G. Nakhaeizadeh, "Algorithms for Association Rule Mining - A General Survey and Comparison" ACM SIGKDD Explorations, vol.2, no.1, pp.58-64, 2000.

[82] Hussein, J. and R. Patel, "MultiBoot with Virtex-5 FPGAs and Platform Flash XL", XAPP1100 (v1.0), Nov. 2008.

[83] Hussein, J., "Multiple-Boot with Platform Flash PROMs", XAPP483 (v2.0.1), Nov. 2007.

[84] Hyvarinen, A., "A Survey on Independent Component Analysis" Neural Computing Survey, vol.2, pp.94-128, 1999.

[85] IBM, Inc., www.ibm.com

[86] Inoue, U., T. Satoh, H. Hayami, H. Takeda, T. Nakamura, and H. Fukuoka, "Rinda: A Relational Database Processor with Hardware Specialized for Searching and Sorting", IEEE Micro, pp.61-70, Dec. 1991.

[87] Jackson, J.E., "A User's Guide to Principal Components", John Wiley & Sons, Inc, 2003.

[88] Jain, A.K., Murty, M.N., and Flynn, P.J., "Data Clustering: A Review", ACM Computing Surveys, vol.31, no.3, pp.264-323, 1999.

[89] Jin, R., G. Yang, and G. Agrawal, "Shared Memory Parallelization of Data Mining Algorithms: Techniques, Programming Interface, and Performance", IEEE Transactions of Knowledge and Data Engineering, vol.16, no.10, Oct. 2004.

[90] Jolliffe, I.T., "Principal Component Analysis", New York: Springer-Verlag, 2002.

[91] Joshi, M.V., G. Karypis, and V. Kumar, "ScalParc: A New Scalable and Efficient Parallel Classification Algorithms for Mining Large Datasets", In proc. of Int. Symp. on Parallel Processing, pp.573-579, 1998.

[92] Kalte, H., D. Langen, E. Vonahme, A. Brinkmann, and U. Ruckert, "Dynamically Reconfigurable System-on-Programmable-Chip", in Proc. of 10th Euromicro Workshop on Parallel, Distributed and Networked-based Processing (EUROMICOR-PDP'02), pp.235-242, Jan. 2002.

[93] Kao, C., "Benefits of Partial Reconfiguration", Xcell Journal Online, Xilinx Inc., Jan. 2005.

[94] Khawam, S., I. Nousias, M. Milward, Y. Yi, M. Muir, and T. Arslan, "The Reconfigurable Instruction Cell Array", IEEE Transaction on Very Large Scale Integration (VLSI) Systems, vol.16, no.1, pp.75-85, Jan. 2008.

[95] Kim, Y., and R.N. Mahapatra, "Hierarchical Reconfigurable Computing Arrays for Efficient CGRA-based Embedded Systems", In Proc. of 46th IEEE Annual Conf. on Design Automation, pp.826-831, 2009.

[96] Kim, Y., W.N. Street, and F. Menczer, "Feature Selection in Data Mining", University of Iowa, USA.

[97] Kindratenko, V., "High-Performance Computing on FPGAs – Challenges and Opportunities", National Centre for Supercomputing Applications (NCSA).

[98] Koran, Y. and L. Carmel, "Robust Linear Dimensionality Reduction" IEEE Transactions on Visualization and Computer Graphics, vol.10, no.4, pp: 459-470, Aug. 2004.

[99] Kriegel, H.P., P. Kröger, A. Zimek, "Clustering High-Dimensional Data: A Survey on Subspace Clustering, pattern-Based Clustering, and Correlation Clustering", ACM Transactions on Knowledge Discovery from Data, (TKDD), vol.3, no.1, pp.1-58, Mar. 2009.

[100] Kuon, I, and J. Rose, "Measuring the Gap Between FPGAs and ASICs", IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems, vol.26, no.2, pp.203-215, Feb. 2007.

[101] Lai, Y.K., L.F. Chen, J.C. Chen, and C.W. Chiu, "A SIMD-Based Reconfigurable Computing Architecture with Two-Way Pipelined Reconfiguration for Multimedia Applications" In Proc. of 9th Int. Workshop on Cellular Neural Networks and Their Applications, pp.261-264, May 2005.

[102] LaMeres, B.J., and C. Gauer, "Dynamic Reconfigurable Computing Architecture for Aerospace Applications", In Proc. of IEEE Int. Conf. on Aerospace, pp.1-6, Mar. 2009.

[103] Leung, K., M. Ercegovac and R. Muntz, "Exploiting Reconfigurable FPGA for Parallel Query Processing in Computation Intensive Data Mining Applications", UC MICRO Technical Report, Feb. 1999.

[104] Li, K.F. and D.G. Perera, "An Investigation of Chip-Level Hardware Support for Web Mining", in Proc. of IEEE Int. Symp. on Data Mining and Information Retrieval, pp.341-348, May 2007.

[105] Li, K.F., and D.G. Perera, "A Hardware Collective Intelligent Agent", to appear in Springer LNCS Transactions on Computational Collective Intelligence, 2012.

[106] Lim, D., and M. Peattie, "Two Flows for Partial Reconfiguration: Module Based and Small Bit Manipulation", XAPP290, 2002.

[107] Liu, H., and L. Yu, "Towards Integrating Feature Selection Algorithms for Classification and Clustering", IEEE Transactions on Knowledge and Data Engineering, vol.17, no.4, pp.491-502, Apr. 2005.

[108] Lodi, A., C. Mucci, M. Bocchi, A. Cappelli, M. De Dominicis, and L. Ciccarelli, "A Multi-Context Pipelined Array for Embedded Systems", in Proc of Int. Conf. on Field Programmable Logic and Applications, (FPL'06), pp.1-8, Aug. 2006.

[109] Mangione-Smitth, W.H., B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V.K. Prasanna, and H.A.E. Spaanenburg, "Seeking Solutions in Configurable Computing", IEEE Computer, vol.30, no.12, pp.38-43, 1997.

[110] Manning, D.C., Raghvan, P., and Schutze, H., "Introduction to Information Retrieval", Cambridge University Press, 2008.

[111] Masselos, K., and N. Voros, "System Level Design of Reconfigurable Systems-on-Chip", Springer, 1st edition, 2005.

[112] McDonald, E.J., "Runtime FPGA Partial Reconfiguration", IEEE Aerospace and Electronic Systems Magazine, vol. 23, no.7, pp.10-15, July 2008.

[113] Mecker, J., M. Hubber, G. Hettich, R. Constapel, J. Eisenmann, and J. Luka, "Dynamic and Partial FPGA Exploitation", in Proc. of IEEE, vol.95, no.2, pp.438-452, Feb. 2007.

[114] Mei, B., S. Vernalde, D. Verkest, H. De Man and R. Lauwereins, "ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix", Field-Programmable Logic and Applications, LNCS 2778, Springer, 2003.

[115] Micron Technology, Inc., "Double Data Rate SDRAM", Data Sheet, 2000.

[116] Mirsky, E., and A. DeHon, "MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources", In Proc. of IEEE Symp. on FPGA for Custom Computing Machines, pp.157-166, Apr. 1996.

[117] Miyamori, T. and K. Olukutun, "A Quantitative Analysis of Reconfigurable Coprocessor for Multimedia Applications", in Proc. of IEEE Symp. on Field Programmable Custom Computing Machine, pp.12-27, 1998.

[118] Miyamori, T., and K. Olukotun, "REMARC: Reconfigurable Multimedia Array Coprocessor", IEICE Transactions on Information and Systems, vol.E82-D, no.2, pp.389-397, Feb.1999.

[119] Narayanan, R., D. Honbo, G. Memik, A, Choudhary, and J. Zambreno, "An FPGA Implementation of Decision Tree Classification", in Proc. of IEEE Int. Conf. on Design, Automation and Test in Europe (DATE), pp.189-194, Apr. 2007.

[120] Ng, M., and M. Peattie, "Using a Microprocessor to Configure Xilinx FPGA via Slave Serial or SelectMap Mode", XAPP502, v1.5, Dec. 2007.

[121] Olver, P.J., "Orthogonal Bases and the QR Algorithm", University of Minnesota, 2008.

[122] Patterson, D.A., and J.L. Hennessay, and "Computer Organization and Design: The Hardware/Software Interface", Morgan Kaufmann, 3rd edition, 2007.

[123] Pechenizkiy, M., S. Puuronen, and A. Tsymbal, "Feature Extraction for Classification in the Data Mining Process", Int. journal Information Theories and Applications, vol.10. pp.271-278.

[124] Perera, D.G. and K.F. Li, "Hardware Acceleration for Similarity Computation of Feature Vectors", IEEE Canadian Journal for Electrical and Computer Engineering (CJECE), vol.33:1, pp.21-32, Winter 2008.

[125] Perera, D.G. and K.F. Li, "On-Chip Hardware Support for Similarity Measures", in Proc. of IEEE Pacific Rim Int. Conf. on Communication, Computers and Signal Processing, pp.354-358, Aug. 2007.

[126] Perera, D.G. and K.F. Li, "Parallel Computation of Similarity Measures Using an FPGA-Based Processor Array", in Proc. of 22nd Int. Conf. on Advanced Information Networking and Application, pp.955-962, Mar. 2008.

[127] Perera, D.G. and K.F. Li, "Similarity Computation Using Reconfigurable Embedded Hardware", in Proc. of 8th IEEE Int. Conf. on Dependable, Autonomic and Secure Computing, DASC'09, pp.323-329, Dec. 2009.

[128] Perera, D.G., and K.F. Li, "Embedded Hardware Solution for Principal Component Analysis", in Proc. of IEEE Pacific Rim Int. Conf. on Communication, Computers and Signal Processing (PacRim'11), pp.730-735, Aug. 2011.

[129] Perera, D.G., and K.F. Li, "FPGA-Based Reconfigurable Hardware for Compute Intensive Data Mining Applications", In Proc. of 6th IEEE Int. Conf. on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC'11), pp.100-108, Oct. 2011, **(Best Paper Award).**

[130] Pisharath, J., and A. Choudhary. "Design of a Hardware Accelerator for Density Based Clustering Applications", in Proc. of 16th Int. Conf. on Application-specific Systems, Architectures and Processors (ASAP), pp.101-106, July 2005,.

[131] Prasanna, V.K., and G.R. Morris, "Sparse Matrix Computations on Reconfigurable Hardware", IEEE Transaction on Computers, vol.40, no.3, pp. 58-64, Mar. 2007.

[132] Quickturn, A Cadence Company, "Mercury Design Verification System Technology Backgrounder", 1999.

[133] Reddy, K., and T. Herron, "Computing the Eigen Decomposition of a Symmetric Matrix in Fixed-point Arithmetic", in Proc. of 10th Annual Symp. on Multimedia Communication and Signal Processing, 2001.

[134] Salton, G., and M.J. McGill, "Introduction to Modern Information Retrieval", McGraw-Hill, New York, 1983.

[135] Saravan, M., S. Govindan, B. Robatmili, H. Esmaeilzadeh, B. Maher, D. Li, A. Smith, D. Burger, and S.W. Keckler, "Scaling Power and Performance via Processor Composability", Technical Report TR-10-14, Department of Computer Sciences, University of Texas at Austin, Apr. 2010.

[136] Scalera, S.M., and J.R. Vazquez, "The Design and Implementation of a Context Switching FPGA", in Proc. of IEEE Symp. on FPGAs for Custom Computing Machines, pp.78-85, Apr. 1998.

[137] Sedcole, P., B. Blodget, T. Becker, J. Anderson, and P. Lysaght, "Modular Dynamic Reconfiguration in Virtex FPGAs", IEE Computers and Digital Techniques, vol.153, no.3, pp.157-164, May 2006.

[138] Shafer, J.C., R. Agrawal, and M. Mehta, "SPRINT: A Sclable Parallel Classifier for Data Mining", In Proc. of 22nd Int. Conf. On Very Large Data Bases, pp.544-555, 1996.

[139] Sharma, A., and K.K. Paliwal, "Fast Principal Component Analysis Using Fixed Point Algorithm", Pattern Recognition Letters, vol.28, no.10, pp.1151-1155, July 2007.

[140] Shlens, J., "A Tutorial on Principal Component Analysis", Institute on Nonlinear Science, UCSD, Dec. 2005.

[141] Sidhu, R.P.S., A. Mei, V.K. Prasanna, "String Matching on Multicontext FPGAs using Self-Reconfiguration", In Proc. of ACM/SIGDA 7th Int. Symp. on Filed Programmable Gate Arrays, pp.217-226, 1999.

[142] Sima, D., T. Fountain, and P. Kacsuk, "Advance Computer Architectures – A Design Space Approach", Addison-Wesley Longman, 1998.

[143] Simunic, T., K. Mihic, and G. De Micheli, "Optimization of Reliability and Power Consumption in Systems on a Chip", Springer, 2005.

[144] Singh, H., M.H. Lee, G. Lu, F. J. Kurdahi, and N. Bagherzadeh, "MorphoSys: An Integrated Reconfigurable Systems for Data-Parallel and Compute-Intensive Applications", IEEE Transaction on Computers, vol. 49, no.5, pp.465-481, May 2000.

[145] Skliarova, I., and A.B. Ferari, "Reconfigurable Hardware for SAT Solvers: A Survey of System", IEEE Transactions on Computers, 53 (11), Nov. 2004.

[146] Smith, L.I., "A Tutorial on Principal Component Analysis", Cornell University, Feb. 2002.

[147] Standaert, F.X., G. Rouvroy, J.J. Quisquater, and J.D. Legat, "Efficient Implementation of Rijndael Encryption in Reconfigurable Hardware: Improvements and Design Tradeoffs", In Proc. of CHES, pp.334-350, 2003.

[148] StatLib Data Sets Archive, available at lib.stat.cmu.edu/datasets/.

[149] Stitt, G., F. Vahid, and S. Nematbakhsh, "Energy Savings and Speedups from Partitional Critical Software Loops to Hardware in Embedded Systems", ACM Trans. on Embedded Computing System, vol.3, no.1, pp.218-232, Feb. 2004.

[150] Strehl, A., J. Ghosh, and R. Mooney, "Impact on Similarity Measures on Web-page Clustering", In Proc. of Workshop of Artificial Intelligence for Web Search, pp.58-64, July 2000.

[151] Sykes, A.O., "An Introduction to Regression Analysis", The Inaugural Course Lecture.

[152] TechTC – Technion Repository of Text Categorization Datasets, available at techtc.cs.technion.ac.il/.

[153] Telle, N., C.C. Cheung, and W. Luk, "Customizing Hardware Designs for Elliptical Curve Cryptography", Computer Systems: Architectures, Modeling, and Simulation, LNCS 3133, Springer, 2004.

[154] Texas Instrument, Inc., www.ti.com

[155] Thangavelu, M. and R. Raich, "On Liner Dimension Reduction for Multiclass Classification of Gaussian Mixtures", in Proc. of IEEE Int. Conf. on Machine Learning and Signal Processing, pp.1-6, Sep. 2009.

[156] Todman, T.J., G.A. Constantinides, S.J.E. Wilton, O. Mencer, W. Luk, and P.Y.K. Cheung, "Reconfigurable Computing: Architectures and Design

Methods", IEE Computer and Digital Techniques, vol.152, no.2, pp.193-207, Mar. 2005.
[157] Trefethen, L.N., and D. Bau, "Numerical Liner Algebra", SIAM, 1997.
[158] Trimberger, S., D. Carberry, A. Johnson, and J. Wong, "A Time-Multiplexed FPGA", in Proc. of 5th Annual IEEE Symp. on FPGAs for Custom Computing Machines, pp.22-28, Apr. 1997.
[159] UCI KDD Archive, available at kdd.ics.uci.edu/.
[160] UCI Machine Learning Repository, available at archive.ics.uci.edu/ml/.
[161] Vakali, A., "New Direction in Web Data Management", Springer, 2011.
[162] Valarmathie, P., M.V. Srinath, and K. Dinakaran, "An Increased Performance of Clustering High Dimensional Data Through Dimensionality Reduction Technique", Theoretical and Applied Information Technology, vol.5, no.6, pp.731-733, 2005.
[163] van der Putten, P., and M. van Someren, "CoIL Challenge 2000: The Insurance Company Case", Published by Sentient Machine Research, Amsterdam. Also a Leiden Institute of Advanced Computer Science Technical Report 2000-09, kdd.ics.uci.edu/database/tic/tic.html, June 2000.
[164] Vuillemin, J., P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard, "Programmable Active Memories: Reconfigurable Systems Come of Age," IEEE Transactions on VLSI Systems, vol.4, no.1, pp.56–69, Mar. 1996.
[165] Ward, S., "Electrical Engineering", Technology & Engineering, 2007.
[166] Wesselkamper, J., "Fail-Safe MultiBoot Reference Design", XAPP468 (v1.0), Nov. 2008.
[167] West, B., R.D. Chamberlain, and R.S. Indeck, "An FPGA-Based Search Engine for Unstructured Database", in Proc. of 2nd Workshop on Application Specific Processors, pp.25-32, Dec. 2003.
[168] Wirthlin, M.J., and B.L. Hutchings, "A Dynamic Instruction Set Computer", In Proc. of IEEE Symp. on FPGAs for Custom Computing Machines, pp.99-107, Apr. 1995.
[169] Wittig, R.D., and P. Chow, "OneChip: An FPGA Processor with Reconfigurable Logic", In Proc. of IEEE Symp. on FPGAs for Custom Computing Machines, pp.126-135, Apr. 1996.
[170] Xiinx, Inc., "Virtex 6 GTX Transceivers User Guide", UG366 (v2.6), July 2011.
[171] Xilinx, Inc, "AR #7662 - XPLA2: Length of Time for Configuration or Download", www.xilinx.com/support/answers/7662.htm.
[172] Xilinx, Inc. "25 Microchips that Shock the World" IEEE Spectrum, 2009, http://www.xilinx.com/publications/IEEE_Top25Chips_May09_eprint.pdf, Retrieved Dec. 2011.
[173] Xilinx, Inc. "Virtex-II Pro and Virtex-II Pro X Platform FPGA: Complete Data Sheet", DS083 (v5.0), June 2002.
[174] Xilinx, Inc. "XPS Multi-Channel External Memory Controller (XPS MCH EMC) (v3.01a)", DS575, Apr.2010.
[175] Xilinx, Inc., "AR #41169 – MIG 7 Series DDR3 SDRAM – Calculating Efficiency and Effective Bandwidth", Mar. 2011, http://www.xilinx.com/support/answers/41169.htm, Retrieved Dec. 2011.

[176] Xilinx, Inc., "Divider Generator v3.0", DS530, June 2009, www.xilinx.com/support/documentation/ip_documentation/div_gen_ds530.pdf.
[177] Xilinx, Inc., "LogiCORE IP XPS HWICAP (v5.00a)", DS586, July 2010, http://www.xilinx.com/support/documentation/ip_documentation/xps_hwicap.pdf.
[178] Xilinx, Inc., "LogiCORE IP XPS Timer/Counter (v1.02a)", DS573, April, 2010. www.xilinx.com/support/documentation/ip_documentation/xps_timer.pdf.
[179] Xilinx, Inc., "MicroBlaze Processor Reference Guide, Embedded Development Kit – EDK 12.2", UG081 (v11.1), http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_2/mb_ref_guide.pdf.
[180] Xilinx, Inc., "ML605 Hardware User Guide", UG534 (v1.5), Feb. 2011, www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf.
[181] Xilinx, Inc., "Multi Channel OPB SDRAM", (v1.00a), DS492, April 2005, www.xilinx.com/support/documentation/ip_documentation/mch_opb_sdram.pdf.
[182] Xilinx, Inc., "Multi-Port Memory Controller", (MPMC) (v6.01.a), DS643, July, 2010.
[183] Xilinx, Inc., "Partial Reconfiguration User Guide" UG702 (v12.3), Oct. 2010, http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_3/ug702.pdf .
[184] Xilinx, Inc., "PlanAhead User Guide", UG632 (v 11.4), Dec. 2009. http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/PlanAhead_UserGuide.pdf.
[185] Xilinx, Inc., "Platform Studio User Guide – Embedded Developemnt Kit EDK 8.2i.", www.xilinx.com/ise/embedded/edk7_1docs/ps_ug.pdf
[186] Xilinx, Inc., "PLB Double Data Rate (DDR) Synchronous DRAM (SDRAM) Controller", www.xilinx.com/support/documentation/ip_document/plb_ddr.pdf.
[187] Xilinx, Inc., "PLB IPIF (v2.01a) data sheet", www.xilinx.com/support/documentation/ip_documentation/plb_ipif.doc
[188] Xilinx, Inc., "PowerPC 405 Processor Block Reference Guide, Embedded Development Kit", UG018 (v2.4), Jan. 2010, http://www.xilinx.com/support/documentation/user_guides/ug018.pdf.
[189] Xilinx, Inc., "Processor Local Bus" (PLB) v4.6 (v1.04a), - DS531. Apr. 2009.
[190] Xilinx, Inc., "Rocket I/O X Transceiver User Guide", Feb. 2007, www.xilinx.com/support/documentation/user_guides/ug035.pdf.
[191] Xilinx, Inc., "Virtex-6 FPGA Configuration User Guide" UG360 (v3.2) Nov. 2010, http://www.xilinx.com/support/documentation/user_guides/ug360.pdf.
[192] Xilinx, Inc., "XPS multi-channel external memory controller", (XPS MCH EMC) (v3.01a), DS575, Apr. 2010.
[193] Xilinx, Inc., "XPS SYSACE (System ACE) interface controller (v1.01a)", DS583, July 2009, http://www.xilinx.com/support/documentation/ip_documentation/xps_sysace.pdf.
[194] Xilinx, Inc., Platform Studio User Guide: Embedded Development Kit EDK 7.1i, 2005, www.xilinx.com/ise/embedded/edk7_1docs/ps_ug.pdf.
[195] Xilinx, Inc., www.xilinx.com
[196] Yao, J.T., "Sensitivity Analysis for Data Mining", in Proc. of 22nd Int. Conf. of Fuzzy Information Processing Society, pp.272-277, July 2003.

[197] Yeung, K.Y., and W.L. Ruzzo, "Principal Component Analysis for Clustering Gene Expression Data" Bioinformatics, pp.763-774, 2001.

[198] Zha, H., X. He, C.H.Q. Ding., H. Simon, and M. Gu., "Spectral Relaxation for K-means Clustering", Advances in Neural Information Processing Systems, vol.14, 2002.

[199] Zobel, J, and A. Moffat, "Exploring the Similarity Space", ACM SIGIR Forum, vol.32, no.1, pp.18-34, 1998.

Appendix A

A. List of Publications

A.1. Peer Reviewed Conference Papers

1. Darshika G. Perera and Kin Fun Li, "FPGA-Based Reconfigurable Hardware for Compute Intensive Data Mining Applications", in Proceedings of 6th IEEE International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC'11), pp.100-108, October 2011, **(Best Paper Award).**

2. Darshika G. Perera and Kin Fun Li, "Embedded Hardware Solution for Principal Component Analysis", in Proceedings of IEEE Pacific Rim International Conference on Communication, Computers and Signal Processing (PacRim'11), pp.730-735, August 2011.

3. Darshika G. Perera and Kin Fun Li, "Similarity Computation Using Reconfigurable Embedded Hardware", in Proceedings of 8th IEEE International Conference on Dependable, Autonomic and Secure Computing, DASC'09, pp.323-329, December 2009.

4. Darshika G. Perera and Kin Fun Li, "Parallel Computation of Similarity Measures Using an FPGA-Based Processor Array", in Proceedings of 22nd International Conference on Advanced Information Networking and Application, pp.955-962, March 2008.

5. Darshika G. Perera and Kin Fun Li, "On-Chip Hardware Support for Similarity Measures", in Proceedings of IEEE Pacific Rim International Conference on Communication, Computers and Signal Processing, pp.354-358, August 2007.

6. Kin Fun Li and Darshika G. Perera, "An Investigation of Chip-Level Hardware Support for Web Mining", in Proceedings of IEEE International Symposium on Data Mining and Information Retrieval, pp.341-348, May 2007.

A.2. Peer Reviewed Journal Papers

7. Kin Fun Li and Darshika G. Perera, "A Hardware Collective Intelligent Agent", to appear in Springer LNCS Transactions on Computational Collective Intelligence, 2012.
8. Darshika G. Perera and Kin Fun Li, "Hardware Acceleration for Similarity Computation of Feature Vectors", IEEE Canadian Journal for Electrical and Computer Engineering (CJECE), vol. 33:1, pp.21-32, Winter 2008.

A.3. Application Notes

9. Darshika G. Perera, "Reconfigurable Custom IP using Xilinx Partial Reconfiguration Tools", to be submitted by an invitation from Canadian Microelectronic Corporation (CMC), May 2012.

Appendix B

B. The Evolution of FPGA

B.1. Roadmap of FPGA

As mentioned in Chapter 2, FPGAs were introduced in mid-1980s and have been evolving ever since. The first FPGA, XC2064, was introduced by Xilinx in 1985 [38],[172]. It has programmable gates and interconnects between gates. It consists of 64 configurable logic blocks (CLBs), with two 3-input LUTs. From mid-1980s to early 2000s, the number of gates on FPGAs increased from thousands (of XC2064) to millions (of Virtex series) [195].

In 2004, Xilinx Virtex-II Pro [173] FPGAs based on the 0.13 micro-meter CMOS technology, can be executed at 80 MHz. It consists of advanced features such as embedded processors (PowerPC and MicroBlaze), substantial amount of embedded memory, clock management circuitry, and high-level embedded functions such as multipliers, adders, etc. The chip has 44096 CLBs and has the capability of transferring data at 2.4 to 6.25 Gb/s full-duplex through the Rocket I/O interface with an external hard-drive. An embedded PowerPC processor provides performance up to 400 MHz. These FPGAs can be reconfigured at run-time, leading to the idea of dynamic reconfiguration. Additionally, they allow partial reconfiguration, where some parts of the chip can be reconfigured while other parts are still operational, which is another advanced feature of modern FPGAs.

Today (circa 2012), Xilinx Virtex-6 FPGA [191] provides the newest, most advanced features in the FPGA market. This chip is one of the most efficient in terms of speed performance, power, and area. Virtex-6 is based on 40-nm process technology and can be executed at 600 MHz. It is capable of transferring data at a rate of 11.18 Gb/s (full duplex) and has 88560 CLBs. In addition to having all the advanced features from previous FPGAs, this chip includes high-performance embedded DSP blocks, enhanced features for dynamic and partial reconfiguration, and much more. Advanced 40-nm process, architectures, tools and system level optimizations enable the reduction of core power by

30%. Furthermore, Virtex-6 provides a low-voltage option, which can potentially increase power savings by 50% without significant performance penalties [191].

These features as well as the millions of logic gates that exist in current FPGAs (early 2010s), allow very complex circuits to be implemented on a single chip. FPGAs still have much room to grow, offering flexibility, performance, area, and power improvement over other means of hardware support in various situations.

B.2. FPGA Performance Review

From our discussions, it is evident that FPGAs have numerous advantages that are important for portable and embedded computing. FPGAs are often compared to custom-designed ASICs. FPGAs undoubtedly can outperform ASICs in terms of flexibility, and area-efficiency. However, for applications that require some form of programmability, it is imperative to investigate whether FPGA can provide the required speed performance compared to other means of processing support such as microprocessor, while considering the constraints associated with portable and embedded devices.

Clock rate of FPGA has always been slower than that of CPU's, typically by about a factor of 10 [77],[80]. According to the 2008 study by Hauck and Dehon [77], in order to achieve the same performance as microprocessor, FPGAs must perform "at least 10 times the computational work per cycle" and "to be a compelling alternative an FPGA-based design should exceed the performance of a processor-based design by 5 to 10 times and hence must actually perform 50 to 100 times the computational work per clock cycle". However, this gap has been lessening. For instance, in late 2000s, FPGAs were running at 80 MHz, and state-of-the-art FPGAs in early 2010s are running at 600 MHz. In comparison, CPUs in late 2000s were operating at 1.8 GHz and, and in early 2010s they are operating at 3.8 GHz.

Another performance challenge is the Amdahl's law: A study shows [80] that in order "to achieve the speedup factors required for user acceptance of a new technology (preferably 50 times), at least 98 percent of the target application must lend itself to substantial acceleration". Despite avoiding the overheads associated with instruction fetch/decode/execute and with the operating system (OS) of the microprocessor, this kind

of speed performance is possible only if the application exhibits a significant amount of parallelism [77].

Unlike microprocessors, FPGAs do not have the overhead of instruction cycles and OS, rather it uses special-purpose hardware for processing logic, leading to higher speed performance. Similar to ASICs, with FPGAs, computations are implemented (spatially), distributed throughout the chip, instead of implementing them temporally [23],[49], which allows exploiting parallelism in computations. FPGA-based designs can provide good speed performance by exploiting: parallelism at different levels of granularity (from coarse-grain to fine-grain); data parallelism; and pipelining. This massive amount of on-chip parallelism using custom-circuits is more than enough to compensate for the slower clock rates on FPGAs. These FPGA-based designs can be significantly faster than microprocessor-based (software-only) designs. Additionally, it is possible to keep the reconfiguration time overhead minimal; in some cases reconfiguration can be done in a single cycle [158]. Many groups [145],[153] have demonstrated that FPGA, compared to microprocessor, can achieve significant speed performance in solving problems due to massive parallelism and fine-grained operation using customized hardware. One example [153] is a reconfigurable design of a cryptography operation on FPGA running at 66 MHz that achieves 540 times speedup compared to an optimized software implementation on a dual-Xeon computer running at 2.6 GHz.

The computation models and the characteristics of the applications play significant roles in determining the FPGA's performance [77],[80]. This information could potentially tell us the amount of existing parallelism in an application, whether this parallelism could be harvested, and the clock speed that could be achieved.

In summary, FPGAs provide a flexible, high performance, low power, and area-efficient means of implementing a variety of applications. Typically, FPGAs achieve significant speed performance when applications are implemented with many parallel hardware modules executing simultaneously.

Appendix C

C. Experimental Results and Analysis for SRH and DRH for Adder and Multiplier

In 4.4, we include the summary of the experiments and analysis for hw_v1 (SRH) and hw_v2 (DRH) for Adder and Multiplier. In this Appendix, these experimental results and analysis are presented and discussed in detail.

C.1. Execution Times for Adder and Multiplier

C.1.1. Total Execution Time

C.1.1.1. For hw_v1 (SRH)

We designed and implemented static reconfigurable hardware (SRH) for Adder and Multiplier as separate entities. With the SRH design, a full bitstream of the Adder was downloaded to the chip and the chip was reconfigured only once. To run the Multiplier design, a full bitstream of that design had to be downloaded again and the entire chip had to be reconfigured. System had to be interrupted for every download and reconfiguration process.

Data size		Execution time (plb_clk_cycles)		Time per item (plb_clk_cycles)	
		Adder	Multiplier	Adder	Multiplier
0.1x	24467	1992386	1992402	81.432	81.432
0.2x	48934	3984702	3984703	81.430	81.430
0.3x	73401	5977016	5977018	81.430	81.430
0.4x	97869	7969408	7969410	81.429	81.429
0.5x	122336	9961721	9961720	81.429	81.429
1x	244672	19923361	19923376	81.429	81.429
2x	489344	39846660	39846662	81.429	81.429
3x	734016	59769947	59769945	81.429	81.429
4x	978688	79693255	79693254	81.429	81.429
6x	1468032	119539832	119539831	81.429	81.429
8x	1957376	159386416	159386415	81.429	81.429
10x	2446720	199233007	199233003	81.429	81.432

Table 31 Execution Times for Adder and Multiplier on hw_v1

Experiments were performed separately on Adder and Multiplier SRH designs with varying data sizes and their execution times are shown in Table 31. Times per item for

addition and multiplication operations are also presented in Table 31. These values are obtained by dividing the total time by the data size. Since both the Adder and Multiplier take one clock cycle for their operation, execution times for the adder and multiplier are quite similar as expected.

C.1.1.2. For hw_v2 (DRH)

Using dynamic reconfigurable hardware (DRH) design, a full bitstream with Addition was downloaded. Addition operation was performed. Then the hardware was reconfigured to Multiplication and Multiplication operation was performed. In order to process different data sets, the hardware was again reconfigured to Addition and so on without downloading the full bitstream or without interrupting the system operation. Partial bitstreams for Addition and Multiplication were stored in external non-volatile memory and downloaded to the region of the reconfigurable module (RM) when necessary.

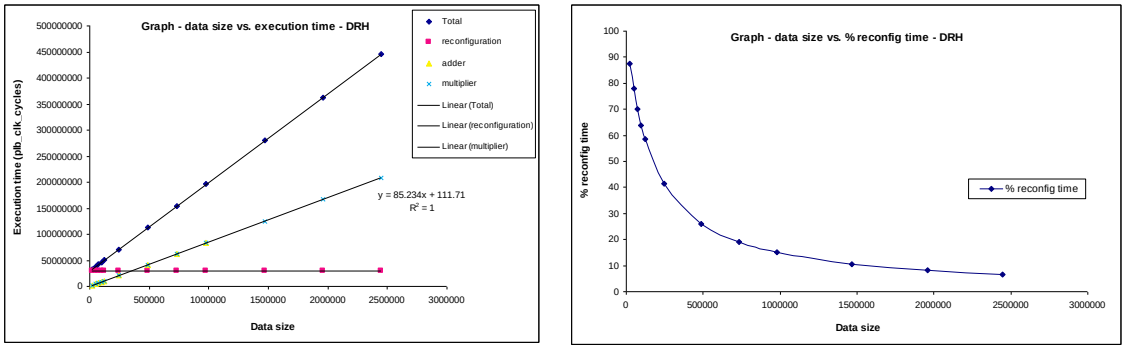
Data size		Execution time (plb_clk_cycles)				Time per item (plb_clk_cycles)	
		Adder	Reconfiguration	Multiplier	Total	Add	Mult
0.1x	24467	2085520	29387176	2085515	33558211	85.238	85.238
0.2x	48934	4170939	29387171	4170937	37729047	85.236	85.236
0.3x	73401	6256351	29387249	6256347	41899947	85.235	85.235
0.4x	97869	8341843	29387201	8341851	46070895	85.235	85.235
0.5x	122336	10427266	29387239	10427261	50241766	85.235	85.235
1x	244672	20854402	29387210	20854407	71096019	85.234	85.234
2x	489344	41708695	29387187	41708700	112804582	85.234	85.234
3x	734016	62562981	29387224	62562978	154513183	85.234	85.234
4x	978688	83417292	29387262	83417298	196221852	85.234	85.234
6x	1468032	125125859	29387230	125125857	279638946	85.234	85.234
8x	1957376	166834503	29387177	166834502	363056182	85.234	85.234
10x	2446720	208543019	29387253	208543020	446473292	85.234	85.234

Table 32 Execution Times for Adder, Reconfiguration, and Multiplier on hw_v2

Unlike hw_v1 (SRH), for hw_v2 (DRH), execution times were measured sequentially: execution of Addition, reconfiguration from Addition to Multiplication, execution of Multiplication. These execution times for different data sizes are presented in Table 32. Times per item for addition and multiplication operations are also presented in Table 32. These values are obtained by dividing the total time by the data size.

It is evident from Table 32 and Figure 33(a) that the reconfiguration time does not vary with the size of the data set. There is a slight variation but it is less than 100 clock cycles

(i.e., 1000 ns). This is expected, since reconfiguration time depends on the size (i.e., area) of the reconfiguration module and not on the number of data being processed.



(a) (b)

Figure 33 (a) Execution Time for Adder/Multiplier (b) Percentage of Reconfiguration Time from Total

As shown in Figure 33(a), execution times for both the adder and multiplier increase linearly with the size of the data set. Also the total execution time of the whole process increase linearly with the size of the data set. This ascertains our expectation that reconfigurable hardware scales linearly with data size. Since both the Adder and Multiplier take one clock cycle for their operation, execution times for the adder and multiplier are quite similar as expected.

In hw_v2 (DRH), for smaller data sets, a significant percentage of total time is spent on reconfiguration. From Figure 33(b), it can be seen that reconfiguration time is amortized and decreases as the size of the data increases.

C.1.2. Breakdown of Execution Time for One Item – hw_v1 & hw_v2

In this section, we performed experiments for hw_v1 (SRH) and hw_v2 (DRH) to distinguish the time taken for the actual additions/multiplications; time taken at the operation count (op_cnt) state; time taken to read an operand from the external memory; and time taken to write a result to the external memory.

In both static and dynamic reconfigurable hardware designs, addition and multiplication operations are controlled by a finite state machine (FSM). The FSM consists of several states:

- First state – checks the operation count (op_cnt), i.e., counts the number of operations or the number of data being processed.
- Second and third states – reads the first operand from the external memory.
- Fourth and fifth states – reads the second operand from the external memory.
- Sixth state – performs the addition operation.
- Seventh and eighth states – writes the results to the external memory.

For static reconfigurable hardware (SRH): op_cnt; op_cnt+add; op_cnt+mult; op_cnt+1read; op_cnt+2reads; op_cnt+1write, were designed and implemented as separate entities and experiments were performed on these entities separately with varying data sizes. For dynamic reconfigurable hardware (DRH): op_cnt; op_cnt+add; op_cnt+mult; op_cnt+1read; op_cnt+2reads; op_cnt+1write, were designed and implemented as different reconfigurable modules (RMs) and experiments were performed on these RMs with varying data sizes by reconfiguring the on-chip hardware from one module to another. The execution times obtained are presented in the following sub sections.

C.1.2.1. Time for One Op_cnt

Data size		Time (plb_clk_cycles)		
		Time for the first state, i.e., op_cnt	Time for one item first state	Additional time taken
0.1x	24467	24540	1.003	73
1x	244672	244749	1.000	77
10x	2446720	2446795	1.000	75

Table 33 Execution Time for the First State (Op_cnt) for hw_v1

Data size		Time (plb_clk_cycles)		
		Time for the first state, i.e., op_cnt	Time for one item first state	Additional time taken
0.1x	24467	24590	1.005	123
1x	244672	244784	1.000	112
10x	2446720	2446854	1.000	134

Table 34 Execution Time for the First State (Op_cnt) for hw_v2

First, we performed experiments using only the first state, which checks the op_cnt. In this case, we designed it in such way that it waits in the same state and checks whether the op_cnt is zero, and decrements the op_cnt until op_cnt is equal to zero. The execution time for the op_cnt is presented in Table 33 and Table 34 for hw_v1 and hw_v2 respectively. As anticipated, the op_cnt only takes one clock cycle (column 4). However,

there is an additional time overhead associated with all the operations. Additional time taken is the time for the processor to send a signal to the hardware to start the computation, plus the time for the hardware to send a signal to inform the processor after finish processing all the data. These are one-time overheads.

C.1.2.2. Time for One Addition/Multiplication

In this case, experiments were performed using the op_cnt state and a second state for the addition or multiplication operation, in order to find out the actual time spent on addition/multiplication operation, without the extra overhead associated with reading/writing data/results from/to external memory. In the first state, it checks whether the op_cnt is zero, goes to the second state, performs the addition or multiplication function and decrements the op_cnt, and goes back to the first state. The execution times for the op_cnt+add and op_cnt+mult are presented in Table 35 and Table 36, for hw_v1 and hw_v2 respectively.

Data size		Time (plb_clk_cycles)					
		Total time first state + add	Total time first state + mult	Per one item first state + add	Per one item first state + mult	Per one add	Per one mult
0.1x	24467	49012	49019	2.003	2.003	1.000	1.000
1x	244672	489427	489430	2.000	2.000	1.000	1.000
10x	2446720	4893519	4893536	2.000	2.000	1.000	1.000

Table 35 Execution Time for Addition and Multiplication for hw_v1

Data size		Time (plb_clk_cycles)					
		Total time first state + add	Total time first state + mult	Per one item first state + add	Per one item first state + mult	Per one add	Per one mult
0.1x	24467	49046	49046	2.005	2.005	1.000	1.000
1x	244672	489460	489460	2.000	2.000	1.000	1.000
10x	2446720	4893548	4893548	2.000	2.000	1.000	1.000

Table 36 Execution Time for Addition and Multiplication for hw_v2

As illustrated in Table 35 and Table 36, first_state+add and first_state+mult each takes only 2 clock cycles, hence addition and multiplication each takes only one clock cycle. This affirms our theoretical estimation for the adder and multiplier computation time.

C.1.2.3. Time for One Read

These experiments were performed using the op_cnt state, and the next two states for one read operation. In the first state, it checks whether the op_cnt is zero, goes to the next states to read one operand from the external memory, decrements the op_cnt, and goes back to the first state. The execution times for the op_cnt+1read are presented in Table 37 and Table 38, for hw_v1 and hw_v2 respectively.

Data size		Time (plb_clk_cycles)					
		Total time first state + one read	Total time first state + one write	Per one item first state + one read	Per one item first state + one write	Per one read	Per one write
0.1x	24467	785788	442723	32.116	18.095	31.113	17.092
1x	244672	7857162	4427434	32.113	18.095	31.113	17.095
10x	2446720	78570896	44274058	32.113	18.095	31.113	17.095

Table 37 Execution Time for One Read and One Write for hw_v1

Data size		Time (plb_clk_cycles)					
		Total time first state + one read	Total time first state + one write	Per one item first state + one read	Per one item first state + one write	Per one read	Per one write
0.1x	24467	808618	440780	33.049	18.015	32.044	17.010
1x	244672	8084961	4406549	33.044	18.010	32.044	17.010
10x	2446720	80848391	44064392	33.044	18.010	32.044	17.010

Table 38 Execution Time for One Read and One Write for hw_v2

C.1.2.4. Time for One Write

These experiments were performed using the op_cnt state, and next two states for one write operation. In the first state, it checks whether the op_cnt is zero, goes to the next states and writes a value to the external memory, decrements the op_cnt, and goes back the first state. The execution times for the op_cnt+1write are also presented in Table 37 and Table 38, for hw_v1 and hw_v2 respectively.

C.1.2.5. Time for Two Consecutive Reads

We performed experiments using the op_cnt state, and next two states for the read operations. In the first state, it checks whether the op_cnt is zero, goes to the next two states and consecutively reads two operands from the external memory, decrements the op_cnt, and goes back to the first state. The execution times for the op_cnt+2reads are presented in Table 39 and Table 40 for hw_v1 and hw_v2 respectively.

For hw_v1, time per read is 31.113 clock cycles when there is one read (Table 37), whereas time per read is 31.167 clock cycles when there are two consecutive reads (Table 39). For hw_v2, time per read is 32.044 clock cycles when there is one read (Table 38), whereas time per read is 32.543 clock cycles when there are two consecutive reads (Table 40). Hence, time per read is higher when there are two consecutive reads.

Data size		Time (plb_clk_cycles)			
		Time for the first state + 2 reads	Time for one item first state + 2 reads	Time for 2 reads	Time for 1 read
0.1x	24467	1549647	63.333	62.333	31.167
1x	244672	15495984	63.333	62.333	31.167
10x	2446720	154959010	63.333	62.333	31.167

Table 39 Execution Time for Two Consecutive Reads for hw_v1

Data size		Time (plb_clk_cycles)			
		Time for the first state + 2 reads	Time for one item first state + 2 reads	Time for 2 reads	Time for 1 read
0.1x	24467	1617073	66.092	65.087	32.543
1x	244672	16169747	66.087	65.087	32.543
10x	2446720	161696422	66.087	65.087	32.543

Table 40 Execution Time for Two Consecutive Reads for hw_v2

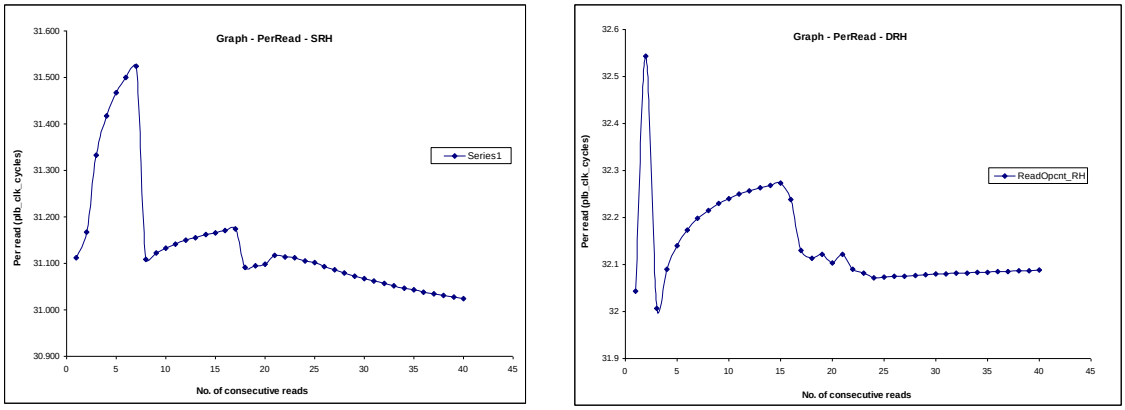
C.1.3. Number of Consecutive Reads vs. Per Read – hw_v1 & hw_v2

Since time per read is different when read once (Table 37 and Table 38) and read twice consecutively (Table 39 and Table 40) for both reconfigurable hardware designs, additional experiments were performed to investigate the number of consecutive reads versus per read time.

Both SRH and the DRH were designed and implemented for op_cnt+n_reads. We performed experiments using the op_cnt state, and the next two states for the read operations. In the first state, it checks whether the op_cnt is zero, goes to the next two states and consecutively reads *n* number of operands from the external memory, decrements the op_cnt, and goes back to the first state. The execution times obtained for the op_cnt+n_reads are shown in Figure 34(a) and Figure 34(b), for hw_v1 and hw_v2 respectively. Although, we performed experiments on varying data sizes, only the results from the largest data size (2446720) are presented here.

As can be seen from Figure 34(a) and Figure 34(b), in general, time per read varies with different number of consecutive reads. For hw_v1 (Figure 34(a)), there is a big

variation for per read time from 7 to 8 consecutive reads, as well as for 17 to 18 consecutive reads. There are some slight variations from 19 to 25 consecutive reads; however, from 26 consecutive reads onwards per read time is decreasing consistently. For hw_v2 (Figure 34(b)), per read time is the highest at 2 consecutive reads. Per read time varies significantly from 1 to 2 consecutive reads, as well as from 2 to 3 consecutive reads. Also, there is a sharp variation for per read time from 15 to 16 consecutive reads. There are some slight variations from 22 to 25 consecutive reads; however, from 26 consecutive reads onwards per read time seems consistent.



(a) (b)

Figure 34 No. Consecutive Reads vs. Per Read Time (a) hw_v1 (b) hw_v2

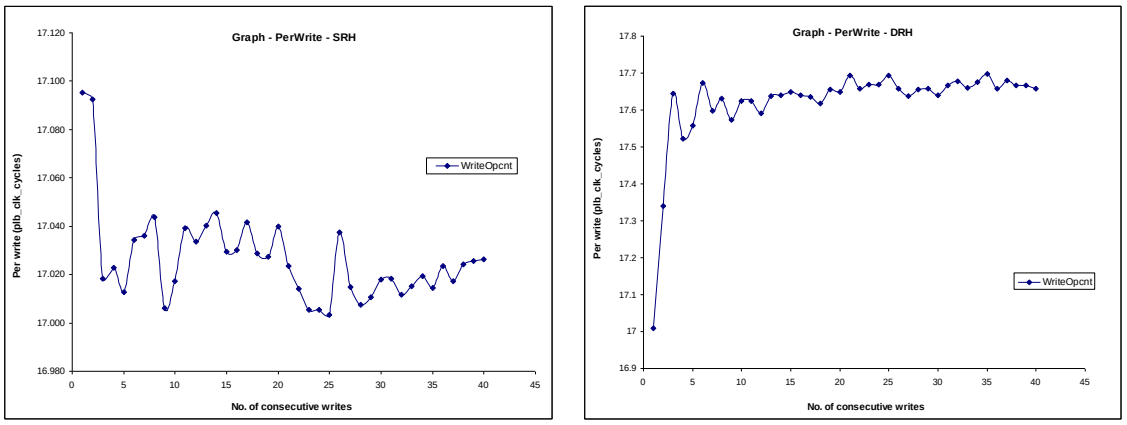
Read operations are performed through PLB bus interfacing with the external memory. Factors such as the asynchronous nature of the read operation, completing read operation just after the rising-edge of the clock which delays the state change, and refresh operations, all have an impact on the data transfer time. Hence, per read time varies for different number of consecutive reads. These reasons are explained in detail in 4.4.1.1.

C.1.4. Number of Consecutive Writes vs. Per Write – hw_v1 & hw_v2

Similar to consecutive reads, additional experiments were performed to investigate the number of consecutive writes versus per write time.

Both the SRH and the DRH were designed and implemented for op_cnt+n_writes. Experiments were performed using the op_cnt state, and the next two states for the write operations. In the first state, it checks whether the op_cnt is zero, goes to the next two states and consecutively write *n* number of values to the external memory, decrements the

op_cnt, and goes back to the first state. The execution times obtained for the op_cnt+n_write are shown in Figure 35(a) and Figure 35(b), for hw_v1 and hw_v2 respectively. Although, we performed experiments on varying data sizes, only the results of data set size 2446720 are presented here.



(a) (b)

Figure 35 No. Consecutive Writes vs. Per Write Time (a) hw_v1 (b) hw_v2

As can be seen from Figure 35(a) and Figure 35(b), in general, time per write varies with different number of consecutive writes. For hw_v1 (Figure 35(a)), initially per write times for 1 and 2 consecutive writes (17.095 and 17.092 clock cycles) are quite high. However, it decreases significantly after 2 consecutive writes and from 3 consecutive writes onwards per write time fluctuates between 17.046 and 17.003 clock cycles. For hw_v2 (Figure 35(b)), initially per write time for 1 consecutive write (17.010 clock cycles) is quite low. However, per write time increases significantly from 1 to 3 consecutive writes. From 3 consecutive writes onwards, per write time fluctuates between 17.693 and 17.522 clock cycles.

Write operations are also performed through PLB bus interfacing with the external memory. Factors such as the asynchronous nature of the write operation, completing write operation just after the rising-edge of the clock which delays the state change, refresh operations, all have an impact on the data transfer time. Hence, per write time varies for different number of consecutive writes. These reasons are explained in detail in 4.4.1.1.

Unlike read operations, with write operations, user_ip (in this case, the reconfigurable module) does not have to wait for the data to be written to the SDRAM. The SDRAM

memory controller allows the write data to be pushed in before or after the address request.

C.1.5. Number of Consecutive Additions/Multiplication vs. Per Addition/Multiplication – hw_v1 & hw_v2

Similar to consecutive reads and writes, additional experiments were performed to investigate the number of consecutive additions/multiplication versus per addition/multiplication time.

Number of consecutive adds/mults = n	Time (plb_clk_cycles)					
	Total time first state + n adds	Total time first state + n mults	Per item (n adds) without first state	Per item (n mults) without first state	Per one add	Per one mult
1	4893535	4893545	1.000	1.000	1.000	1.000
5	14680414	14680425	5.000	5.000	1.000	1.000
10	26914006	26914017	10.000	10.000	1.000	1.000
15	39147612	39147623	15.000	15.000	1.000	1.000
20	51381204	51381215	20.000	20.000	1.000	1.000
25	63614810	63614821	25.000	25.000	1.000	1.000
30	75848402	75848413	30.000	30.000	1.000	1.000
35	88082008	88082019	35.000	35.000	1.000	1.000
40	100315614	100315625	40.000	40.000	1.000	1.000

Table 41 Execution Time for *n* Consecutive Additions/Multiplication for hw_v1

Number of consecutive adds/mults = n	Time (plb_clk_cycles)					
	Total time first state + n adds	Total time first state + n mults	Per item (n adds) without first state	Per item (n mults) without first state	Per one add	Per one mult
1	4893564	4893564	1.000	1.000	1.000	1.000
5	14680458	14680458	5.000	5.000	1.000	1.000
10	26914056	26914056	10.000	10.000	1.000	1.000
15	39147654	39147654	15.000	15.000	1.000	1.000
20	51381252	51381252	20.000	20.000	1.000	1.000
25	63614850	63614850	25.000	25.000	1.000	1.000
30	75848448	75848448	30.000	30.000	1.000	1.000
35	88082046	88082046	35.000	35.000	1.000	1.000
40	100315644	100315644	40.000	40.000	1.000	1.000

Table 42 Execution Time for *n* Consecutive Additions/Multiplication for hw_v2

Both SRH and the DRH were designed and implemented for op_cnt+n_adds and op_cnt+n_mults, separately. We performed experiments using the op_cnt state, next state for the addition/multiplication operations. In the first state, it checks whether the op_cnt is zero, goes to the next states and performs the addition/multiplication operation

consecutively, decrements the op_cnt, and goes back to the first state. The execution times for the op_cnt+n_add and op_cnt+n_mults are presented in Table 41 and Table 42 for hw_v1 and hw_v2 respectively. Although, we performed experiments on varying data sizes, only the results from the largest data size 2446720 are presented here.

From Table 41 and Table 42, it is evident that per addition/multiplication time is only one clock cycle, regardless of the number of consecutive additions/multiplication for both SRH and DRH. This is expected, since addition/multiplication operations are designed to execute in only one clock cycle, and it is synchronous with the system clock.

C.2. Comparison Among Different Cases of hw_v1 and hw_v2

In this section, we made comparison among different cases of hw_v1 (SRH) and hw_v2 (DRH). These cases are as follows:

- Case 1a and Case 2a are referred to as the experiments performed on the whole process, as a single process, which goes through all the stages; i.e., check the op_cnt + read two operands consecutively, perform addition/multiplication operation, and write the results; for hw_v1 (SRH) and hw_v2 (DRH) respectively.
- Case 1b and Case 2b are referred to as the experiments performed on each stage of the above process separately, i.e., the breakdown of execution time per item, for hw_v1 (SRH) and hw_v2 (DRH) respectively.

C.2.1. Case 1b vs. Case 2b – Breakdown of Execution Time Per Item for hw_v1 and hw_v2 for Adder and Multiplier

Operation	Time (plb_clk_cycles)		Difference (plb_clk_cycles)
	For Case 1b	For Case 2b	
Checking op_cnt (Table 33 & Table 34)	1.000	1.000	0.000
2 consecutive reads (Table 39 & Table 40)	62.333	65.087	2.754
Add/mult operation (Table 35 & Table 36)	1.000	1.000	0.000
One write (Table 37 & Table 38)	17.095	17.010	-0.086
Total	81.429	84.097	2.668

Table 43 Case 1b vs. Case 2b

We analyzed the breakdown of execution times for hw_v1 (SRH) and hw_v2 (DRH), from the experiments performed on each stage, i.e., Case 1b vs. Case 2b. These results are presented in Table 43 for hw_v1 (column 2) and hw_v2 (column 3). The results for hw_v1

are obtained from Table 33, Table 35, Table 37, and Table 39. The results for hw_v2 are obtained from Table 34, Table 36, Table 38, and Table 40.

C.2.1.1. Separate Timing for Addition and Multiplication

From Table 43, it is evident that the execution times to check the op_cnt and the execution times for addition/multiplication operations are the same for both the hw_v1 and hw_v2. We performed additional experiments to check whether adder and multiplier indeed take only one clock cycle. For these experiments, addition and multiplication operations were performed consecutively after the op_cnt state for both hw_v1 and hw_ v2 with different data sizes. These results are presented in Table 41 and Table 42 for hw_v1 and hw_v2 respectively. From these results, it is evident that per addition or per multiplication time is only one clock cycle, regardless of the number of consecutive additions or multiplications. This is expected, since addition or multiplication operation (in the design) only takes one clock cycle, and it is synchronous with the system clock.

C.2.1.2. Separate Timing for Read and Write Operations

Again from Table 43, times for 2 consecutive read operations for hw_v1 and hw_v2 are 62.333 and 65.087 clock cycles respectively. These times are significantly different (2.754 clock cycles difference). In order to gain insight about the read operations, additional experiments were performed, in which read operations were performed consecutively after the op_cnt state for both hw_v1 and hw_ v2 with different data sizes. These results are presented in Figure 34(a) and Figure 34(b) for hw_v1 and hw_v2 respectively. From Figure 34(a), Figure 34(b) and Table 44, for hw_v1, time per read varies from 31.024 to 31.524 clock cycles, and for hw_v2, time per read varies from 32.007 to 32.543 clock cycles. In some cases, there are significant variations in the read operations.

Also from Table 43, times for one write operation hw_v1 and hw_v2 are 17.095 and 17.010 clock cycles respectively. These times are slightly different (0.086 clock cycles difference). In order to gain insight about the write operations, additional experiments were performed, in which write operations are performed consecutively after the op_cnt state for both hw_v1 and hw_ v2 with different data sizes. These results are presented in Figure 35(a) and Figure 35(b) for hw_v1 and hw_v2 respectively. From Figure 35(a), Figure 35(b), and Table 44, for hw_v1, time per write varies from 17.003 to 17.095 clock

cycles; and for hw_v2, time per write varies from 17.010 to 17.693 clock cycles. In case of hw_v2, the per write time variation is more significant than in hw_v1, especially after one write.

No. of reads	Time per read		Per read difference (v2-v1)	Time per write		Per write difference (v2-v1)	2 reads + 1 write difference total
	hw_v1	hw_v2		hw_v1	hw_v2		
1	31.113	32.044	0.931	17.095	17.010	-0.086	1.776
2	31.167	32.543	1.377	17.093	17.340	0.248	3.001
3	31.333	32.007	0.674	17.018	17.643	0.625	1.973
4	31.417	32.090	0.674	17.023	17.522	0.499	1.847
5	31.467	32.140	0.674	17.013	17.557	0.544	1.892
6	31.500	32.174	0.674	17.034	17.674	0.639	1.987
7	31.524	32.198	0.674	17.036	17.598	0.561	1.909
8	31.108	32.215	1.108	17.044	17.632	0.588	2.803
9	31.122	32.229	1.108	17.006	17.574	0.568	2.783
10	31.133	32.240	1.108	17.017	17.624	0.607	2.822
11	31.142	32.250	1.108	17.039	17.625	0.586	2.801
12	31.150	32.257	1.108	17.034	17.591	0.557	2.772
13	31.156	32.264	1.108	17.040	17.639	0.598	2.814
14	31.161	32.269	1.108	17.046	17.640	0.594	2.810
15	31.166	32.274	1.108	17.029	17.649	0.620	2.835
16	31.170	32.238	1.067	17.030	17.639	0.609	2.743
17	31.174	32.129	0.955	17.042	17.635	0.593	2.504
18	31.092	32.114	1.022	17.029	17.619	0.590	2.634
19	31.095	32.121	1.026	17.027	17.656	0.628	2.681
20	31.098	32.104	1.006	17.040	17.649	0.609	2.621
21	31.116	32.122	1.005	17.023	17.693	0.669	2.680
22	31.113	32.090	0.977	17.014	17.658	0.644	2.597
23	31.112	32.082	0.970	17.005	17.668	0.663	2.604
24	31.106	32.071	0.965	17.006	17.669	0.664	2.594
25	31.101	32.073	0.971	17.003	17.693	0.690	2.632
26	31.094	32.074	0.981	17.038	17.657	0.620	2.581
27	31.086	32.076	0.990	17.015	17.638	0.623	2.602
28	31.079	32.077	0.998	17.007	17.655	0.647	2.643
29	31.073	32.078	1.005	17.011	17.659	0.648	2.659
30	31.067	32.079	1.012	17.018	17.641	0.623	2.648
31	31.061	32.080	1.019	17.018	17.667	0.648	2.686
32	31.056	32.081	1.025	17.012	17.677	0.666	2.716
33	31.051	32.082	1.031	17.015	17.661	0.646	2.708
34	31.047	32.083	1.037	17.019	17.676	0.656	2.730
35	31.042	32.084	1.042	17.015	17.697	0.683	2.766
36	31.038	32.085	1.047	17.024	17.657	0.633	2.727
37	31.034	32.086	1.051	17.017	17.680	0.663	2.766
38	31.031	32.086	1.056	17.024	17.666	0.642	2.753
39	31.027	32.087	1.060	17.026	17.666	0.641	2.761
40	31.024	32.088	1.064	17.026	17.658	0.632	2.760

Table 44 Read Time Difference and Write Time Difference

From these results, it is evident that per read/write time varies with different number of consecutive reads/write for both hw_v1 and hw_v2. These variations are due to

asynchronous nature of these operations, state changing before the operation is complete, and refresh operations. Explanations for these phenomena are presented in 4.4.1.1.

From Table 44, per read time difference (between hw_v1 and hw_v2) varies from 0.674 to 1.377 clock cycles; and per write time difference (between hw_v1 and hw_v2) varies from 0.086 to 0.690 clock cycles. This difference might be due to the fact that hw_v2 is being reconfigured dynamically, whereas hw_v1 is not. For instance, these two hardware versions were designed and implemented using different design flows and the interfacing of dynamic reconfigurable hardware with the rest of the system is different from the interfacing of the static reconfigurable hardware with the rest of the system. These might cause the difference in execution times between hw_v1 and hw_v2. These reasons are explained in detail in 4.4.1.1.

From the last column of Table 44, the maximum value for the total difference for 2 consecutive reads and one write is 3.001 clock cycles. From Table 43, the total difference is 2.668 clock cycles. This is less than the maximum value.

From the above details, it is evident that the read/write operation has a significant impact on the execution time of the whole process of the addition/multiplication. It is also evident that reconfiguration somehow affects the timing of data transfer operations. This contributes to the difference in execution time (of the whole process of addition/multiplication) of hw_v1 and hw_v2.

C.2.2. Case 1a vs. Case 1b –Per Item Execution Time for hw_v1 for Adder and Multiplier

We analyzed the per item execution times for hw_v1, static reconfigurable hardware (SRH), from the experiments performed for the whole process (Case 1a) vs. experiments performed on each stage (Case 1b). These results are presented in Table 45. The results for hw_v1 are obtained from Table 31, Table 33, Table 35, Table 37, and Table 39.

From Table 45, the total time per item for the whole process (from Case 1a – first row) is equal to the summation of the execution times for separate stages (from Case 1b – last row). In both case, total time per item is 81.429 clock cycles.

This also explains the additional overhead of 0.429 clock cycles (from Table 46). From the simulation and timing diagrams, one read operation takes 31 clock cycles and one

write operation takes 17 clock cycles; hence both addition and multiplication have an overhead of 0.429 clock cycles. This overhead is not due to the actual addition/multiplication operation, nor due to the state that checks the op_cnt, but due to read and write operations.

Description	Time (plb_clk_cycles)
Case 1a	
From Table 31, execution time per item for the whole process of add/mult	81.429
Case 1b	
Breakdown of execution time - for one item (for both addition and multiplication)	
From Table 33, checking the op_cnt only takes 1 cycle	1.000
From Table 39, number of cycles for two consecutive reads	62.333
From Table 35 addition or multiplication only takes 1 cycle	1.000
From Table 37, number of cycles for 1 write	17.095
Total time (from summation of separate execution times)	81.429

Table 45 Case 1a vs. Case 1b – Per Item Execution Time for hw_v1

Also, with hw_v1 (SRH), execution time for the whole process is consistent, i.e., the time for the whole process is equal to the summation of execution times for separate stages.

Description	Time (plb_clk_cycles)
From Table 39, additional cycles for two consecutive reads = (31.167 – 31) * 2 =	0.333
From Table 37, additional cycles for 1 write = 17.095 – 17 =	0.095
Hence, additional cycles for (2 reads + 1 write) = 0.333 + 0.095 =	0.429

Table 46 Additional Overhead for hw_v1

C.2.3. Case 2a vs. Case 2b –Per Item Execution Time for hw_v2 for Adder and Multiplier

We analyzed the per item execution times for hw_v2, dynamic reconfigurable hardware (DRH), from the experiments performed for the whole process (Case 2a) vs. experiments performed on each stage (Case 2b). These results are presented in Table 47. The results for hw_v2 are obtained from Table 32, Table 34, Table 36, Table 38, and Table 40.

From Table 47, unlike hw_v1, for hw_v2 (DRH), the total time per item for the whole process (from Case 2a – first row) is not equal to the summation of the execution times for separate stages (from Case 2b – last row). The execution time difference between Case 2a and Case 2b is 1.137 clock cycles.

From the above details, it is evident that unlike hw_v1 (SRH), the timings for the Case 2a and Case 2b are not the same. In Case 2a, the total time for the whole process for the addition/multiplication is measured after the reconfiguration of that module. In Case 2b, times for each stage are also taken after the reconfiguration of each and every module. From previous analysis with Table 44, it is evident that dynamic reconfiguration affects the timing of read and write operations. This might cause the difference in execution times per item for the above two case in hw_v2. Explanations for these phenomena are presented in 4.4.1.1.

Description	Time (plb_clk_cycles)
Case 2a	
From Table 32, execution time per item for the whole process of add/mult	85.234
Case 2b	
Breakdown of execution time - for one item (for both addition and multiplication)	
From Table 34, checking the op_cnt only takes 1 cycle	1.000
From Table 40, number of cycles for two consecutive reads	65.087
From Table 36, addition or multiplication only takes 1 cycle	1.000
From Table 38, number of cycles for 1 write	17.010
Total time (from summation of separate execution times)	84.097

Table 47 Case 2a vs. Case 2b – Per Item Execution Time for hw_v2

C.2.4. Case 1a vs. Case 2a –Per Item Execution Time for hw_v1 and hw_v2 for Adder and Multiplier

Description	Time (plb_clk_cycles)
Case 1a	
From Table 31, execution time per item for the whole process of add/mult	81.429
Case 2a	
From Table 32, execution time per item for the whole process of add/mult	85.234
Execution time difference	3.805

Table 48 Case 1a vs. Case 2a – Per Item Execution Time for hw_v1 and hw_v2

We analyzed the total per item execution times for hw_v1, static reconfigurable hardware (SRH), and for hw_v2, dynamic reconfigurable hardware (DRH), from the experiments performed on the whole process, i.e., Case 1a vs. Case 2a. These results are presented in Table 48. The results for hw_v1 are obtained from Table 31, and the results for hw_v2 are obtained from Table 32.

For Case 2a for hw_v2, the total time for the whole process for the addition or multiplication is measured after the reconfiguration of that module. However, for Case 1a

for hw_v1 this is not the case. From Table 44, it is evident that dynamic reconfiguration somehow affects the timing of read and write operations. Due to different design flows, interfacing to the rest of the system being different, etc. might cause the difference in execution times per item (of the whole process of addition/multiplication) between hw_v1 and hw_v2. Explanations for these phenomena are discussed in 4.4.1.1.

From Table 48, execution time difference between hw_v1 and hw_v2 for per item addition or multiplication is 3.805 clock cycles. From our previous analysis, we know that this difference is due to the read/write operations, specifically, 2 consecutive reads and one write. However, in Case 2b, from Table 44 (last column), the maximum value for the total difference for 2 consecutive reads and one write between hw_v1 and hw_v2 is 3.001 clock cycles. In Table 44, in Case 2b, the 2 consecutive reads are reading from the same address locations, whereas in Case 2a (Table 48), the 2 consecutive reads are reading from 2 separate address locations. Address patterns can also have a significant impact on the overhead. As mentioned in [175], "sequentially bursting across a row has little or no overhead, whereas, a random address pattern results in high overhead due to Activate and Pre-charge times". These are elaborated in 4.4.1.1.

Further, in Case2b (Table 44), the reads are done separately, as well as the writes. In Case2a (Table 48), 2 consecutive reads, addition/multiplication, and then a write operations are performed in a sequence. This sequence is followed to process the whole data set. With regard to the SDRAM, the traffic patterns (both the command and the address patterns) would have an impact on the read/write operation. Hence, the maximum difference in Table 44 (3.001 clock cycles) is less than the difference in Table 48 (3.802 clock cycles). How and why the traffic patterns affect these operations are discussed in detail in 4.4.1.1.