NORTHWESTERN UNIVERSITY

Configuration Management Techniques for Reconfigurable Computing

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

For the degree

DOCTOR OF PHILOSOPHY

Field of Computer Engineering

By

Zhiyuan Li

EVANSTON, ILLINOIS

June 2002

ABSTRACT

Configuration Management Techniques for Reconfigurable Computing

Zhiyuan Li

Reconfigurable computing is becoming an important part of research in computer architectures and software systems. By placing the computationally intense portions of an application onto the reconfigurable hardware, that application can be greatly accelerated. Gains are realized because reconfigurable computing combines the benefits of both software and ASIC implementations. However, the advantages of reconfigurable computing do not come without a cost. By requiring multiple reconfigurations to complete a computation, the time to reconfigure the hardware significantly degraded performance of such systems. This thesis examines a complete strategy that attacks this reconfiguration bottleneck from different perspectives. Compression techniques are introduced to decrease the amount of configuration data that must be transferred to the system. Configuration caching approaches are investigated to retain configurations on-chip. Configuration prefetching techniques are developed to hide reconfiguration latency. Reconfiguration overhead is virtually eliminated by using these techniques.

Acknowledgements

I am greatly indebted to my thesis advisor, Professor Scott A. Hauck. Scott has provided me with countless opportunities to learn and to research that I doubt I would have found otherwise. I am thankful to Scott for his guidance throughout this work, and for showing me how to perform research. I am also thankful to Scott for providing me excellent facilities and financial support throughout my studies. Thanks also to the members of my thesis committee at Northwestern University, Professors Prithviraj Banerjee and Lawrence Henschen.

I would like to thank all the members of the ACME group in University of Washington and Northwestern University, from whom I have learned so much. They not only put up with me all of those years, but also made my days as graduate student enjoyable. I am grateful for their valuable suggestions and discussions. In particular, I wish to thank Katherine Compton, Melany Richmond and others who contributed along the way.

This work has benefited from various researchers. I would like to acknowledge following people for providing benchmarks and feedbacks for this work: Professor Gordon Brebner, Dr. Timothy Callahan, Dr. Steve Trimberger, Dr. Andreas Dandalis, Professor Carl Ebeling and many others. I would also like to acknowledge financial support provided the Defense Advanced Research Projects Agency and Xilinx Inc..

I wish to thank my parents for all their love, support and encouragement during my graduate studies. There were many struggles and accomplishments along the way, but through them all, they were always there.

Contents

List of Figures

List of Tables

Chapter 1 Introduction **1**

Chapter 2 Background and Motivation **6**

2.1 Reconfigurable Computing Systems 6

2.2 Reconfiguration Models 9

2.2.1 System Example—Garp 13

2.2.2 System Example—DISC 15

2.3 Reconfiguration Overhead Reduction Techniques 16

2.3.1 Configuration Cloning 16

2.3.2 Configuration Sharing 17

2.3.3 Configuration Scheduling 18

2.4 Research Focus 19

Chapter 3 Configuration Compression **22**

3.1 General Data Compression Techniques 22

3.2 Configuration Compression Overview 23

3.3 Configuration Compression Vs. Data Compression 24

3.4 Compression for the Xilinx 6200 FPGAs 26

3.4.1 Algorithm Overview 28

3.4.2 The First Stage of the Algorithm 29

3.4.3 Wildcarded Address Creation via Logic Minimization 32

3.4.4 The Second Stage of the Compression Algorithm 36

3.4.5 Experimental Results 40

3.5 Compression for the Xilinx Virtex FPGAs 43

3.5.1 Algorithms Overview 44

3.5.2 Regularity Analysis 45

3.5.3 Symbol Length 46

3.5.4 Huffman coding 47
3.5.5 Arithmetic Coding 48
3.5.6 Lempel-Ziv Based Compression 50
3.5.7 The Readback Algorithm 54
3.5.8 Active Frame Reordering Algorithm 59
3.5.9 Fixed Frame Reordering Algorithm 61
3.5.10 Wildcarded Compression for Virtex 61
3.5.11 Simulation Results 63
3.5.12 Hardware Costs 67
3.6 Related Works 68
3.7 Summary 68
Chapter 4 Don't Care Discovery for Configuration Compression 70
4.1 Don't Cares 70
4.2 The Backtracing Algorithm 72
4.3 Don't Care Discovery for the Xilinx 6200 FPGAs 74
4.3.1 Don't Care Discovery Algorithm 74
4.3.2 The Modification of the Compression Algorithm 78
4.3.3 Experimental Results 81
4.4 Virtex Compression with Don't Cares 83
4.5 Summary 84
Chapter 5 Configuration Caching 86
5.1 Configuration Caching Overview 86
5.2 Reconfigurable Models Review 88
5.3 Experimental Setup 90
5.4 Capacity Analysis 91
5.5 Configuration Sequence Generation 92
5.6 Configuration Caching Algorithms 93
5.7 Single Context Algorithms 94
5.7.1 Simulated Annealing Algorithm for Single Context FPGA 94
5.7.2 General Off-line Algorithm for Single Context FPGA 96

5.8 Multi-Context Algorithms 98
5.8.1 Complete Prediction Algorithm for Multi-Context FPGA 98
5.8.2 Least Recently Used (LRU) Algorithm for Multi-Context 101
5.9 Algorithms for the PRTR FPGAs 101
5.9.1 A Simulated Annealing Algorithm for the PRTR FPGA 102
5.9.2 An Alternate Annealing Algorithm for the PRTR FPGA 103
5.10 Algorithms for the PRTR R+D Model 104
5.10.1 A Lower-bound Algorithm for the PRTR R+D FPGA 105
5.10.2 A General Off-line Algorithm for the PRTR R+D FPGA 106
5.10.3 LRU Algorithm for the PRTR R+D FPGA 107
5.10.4 Penalty-oriented Algorithm for the PRTR R+D FPGA 107
5.11 A General Off-line Algorithm for the Relocation FPGA 108
5.12 Simulation Results and Discussion 109
5.13 Summary 112
Chapter 6 Configuration Prefetching 114
6.1 Prefetching Overview 114
6.2 Factors Affecting the Configuration Prefetching 117
6.3 Configuration Prefetching Techniques 118
6.4 Configuration Prefetching for Single Context Model 119
6.4.1 Experiment Setup 120
6.4.2 Cost Function 122
6.4.3 The Bottom-up Algorithm for Prefetching 124
6.4.4 Loop Detection and Conversion 128
6.4.5 Prefetch Insertion 130
6.4.6 Results and Analysis 130
6.5 Configuration Prefetching for Partial R+D FPGA 132
6.5.1 Static Configuration Prefetching 133
6.5.2 Dynamic Configuration Prefetching 138
6.5.3 Hardware Requirements of Dynamic Prefetching 141
6.5.4 Hybrid Configuration Prefetching 142

6.5.5 Results and Analysis....................145
6.6 Summary.................... 147
Chapter 7 Conclusions....................148
7.1 Summary of Contributions.................... 149
7.2 Future work.................... 151
References 153
Appendix A 161

List of Figures

2.1 Architecture of a reconfigurable computing system 6
2.2 A programming bit for SRAM FPGAs 7
2.3 A two-input look-up table 8
2.4 The structure of a Single Context FPGA 9
2.5 A four-context FPGA 10
2.6 The structure of a Partial Run-time Reconfigurable FPGA 11
2.7 The architecture of the Relocation + Defragmentation model 12
2.8 An example of configuration relocation 12
2.9 An example of defragmentation 13
2.10 Block diagram Garp 14
2.11 Block diagram of DISC system 16
2.12 An example of Configuration Cloning 17
2.13 Execution using configuration caching approach 19
2.14 Execution using data caching approach 19
3.1 The flow of compression 23
3.2 XC6216 simplified block diagram 26
3.3 Example of the transformation of 2-level logic minimization into the simplified configuration compression problem 30
3.4 Example for demonstrating the potential for configuration compression 31
3.5 Example of the use of Don't Cares in configuration compression 32
3.6 Espresso input (a), and the result output (b) 33
3.7 An example that illustrates the reason for selecting bigger groups 34
3.8 An example of Wildcard reduction 36
3.9 Graph of compressed file size as a percentage of original file size 42
3.10 Virtex architecture 43
3.11 Virtex frame organization 44

3.12 An example of Huffman coding 48
3.13 An example of Arithmetic coding 49
3.14 The LZ77 sliding window compression example 52
3.15 The hardware model for LZ77 compression 53
3.16 Example to illustrate the benefit of readback 55
3.17 Seeking optimal configuration sequence 56
3.18 An example of memory sharing 58
3.19 An example to illustrate Memory Requirement Calculation algorithm 59
3.20 An example of inter-frame compression using addressable FDR 62
3.21 The simulation results for 6-bit symbol 65
3.22 The simulation results for 9-bit symbol 65
3.23 Unaligned regularity between frames 66
4.1 Sample circuit for backtracing 73
4.2 Xilinx 6200 function unit and cell routing 75
4.3 The Xilinx 6200 North switch at 4×4 block boundaries 76
4.4 Experimental results of the compression algorithms 83
4.5 The effect of Don't Cares on benchmarks in Table 3.2 for Virtex compression 84
5.1 An example illustrating the effect of defragmentation 90
5.2 An example to illustrate the General Off-line algorithm for Single Context FPGAs 97
5.3 Reconfiguration overheads of the Single Context FPGA, the PRTR and the Multi-Context models. 110
5.4 Reconfiguration overheads for the Relocation and the PRTR R+D FPGA 111
5.5 Comparison between the PRTR with Relocation + Defragmentation model and the Multi-Context model 112
6.1 An example for illustrating the ineffectiveness of the directed shortest-path algorithm 122
6.2 The control flow graph for illustrating the const calculation 123

6.3 An example of multiple children nodes reaching the same configuration 127

6.4 Loop conversion 129

6.5 Example of prefetching operation control 134

6.6 An example of prefetch scheduling and generation after the probability calculation 136

6.7 The Markov model generated from access string A B C D C C C A B D E 138

6.8 A table is used to represent the Markov graph 141

6.9 An example illustrates the ineffectiveness of the dynamic prefetching 143

6.10 Reconfiguration overhead comparison 146

6.11 Effect of the replacement algorithms for the static prefetching 147

List of Tables

The results of the compression algorithm on benchmark circuits............. 41

Information for Virtex benchmarks.. 64

4.1 The results of the compression algorithms....................................... 82

6.1 Results of the prefetching algorithm 132

6.2 Probability calculation for Figure 6.6... 136

Chapter 1

Introduction

As we approach the era in which a single chip can hold more than 100 million transistors, current general-purpose processor systems will not reach their full potential despite the great flexibility they can provide. On the other hand, *application specific integrated circuits* (ASICs) achieve exceptionally high performance by targeting every application on custom circuitry. However, no one can afford to design and implement a custom chip for every application because of the enormous expense.

Reconfigurable computing systems have become an alternative to fill the gap between ASICs and general-purpose computing systems. Although the basic concept was proposed in the 1960s [Estrin63], reconfigurable computing systems have only recently become feasible. This is due to the availability of high-density VLSI devices that use programmable switches to implement flexible hardware architectures.

Most reconfigurable systems consist of a general-purpose processor, tightly or loosely coupled with reconfigurable hardware. These systems can implement specific functionality of applications on reconfigurable hardware rather than on the general-purpose processor, providing significantly better performance. The general-purpose processor in such systems no longer provides the major computational power; rather it mainly performs tasks such as data collection and synchronization. Though the performance of reconfigurable computing systems on a specific application is not as high as on an ASIC, their promise to deliver flexibility along with high performance has attracted a lot of attention. Moreover, in recent years such system can achieve high

performance for a range of applications, such as image processing [Huelsbergen97], pattern recognition [Rencher97], and encryption [Elebirt00, Leung00].

Field programmable gate arrays (FPGAs) [Brown92] or FPGA-like devices are the most common hardware used for reconfigurable computing. A FPGA contains an array of computational elements whose functionality is determined through multiple SRAM configuration bits. These elements, also known as logic blocks, are connected using a set of routing resources that is also programmable. In this way, custom circuits can be mapped to the FPGA by computing the logic functions of the circuit within the logic blocks, then using the configurable routing to connect the blocks to form the necessary circuit.

Although the logic capacity of FPGAs is lower than that of ASICs because of the area overhead for providing undedicated logic and routing, FPGAs provide significantly higher flexibility than ASICs, while still offering considerable speedup over general-purpose systems. In addition, the run-time reconfigurability provided by the advanced FPGAs greatly improves hardware utilization.

In the first generation of reconfigurable computing systems, a single configuration was created for the FPGA, and this configuration was the only one loaded into it. These are called *static reconfigurable systems* [Sanchez99]. In contrast, *run-time reconfigurable systems* can change configurations multiple times during the course of a computation. Such systems are capable of reducing under-utilized hardware and fitting large applications onto FPGAs.

In a static reconfigurable system, individual operations of an application will remain idle when they are not required. For example, data dependencies within an application may cause an operation idle, waiting for data inputs from other operations. Therefore, placing all operations onto the FPGA at once is a poor choice, wasting precious hardware resources. Run-time reconfiguration can be used to remove such idle

operations by making them share limited hardware resources. Moreover, run-time reconfiguration provides a design methodology for large applications that are too big for the available hardware resources on the FPGA.

Many recent reconfigurable systems, such as Garp [Hauser97], PipeRench [Schmit97], and Chimaera [Hauck97], involve run-time reconfiguration. In such systems, hardware configuration may change frequently at run-time to reuse silicon resources for several different parts of a computation. Such systems have the potential to make more effective use of chip resources than even standard ASICs, where fixed hardware may be used only in a portion of the computation.

In addition, run-time reconfigurable systems have been shown to accelerate a variety of applications. An example is the run-time reconfiguration within an automatic target recognition (ATR) application developed at UCLA to accelerate a template-matching [Villasenor97]. The algorithm in this system is based on a correlation between incoming radar image data and a set of target templates. Without considering the reconfiguration time, this system improves performance by a factor of 30 over a general-purpose computing system.

However, the advantages of run-time reconfiguration do not come without a cost. By requiring multiple reconfigurations to complete a computation, the time it takes to reconfigure the FPGA becomes a significant concern. The serial-shift configuration approach, as its name indicated, transfers all programming bits into the FPGA in a serial fashion. This very slow approach is still used by many existing FPGAs [Xilinx94, Altera98, Lucent98]. Recent devices have moved to cutting-edge technology, resulting in FPGAs with over one million gates. The configuration's size for such devices is over one megabyte [Xilinx00]. It could take milliseconds to seconds to transfer such a large configuration using the serial-shift approach.

In most reconfigurable systems the devices must sit idle while they are being reconfigured, wasting cycles that could otherwise be performing useful work. For example, the ATR system developed at UCLA uses 98% of its execution time performing reconfiguration, meaning that it uses merely 2% time doing computation. DISC and DISC II systems developed at BYU have spent up to 90% [Wirthlin95, Wirthlin96] of their execution time performing reconfiguration. It is obvious that a significant improvement in reconfigurable system performance can be achieved by eliminating or reducing this overhead associated with reconfiguration delays.

To deal with the reconfiguration overhead this thesis develops an integrated configuration management strategy for reconfigurable computing. Note that this strategy will not only reduce or eliminate the time wasted in performing reconfigurations, it can also increase the potential performance gains and applicability of reconfigurable computing. Specifically, because of reconfiguration overhead, there are applications where the performance benefits of using reconfigurable devices are overwhelmed by the reconfiguration latencies involved; even worse, some applications well suited to reconfigurable computing cannot run on current reconfigurable systems simply because their performance gain is overwhelmed by reconfiguration time. Therefore, eliminating reconfiguration overhead allows more applications to be mapped into reconfigurable hardware, significantly increasing the benefits of reconfigurable systems.

This thesis attempts to develop a complete set of configuration management techniques. The succeeding chapters present:

- A review of reconfigurable computing systems, focusing on various popular reconfigurable models, and of existing studies and techniques for reducing the reconfiguration overhead (Chapter 2).

- A discussion of an integrated configuration management strategy, including configuration compression, configuration caching, and configuration prefetching (Chapter 2).
- An exploration of configuration compression techniques (Chapter 3).
- An investigation of Don't Care discovery technique to improve configuration compression (Chapter 4)
- An investigation of configuration caching techniques for a range of reconfigurable models (Chapter 5).
- An examination of configuration prefetching techniques (Chapter 6).
- A summary of the proposed techniques (Chapter 7).

This thesis closes with conclusions and opinions about the directions of future research in configuration management techniques.

Chapter 2

Background and Research Focus

This chapter first reviews the fundamentals of reconfigurable computing, and introduces a variety of reconfigurable models and systems. It then summarizes previous techniques and concludes by presenting the strategy and focus of our research.

2.1 Reconfigurable Computing Systems

Figure 2.1 illustrates the basic architectural components of a typical reconfigurable computing system. The main component is the field programmable gate array (FPGA).

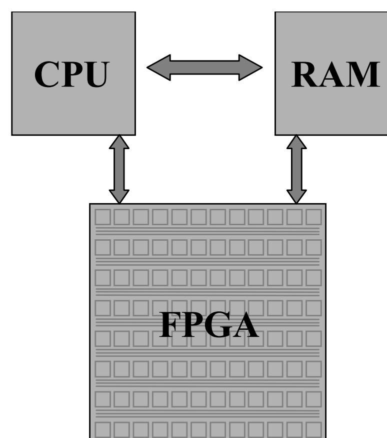


Figure 2.1: Architecture of a reconfigurable computing system.

FPGAs contain configurable logic blocks (CLBs), input-output blocks (IOBs), memory, clock resources, programmable routing, and configuration circuitry. These logic resources are configured through the *configuration bit-stream* allowing a very complex

circuit to be programmed onto a single chip. The configuration bit-stream can be read or written through one of the configuration interfaces on the device.

At this time, SRAM programmable FPGAs are very popular for reconfigurable applications. For such devices, SRAM cells, as shown in Figure 2.2, are connected to the configuration points within the FPGA. Configuration data from the input bit-stream is written to the SRAM cell. The outputs connect to the FPGA logic and interconnect structures. Control of the FPGA is therefore handled by the outputs of the SRAM cells scattered throughout the device. Thus, an FPGA can be programmed and reprogrammed as simply as writing to and reading from a standard SRAM.

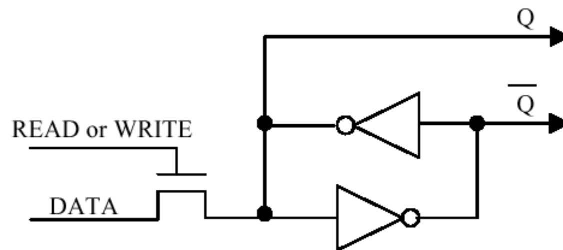


Figure 2.2: A programming bit for SRAM FPGAs

The logic block is often considered to be a *lookup table* (LUT) that takes a number of bits of input and generates one bit of output. By design, LUTs can compute any Boolean logic function with n inputs. The LUT holds truth table outputs in the memory instead of computing the output directly through combinational logic. In a LUT, multiplexers implement logic function by choosing from the program bits in the table. Figure 2.3 shows a two-input lookup table.

Logic blocks in commercial FPGAs are more complex than a single lookup table. For example, a logic block of a Xilinx Virtex FPGA consists of 2 four-input lookup tables as well as a dedicated carry chain circuitry that forms a fast adder. In addition, several multiplexers are also included to combine with the lookup table, providing any logic

function of five, six, seven, or eight inputs. Each logic block also contains flip-flops for state-holding.

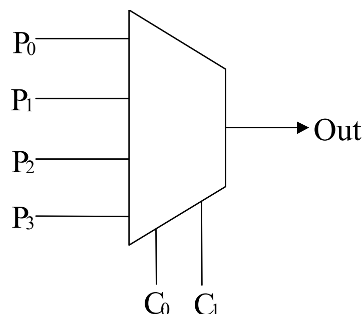


Figure 2.3: A two-input look-up table. P_0, P_1, P_2, and P_3 are program bits. By giving them the right values, any two-input logic function can be realized.

Besides logic blocks, the other key feature that characterizes an FPGA is its interconnect structure. Generally interconnect is arranged in horizontal and vertical channels that are capable of connecting any two logic blocks. Each routing channel contains short wire segments that often connect adjacent logic blocks, medium wire segments that span multiple logic blocks, and long wire segments that run the entire length of the chip. Most interconnect architectures use switches for signals from one logic block to reach another.

As technology advances, the computational power of FPGAs has grown significantly. For example, the new Xilinx Virtex FPGA has enough hardware to implement over two thousand 32-bit adders. As a consequence, the time need to configure larger FPGAs also increases. For example, the configuration time for the Xilinx Virtex 1000 FPGA is 15ms [Xilinx99]. This may not be an issue if the FPGAs were used as logic emulators. However, for a reconfigurable computing system with configuration operations that occur frequently, this latency represents an overhead that significantly degrades the performance as well as limits the utilization of such systems.

2.2 Reconfiguration Models

Frequently, the areas of a program that can be accelerated through the use of reconfigurable hardware are too numerous or complex to be loaded simultaneously onto the available hardware. For these cases, it is helpful to swap different configurations in and out of the reconfigurable hardware as they are needed during program execution, performing a run-time reconfiguration of the hardware. Because run-time reconfiguration allows more sections of an application to be mapped into hardware than can be fit in a non-run-time reconfigurable system, a greater portion of the program can be accelerated in the run-time reconfigurable systems. This can lead to an overall improvement in performance.

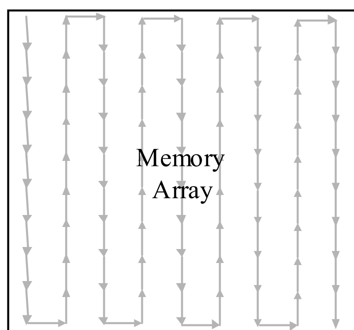


Figure 2.4: The structure of a Single Context FPGA.

There are a few traditional configuration memory styles that can be used with reconfigurable systems, including the *Single Context* model [Xilinx94, Altera98, Lucent98], the *Partial Run-time Reconfigurable* model (PRTR) [Ebeling96, Schmit97, Hauck97] and the *Multi-Context* model [DeHon94, Trimberger97]. For the Single Context FPGA shown in Figure 2.4, the whole array can be viewed as a shift register, and the whole chip area must be reconfigured during each reconfiguration. This means that even if only a small portion of the chip needs to be reconfigured, the whole chip is rewritten. Since many of the applications being developed with FPGAs today involve run-time reconfiguration, the reconfiguration of the Single Context architecture incurs a

significant overhead. Therefore, much research has focused on the new generation of architectures or tools that can reduce this reconfiguration overhead.

A Multi-Context device [DeHon 94, Trimberger 97] has multiple layers of programming bits, where each layer can be active at a different point in time. An advantage of the Multi-Context FPGA over a Single Context architecture is that it allows for an extremely fast context switch (on the order of nanoseconds), whereas the Single Context may take milliseconds or more to reprogram. The Multi-Context design allows for background loading, permitting one context to be configuring while another is executing. Each context of a Multi-Context device can be viewed as a separate Single Context device. A four-context FPGA is shown in Figure 2.5.

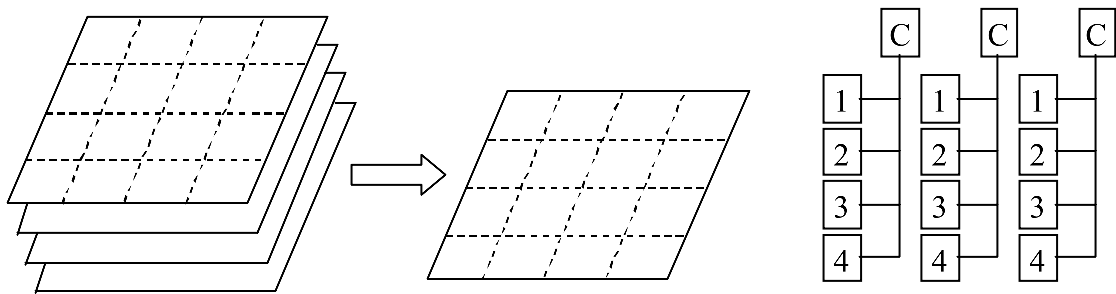


Figure 2.5: A four-context FPGA. At left is the four-context FPGA model, and at right is the memory structure of the four-context FPGA.

Partial Run-time Reconfiguration (PRTR) is another typical reconfiguration model. By changing only a portion of the reconfigurable logic while other sections continue operating, the reconfiguration latency can be hidden by other computations. In recent years, many commercial devices (Xilinx Virtex series, Xilinx 6200 series) and systems (Garp [Hauser 97], Chimaera [Hauck 97], DISC [Wirthlin 96]) have applied the PRTR model.

The structure of a PRTR FPGA is shown in Figure 2.6. The entire SRAM control store memory maps into the host processor's address space. By sending configuration values with row and column addresses to the FPGA, a certain location can be configured.

Based on the Partial Run-time Reconfigurable model, a new model, called Relocation + Defragmentation (Partial R + D) [Compton00, Hauser97], was built to further improve hardware utilization. Relocation allows the final placement of a configuration within the FPGA to be determined at run-time, while defragmentation provides a way to consolidate unused area within an FPGA during run-time without unloading useful configurations.

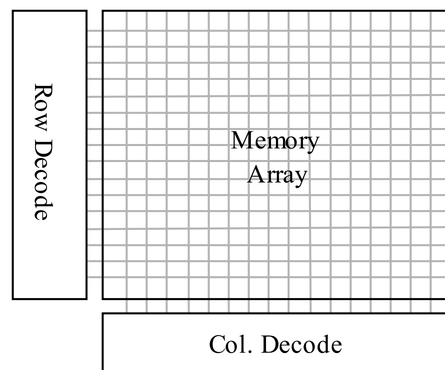


Figure 2.6: The structure of a Partial Run-time Reconfigurable FPGA.

Like the PRTR FPGA, the memory array of the Partial R+D FPGA is composed of an array of SRAM bits. These bits are read/write enabled by the decoded row address for the programming data. However, instead of using a column decoder, an SRAM buffer called the "staging area" is built. This buffer is essentially a set of memory cells equal in number to one row of programming bits in the FPGA memory array. Its values are transferred in parallel to the row location indicated by the row address. The structural view of the Partial R + D model is shown in Figure 2.7.

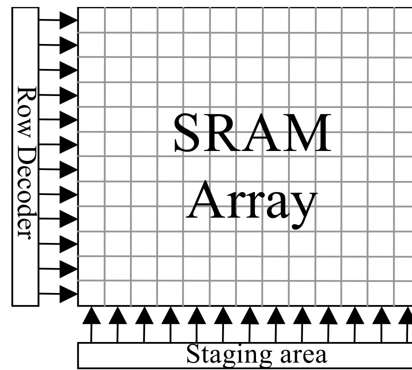


Figure 2.7: The architecture of the Relocation + Defragmentation model. Each row of the configuration bit-stream is loaded into the staging area and then moved into the array.

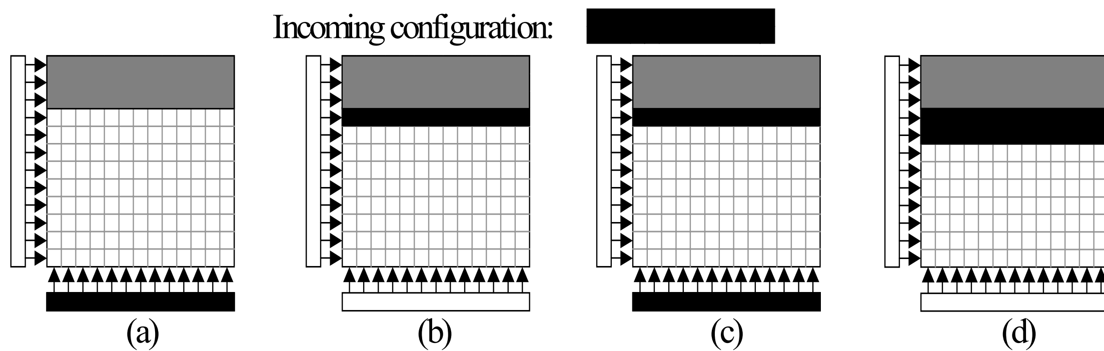


Figure 2.8: An example of configuration relocation. The incoming configuration contains two rows. The first row is loaded into the staging area (a) and then transferred to the desired location that was determined at run-time (b). Then the second row of the incoming configuration is loaded to the staging area (c) and transferred into the array (d).

To configure a chip, every row of a configuration is loaded into the staging area and then transferred to the array. By providing the run-time determined row address to the row decoder, rows of a configuration can be relocated to locations specified by the system. Figure 2.8 shows the steps of relocating a configuration into the array. The defragmentation operation is slightly more complicated than a simple relocation operation. To collect the fragments within that array, each row of a particular configuration is read back into the staging area and then moved to a new location in the array. Figure 2.9 presents the steps of a defragmentation operation. [Compton00] has

shown that with a very minor area increase, the Relocation + Defragmentation model has a considerably lower reconfiguration overhead than the Partial Run-Time Reconfigurable model.

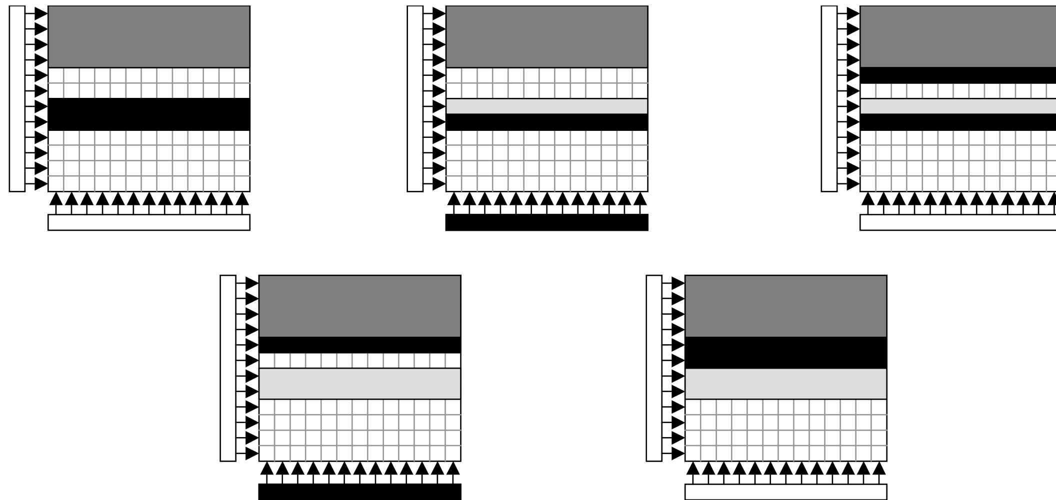


Figure 2.9: An example of defragmentation. By moving the rows in a top-down fashion into staging area and then moving upwards in the array, the smaller fragments are collected.

Many recent reconfigurable systems are built based on these reconfiguration models. In the next two sections, successful projects are briefly reviewed, illustrating high performance and flexibility that reconfigurable systems can provide. Moreover, performance degradation caused by reconfiguration overhead will also be discussed.

2.2.1 System Example -- Garp

The Garp project [Hauser97] focuses on the integration of a reconfigurable computing unit with an ordinary RISC processor to form a single combined processor chip. The research aims to demonstrate a tentative viable architecture that speeds up applications. The Garp architecture is illustrated in Figure 2.10.

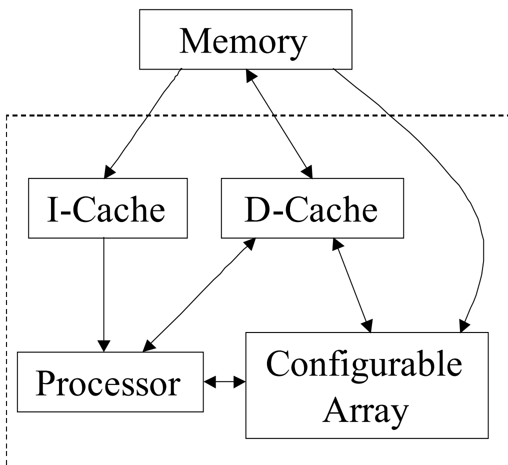


Figure 2.10: Block diagram of Garp.

Garp's main processor executes a MIPS-II instruction set extended for Garp. The rest of the blocks in the system are logic blocks, which correspond roughly to the logic blocks of the Xilinx 4000 series [Xilinx94]. The Garp architecture fixes the number of columns of blocks at 24. The number of rows is implementation-specific. The architecture is defined so that the number of rows can grow in an upward-compatible fashion.

A C compiler is developed to discover the portions (kernels) of applications that can be mapped onto Garp's configurable array. The kernels are synthesized to run on the configurable array, while the rest of the code is executed on the MIPS processor.

The loading and execution of configurations is under the control of the main processor. Instructions are added to the MIPS-II instruction set for this purpose, including ones that allow the processor to move data between the array and the processor's own registers. In the GARP system, configuration is loaded from memory. Since a significant amount of time is needed to load the whole configuration, several steps have been done to shorten the reconfiguration latency. One of the steps is to build a PRTR array with relocation. Hardware translates a physical row numbers into logical ones, so one can load several smaller configurations into the array and switch between them simply

by changing a starting row's address. In addition, simple cache memory units that contain several recently used configurations are distributed within the array. However, the performance of the system still suffers, since no sophisticated strategy was developed to attack the reconfiguration bottleneck.

2.2.2 System Example — DISC

The dynamic instruction set computer (DISC) [Wirthlin95] successfully demonstrated that application specific processors with large instruction set could be built on partial reconfigurable FPGAs. DISC presented the concept of alleviating the density constraint of FPGAs by dynamically reconfiguring the system at run-time.

The block diagram of the DISC system is showed in Figure 2.11. A CLAy31 FPGA [National93] is used as the reconfigurable array. Bus interface circuitry is built for communications between the host processor and the reconfigurable array. A configuration controller is implemented on another CLAy31 FPGA to manage configuration loadings from the RAM. The reconfigurable array runs each instruction during execution. If an instruction requested is not presented on the array, the system enters a halting state and sends a request for the instruction to the host processor.

Upon receiving a request from the reconfigurable array, the host processor chooses a physical location in the array to hold the requested mode. The physical location is chosen based on evaluation of the idle FPGA resources and the size of the requested instruction. If another instruction currently occupies the location selected to hold the requested instruction, the configuration of that instruction is replaced by that of the requested instruction.

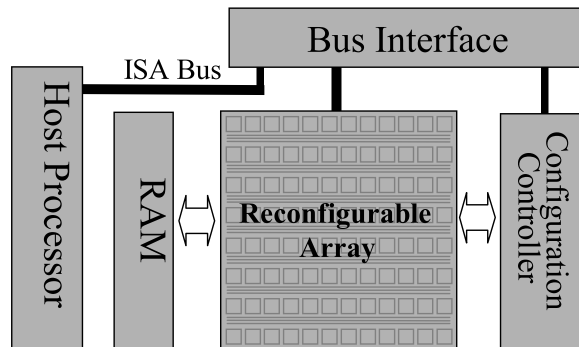


Figure 2.11: Block diagram of DISC system.

Without consideration of the reconfiguration overhead, the DISC system achieves a factor of 80 speedup over the general-purpose approach. However, when considering the reconfiguration overhead, DISC only provides a 23 times speedup. This means the reconfiguration overhead causes a factor of 3.8 performance degradation. Further analysis on a range of applications shows that 25%--91% of execution time is used to perform reconfiguration.

2.3 Reconfiguration Overhead Reduction Techniques

As demonstrated, reconfiguration overhead can severely degrade the performance of reconfigurable systems. In addition, it also limits the system's utilization simply because it can overwhelm the speedup achieved by running applications on reconfigurable hardware. Therefore, eliminating or reducing this overhead becomes a very critical issue for reconfigurable systems. This section describes related research work that attempts to reduce reconfiguration overhead.

2.3.1 Configuration Cloning

Configuration Cloning [Park99] exploits regularity and locality during reconfiguration and is implemented by copying the bit-stream from one region of an FPGA to one or several other regions. Therefore, without loading entire bit-stream the chip can be

configured. By using this cloning technique, configuration overhead can be reduced. Figure 2.12 shows an example of configuration cloning.

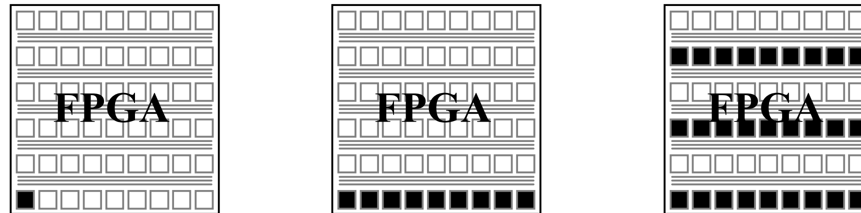


Figure 2.12: An example of Configuration Cloning. Left shows the initial loading, followed by a horizontal copy (middle) and a vertical copy (right).

However, this method is not very realistic. First, it requires the FPGA to send all bits from multiple cells in a row/column to several other cells in the same row/column in parallel. This necessitates a very large amount of routing hardware, complex control circuitry, and perhaps some large switch matrixes, all of which could impose a significant area overhead. Second, this method requires very high regularity; it is best suited only to hand-mapped circuits and those circuits of arrays of replicated cells. This is a significant set of restrictions, and may make this method of very limited utility. Finally, in order to implement configuration cloning, the instruction set of the host processor has to be extended, and the system requires a command interpreter in the FPGA that can decode the command from a host processor and broadcast sender and receiver addresses to a proper configuration bit-stream line. However, because of uncertainty about the number of operands in a command, it is hard for the system to decode commands. These drawbacks not only limit the utilization of this approach, but also result in poor performance, as experimental results indicated.

2.3.2 Configuration Sharing

Configuration sharing [Heron99] is a way to efficiently exploit partial reconfiguration. It involves locating the similarity between circuits that will reduce the amount of reconfiguration. Maximizing the amount of static circuitry (the common circuitry

shared by multiple configurations) can reduce the amount of reconfiguration necessary to switch between them.

The basic idea of this method is simple, but it is hard to implement. Configuration sharing requires applications with similar sub-circuitry, which makes this method of very limited utility. In addition, it requires software tools that can identify the similarities between the applications and map the applications in a way that permits them to share the static circuitry. This will limit the utilization of hardware resources. Furthermore, common circuitry development is very sensitive to the device and algorithm. The static circuitry found in one device probably will not apply to another. Using different algorithms for a function or application can also affect performance, requiring designers to recognize the possible static circuitry produced by different algorithms.

2.3.3 Configuration Scheduling

The configuration scheduling technique [Deshpande99] is suitable for pipelined applications and was designed on a Striped FPGA [Schmit97]. There are two major configuration scheduling approaches: configuration caching and data caching.

In configuration caching, all configurations must be stored in the cache, and cached configurations are circulated through the fabric (Figure 2.13). In data caching, cached intermediate data is circulated through the fabric (Figure 2.14). For configuration caching, a stripe of the fabric is configured each pipeline stage. Therefore, that stripe does not provide useful computation at the pipeline stage. Data caching, on the other hand, does not perform reconfiguration at all pipeline stages. Therefore, when the number of data elements processed by the application exceeds the number of pipeline stages, the number of execution cycles is lower for data caching than for configuration caching.

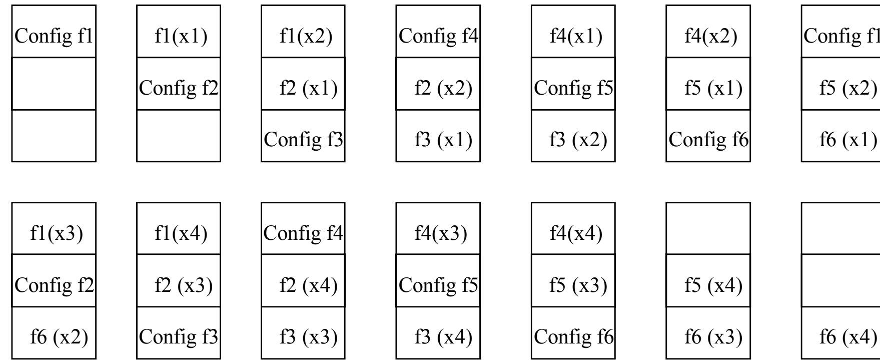


Figure 2.13: Execution using configuration caching approach. A strip is configured at each pipeline stage, meaning that the strip does not contribute to computation at that stage.

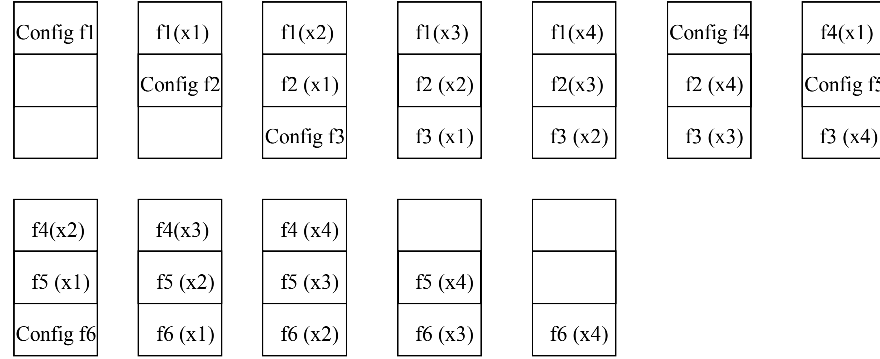


Figure 2.14: Execution using data caching approach. At stages 4 and 5, no strip needs to be configured, thus all strips contribute to computation.

2.4 Research Focus

Section 2.3 illustrated a number of different tactics for reducing configuration overhead. However, each of these techniques attempts to attack only one facet of the problem and does not show significant overhead reduction. In addition, these approaches rely heavily on specific architectures or applications, greatly limiting their utility.

In this thesis, we focus on developing a complete strategy that attacks this reconfiguration bottleneck from multiple perspectives. First, compression techniques can be introduced to decrease the amount of configuration data that must be transferred

to the system. Second, the actual process of transferring configuration data from the host processor to the reconfigurable hardware can be modified to include a configuration cache. Third, configurations can be preloaded to overlap as much as possible with the execution of instructions by the host processor. Finally, high bandwidth configuration bus can be built to speed up configuration transferring process.

Loading a configuration requires that a large amount of data be transferred as quickly as possible into the reconfigurable hardware. Increasing the number of pins for configuration can provide higher configuration bandwidth. In addition, minimizing the clock cycle of the configuration bus presents another alternative to increase configuration bandwidth. However, these approaches are greatly limited by process technology or device vendors, and thus they will not be the major focus of this thesis.

While high-bandwidth buses can get data into the reconfigurable device quickly, the amount of data moved in a reconfiguration is often quite large. To deal with this, we will investigate *configuration compression* techniques that can minimize the size of the configuration bit-streams. FPGA configuration bit-streams tend to be sparse, with significant amounts of regularity. We will develop algorithms to discover regularities within the configuration bit-streams and compress the bit-streams using them.

To maximize the likelihood that a required configuration is already present on reconfigurable devices, we will develop *configuration caching* techniques. By storing the configurations on-chip, the number of configuration loading operations can be reduced, and thus the overall time required is reduced. The challenge in configuration caching is to determine which configurations should remain on the chip and which should be replaced when a reconfiguration occurs. An incorrect decision will fail to reduce the reconfiguration overhead and lead to a much higher reconfiguration overhead than a correct decision. In addition, the different features of various FPGA programming models (discussed in Section 2.2) add complexity to configuration caching because each FPGA model may require unique caching algorithms.

Performance can be further improved when the actual configuration operation is overlapped with computations performed by the host processor. Overlapping configuration and execution prevents the host processor from stalling while it is waiting for the configuration to finish, and hides the configuration time from the program execution.

Configuration prefetching attempts to leverage this overlap by determining when to initiate reconfiguration in order to maximize overlap with useful computation on the host processor. It also seeks to minimize the chance that a configuration will be prefetched falsely, overwriting the configuration that is actually used next. The challenge in configuration prefetching is to determine far enough in advance which configuration will be required next. Many applications can have very complex control flows, with multiple execution paths branching off from any point in the computation, each potentially leading to a different next configuration. In addition, it is very important to correctly predict which configuration will be required. In order to load a configuration, configuration data that is already in the FPGA can be overwritten. An incorrect decision on what configuration to load can not only fail to reduce the reconfiguration delay, but also in fact can greatly increase the reconfiguration overhead when compared to a non-prefetching system.

All of these techniques will be brought together to create a complete configuration management system. Configuration compression will reduce the amount of data that needs to be transferred for each configuration. Configuration caching will increase the likelihood that a required configuration is present on-chip. Configuration prefetching will overlap computation with reconfiguration, avoiding system stalls. By using these techniques together, we can virtually eliminate reconfiguration overhead from reconfigurable computing systems.

Chapter 3

Configuration Compression

For each configuration operation, a significant amount of data is transferred onto the reconfigurable device (FPGA) through communication links. Configuration compression makes it possible to speed up data transfer by reducing the amount of space consumed by the information being sent. In this chapter, we will investigate a variety of effective configuration compression techniques for common reconfigurable devices.

3.1 General Data Compression Techniques

Data compression has important application in the areas of data transmission and data storage. Many data processing applications store large volumes of data, and the number of such applications is constantly increasing as the use of computers extends to new disciplines. At the same time, the proliferation of computer communication networks is resulting in massive transfers of data over communication links. Compressing data to be stored or transmitted reduces storage and/or communication costs. When the amount of data to be transmitted is reduced, the bandwidth of the communication link is effectively increased.

Compression techniques are divided into two categories: *lossless compression* and *lossy compression* [Nelson95]. No information of the original is lost for lossless compression, meaning that a perfect reproduction of the original can be achieved from the compressed data. This is generally the technique of choice for text or spreadsheet files, where losing words or financial data could pose a problem. Lossy compression, on the other hand, involves the loss of some information. Data reconstructed from the

lossy compression is similar to, but not exactly the same as, the original. Lossy compression is generally used for video and sound, where a certain amount of information loss can be tolerated by users. In general, lossy compression techniques achieve better compression ratios than lossless ones.

3.2 Configuration Compression Overview

The goal of configuration compression for reconfigurable systems is to minimize the amount of configuration data that must be transferred. Configuration compression is performed at compile-time. Once compressed, the bit-streams are stored in off-chip memory. During reconfiguration at run-time, the compressed bit-stream is transferred onto the reconfigurable device and then decompressed. The processes of compression and decompression are shown in Figure 3.1.

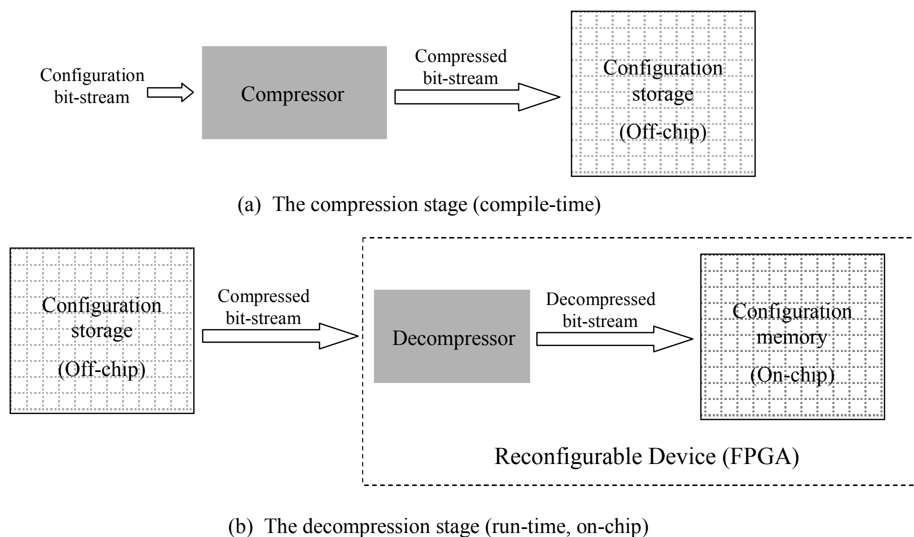


Figure 3.1: The flow of compression. The original configuration data is compressed at compile-time (a). When reconfigurations occur, the compressed data is transferred to the decompressor on the reconfigurable device (b).

As can be seen in Figure 3.1, two issues must be resolved for configuration compression. First, an efficient compression algorithm must be developed. Second,

since decompression is performed on-chip, building a decompressor should not result in significant hardware overhead.

Furthermore, any configuration compression technique must satisfy the following two conditions: (1) the circuitry generated from the decompressed bit-stream must not cause any damage to the reconfigurable devices, and (2) the circuitry generated must result in the same outputs as those produced by circuitry generated from the original configuration data. Consequently, most configuration compression research does not involve lossy techniques since any information loss in a configuration bit-stream may generate undesired circuitry on reconfigurable devices, and, even worse, may severely damage the chips.

Lossless compression techniques satisfy the above conditions naturally, because the decompressed data is exactly the same as the original configuration data. Lossless data compression is a well-studied field, with a variety of very efficient coding algorithms. However, applying these algorithms directly may not significantly reduce the size of the configuration bit-stream, because a number of differences exist between configuration compression and general data compression.

3.3 Configuration Compression Vs. Data Compression

The fundamental strategy of compression is to discover regularities in the original input and then design algorithms to take advantage of these regularities. Since different data types possess different types of regularities, a compression algorithm that works well for a certain data input may not be as efficient as it is for other inputs. For example, Lempel-Ziv compression does not compress image inputs as effectively as it does text inputs. Therefore, in order to better discover and utilize regularities within a certain data type, a specific technique must be developed. Existing lossless compression algorithms may not be able to compress configuration data effectively, because those algorithms cannot discover the potential specific regularities within configuration bit-streams.

Since decompression is performed on-chip, the architecture of a specific device can have an equally significant impact on compression algorithm design. Lossless data compression algorithms do not consider this architecture factor, causing the following problems:

(1) Significant hardware overhead can result from building the decompressor on-chip. For example, a dictionary-based approach, such as Lempel-Ziv-Welch coding requires a significant amount of hardware to maintain a large lookup table during decompression.

(2) The decompression speed at run-time may offset the effectiveness of the compression. For example, in Huffman compression, each code word is decompressed by scanning through the Huffman tree. It is very hard to pipeline the decompression process, and therefore it could take multiple cycles to produce a symbol. As the result, the time saved from transferring compressed data is overwhelmed by slow decompression.

(3) Certain special on-chip hardware that can be used as decompressor may be wasted. For example, wildcard registers on the Xilinx 6200 series FPGAs can be used as decompressors. Unfortunately, no previous algorithm exists to take advantage of this special feature.

Realizing the unique features required for configuration compression, we have focused on exploring regularity and developing proper compression techniques for various devices. However, any technique will be limited if it can merely apply to one device. Therefore, our goal is to investigate the characteristics of different configuration architecture domains, and develop efficient compression algorithms for a given domain.

Two types of base devices (FPGAs) are considered in this work: the Xilinx 6200 series FPGAs and the Xilinx Virtex FPGAs. A first generation partial run-time reconfigurable FPGA, the Xilinx 6200 series provides a special hardware, called wildcard registers, that

allows multiple locations to be configured simultaneously. The *wildcard compression* algorithm we developed not only efficiently compresses configuration bit-streams for all members of the Xilinx 6200 family, but also works for any devices with similar feature. The Xilinx Virtex FPGAs are the most widely used reconfigurable devices, with millions of gates. The architecture of the Virtex family possesses interesting features for future development of reconfigurable devices. Consequently, our compression research for the Xilinx Virtex FPGAs can be adapted to a number of devices without significant modifications. In the following sections, we will discuss the details of our compression algorithms for these two devices.

3.4 Compression for the Xilinx 6200 FPGAs

The XC6200 FPGA is an SRAM-based, high-performance, Sea-Of-Gates FPGA optimized for datapath designs [Xilinx97]. All user registers and SRAM control-store memory are mapped into a host processor's address space, making it easy to configure and access the chip's state. A simplified block diagram of the XC6216 is shown in Figure 3.2.

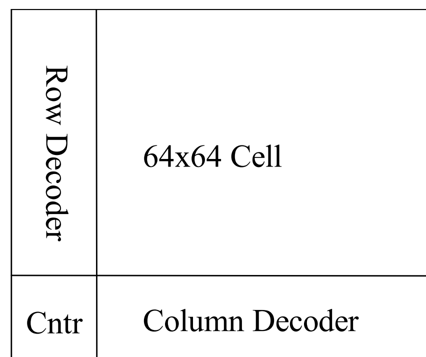


Figure 3.2: XC6216 simplified block diagram.

The XC6200 provides five types of programming control registers. *Device Configuration Registers* control global device functions and modes. *Device Identification Registers* control when computation starts; usually the ID Registers are

written in the final step of configuration. *Map Registers* can map all possible cell outputs from a column onto the external data bus. By correctly setting the Map Registers, the state registers can be easily accessed without complicated mask operations. *Mask Registers* can control which bits on the data bus are valid and which bits are ignored. Finally, *Wildcard Registers* allow some cell configuration memories within the same row or column of cells to be written simultaneously. Since Wildcard Registers are the primary architectural feature used by our algorithm, we will provide more detail about them.

There are two Wildcard Registers: *Row Wildcard Register* and *Column Wildcard Register*, which are associated with the row address decoder and the column address decoder, respectively. Each register has one bit for each bit in the row address or the column address. Wildcard Registers can be viewed as "masks" for the row and column address decoders. Let us consider the effect of the Row Wildcard Register on row address translation. (The Column Wildcard Register has the same effect on column address translation.) A logic one bit in the Row Wildcard Register indicates that the corresponding bit of the row address is a wildcard, which means the address decoder matches rows whose addresses have either a "1" or "0" on the wildcard bits. Thus, the number of cells that will be configured at the same time is 2^n if there are n logic one bits in the Wildcard Register. For example, suppose the Row Wildcard Register is set to "010001" and the address to the row address decoder is set to "110010". In this case the row decoder selects rows 100010, 100011, 110010, and 110011. If these locations share the same computation, and thus would need to be configured with the same value, all four could be configured with a single write operation. Thus, Wildcard Registers permit faster configuration to be achieved.

The Wildcard Registers and the address decoder can be viewed as a configuration decompressor. Given a compressed configuration file, which has Wildcard Register writes followed by address writes, the address is decompressed so that several cells with

the same function get configured simultaneously. The Wildcard Registers can inform the address decoder about which bits in the address can be "wildcarded" and which bits cannot. Theoretically, up to 4096 cells can be configured by only three writes (two Wildcard Registers writes and one address write) if we assume all 4096 cells share the same function. With this "decompressor" hardware available, there is the potential to achieve significant reductions in the required configuration bandwidth. The key is to find an algorithm that can efficiently use this decompression hardware.

3.4.1 Algorithm Overview

Given a normal configuration bit-stream, our algorithm will generate a new configuration file that performs the same configuration with fewer writes by using the Wildcard Registers. Our algorithm contains two stages. In the first, we assume that writes to the Wildcard Registers are free and thus seek the minimum number of writes necessary to configure the array for a given configuration. This will create a series of writes with arbitrary wildcards, meaning that these wildcard writes may add significant overhead. The second stage of the algorithm attempts to reduce this wildcarding overhead by sharing the same wildcard in a series of writes, thus reducing the number of times the Wildcard Registers must be changed.

Before discussing details of this algorithm, we first describe the format of the configuration file we use. The standard Xilinx XC6200 configuration file (.cal file) consists of a series of configuration address-data pairs. Two points must be made about the .cal files. First, a .cal file contains data to configure the entire chip, including both the logic array and the configuration registers. However, the Wildcard Registers operate only on the logic array memory addresses, meaning that it is not possible to compress the configuration register writes. Thus, these register writes represent a fixed overhead for our algorithm. We will ignore these writes during the discussion that follows, although our algorithm does maintain all control register writes from the source file, and our results include these fixed overheads. Second, the XC6200 is partially

reconfigurable, meaning that a .cal file may contain writes to only a portion of the logic array. Thus, there are regions of the array that are not modified by the input configuration. Since these locations may contain data from previous configurations that must be maintained, we treat all locations not written by an input .cal file as "Don't Touches". That is, we do not allow our algorithm to reconfigure these locations, thus restricting the amount of compression possible.

3.4.2 The First Stage of the Algorithm

In the first stage of the algorithm, we assume that both Wildcard Registers can be written during the same cycle as data is written to the logic array's configuration. Thus, we ignore the overhead of wildcard writes in order to simplify compression problem. However, Appendix A shows that even this simplified version of the problem is NP-hard by transforming two-level logic minimization into this compression problem. Although this will demonstrate that an optimal algorithm is unlikely for this problem, it will also point the way towards an efficient heuristic via standard logic minimization techniques.

In the standard two-level logic minimization problem, the goal is to find the minimum number of cubes that cover the ON set of a function, while covering none of the OFF set. In the configuration compression problem, we seek the fewest wildcard-augmented writes that will set the memory to the proper state. Figure 3.3 shows a transformation of the two-level logic minimization problem into the wildcard compression problem.

Since even the simpler decision version of the problem of finding the smallest set of wildcard writes that implements a particular configuration is NP-complete, we are unlikely to find an efficient (polynomial-time) algorithm to construct the smallest such set of writes. Consequently, we focus our attention on heuristic techniques instead.

	00	01	11	10
00	0	1	0	0
01	0	0	0	0
11	1	1	1	0
10	0	1	1	0

	00	01	10	11
00	DT	1	DT	DT
01	DT	DT	DT	DT
10	DT	1	DT	1
11	1	1	DT	1

Figure 3.3: Example of the transformation of 2-level logic minimization into the simplified configuration compression problem. The Karnaugh Map (left) of the circuit is transformed into the configuration to be compressed (right). "DT" indicates don't touches in the configuration.

Because of the similarity of the two problems, we should be able to use standard logic minimization techniques to find the wildcards for the configuration problem. For the example in Figure 3.4, normal configuration will need four writes to configure all cells with the function "2". However, by using logic minimization techniques we can find a single cube that covers the corresponding cells in the Karnaugh map. Since we have Wildcard Registers, we can compress the four configuration memory addresses in the cube into one address "--10", where "-" means wildcard. Before configuring these four cells, we first set the Row Wildcard Register to "11" (which means the row address following is read as "--") and the Column Wildcard Register to "00". The row address decoder then automatically decompresses the address, configuring all four cells at the same time.

Even though this configuration problem can be viewed as a logic minimization problem, there is a difference between these two problems. In logic minimization the logic is static, which means all "1" terms are written in the Karnaugh map at the same time, and the sum of the product terms (cubes) exactly covers the logic for each output. However, in configuration compression the configuration is done dynamically, which means that later writes can overwrite previous values. Thus, we can consider the values of the cells that have not yet been written into the FPGA as Don't Cares.

With these Don't Cares, we may be able to use fewer product terms (cubes) to cover the cells that need to be written to the FPGA, reducing the number of writes in the configuration. For example, in Figure 3.4, suppose data "1" is written before data "3". We can find a single cube to cover all the "1"s, instead of two, if we consider the cells with data "3" as Don't Cares (Figure 3.5a). This means we need just one address write to configure all "1"s. Of course, all cells covered by the cube shaded in Figure 3.5a are configured with data "1", including those cells that actually require data "3". However, since the XC6200 FPGA is a reconfigurable device, those cells with the wrong data can be rewritten with the correct configuration later, as shown in Figure 3.5b.

	00	01	10	11
00	1	1	2	DT
01	1	1	2	DT
10	1	3	2	3
11	3	3	2	DT

Figure 3.4: Example for demonstrating the potential for configuration compression.

From the example in Figure 3.5, we can see that the order in which specific values are written can affect the total number of writes needed. If we ignore Wildcard Register writes, the total number of writes needed to complete the configuration in Figure 3.4 is four for the case in which the "1"s are written before the "3"s. However, for the case in which the "3"s are written before the "1"s, the total number of writes will be five. This is because we can write all "1"s in one cycle if the "3"s are Don't Cares, while the "3"s will take two writes regardless of whether the "1"s are written before or after the "3"s. Thus, we have to consider not only how to most efficiently write each value into the configuration memory, but also the order of these writes should occur to best compress the data. We can certainly find an optimal sequence for a specific configuration by doing an exhaustive search, but the runtimes would be significant. Thus, heuristic

algorithms are required not just for finding wildcarded addresses, but also to determine the order of wildcard writes. Before we present these heuristics, we first introduce the logic minimization technique we used for our configuration algorithm.

	00	01	10	11
00	1	1		
01	1	1		
10	1	X		
11	X	X		

(a)

	00	01	10	11
00	1	1		
01	1	1		
10	1	3		3
11	3	3		

(b)

Figure 3.5: Example of the use of Don't Cares in configuration compression. By making the locations with "3" as Don't Cares (a), one write, rather than two, is sufficient to configure all "1"s. "3"s are written into required locations later (b).

3.4.3 Wildcarded Address Creation via Logic Minimization

The logic minimization problem is a well-known NP-complete problem, and heuristic algorithms exist to find near optimal solutions. The Espresso algorithm [Brayton84] is widely used for single-output logic optimization, and it is claimed that optimal solutions will be produced in most cases. We use Espresso as a major portion of our configuration compression algorithm. The input required by Espresso is an encoded truth table, as shown in Figure 3.6 (a). Each line consists of a minterm index encoded in binary, followed by either a "1" (for members of the On set) or a "-" (for members of the Don't Care set). The corresponding minimized truth table is shown in Figure 3.6 (b).

The configuration memory addresses in the .cal file can be viewed as minterms for the Espresso input file. Assume, for example, that we decide that the "3"s are the next values to write to the array, and that the "1"s have already been written, though the "2"s have not. We can use Espresso to find the proper wildcarded writes by assigning all addresses with the value to be written assigned to the On set, all Don't Touch and

already written values to the Off set, and all values not yet written to the Don't Care set. Thus, the "3" addresses would be passed to Espresso with a "1", and the "2" addresses would be passed with a "-". The results of Espresso will be a set of cubes that correspond to wildcarded writes. These writes contain all of the addresses that need to be set to the value to be written, as well as locations that will be written in future writes, but will not contain the Don't Touch or already written addresses.

(a)		(b)	
1000	1	00--	1
0001	1	-000	1
0010	1		
0011	1		
0000	-		

Figure 3.6: Espresso input (a), and the resulting output (b).

We now present the first stage of our algorithm:

1. Read the input .cal file and group together all configuration memory addresses with the same value. Mark all address locations as "unoccupied".
2. Sort the groups in decreasing order of the number of addresses to be written in that group.
3. Pick the first group, and write the addresses in the group to the Espresso input file as part of the On set.
4. Write all other addresses marked "unoccupied" to the Espresso input file as part of the Don't Care set.
5. Write all addresses marked "occupied", yet with the same value as the first group, to the Espresso input file as part of the Don't Care set.

6. Run Espresso.
7. Pick the cube from the Espresso output that covers the most unoccupied addresses in the first group and add it to the compressed .cal file. Mark all covered addresses as "occupied", and remove them from the group.
8. If the cube did not cover all of the addresses in the group, reinsert the group into the sorted list.
9. If any addresses remain to be compressed, go to Step 2.

This algorithm uses the Espresso-based techniques discussed earlier, with a greedy choice of the order in which to write the different values. We greedily pick the group with the most addresses in it because this group should benefit the most from having as many Don't Cares as possible, since the values may be scattered throughout the array. An example of this is shown in Figure 3.7. If we choose to write the "5"s first, the total number of writes (excluding Wildcard Register writes) is five, while it requires only three writes if the "6"s are written first. This greedy method has been as efficient as other more complete heuristic methods we have implemented.

	00	01	10	11
00	6	6	6	6
01	6	6	6	6
10	6	6	6	5
11	6	6	5	6

Figure 3.7: An example that illustrates the reason for selecting bigger groups.

Since a single cube may not cover all the addresses in the currently picked group, we pick the cube that covers the most addresses, since it provides the greatest compression factor. When this group is picked again (in order to cover the rest of the addresses), we will put Don't Cares for those configuration memory addresses "occupied" by the same

function data. Thus, later cubes are still allowed to cover these earlier addresses, since writing the same value twice does not cause any problems.

One additional optimization we have added to the algorithm is to perform a preprocessing to determine if any of the groups will never benefit from any Don't Cares, and thus can be scheduled last. For each group, we run Espresso twice. In the first run, all locations that will be configured, except for the members of the group, are assigned to the Don't Care set. In the second run, these nodes instead form the Off set. In both cases the group members are assigned to the On set. If the numbers of cubes found in both runs are identical, it is clear that the Don't Cares do not help to reduce the number of writes for this value. Thus, this group is always scheduled last.

One final concern for the first stage of our algorithm is the XC6216 column wildcard restriction. Because of the electrical properties of the memory write logic, the architecture restricts the number of wildcards in the column address to at most four bits. To handle this, we examine the cube picked in Step 7 and see if it meets this restriction. If there are too many wildcards in the column bits, we iteratively pick one wildcard to remove until the restriction is met. To pick the wildcard to remove, we determine how many addresses have a "0" in a given wildcard bit and how many have a "1". The wildcard removed is the one with the most addresses with a specific value ("1" or "0"), and that value replaces the Wildcard.

Once the first stage of the algorithm is completed, we have a list of address data pairs, with wildcards in most of the addresses, which will produce the desired configuration. However, while this series of writes assumes that the Wildcard Registers can be set in the same cycle as the configuration memory write, it actually takes three cycles to perform this operation: Row Wildcard Register write, Column Wildcard Register write, and configuration memory write. Thus, wildcard writes will triple the total number of writes. In stage two of the algorithm, we use techniques for sharing Wildcard Register

writes between multiple configuration memory writes, significantly reducing this overhead.

3.4.4 The Second Stage of the Compression Algorithm

The objective of this stage is to reorder the sequence of writes created in the first stage in order to share Wildcard Register writes between configuration memory writes. Also, since Espresso will find the largest cube that covers the required configuration addresses, there may be some wildcard bits that can be changed into "0" or "1" while still covering all required memory addresses. Performing such reductions may increase the number of compatible Wildcard Register values, again increasing Wildcard Register value sharing. We call this second transformation "wildcard reduction". Figure 3.8 gives an example of two consecutive writes that cannot share any Wildcard Register values after the first stage, yet after wildcard reduction both wildcards can be shared. The number of writes needed for writing the 6 configuration memory addresses is down to four, two less than that without wildcard sharing.

Write 1 Addresses	**Write 2 Addresses**
(000000, 000100)	(100000, 100100)
(010000, 000100)	(100000, 100100)
(010000, 001000)	(110000, 101000)

(a)

Original Writes
(0-0-00, 00--00)
(1-0000, 1---00)

(b)

Reduced Writes
(0-0000, 00--00)
(1-0000, 10--00)

(c)

Figure 3.8. An example of Wildcard reduction. The addresses to be configured are shown in (a). (b) shows the set of writes given by the first stage, which requires unique row and column wildcards. The reduced version (c) can share both row and column Wildcards by removing some Wildcard bits.

Before we continue the discussion, we first need to define some terms:

- *Required Addresses Set*: the set of addresses that become occupied because of this write (the addresses this write is used to set).
- *Maximum Address*: the wildcarded address found by Espresso.

- *Minimum Address*: the wildcarded address with the minimum number of wildcards that still covers the Required Address Set.
- *Intersect(Addr1, Addr2)*: the set of addresses covered by both addresses Addr1 and Addr2.
- *And(Wild1, Wild2)*: the bitwise AND of two Wildcard Register values. Retains a wildcard bit only when it appears in both values.
- *Or(Wild1, Wild2)*: the bitwise OR of two Wildcard Register values. Contains a wildcard bit when either source wildcard value has a wildcard at that bit.
- *Superset(Wild1, Wild2)*: true if every wildcard bit in Wild2 is also in Wild1.

In the second stage, we reorder the sequence of writes found in stage one and apply the wildcard reduction selectively to find a new order with a much lower Wildcard Register write overhead. In order to do this, we convert the totally ordered sequence of writes from the first stage into a partial order that captures only those ordering constraints necessary to maintain correctness. We then create a new order and apply the wildcard reduction.

In the first stage, the sequence we created is not necessarily the only order in which the sequence of writes can correctly be applied. For example, the writes in Figure 3.8 can be reversed without altering the resulting configuration since neither write overwrites relevant data from the other. Of course, there are some writes that are not swappable, so we must determine which writes must be kept in sequence and which can be reordered. Once we have this information, we can reorder the writes to increase Wildcard Register value sharing. The following condition gives one situation in which writes can be reordered and forms the basis for our partial order generation algorithm. In the paragraphs that follow, we assume that write A preceded write B in the original order.

Condition 1: If Intersect(Maximum Address(A), Required Addresses Set(B)) = {}, then A and B can be reordered.

In order to create a partial order, we investigate each (not necessarily consecutive) pair of nodes in the original order. If condition 1 does not hold for this pair of nodes, an edge is inserted into the partial order group, requiring that the earlier write must occur before the later write. Once all pairs have been considered, we have created a partial order for the entire set of writes. Only those nodes without any incoming edges can be scheduled first. After a node is scheduled, that node and any edges connected to it are removed, potentially allowing other nodes to be scheduled. All nodes that become schedulable once a given node is removed from the partial order are called the "children" of that node.

At any given point in the scheduling process, the partial order graph determines which nodes are candidates to be scheduled. Now, we must develop an algorithm for choosing the best candidate node to schedule. We use the following rules as our scheduling heuristics. The rules are applied in order, with ties at an earlier rule broken by the rules that follow. Thus, losers at any rule are eliminated. Only the winners are compared with the following rules:

1. The candidate can share both row and column wildcards with the preceding writes.
2. A child of the candidate can share both wildcards with a different current candidate.
3. The candidate can share either the row or column wildcard with the preceding writes.
4. The candidate with the greatest number of other candidates and children that can share both row and column wildcards with it.

5. the candidate with the greatest number of other candidates and children that can share either the row or column wildcard with it.

6. Candidate with the greatest number of children.

Rules 1 and 3 measure the immediate impact of scheduling the candidate on the number of wildcard writes. Rule 2 adds some lookahead, scheduling a candidate early in order to allow its children to share wildcards with another current candidate. Rules 4 to 6 attempt to increase the number of good candidates, hoping that the greater flexibility will result in lower wildcard overheads.

In order to implement these rules, we must determine when two writes can share a row or column wildcard. To do this, we use the following condition:

Condition 2: If (Maximum Wildcard of A *And* Maximum Wildcard of B) is the superset of (Minimum Wildcard of A *Or* Minimum Wildcard of B), then A and B can share the wildcard.

The intuition behind this condition is that if A and B can share a wildcard, then the Maximum Wildcard of A must be the superset of the Minimum Wildcard of B, and the Maximum Wildcard of B must be the superset of the Minimum Wildcard of A. Otherwise, they cannot share the wildcard. Notice that the wildcard sharing is not transitive. That is, if A and B can share a wildcard, and B and C can share a wildcard, it is not always true that A and C can share a wildcard. For example, B might have all bits as wildcards, while A and C each have only one wildcarded position, and the position differs for A and C.

The non-transitivity of wildcards is an important consideration. If we apply the scheduling rules discussed earlier pairwise, we may schedule three writes in a series and expect them to share all wildcards, when in fact we require new wildcard writes before the third write. To deal with this, when a node is scheduled, we generate a new

Minimum Wildcard and Maximum Wildcard bounds for the schedule so far. These wildcard bounds must represent all possible values in the Wildcard Registers at this point in the schedule. This process is captured by the following rules:

1. If the scheduled candidate cannot share the current wildcard:
   Minimum Wildcard(schedule) = Minimum Wildcard(candidate)
   Maximum Wildcard(schedule) = Maximum Wildcard(candidate)
2. If the scheduled candidate can share the current wildcard:
   Min Wildcard(schedule) = Or(Min Wildcard(schedule), Min Wildcard(candidate))
   Max Wildcard(schedule) = And(Max Wildcard(schedule), Max Wildcard(candidate))

These rules maintain the Minimum and Maximum Wildcards in order to more accurately determine which candidate can share a wildcard with the preceding writes. Thus, whenever we apply the rules for determining which candidate to choose, we always use the schedule's Minimum and Maximum Wildcards to determine whether a candidate can share a wildcard.

3.4.5 Experimental Results

The algorithm described above was implemented in C++ on a Sun Sparc20 and was run on a set of benchmarks collected from XC6200 users. These benchmarks are real applications that are either hand-mapped or generated by automatic tools.

The results are shown in Table 3.1 (as well as graphically in Figure 3.9). The size of the initial circuit is given in the "Input size" column in terms of the number of configuration writes in the original .cal files. This size includes all writes required to configure the FPGA, including both compressible writes to the logic array as well as non-compressible control register writes. The "Control writes" column represents the number of non-compressible writes, and is a fixed overhead for both the original and

compressed file. The size of the compressed file is contained in the "Total writes" column, which includes control writes, writes to the logic array ("Config. Writes"), and writes to the Wildcard Registers ("Wildcard Writes"). The "ratio" column is the ratio of the compressed file size to the original file size. The "CPU time" for compressing each benchmark is represented in the last column. As can be seen, the algorithm achieves an average compression factor of almost four. This represents a significant reduction in the bandwidth requirements for reconfiguration in reconfigurable systems.

Table 3.1: The results of the compression algorithm on benchmark circuits.

Benchmark	**Input size**	**Control writes**	**Config. writes**	**Wildcard writes**	**Total Writes**	**Ratio**	**CPU time(ms)**
counter	199	40	53	13	106	53.2%	1.3E3
parity	208	16	9	3	28	13.5%	3.0E2
adder4	214	40	43	14	97	45.3%	4.5E3
zero32	238	42	12	3	57	23.9%	4.0E2
adder32	384	31	28	14	73	19.0%	1.7E3
smear	696	44	224	37	305	43.8%	4.5E4
adder4rm	908	46	473	45	564	62.1%	8.3E4
gray	1201	44	530	74	648	52.2%	2.6E5
top	1367	70	812	87	969	70.8%	1.3E6
demo	2233	31	423	91	545	24.4%	2.8E6
ccitt	2684	31	346	84	461	17.2%	2.2E6
tally	3366	42	211	42	295	8.7%	4.5E6
t	5819	31	834	192	1057	18.2%	1.1E7
correlator	11011	38	1663	225	1926	17.4%	5.0E7
Geometric Mean:						27.7%	

In addition, we notice a significant disparity in the compression ratios for different benchmarks. After examining the configuration bit-streams and their corresponding circuits, we discovered that this disparity was caused by the varied levels of the

regularities existing within different configuration bit-streams. More specifically, the hand-mapped applications consist of more regular logic and routing structures than the automatically generated applications. As a result, the benchmarks that are hand-mapped are more compressible in general than the automatically generated ones.

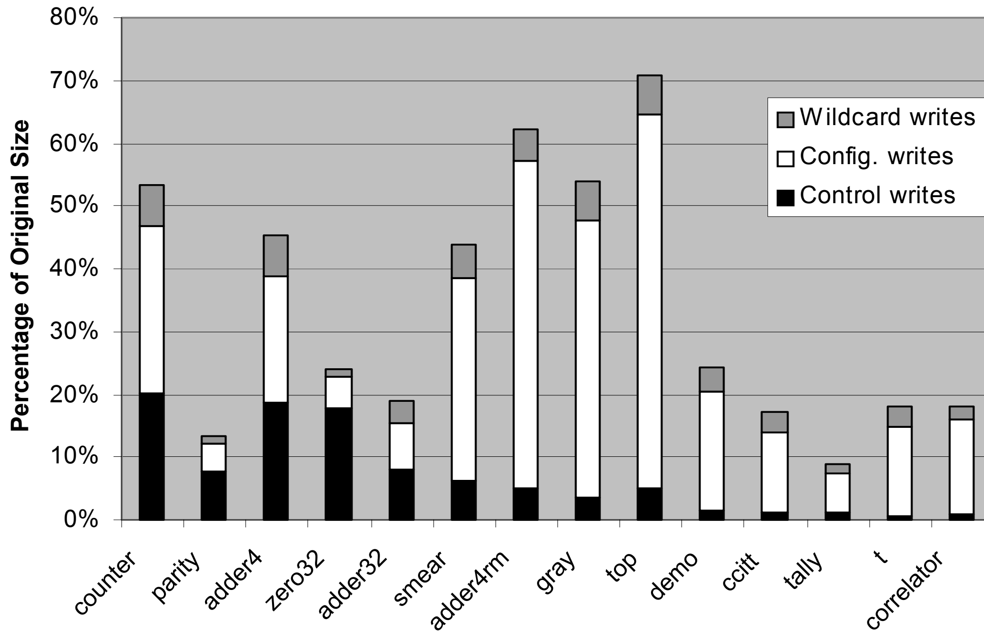


Figure 3.9: Graph of compressed file size as a percentage of original file size. Bar height represents the total resulting file size and is broken into components. The gray portion represents the writes to the Wildcard Register, white represents actual writes to the array, and black represents the fixed, non-compressible portions of the files.

Figure 3.9 also demonstrates the effectiveness of the second stage of our compression algorithm. "Config. writes" represent the number of writes necessary to configure the logic, which are produced by the first stage of our compression algorithm. A large number of "Config. writes" require Wildcard Register writes, generating a significant overhead. This overhead is minimized by applying the second stage of our compression algorithm. As can be seen in Figure 3.9, "wildcard writes" represent only a small percentage of total writes.

3.5 Compression for the Xilinx Virtex FPGAs

Each Virtex [Xilinx99] device contains configurable logic blocks (CLBs), input-output blocks (IOBs), block RAMs, clock resources, programmable routing, and configuration circuitry. These logic functions are configurable through the configuration bit-stream. Configuration bit-streams that contain a mix of commands and data can be read and written through one of the configuration interfaces on the device. A simplified block diagram of a Virtex FPGA is shown in Figure 3.10.

The Virtex configuration memory can be visualized as a rectangular array of bits. The bits are grouped into vertical frames that are one-bit wide and extend from the top of the array to the bottom. A *frame* is the atomic unit of configuration, meaning that it is the smallest portion of the configuration memory that can be written to or read from. Frames are grouped together into larger units, called *columns*. In Virtex devices, there are several different types of columns, including one center column, two IOB columns, multiple block RAM columns, and multiple CLB columns. As shown in Figure 3.11, each frame sits vertically, with IOBs on the top and bottom. For each frame, the first 18 bits control the two IOBs on the top of the frame, then 18 bits are allocated for each CLB row, and another 18 bits control the two IOBs at the bottom of the frame. The frame then contains enough "pad" bits to make it an integral multiple of 32 bits.

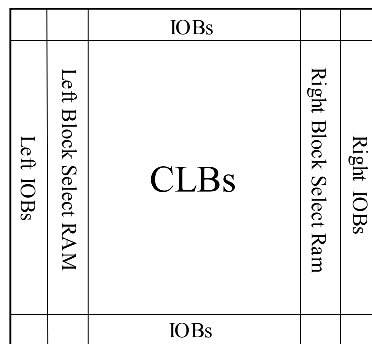


Figure 3.10: Virtex architecture.

The configuration for the Virtex device is done through the *Frame Data Input Register* (FDR). The FDR is essentially a shift register into which data is loaded prior to transfer to configuration memory. Specifically, given the starting address of the consecutive frames to be configured, configuration data for each frame is loaded into the FDR and then transferred to the frames in order. The FDR allows multiple frames to be configured with identical information, requiring only a few cycles for each additional frame, thus accelerating the configuration. However, if even one bit of the configuration data for the current frame differs from the previous frame, the entire frame must be reloaded.

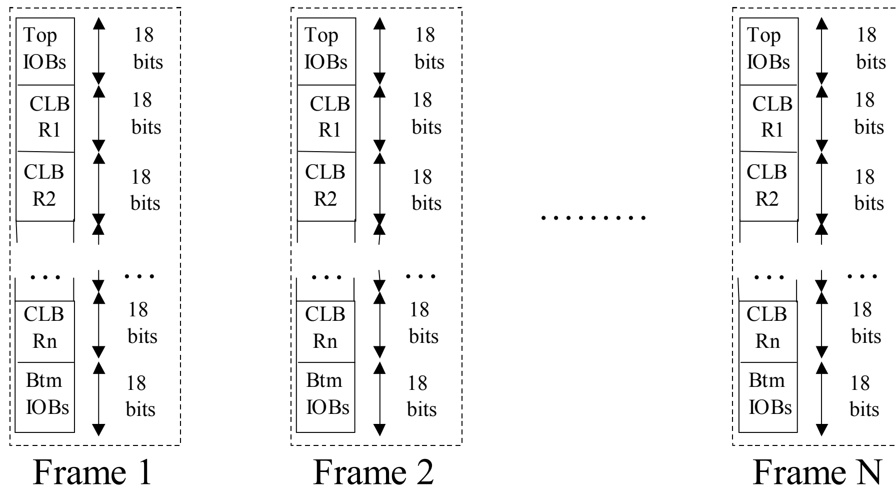


Figure 3.11: Virtex frame organization.

3.5.1 Algorithms Overview

As we mentioned, well-known techniques -- including Huffman [Huffman52], Arithmetic [Witten87] and LZ [Ziv77] coding -- are very efficient for general-purpose compression, such as text compression. However, without considering features of the bit-stream, applying these techniques directly will not necessarily reduce the size of the configuration file. Given the frame organization described above, it is likely that

traditional compression will either miss or destroy the regularities contained in the configuration files. For example, the commercial tool gzip achieves a compression factor of 1.85 in our benchmark set, much less than is achievable.

In this section, we will consider general-purpose compression approaches including Huffman, Arithmetic and Lempel-Ziv coding because of their proven effectiveness. In addition, we will extend our wildcard approach used for Xilinx 6200 bit-stream compression. Before we discuss the details of our compression algorithms, we will first analyze the potential regularities in the configuration files.

3.5.2 Regularity Analysis

Current Virtex devices load whole frames of data at a time. Because of the similarity of resources in the array, we can expect some regularity between different frames of data. We call this similarity *inter-frame* regularity. In order to take advantage of this regularity, the frames containing the same or similar configuration data should be loaded consecutively. For example, an LZ77 compression algorithm uses recently loaded data as a fixed-sized dictionary for subsequent writes, and by loading similar frames consecutively, the size of the configuration files can be greatly reduced. The current Virtex frame numbering scheme, where consecutive frames of a column are loaded in sequence, can be a poor choice for compression. After analyzing multiple configuration files, we discovered that the *Nth* frame of the columns is more likely to contain similar configuration data since it controls identical resources. Therefore, if we clustered together all of the *Nth* frames of the columns in the architecture, we can achieve a better compression ratio. Of course, changing the order of the frames will incur an additional overhead by providing the frame address, but the compression of frame data may more than compensate for this overhead. Note that Huffman and Arithmetic coding are probability-based compression approach, meaning that the sequence that the configuration data is written will not affect the compression ratio.

Regularity within frames may be as important as regularity between frames. This *intra-frame* regularity exists in circuits that contain similar structures between rows. To exploit this regularity we will modify the current FDR with different frame buffer structures and develop the corresponding compression algorithms. For Lempel-Ziv compression, the shift-based FDR fits the algorithm naturally. However, extending the size of the FDR structure to a larger window can provide even greater compression ratios, though this must be balanced against potential hardware overheads. For our wildcarded approach, the structure of the Wildcard Registers used in Xilinx 6200 can be applied to the FDR to allow multiple locations within the FDR to be written at the same time.

3.5.3 Symbol Length

Even though the configuration bit-stream is packed with 32-bit words for the Virtex devices, much of the regularity will be missed if the symbol length is set to 32-bit or other powers of two. As was shown in Figure 3.11, each CLB row within a frame is controlled by an 18-bit value, and the regularities we discussed above exist in the 18-bit fragments rather than 32-bit ones. In order to preserve those regularities we will break the 32-bit original configuration bit-stream. Except for the regularity, two other factors are considered to determine the length of the basic symbol. First, for Lempel-Ziv, Arithmetic and Huffman coding, the length of the symbol could affect the compression ratio. If the symbol is too long, the potential intra-symbol similarities will likely be overwhelmed. On the other hand, very short symbols, though retaining all the similarities, will significantly increase coding overhead. Second, since decompression is done at run-time, the potential hardware cost should be considered. For example, both Huffman and Arithmetic coding are probability-based approaches and require that the probabilities of symbols be known during decompression. Retaining long symbols and their probabilities on-chip could consume significant hardware resources. In

addition, transferring the probability values to the chip could also represent additional configuration overhead.

As discussed above, using 18-bit symbols will retain the regularities in the configuration bits-stream. However, for Huffman and Arithmetic coding, the probabilities of 2^{18} symbols need to be transferred and then retained on-chip to correctly decompress the bit-stream. Clearly, this is not possible to implement and will increase configuration overhead. Therefore, we choose to use 6-bit or 9-bit symbols for Huffman, Arithmetic and Lempel-Ziv compressions. Using 6-bit or 9-bit symbols will preserve the potential regularities in the bit-streams and limit additional overheads.

Notice that the 32-bit words packed in each frame may not necessarily be multiples of six or nine. Therefore, if we simply take the bit-steams and break them into 6-bit or 9-bit symbols, we will likely to destroy *inter-frame* and regularity. To avoid this, during the compression stage, we will attach the necessary pad bits to each frame to make it a multiple of six or nine. This represents a pre-processing step for each of the compression algorithms.

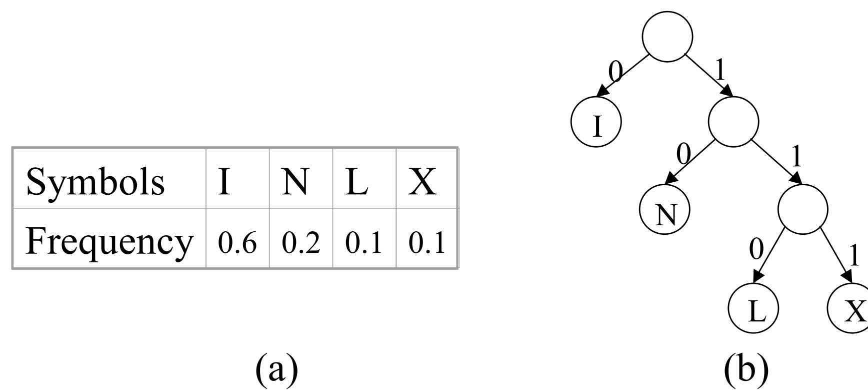
3.5.4 Huffman coding

The goal of Huffman coding is to provide shorter codes to symbols with higher frequency. Huffman coding assigns an output code to each symbol, with the output codes being as short as one bit or considerably longer than the original symbols, depending on their probabilities. The optimal number of bits to be used for each symbol is $log_2(1/p)$, where p is the probability of a given symbol. The probabilities of symbols are sorted, and a prefix binary tree is built based on the sorted probabilities, with the highest probability symbol at the top and the lowest at the bottom. Scanning the tree will produce the Huffman code. Figure 3.12 shows a set of symbols (a) and its corresponding Huffman tree (b). Given a string “XILINX” the resultant Huffman code is 1110110010111, using 13 bits.

Huffman compression for Virtex devices consists of two simple steps:

1. Convert the input bit-stream into a symbol stream.
2. Perform Huffman coding over the symbol stream.

The problem with this scheme lies in the fact that the Huffman codes must be an integral number of bits long. For example, if the probability of a symbol is *1/3*, the optimum number of bits to code that symbol is around *1.6*. Since Huffman coding requires an integral number of bits to the code, assigning a 2-bit symbol leads to a longer compressed code than is theoretically possible.



Symbols	I	N	L	X
Frequency	0.6	0.2	0.1	0.1

Figure 3.12: An example of Huffman coding. A set of 4 symbols and their frequencies are shown in (a). The corresponding Huffman tree is shown in (b).

Another factor that needs to be considered is decompression speed. Since each code word is decompressed by scanning through the Huffman tree, it is very hard to pipeline the decompression process. Therefore it could take multiple cycles to produce a symbol. Also, it is difficult to parallelize the decoding process, because Huffman is a variable-length code.

3.5.5 Arithmetic Coding

Unlike Huffman coding, which replaces each input symbol by a code word, Arithmetic coding completely takes a series of input symbols and replaces it with a single output

number. The symbols contained in the stream may not be coded to an integral number of bits. For example, a stream of five symbols can be coded in 8 bits, with 1.6-bit average per symbol. Like Huffman coding, Arithmetic coding is a statistical compression scheme. Once the probabilities of symbols are known, the individual symbols are assigned to an interval along a probability line, and the algorithm works by keeping track of a high and low number that bracket the interval of the possible output number. Each input symbol narrows the interval and as the interval becomes smaller, the number of bits needed to specify it grows. The size of the final interval determines the number of bits needed to specify a stream. Since the size of the final interval is the product of the probabilities of the input stream, the number of bits generated by Arithmetic coding is equal to the entropy. Figure 3.13 shows the process of Arithmetic coding for string "XILINX" over the same symbol set used for Huffman coding. The generated code is 11110011011, two bits shorter than the Huffman code.

Note that the basic idea described above is difficult to implement, because the shrinking interval requires the use of high precision arithmetic. In practice, mechanisms for fixed precision arithmetic have been widely used.

Symbols	I	N	L	X
Frequency	0.6	0.2	0.1	0.1

(a)

Symbol	LowRange	HighRange
	0.0	1.0
X	0.9	1.0
I	0.9	0.954
L	0.9432	0.9486
I	0.9432	0.94644
N	0.945144	0.951624
X	0.950994	0.951644

(b)

Figure 3.13: An example of Arithmetic coding. The same symbol set used for the Huffman coding is shown in (a). The coding process for string "XILINX" is shown in (b). The final interval, represented by the last row in (b), determines the number of bits needed.

The Arithmetic compression for Virtex devices consists of two steps:

1. Convert the input bit-stream into a symbol stream.
2. Perform the fixed-precision Arithmetic coding over the symbol stream.

The problem with this algorithm is that Arithmetic coding considers the symbols to be mutually unrelated. However, the regularities existing in the configuration bit-stream may cause certain symbols to be related to each other. Therefore, this approach may not be able to yield the best solution for configuration compression. One solution to this problem is to combine multiple symbols together and discover the accurate probabilities of the combined symbols. However, this will cause additional overhead by transferring and retaining a significant amount of probability values. Another way to improve performance is to calculate the probabilities of combined symbols by simply multiplying the probabilities of individual symbols. This dynamic approach will increase the precision of the interval without considering the correlation between symbols. However, performing additional multiplications on the decompression end will slow down decompression.

3.5.6 Lempel-Ziv-Based (LZ) Compression

Recall that Arithmetic coding is a compression algorithm that performs better on a stream of unrelated symbols. LZ compression is an algorithm that more effectively represents groups of symbols that occur frequently. This dictionary-based compression algorithm maintains a group of symbols that can be used to code recurring patterns in the stream. If the algorithm spots a sub-stream of the input that has been stored as part of the dictionary, the sub-stream can be represented in a shorter code word. The related symbols caused by the regularities in the configuration bit-stream make LZ algorithms an effective compression approach.

There are variations of LZ compression, including LZ77 [Ziv77], LZ78 [Ziv78] and LZW [Welch84]. In general, LZ78 and LZW will achieve better compression than LZ77 over a finite data stream. A lookup table is used to maintain occurred patterns for

LZ78 and LZW. However, the excessive amount of hardware resources required to retain the table for LZ78 and LZW during decompression prevent us from considering those schemes for configuration compression. The “sliding window” compression of LZ77 requires only a buffer, and the shift-based FDR fits the scheme naturally, though hardware must be added to allow reading of specific frame locations during execution.

The LZ77 compression algorithm tracks the last n symbols of data previously seen, where n is the size of the sliding window buffer. When an incoming string is found to match part of the buffer, a triple of values corresponding to the matching position, the matching length, and the symbol that follows the match is output. For example, in Figure 3.14, we find that the incoming string 3011 is in buffer position 2 with match length 4, and the next symbol is 0. So the algorithm will output codeword (3, 4, 0).

Standard LZ77 compression containing the three fields will reach entropy over an infinite data stream. However, for a finite data stream, this format is not very efficient in practice. For the case when no matching is found, rather than outputing the symbol, the algorithm will produce a codeword containing three fields, wasting bits and worsening the compression ratio. An extension of LZ77, called LZSS [Storer82], will improve coding efficiency. A threshold is given and if the matching length is shorter than the threshold, only the current symbol will be output. When the matching length is longer than the threshold, the output codeword will consist of the index pointer and the length of the matching. In addition, to achieve correct decompression, a flag bit is required for each code word to distinguish the two cases.

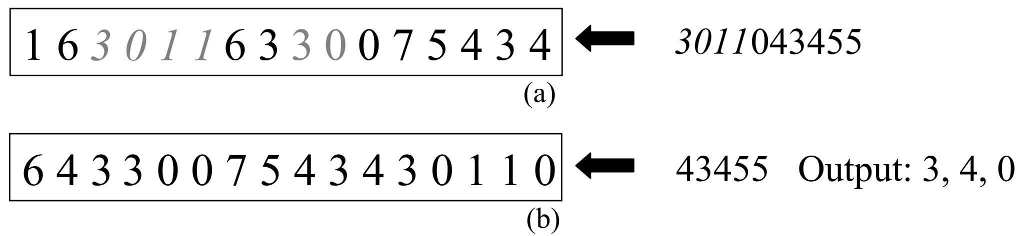


Figure 3.14: The LZ77 sliding window compression example. Two matches found are illustrated in color gray. LZ77 selects the longer match “3011”, and the resultant codeword is (3, 4, 0). (a) shows the sliding window buffer and the input string before encoding. (b) shows the buffer and input string after encoding.

As mentioned above, the FDR in Virtex devices can be used as the sliding window buffer, and LZSS can take advantage of the intra-frame regularity naturally. However, since the current FDR can contain only one frame of configuration data, using it as the sliding window buffer will not take full advantage of inter-frame regularities. Thus, we modify the FDR to the structure shown in Figure 3.15. As can be seen in Figure 3.15, the bottom portion of the modified FDR, which has same size as the original FDR, can transfer data to the configuration memory. During the decompression the compressed bit-stream is decoded and then fed to the bottom of the modified FDR. Incoming data will be shifted upwards in the modified FDR. Configuration data will be transferred to the specified frame once the bottom portion of the modified FDR is filled with newly input data.

In addition, configuration data that is written to the array can be reloaded to the bottom portion of the modified FDR. This lets a previous frame be reused as part of the dictionary, and the inter-frame regularity is better utilized. Specifically, before loading a new frame, we could first read a currently loaded frame from the FPGA array back to the frame buffer, and then load the new frame. By picking a currently loaded frame that most resembles the new frame, we may be able to exploit similarities to compress this new frame.

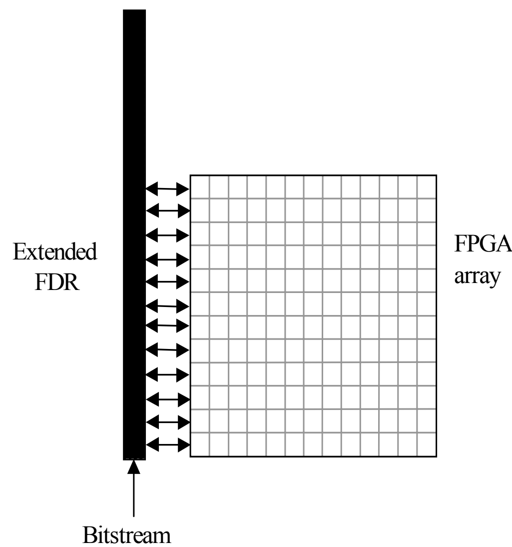


Figure 3.15: The hardware model for LZ77 compression.

While this technique will be slow due to delays in sending data from the FPGA array back to the FDR, there may be ways to accelerate this with moderate hardware costs. In current Virtex devices, the data stored in the Block Select RAMs can be transferred to logic very quickly. We can exploit this feature by slightly modifying the current hardware to allow the values stored in the Block Selected RAMs to be quickly read back to the modified FDR. By providing the fast readback from only the Block Select RAMs, we efficiently use the Block RAMs as caches during reconfiguration to hold commonly requested frames without significant hardware costs. Also, the size of the modified FDR must be balanced against the potential hardware cost. In our research, we allow the modified FDR to contain two frames of data. This will not significantly increase hardware overhead, yet it will utilize the regularities in the configuration stream.

Finding regularities in a configuration file is a major goal. LZ compression performs well only in the case where common strings are found between the sliding window buffer and incoming data. This requires quite a large buffer to find enough matches for general data compression. However, for configuration compression, the hardware costs

will restrict the size of the sliding window buffer. Thus, performing LZ compression directly over the datastream will not render the desired result. In order to make compression work efficiently for a relatively small buffer, we need to carefully exploit the data stream, finding regularities and intelligently rearranging the sequence of frames to maximize matches. In the following sections, we discuss algorithms that apply LZSS compression, targeting the hardware model described above. These algorithms are all realistic but require different amounts of hardware resources and thus provide different compression ratios.

3.5.7 The Readback Algorithm

The goal of configuration compression is to take advantage of both inter-frame and intra-frame regularities. In the configuration stream, some of the frames are very similar. By configuring them consecutively, higher compression ratios can be achieved. The readback feature allows the frame that most resembles the new frame to be read back to the modified FDR and reused as a dictionary, increasing the number of matches for LZSS. This permits us to fully use regularities within the bit-stream. For example, in Figure 3.16, four frames are to be configured, and frames (b), (c) and (d) are more like (a) than like each other. Without readback, inter-frame regularities between (c), (d) and (a) will be missed. However, with the fast readback feature, we can temporarily store frame (a) in the Block Select RAMs, reading it back to the modified FDR and using it as a dictionary when other frames are configured. This fast readback will significantly increase the utilization of inter-frame regularities with negligible overhead. Since the modified FDR is larger than the size of the frame, LZSS will be able to use intra-frame regularities naturally.

Discovering *inter-frame* regularities represents an issue that will influence the effectiveness of compression. Based on the hardware model we proposed above, the similarity between the frame in the modified FDR and the new incoming frame is the key factor for compression. More specifically, we seek to place a certain frame in the

modified FDR so that it will mostly aid the compression of the incoming frame. In order to obtain such information, each frame is used as a fixed dictionary in a preprocessing stage, and LZSS is applied to all other frame, which are called *beneficiary* frames. Note that LZSS is performed without moving the sliding window buffer, meaning that the dictionary will not be changed. This approach excludes potential intra-frame regularities within each beneficiary frame, providing only the inter-frame regularity information. The output code length represents the necessary writes for each beneficiary frame based on the dictionary, and shorter codes will be found if the beneficiary frame resembles the dictionary.

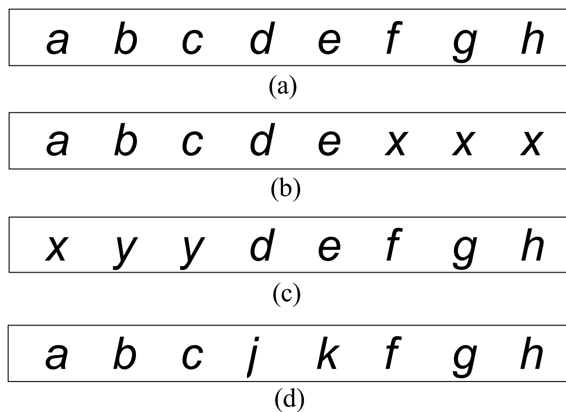


Figure 3.16: Example to illustrate the benefit of readback. (b), (c), and (d) resemble to (a). By reusing (a) as a dictionary, better compression can be achieved.

Once this process is over, a complete directed graph can be built, with each node standing for a frame. The source node of a directed weighted edge represents a dictionary frame, and the destination node represents a beneficiary frame. The weight of each edge denotes the inter-frame regularity between a dictionary frame and a beneficiary frame. One optimization performed is to delete the edges that present no inter-frame regularity between any two frames. Figure 3.17(a) shows an example of the inter-frame regularity graph.

Given an inter-frame regularity graph, our algorithm seeks an optimal configuration sequence that maximizes the inter-frame regularities. Specifically, we seek a subset of the edges in the inter-frame regularity graph such that every node can be reached and the aggregate edge weight is minimized. Solving this problem is equivalent to solving the *directed minimum spanning tree* problem, where every node has one and only one incoming edge, except for the root node. Figure 3.17(b) shows the corresponding optimal configuration sequence graph of Figure 3.17(a). In the configuration sequence graph, a frame with multiple children needs to be stored in Block Select RAMs for future readback. For example, in Figure 3.17 (b), a copy of frame A will be stored in Block Select RAMs and read back to the modified FDR to act as a dictionary.

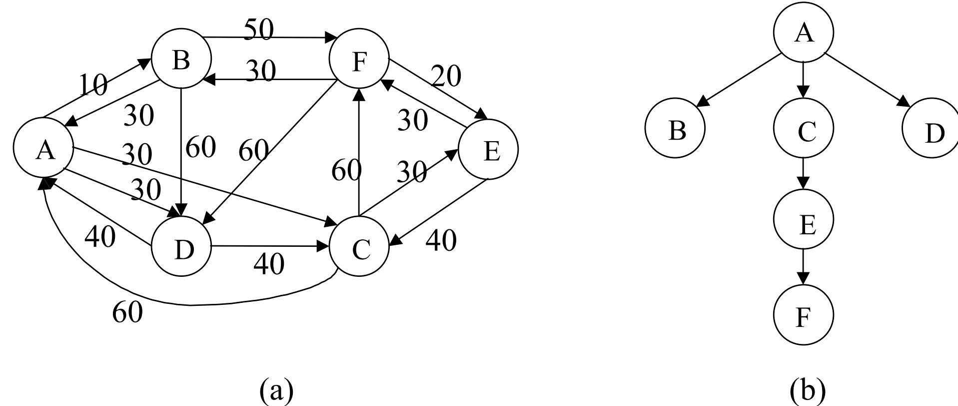


Figure 3.17: Seeking optimal configuration sequence. An inter-frame regularity graph is shown in (a). The corresponding optimal configuration sequence graph is shown in (b).

Now we present our Readback algorithm:

1 Convert the input bit-stream into a symbol stream.

2 For each frame, use it as a fixed dictionary and perform LZSS on every other frame.

3 Build an inter-frame regularity graph using the values computed in step 2.

4 Apply the *standard directed minimum spanning tree algorithm* [Chu65] on the inter-frame regularity graph to create the configuration sequence graph.

5. Perform pre-order traverse starting from the root. For each node that is being traversed:
    5.1. If it has multiple children, a copy of it will be stored in an empty slot of the Block Select RAMs.
    5.2. If its parent node is not in the modified FDR, read the parent back from the Block Select RAMs.
    5.3. Perform LZSS compression.
    5.4. If it is the final child traversed of the parent node, release the memory slot taken by the parent.

Step 2 investigates inter-frame regularities between frames. Results are used to build the inter-frame regularity graph and the corresponding configuration sequence graph in Steps 3 and 4 respectively. The Pre-order traversal performed in Step 5 uses the parent frame of the currently loading frame as a dictionary for LZSS compression. Note that a copy of the currently loading frame will be stored in the Block Select RAMs if it has multiple children in the configuration sequence graph. Also, additional overhead from setting configuration registers will occur if frames to be configured are not contiguous.

One final concern for our Readback algorithm is the storage requirement for the reused frames. Analyzing configuration sequence graphs, we found that although a large number of frames need to be read back, they are not all required to be held in the Block Selected RAM at the same time, and they can share the same memory slot without conflict. For example, in Figure 3.18, both frame A and frame B need to be read back. Suppose the left sub-tree needs to be configured first; then frame A will occupy a slot in the Block Selected RAMs for future readback. Once the configuration of the left tree is complete, the memory slot taken by frame A can be reused by frame B during configuration of the right sub-tree.

We have developed an algorithm using a bottom-up approach that accurately calculates the memory slots necessary. By combining it with our Readback algorithm, usage of the Block Select RAMs can be minimized. The details of the algorithm are as follows:

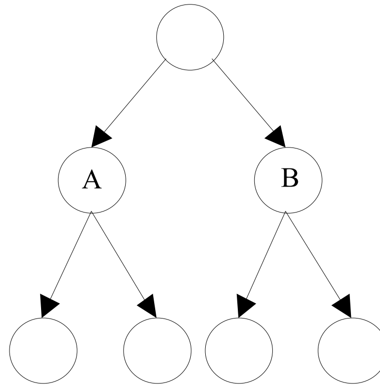


Figure 3.18: An example of memory sharing.

1. For each node in the configuration sequence graph, assign 0 to the variable V and number of children to C.
2. Put each node whose children are all leaves into a queue.
3. While the queue is not empty:
   3.1. Remove a node from the queue.
   3.2. If it has one child, $V = V_{child}$, else $V = max(largest\ V_{child}, (second\ largest\ V_{child} + 1))$.
   3.3. For its parent node, C = C -1. If C = 0, put the parent node into the queue.

Figure 3.19 shows an example of our Memory Requirement Calculation algorithm. At left is the original configuration sequence graph. At right shows the calculation of the memory requirement using a bottom-up approach. The number inside each node represents the number of memory slots necessary for configuring its sub-trees. As can

be seen, only two memory slots are required for this 14-node tree. It is obvious that the memory required by a node depends on the memory required by each of its children. One important observation is that the memory required by the largest sub-tree can overlap with the memory required for other sub-trees. In addition, since the last child of a node to be configured can use the memory slot released by its parent, the memory required by configuring all sub-trees can equal that of configuring the largest sub-tree. Since the pre-order traverse will scan the left sub-trees before the right ones, we should readjust the configuration sequence graph to set each node in the sub-tree that requires the most memory as the rightmost sub-tree. In order to apply the memory minimization to our compression, we modify Step 4 of our readback algorithm as follows:

4. Apply the standard directed minimum spanning tree algorithm on the inter-frame regularity graph to create the configuration sequence graph. Perform the Memory Calculation algorithm, and the largest sub-tree for each node is set as the rightmost sub-tree.

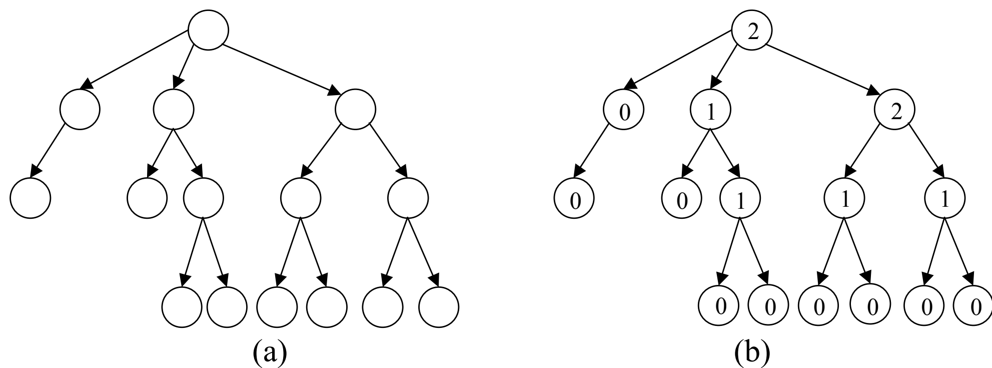


Figure 3.19: An example to illustrate our Memory Requirement Calculation algorithm. A configuration sequence graph is shown in (a), and the corresponding memory requirement calculation procedure is shown in (b).

3.5.8 Active Frame Reordering Algorithm

The Readback algorithm allows frames to be read back to the modified FDR to achieve effective compression. However, the delay and hardware alterations required for the Block Selected RAM readback may not be acceptable. Some applications may restrict

the use of the Block Select RAMs. In order to take advantage of the regularities within the configuration bit-stream, we have developed a frame reordering algorithm that does not require frame readback.

As can be seen in our readback algorithm, frame reordering enhances compression by utilizing inter-frame regularities. This idea can still be applied to applications without the readback feature. In our readback algorithm, once the inter-frame regularity graph is built, a corresponding configuration sequence graph can be generated, and traversing the configuration sequence graph in pre-order can guarantee the maximum utilization of the regularities discovered. However, without frame readback, traversing the configuration sequence graph might not necessarily be the optimal solution, since parent nodes cannot be reused as a dictionary. Our Active Frame Reordering algorithm uses a greedy approach to generate a configuration sequence that allows each frame to be used as a dictionary only once. It still takes the inter-frame regularity graph as input. However, instead of using the directed MST approach to create a configuration sequence, a spanning chain will be generated using a greedy approach. The details of the algorithm are as follows:

1. Convert the input bit-stream into symbol stream.
2. For each frame, use it as fixed dictionary, perform the LZSS on every other frame.
3. Build an inter-frame regularity graph using the values that resulted from Step 2.
4. Put the two frames connected by the minimum weight edge into a set. Let H be the head and T be the tail of this edge.
5. While not all frames are in the set:
    - 5.1. For all incoming edges to H and outgoing edges from T, find the shortest one that connects to a frame not in the set. Put that frame into the set. The frame is set to H if the edge found is an incoming edge to H; otherwise set the frame to T.

6. Perform LZSS compression on the chain discovered in Step 5.

The basic idea of the algorithm is to grow a spanning chain from the two ends. Step 5 finds a frame not in the chain with the shortest edge either coming into an end or going out from the other. This greedy process is repeated until all frames are put in the spanning chain. For example, in Figure 3.17, the order of the frames to be put into the chain discovered by our algorithm is ABDFEC (the configuration sequence will be DABFEC). The cost of the sequence is 160, slightly larger than the optimal spanning chain (150). Starting from one end of the discovered spanning chain, LZSS can be performed to generate a compressed bit-stream.

3.5.9 Fixed Frame Reordering Algorithm

One simple algorithm is to reorder the frames such that the *Nth* frame of each column to be configured in consecutively. Performing LZSS over the sequence generated by the simple reordering takes advantage of the regularities within applications. The overhead of setting the configuration registers can be eliminated using this fixed frame order.

3.5.10 Wildcarded Compression for Virtex

Since our Wildcard Compression achieves good results for the Xilinx 6200 FPGAs, we would like to apply it to Virtex FPGAs. For Virtex configurations, multiple rows within a frame can contain the same configuration data. Instead of configuring them one by one, the wildcarded approach allows these rows to be configured simultaneously. To apply the wildcarded approach to Virtex, an address register and a wildcard register will be added as an augmented structure to the FDR. They will allow specified rows within the FDR to be configured.

For circuits with repetitive structures, multiple frames could be very similar, yet not completely identical. By allowing the FDR to be addressable, we take advantage of this inter-frame regularity. Instead of loading the whole frame, we can load only the

differences between frames. For example, in Figure 3.20, two frames need to be configured, and the second frame has only three different rows from the first one. In this case, only the configuration data for the 3 different rows needs to be loaded. In addition, if the three different rows can be covered by a wildcard, one write is enough to configure the whole second frame. This structure will also support true partial reconfiguration. More specifically, for each frame to be reconfigured, rather than loading the entire frame, we can simply load the difference from the current configuration. Note that adding the Address Register and Wildcard Register represents additional hardware cost. Moreover, extra bits for the address and wildcard need to be transferred for every write.

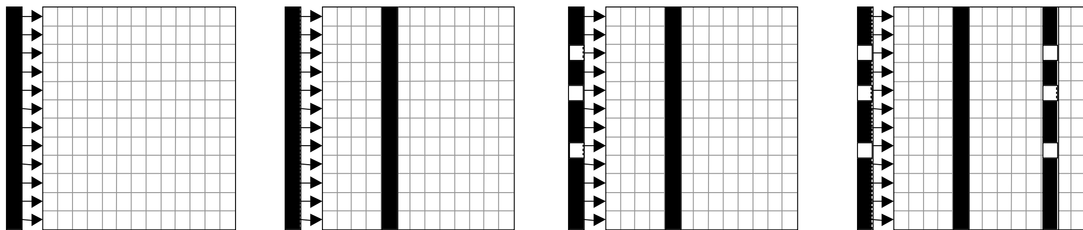


Figure 3.20: An example of inter-frame compression using addressable FDR.

The Wildcard algorithm consists of 2 stages. In the first stage we reorder similar frames so they will be configured consecutively. This creates a sequence in which the number of writes necessary for configuring each frame is greatly reduced. In the second stage, we find the wildcards covering the writes for each frame and thus further reduce the configuration overhead. The first stage takes advantage of inter-frame regularities while the second stage focuses on intra-frame regularities.

In the first stage, we discover the number of non-matching rows between each pair of frames; the result indicates the extent of similarity between the frames. An undirected graph is built to keep track of the regularities, and a near optimal sequence needs to be discovered. Since each frame is configured exactly once, finding the sequence based on

the regularity graph is equivalent to solving the traveling salesman problem. An existing algorithm is an approximation with a ratio bound of two for the traveling-salesman problem with triangle inequality [Lawler]. Given a complete undirected graph $G = (V, E)$ that has a nonnegative integer cost $c(u, v)$ associated with each edge $(u, v) \in E$, cost function c satisfies the triangle inequality if for all vertices u, v, w $\in$V, $c(u, w) \leq c(u, v) + c(v, w)$. Since the differences between frames satisfy the triangle inequality, we can apply the approximation algorithm on our compression algorithm. The details of our Wildcard algorithm are as follows:

1. Convert the input bit-stream into an 18-bit symbol stream.
2. For each pair of frames, identify the different 18-bit symbols between them.
3. Build a regularity graph using the results from Step 2.
4. Perform the Approx-TSP-Tour algorithm [12] to determine the order of frames to be configured.
5. For each frame configuration, use the Wildcard algorithm to find the wildcards to cover the differences.

3.5.11 Simulation Results

All algorithms are implemented in C++ on a Sun Sparc Ultra 5 workstation and were run on a set of benchmarks collected from Virtex users. Detailed information about the benchmarks is shown in Table 3.2.

Figure 3.21 shows simulation results for compression approaches using 6-bit symbols; the wildcard approach uses 18-bit symbols. The left 10 benchmarks are automatically mapped and use more than 50% of the chip area. The “Geo. Mean” column is the geometric mean of the 10 benchmarks. The three right-most benchmarks are either hand mapped or use only a small percentage of the chip area and are included to demonstrate how hand mapping or low utilization affects compression. Figure 3.22

demonstrates simulation results for 9-bit symbols. (The Wildcard algorithm is not shown, since it uses only 18-bit symbols.)

Table 3.2: Information for Virtex benchmarks.

Benchmark	**Source**	**Device**	**Chip Utilization**	**Mapping**
Mt1mem0	Rapid	400	>80%	Auto
Mt1mem1	Rapid	400	>80%	Auto
Mars	USC	600	Unknown	Auto
RC6	USC	400	Unknown	Auto
Serpent	USC	400	Unknown	Auto
Rijndael	USC	600	Unknown	Auto
Design1	HP	1000	>70%	Auto
Pex	Northeastern	1000	93%	Auto
Glidergun	Xilinx	800	>80%	Hand
Random	Xilinx	800	>80%	Hand
U1pc	Xilinx	100	1%	Auto
U50pc	Xilinx	100	50%	Auto
U93	Xilinx	100	>90%	Auto

As can be seen in the figures, the readback algorithm performs better than other algorithms for both 6-bit and 9-bit cases for most of the benchmarks. This is because the Readback algorithm takes full advantage of inter-frame regularities within the configuration bit-stream by reusing certain frames as dictionaries. Though they cannot fully utilize inter-frame regularities, the reordering techniques still provide fairly good results without using the Block Select RAMs as a cache. The Active Reordering algorithm performs better than the Fixed Reordering algorithm since active reordering can better use inter-frame regularities by actively shuffling the sequence of frames, while fixed reordering can utilize only the regularities given by the fixed sequence.

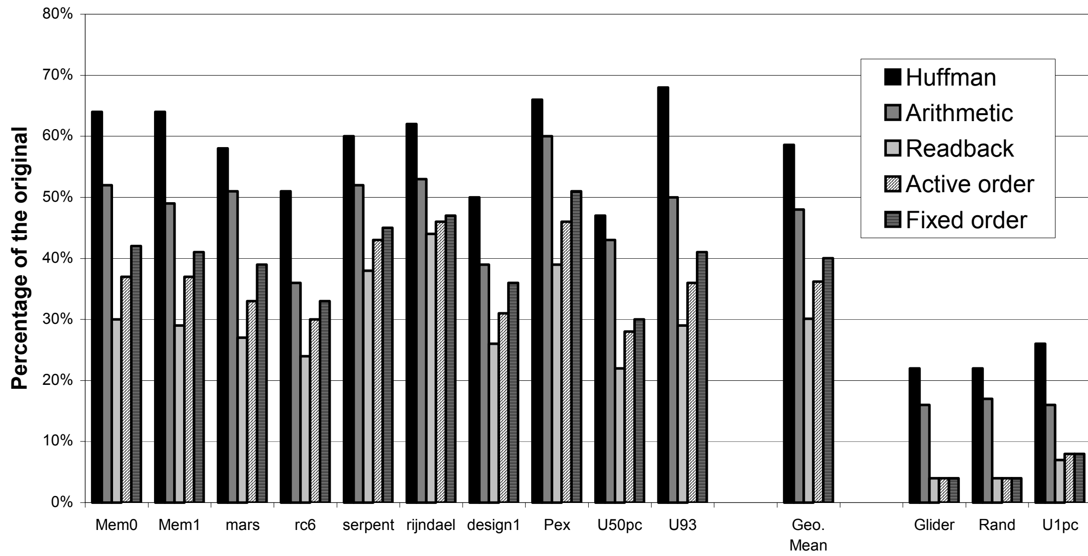


Figure 3.21: The simulation results for 6-bit symbol.

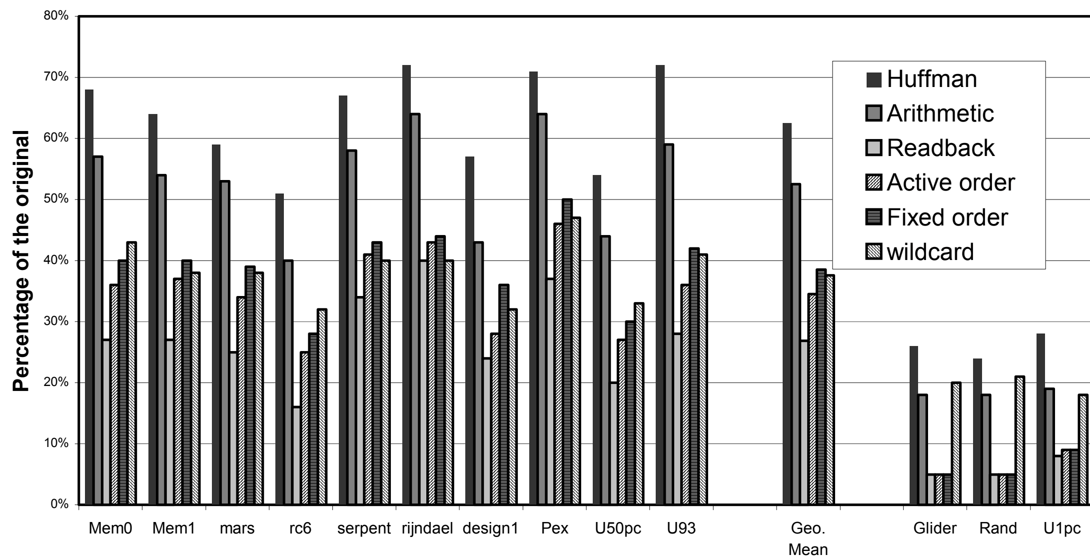


Figure3.22: The simulation results for 9-bit symbol.

Surprisingly, although the wildcard approach can exploit both inter-frame and intra-frame regularities, it still yields worse compression ratios than the active reordering scheme for most of the benchmarks. There are several reasons for this. First, the wildcard approach requires address and wildcard specification for each write, adding significant overhead to the bit-stream. The additional overhead overwhelms the benefits provided by the regularities within the applications. Second, the wildcard approach requires a comparison between the same rows of given frames to discover inter-frame regularities. Consequently, the similarity the wildcard approach can discover is aligned in rows, and any unaligned similarities that benefit the LZ-based approaches will not help it. For example in Figure 3.23, the Wildcard algorithm cannot discover the inter-frame regularity between frame A and frame B. However, the regularity can be exploited for LZ-based approaches. Third, the Wildcard approach requires that enough rows covered by a wildcard share the same configuration value to achieve better compression. However, even the XCV1000, which is a relatively large device, has only 64 rows and it is not likely to find enough rows covered by a wildcard that have the same configuration value. For many cases, each wildcard contains only one row, and the address/wildcard overhead is still applied.

	Frame A values	**Frame B values**
Row 1	1	2
Row 2	2	3
Row 3	3	4
Row 4	4	5
Row 5	5	6

Figure 3.23: Unaligned regularity between frames. The wildcard approach will miss this regularity, which benefits the LZ-based approaches.

The probability-based Huffman and Arithmetic coding techniques perform significantly worse than other techniques, since they do not consider regularities within the bit-stream. The Huffman approach did worse than the Arithmetic approach, simply because of its inefficient coding method. Adding the fact that these two approaches require significant hardware for the decompressor, we will not consider using them for configuration compression.

Comparing the results in Figure 3.21 and Figure 3.22, we found that LZ-based approaches perform better on 6-bit symbols than 9-bit ones for most of the benchmarks. Analyzing the bit-stream, we found that its regularities discovered within the bit-stream may not result in very long matches. Increasing the symbol size will shorten the matches and increase the length of codewords for single symbols. Huffman and Arithmetic approaches perform better on 9-bit symbols, which distribute the probability better.

Most of the benchmarks we tested use a significant amount of FPGA capacity. The only exception is “u1pc”, which uses about 1% of the chip area. As can be seen in the figures, its compression ratio is very high. The other two benchmarks with very high compression ratios use more than 80% of the chip area, but they were hand placed. After analyzing the two benchmarks, we found that the two handcrafted circuits have extremely strong intra-frame regularities. Specifically, most rows within each frame are identical, and very long matches can be found.

3.5.12 Hardware Costs

Since decompression must be performed on-chip, hardware costs for building decompressors must be evaluated to determine whether our compression algorithms are viable techniques. In this work we focus on the hardware implementation of the LZ decompressor because its compression algorithms outperform other approaches. Our fellow graduate student, Melany Richmond has implemented an LZ decompressor in

hardware and demonstrated the hardware cost is minimal. The overall increase in area is less than 1% for Virtex 1000 or larger devices [Richmond01].

3.6 Related Works

A LZ-based approach [Dandalis01] was developed to compress the Virtex bit-stream. As a modified LZW algorithm, this approach requires a significant amount of memory storage on-chip to retain dictionary. Though it can be applied to general data compression, the approach does not compress the Virtex bit-stream effectively, mainly because it does not exploit the specific regularities within the configuration data. Our compression techniques outperform it by a factor of two.

Another approach, called Run-length compression [Hauck99], was developed to compress the Xilinx 6200 bit-stream. A series of addresses with a common offset can be compressed into a form of base, offset, and length. Also, the repeated configuration data values can be compressed with Run-Length encoding. Results demonstrate a factor of 3.6 average compression ratio, slightly worse than our Wildcard algorithm.

3.7 Summary

In this chapter, configuration compression techniques to reduce reconfiguration overhead were discussed. Similar to general data compression, configuration compression takes advantage of regularities and repetitions within the original configuration data. However, using existing lossless compression approaches cannot significantly reduce the size of configuration bit-streams, because there are several fundamental differences between general compression and configuration compression. The unique regularities and on-chip run-time decompression require distinct compression algorithms for different architectures.

In this chapter, we have investigated configuration compression techniques for the Xilinx 6200 FPGAs and the Xilinx Virtex FPGAs. Taking advantage of on-chip

Wildcard Registers, our Wildcard algorithm can achieve a factor of 3.8 compression ratio for the Xilinx 6200 FPGAs without adding extra hardware. A number of compression algorithms were investigated for Virtex FPGA -- the most popular commercial configurable devices. These algorithms can significantly reduce the amount of data that needs to be transferred with a minimum modification of hardware. In order to explore the best compression algorithm, we have extensively researched current compression techniques, including Huffman coding, Arithmetic coding, and LZ coding. Furthermore, we have developed different algorithms targeting different hardware structures. Our Readback algorithm allows certain frames to be reused as a dictionary and sufficiently utilizes the regularities within the configuration bit-stream. Our Frame Reordering algorithms exploit regularities by shuffling the sequence of the configuration. The simulation results demonstrate that a factor of four compression ratio can be achieved. As mentioned earlier, the configuration compression algorithms we developed can be extended to any similar reconfigurable devices without significant modifications.

Chapter 4

Don’t Care Discovery for Configuration Compression

The results in Chapter 3 demonstrate that configuration compression can effectively reduce the amount of configuration data that needs to be transferred. All algorithms we developed are considered lossless compression approaches, since no configuration information is lost during the compression stage. In order to further reduce configuration data, a lossy approach that can increase regularities, called Don’t Care discovery, is discussed in this chapter. By combining Don’t Care discovery with the compression algorithms developed in Chapter 3, higher compression ratios can be achieved.

4.1 Don’t Cares Overview

There are two types of Don’t Cares for each configuration: *True Don’t Cares* and *Partial Don’t Cares*. In practice, most applications do not utilize all function units or routing resources. Therefore, unused function units and routing resources will not affect computational results, and the corresponding configuration bits for these locations can be considered as True Don’t Cares. For other locations, not all configuration bits are important to computation, and some bits can be turned into Don’t Cares without causing errors. We call these bits Partial Don’t Cares. By exploring Don’t Cares and then assigning each Don’t Care bit a certain value, we can increase regularities within each application.

For example, each cell of a Xilinx 6200 FPGA can route signals in four different directions. However, in most cases, only one or two directions are actually used for the computation, so the configuration for unused directions can be treated as Partial Don't Cares. Although none of these locations is a True Don't Care, regularities for different locations may increase by turning bits into Don't Cares, and thus fewer cubes to cover the necessary configuration can be found. Suppose there are only two locations specified in a configuration, with address 1 containing data "00101000" and address 2 containing data "00100000". Obviously, two separate configuration writes are required. However, assume that we can modify the value in address 1 to "0010-000", where "-" means Don't Care. Without considering the overhead of the Wildcard Register write, one write is now sufficient to complete the configuration of both locations.

Obviously, exploring Don't Cares requires detailed information about configuration bit-streams. Additionally, an algorithm that uses the information to discover all configuration bits that sufficiently contribute to correct computation must be developed. Furthermore, all output locations, including IOBs and registers, must be specified–from the user's point of view, these locations contain the information that the user really needs. The outputs of these locations are computed by logic operations on the inputs to them, meaning that the locations providing these inputs could affect the results of the outputs. This identifies all fields within newly specified locations are critical to the computation results. Our algorithm backtraces the inputs to these fields and gets another set of important fields. This backtracing process is repeated until all important fields for the computation are traversed. Notice that these traversed fields normally represent a subset of the given configuration. This is because some configuration bits specified in the configuration file become Don't Cares, meaning that we can assign arbitrary values to them.

As mentioned in Chapter 3, one major concern for any lossy approach is whether the correctness of computation can be held for circuitry generated by decompressed data.

Using our Don't Care discovery technique, the given configuration can be changed to a different configuration, since new values can be assigned to the newly discovered Don't Care bits. However, the resulting computation of the two configurations will be identical. From the user's point of view, if the outputs of both configurations produce the same result, we can safely say that both configurations meet the user's needs. Since the backtracing starting from the outputs for a given configuration covers all important fields necessary to the outputs, the computation correctness is maintained.

We have seen that the original configuration can be changed to a different one. However, we must ensure that the new configuration will not damage the reconfigurable device. Our algorithm in this work is optimized for the Xilinx 6200 and Virtex FPGAs, whose architectures have safeguards that prevent short-circuits from being created in the programming of the FPGAs. In systems where a bad configuration could cause a short-circuit on the device, a simple algorithm that eliminates the side effects of Don't Care discovery is necessary.

One final concern is that the new configuration will overwrite locations that may be used by other configurations. Since the locations traversed during backtracing contain information for the correct computation, those locations must be specified by the original configuration or by initialization (Reset) values. In either case, if the given configuration does not overwrite any locations that are used by other computations, the new configuration also will not, since it is a subset of the given one.

4.2 The Backtracing Algorithm

Given a configuration file, the backtracing algorithm seeks to discover the Don't Cares. Once this stage is complete, with minor modifications, our configuration compression algorithm can be applied to find a compressed version of a configuration. The algorithm starts from the output cells (user-defined registers) and output IOBs, backtracing all configuration bits that contribute to the correct computation. This determines all

programming bits necessary for correct computation, meaning that all other bits don't matter and can thus be considered as Don't Cares.

During backtracing we seek all portions of a circuit that help produce a given signal. Once these regions are found for each circuit output, we have identified all locations that must be configured with a specified value. Thus, all other locations can be treated as Don't Cares. For example, in Figure 4.1, the only output of the circuit is "O". We backtrace this signal, discovering that it is computed by a register. This means that its clock circuitry and its input "A" are important. Backtracing A will show that the function block of this cell is important, requiring B and C to be backtraced. Eventually, we will reach the registers that start this computation. Using this recursive backtracing process, we will identify the entire circuitry shown. For this example all other configuration data is irrelevant to proper circuit function, and can be considered as Don't Care. Thus, all Northward and Westward routing resources, the logic blocks of cells 1 and 2, and the register in cell 3 can be configured arbitrarily. It is this flexibility that helps boost compression ratios significantly.

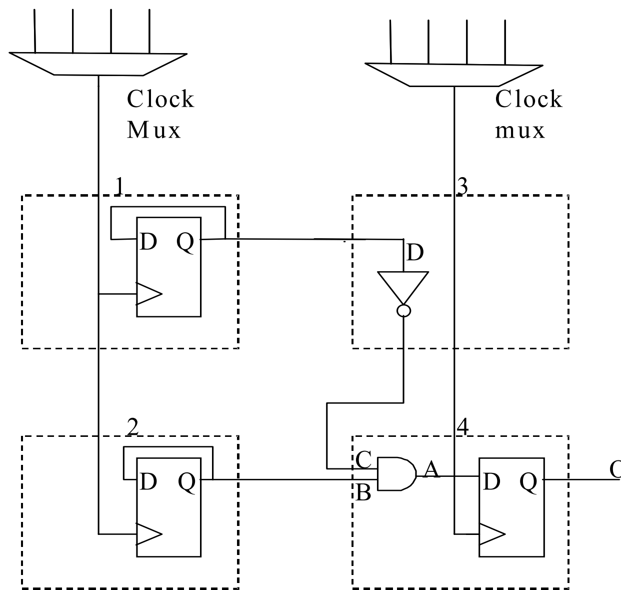


Figure 4.1: Sample circuit for backtracing.

Since Xilinx does not disclose the information necessary for discovering the Don't Cares in Virtex applications, we will focus mainly on algorithm design for the Xilinx 6200 architecture. However, we will still estimate the potential impact of Don't Cares for Virtex compression.

4.3 Don't Care Discovery for the Xilinx 6200 FPGAs

In order to do the backtracing, we need information about output locations. One set of our benchmarks is compiled by XACT6000 tools, which produce a symbol table file (.sym file) that specifies the locations of all circuit inputs and outputs. For another set of benchmarks that is not created by XACT6000 tools, we create the symbol files that consist of output information provided by the designers.

4.3.1 Don't Care Discovery Algorithm

As discussed in Section 4.2, the key technique for Don't Care discovery is to backtrace all components that produce outputs. There are three major components in the array:

cells, switches and IOBs. There are 4096 cells arranged in a 64 × 64 array, and each cell has 3 separate 8-bit configuration bytes. One of these bytes controls the neighbor routing multiplexers, and two others control the functionality. Switches are located at the boundary of blocks of 4 × 4 cells, and they are labeled according to the signal travel direction. Each of the east and west switches has one configuration byte controlling neighbor routing, length 4 wire routing, and length 16 wire routing. Each north and south switch has multiple configuration bytes that control: neighbor routing, length 4 and length 16 routing, and global signals including clock and clear lines. Each IOB consists of multiple configuration bytes controlling routing and some circuit control signals. A configuration can be viewed as the configurations of the multiplexers in cells, switches, and IOBs. If any multiplexer in a specified unit (cells, switches and IOBs) is not used for the computation, then the corresponding configuration bits for that multiplexer are considered Don't Cares. We now present details on how to find Don't Cares for cells, switches and IOBs.

Figure 4.2 shows the basic XC6200 cell in detail, with the function unit at left and cell routing resources at right. Input multiplexers select outputs from neighbors or from length 4 wires to connect to X1, X2, and X3. The Y2 and Y3 multiplexers provide for conditional inversion of the inputs. The CS multiplexer selects a combinatorial or sequential output. The RP multiplexer controls the contents of the register to be "protected". If the register is configured as "protected", then only the user interface can write it.

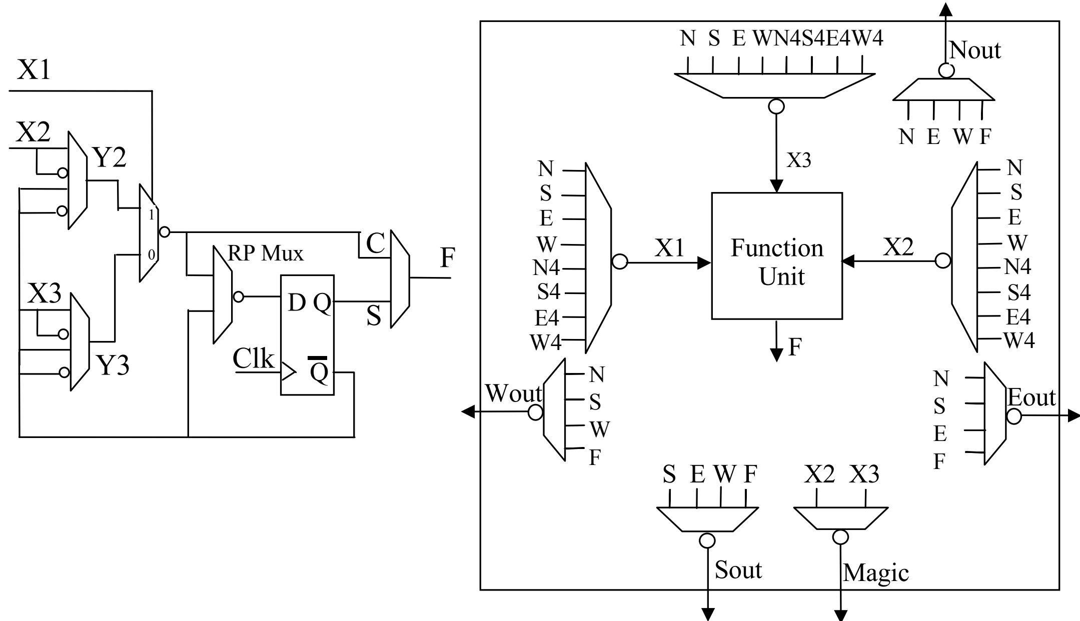


Figure 4.2: Xilinx 6200 function unit and cell routing.

Two configuration bytes control the multiplexers for the function unit. Don't Care discovery depends on the functionality of the cell. For example, if the CS multiplexer selects the sequential output and the RP multiplexer configures the register as protected (feeds the register output back into its input), then all X and Y multiplexers can be set as Don't Cares because the user interface is the only source that can change the F output. If either the Y2 or Y3 multiplexer selects the output of the register, then the corresponding X multiplex can be set to Don't Care. The X1 multiplexer can be set to Don't Care if Y2 and Y3 both select the same signal. For any of the four neighbor routing multiplexers not used for computation or routing, the bits for controlling the multiplexer can be considered Don't Cares.

Figure 4.3 shows the North switch at 4 × 4 block boundaries. Two multiplexers control neighbor routing and length 4 routing to the North, and there is an additional length 16 multiplexer at each 16 × 16 boundary. South, East and West switches have structures that are similar to the North switches. Generally, if any of the multiplexers is not used,

then the configuration bits for that multiplexer can be set to Don't Cares. However, the configuration bits for the Nout multiplexer cannot be set to Don't Cares if the N4out multiplexer selects NCout, since the same programming bits control the upper and lower four input multiplexers. If NCout and Nout select different inputs, both inputs must be backtraced.

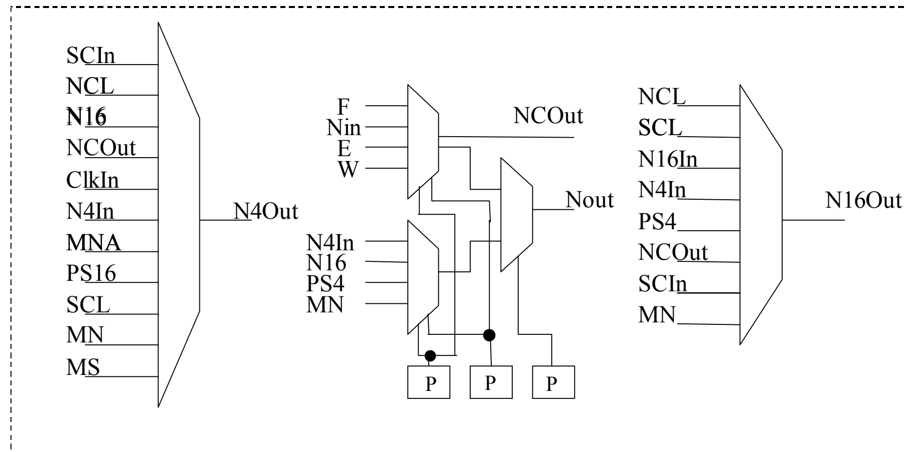


Figure 4.3: The Xilinx 6200 North switch at 4 × 4 block boundaries.

Each North switch contains an additional Clock multiplexer. This multiplexer is traversed only if a cell in the same column within the 4 × 4 block is configured as a register. Each South switch at the 16 × 16 boundary contains a Clear multiplexer. This multiplexer is traversed only if any cell at the same column within the 16 × 16 block is configured as a register.

Our algorithm does not attempt to find Don't Cares in IOBs for two reasons. First, there are only 64 IOBs at each side of the array, meaning that we will not benefit significantly from finding Don't Cares. Second, the architecture of IOB involves many circuit-control signals that cannot be turned to Don't Cares. However, our algorithm does traverse identified IOBs to backtrace other units. Thus, our algorithm is conservative, since it may not discover Don't Cares in IOBs, but will always produce valid output.

We now present the basic steps of our Don't Care discovery algorithm.

1. Read the input .cal file and mark a unit as "touched" if any part of it is specified in the .cal file. Mark all configuration bits as Don't Cares.
2. Read the .sym file and put all output units (IOBs and registers used as outputs) into a queue.
3. Remove a unit from the queue. If it has already been backtraced, ignore it. Otherwise, mark its configuration bits as "no longer Don't Care", and insert its important inputs into the queue. Mark the unit as "touched".
4. If the queue is not empty, goto Step 3.
5. Produce a new target configuration where:
   5.1. All locations that were not marked as touched are considered as Don't Touch.
   5.2. All bits that were marked as "no longer Don't Care" are assigned their values from the .cal file.
   5.3. All other bits are Don't Cares.

Note that in situations where the configuration given to the compression algorithm represents the entire logic that will be mapped to the array, it does not matter what happens to unused cells in the FPGA. In these cases, Step 5 instead sets locations not marked as touched to Don't Care.

4.3.2 Compression Algorithm Modifications

Once the Don't Care discovery algorithm is complete, we have a list of address data pairs, with Don't Care bits contained in many of the data values. In order to take advantage of these Don't Cares we need to modify to our configuration compression algorithm.

In our original configuration compression algorithm, locations with the same data value are placed in the same group. This is because the addresses with the same value represent an On set in the corresponding logic minimization problem. However, by discovering the Don't Care bits, each On set can be represented by a set of locations that does not necessarily consist of the same value. After modifying the Don't Cares to "1" or "0", the locations with different values in the given configuration can be placed into the same group, since they are compatible. Notice that it is now possible for an address to fit into multiple groups instead of just one group in our original compression algorithm because of the Don't Cares, meaning that the flexibility for our configuration compression algorithm has increased. For example, suppose that after the discovery of Don't Care bits, address A contains data "00-000-0". Assume there are 3 groups, where group 1 has the value "00000000", group 2 has the value "00000010" and group 3 has the value "00100000". Address A is compatible with the value of each of the three groups and is placed into them. Writing any value representing the three groups into address A properly configures it. This is because any of the three values can create the configuration necessary for the computation. Even though address A may be overwritten by values from the other two groups, the necessary configuration is maintained. Our original algorithm can take advantage of this feature to find fewer cubes covering the necessary configuration.

In our original configuration compression algorithm, the data associated with an address has a fixed value, so the locations were grouped by their values. However, after running the Don't Care discovery algorithm, a location with Don't Cares can be placed into multiple groups depending on their compatibility. Thus, we need to develop an algorithm to group the locations so that the addresses (locations) in each group are compatible. An address (location) can appear in as many as 2^n groups, where n is the number of Don't Care bits contained in its data value. Notice that compatibility is not transitive. That is, if A and B are compatible, and B and C are compatible, it is not always true that A and C are compatible. For example, assume A, B and C have values

"000100-0", "0-0-0000" and "0100-000", respectively. A and B are compatible, and B and C are compatible, but A and C are not compatible. This non-transitivity property is an important consideration, making grouping decisions complex.

For 8-bit data, the simplest method for grouping is to create 256 groups with the values 0 to 255, and place each address data pair into every group with a compatible value. However, this technique has exponential time complexity; to extend this technique to a 32-bit data bus, the number of groups needed is 2^{32}. It is obvious that a heuristic method is needed. We present our heuristic grouping algorithm as follows:

1. Once Don't Care discovery is complete, put those addresses with Don't Care data bits into a list. Group those addresses without Don't Care Data bits according to their data values.
2. Search the list, removing those addresses that can be fit into any of the current groups, and put them into all compatible groups.
3. Repeat until the list is empty:
   - 3.1. Pick a location from the list with the fewest Don't Care bits.
   - 3.2. The value for the group equals the value for the picked location, but with all Don't Care bits converted to "0" or "1". These bits are converted iteratively, converting to the value that is most compatible with other locations.
   - 3.3. Add all locations compatible with this value to the group. If they are on the unassigned list, remove them.

We also need to modify other steps of the configuration compression algorithm. First, we present the modified algorithm:

1. Apply the Don't Care discovery algorithm to find Don't Cares. Group the address data pairs by using our grouping algorithm. Mark the address locations specified

in given .cal file as "unoccupied". Mark the address locations not specified in the .cal file, but used in the backtrace, as "occupied".

2. Sort the groups in decreasing order of the number of addresses unoccupied in that group.
3. Pick the first group and write the addresses in the group to the Espresso input file as part of the On set.
4. Write all "unoccupied" addresses to the Espresso input file as part of the Don't Care set.
5. Write all addresses marked "occupied", yet with a value compatible with the group, to the Espresso input file as part of the Don't Care set.
6. Run Espresso.
7. Pick the cube from the Espresso output that covers the most unoccupied addresses in the first group and add it to the compressed configuration file. Mark all covered addresses as "occupied".
8. If the cube did not cover all of the addresses in the group, reinsert the group into the sorted list.
9. If any addresses remain unoccupied, go to Step 2.

This Don't Care discovery algorithm has several classes of locations: configured, untouched, and initialized. *Configured locations* are those whose value is set in the input .cal file, and our algorithm will generate a write to set these values. *Untouched locations*, which are not found in either the backtrace or the .cal file, can be viewed as either Don't Touch, if these unused cells may be used for other functions, or Don't Care, if the cells will be left unused. *Initialized locations* are locations that are not set in the .cal file, but are discovered to be important during backtracing. Thus, the initialization value must be used. Our algorithm handles these locations as potential

group members marked as "occupied". As a result, compatible values can overwrite these locations to achieve better compression, but the algorithm is not required to write to these locations if it is not advantageous.

4.3.3. Experimental Results

The results are shown in Table 4.1 (as well as in Figure 4.4). The size of the initial circuit is given in the "Input size" column. This size includes all writes required to configure the FPGA, including both compressible writes to the array, as well as non-compressible control register writes. The "Ctrl" column represents the number of non-compressible writes, and is a fixed overhead for both the original and compressed file. The results of the compressed version achieved by our original algorithm are shown in the column "Original Compression". The results of the compressed version by our new algorithm are shown in the column "New algorithm", with unspecified locations considered as Don't Touch (the configuration bits for these locations cannot be changed) or Don't Care depending on the details of the use of these configurations.

Table 4.1. The results of the compression algorithms.

Bench-mark	Input size	Ctrl	Original compression algorithm				New algorithm (Don't Touch)				New algorithm (Don't Care)			
			Cnfg	Wcrd	Ratio1	Ratio2	Cnfg	Wcrd	Ratio1	Ratio2	Cnfg	Wcrd	Ratio1	Ratio2
Counter	199	40	53	13	53.2%	41.5%	29	5	37.2%	21.4%	22	4	33.2%	16.4%
parity	208	16	9	3	13.5%	6.3%	6	2	11.5%	4.2%	6	2	11.5%	4.2%
Add4	214	40	43	14	45.3%	32.7%	24	7	33.2%	17.8%	16	6	29.0%	12.6%
zero32	238	42	12	3	23.9%	7.7%	8	3	22.3%	5.6%	6	3	21.4%	4.5%
adder32	384	31	28	14	19.0%	11.9%	20	13	16.7%	9.3%	20	13	16.7%	9.3%
Smear	696	44	224	37	43.8%	40.0%	150	36	33.0%	28.5%	121	32	28.3%	23.5%
Add4rm	908	46	473	45	62.1%	60.1%	279	78	44.3%	41.4%	203	65	34.6%	31.1%
Gray	1201	44	530	74	53.9%	52.2%	378	53	39.5%	37.3%	311	44	33.2%	30.4%
Top	1367	70	812	87	70.8%	69.3%	531	65	48.7%	46.0%	419	57	39.9%	36.7%
demo	2233	31	423	91	24.4%	23.3%	281	77	17.4%	16.3%	241	66	15.1%	13.9%
ccitt	2684	31	346	84	17.2%	16.2%	235	55	12.0%	11.0%	204	50	10.6%	9.6%
t	5819	31	834	192	18.2%	17.7%	567	176	13.3%	12.8%	492	162	11.8%	11.3%
correlat	11011	38	1663	225	17.4%	17.2%	1159	187	12.6%	12.3%	1004	176	11.0%	10.8%
Totals:														
w/ctrl	27162			6836	(25.2%)			4928	(18.1%)			4249	(15.6%)	
w/o ctrl	26658			6332	(23.8%)			4424	(16.6%)			3745	(14.0%)	

The number of writes to configure the logic array is shown in the column "Cnfg", the number of Wildcard Register writes is shown in "Wcrd", and "Ratio1" is the ratio of the total number of writes (the summation of "Ctrl", "Cnfg" and "Wcrd") to the size of the input configurations. Notice that the "Ctrl" writes represent a fixed startup cost that can often be ignored during Run-time reconfiguration. Thus, to reflect the compression ratio without this initial startup cost, we use "Ratio2", which equals to ("Cnfg" + "Wcrd")/("Input size" – "Ctrl"), to represent the compression ratio for the compressible parts of the circuits. In the last two rows, the total number of writes and compression ratios of all benchmarks are calculated for two cases, with and without counting the "Ctrl" writes. As can be seen, the use of Don't Care discovery as pre-processing can improve the average compression factor from 4 to 7.

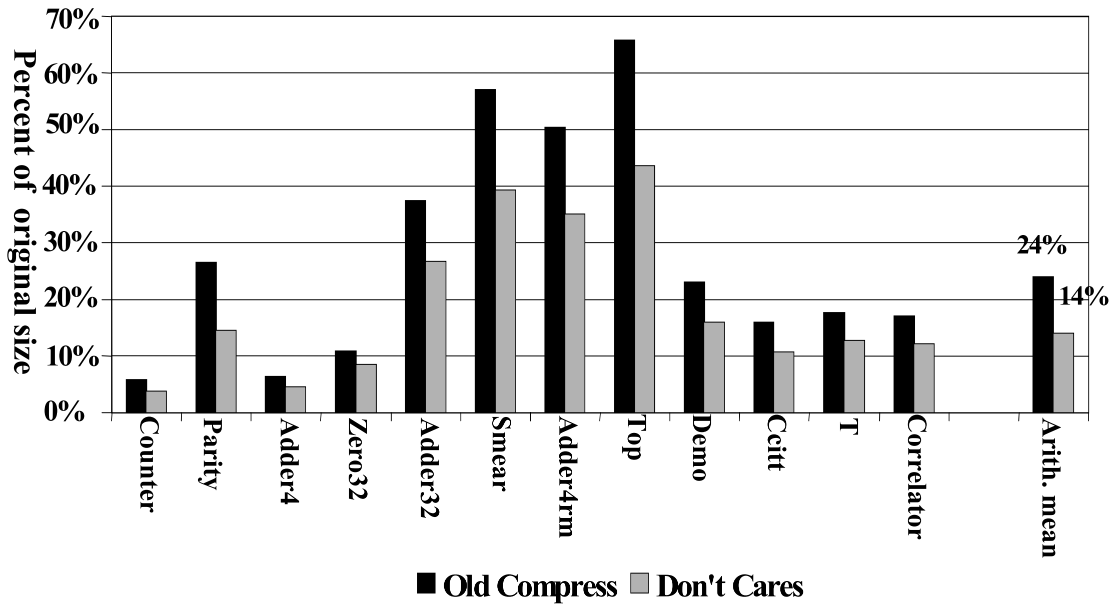


Figure 4.4: Experimental results of the compression algorithms

4.4 Virtex Compression with Don't Cares

Although Xilinx does not disclose the information necessary to discover Don't Cares in the Virtex applications, we can still evaluate the potential impact of the Don't Cares for Virtex compression. In order to make an estimate, we randomly turn some bits of the data stream into Don't Cares and bound the impact of Don't Cares on our Readback algorithm.

In practice, the discovered Don't Care bits need to be turned to '0' or '1' to produce a valid configuration bit-stream. The way that the bits are turned affects the frame sequence and thus the compression ratio. Finding the optimal way to turn the bits takes exponential time. We have used a simple greedy approach to turn these bits to create an upper-bound for our Readback algorithm. The configuration sequence graph is built taking into account the Don't Cares. We greedily turn the Don't Care bits into '0' or '1' to find the best matches. Note that once a bit is turned, it can no longer be used as a

Don't Care. To discover the lower-bound, we do not turn the Don't Care bits; thus, they can be used again to discover better matches.

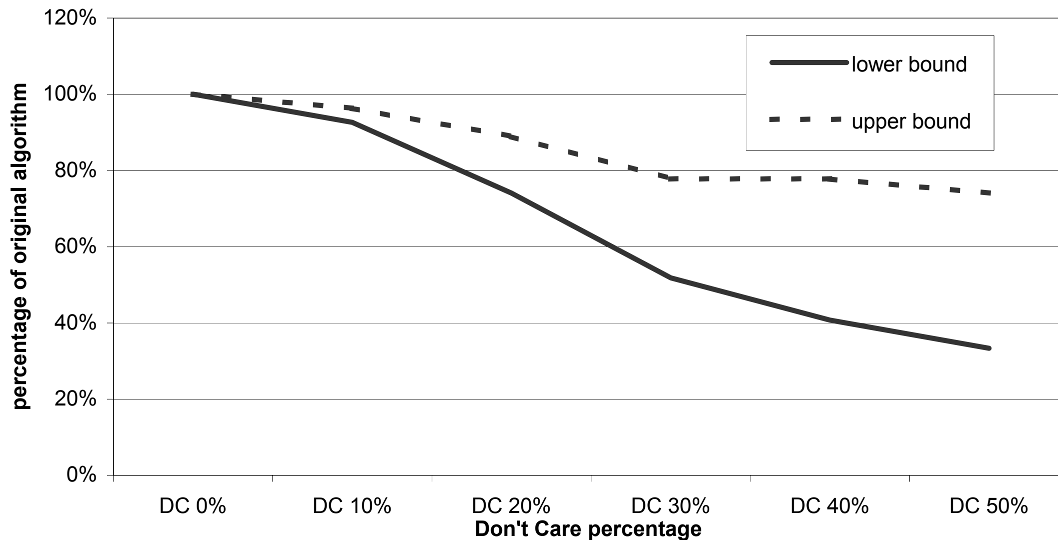


Figure 4.5: The effect of Don't Cares on benchmarks in Table 3.2 for Virtex compression.

Figure 4.5 demonstrates the potential effect of Don't Cares over the benchmarks listed in Table 3.2. The X-axis is the percentage of the don't cares we randomly create and the Y-axis is the normalization over the results without considering Don't Cares. As can be seen in Figure 4.5, by using upper-bound approach a factor of 1.3 improvement can be achieved on applications containing 30% Don't Cares, while a factor of 2 improvement can be achieved using the lower-bound approach.

4.5 Summary

Lossy approaches can be applied to achieve better configuration compression. However, the configuration changed by a lossy compression must generate same computational results as the original one and must not damage the reconfigurable device. In this chapter, we have presented an efficient lossy approach, called Don't Care discovery.

Realizing that Don't Cares increase regularities, our Don't Care discovery technique backtraces important locations starting from the outputs, generating a new configuration. A subset of the original configuration, this new configuration discovers the locations that are sufficient to produce the correct computation. All other locations can be treated as Don't Cares, increasing regularities within the configuration data. A significant improvement in compression ratios is achieved by combining this technique with our lossless techniques presented in Chapter 3.

Chapter 5

Configuration Caching

Configuration caching — the process of determining which configurations that are retained on the reconfigurable hardware until they are required again -- is another technique to reduce reconfiguration overhead. However, the limited the on-chip memory and the non-uniform configuration latency add complexity to decide which configurations to retain to maximize the odds that the required data is present in the cache. In this chapter, we present new caching algorithms targeted to a number of different FPGA models.

5.1 Configuration Caching Overview

Caching configurations on an FPGA is similar to caching instructions or data in a general memory. It retains the configurations on reconfigurable hardware so the amount of data that needs to be transferred to the chip can be reduced. In configuration caching, we view the area of the FPGA as a cache. If this cache is large enough to hold more than one computation, configuration cache management techniques can be used to determine when configurations should be loaded and unloaded to best minimize overall reconfiguration times.

In a general-purpose computational system, caching is an important approach to hide memory latency by taking advantage of two types of locality: *spatial locality* and *temporal locality*. Spatial locality states that items whose addresses are near one another tend to be referenced close together in time. Temporal locality addresses the tendency of recently accessed items to be accessed again in the near future. These two localities also apply to the caching of configurations for reconfigurable systems. However, the

traditional caching approaches for general-purpose computational systems are unsuited to configuration caching for the following reasons:

1) In general-purpose systems, the data loading latency is fixed, because the block represents the atomic data transfer unit; in reconfigurable systems, the loading latency of configurations may vary due to non-uniform configuration sizes. This variable latency factor could have a great impact on the effectiveness of caching approaches. In traditional memory caching, frequently accessed data items are kept in the cache in order to minimize the memory latency. However, this might not be true in reconfigurable systems because of the non-uniform configuration latency. For example, suppose that we have two configurations with latencies 10ms and 1000ms, respectively. Even though the first configuration is executed 10 times as often as the second, the second is likely to be cached in order to minimize the total configuration overhead.

2) In configuration Caching, the large size of each configuration means that only a small number of configurations can be retained on-chip. This makes the system more likely to suffer from the "thrashing problem", in which configurations are swapped excessively between the configuration memory and the reconfigurable device.

The challenge in configuration caching is to determine which configurations should remain on the chip and which should be replaced when reconfiguration occurs. An incorrect decision will fail to reduce reconfiguration overhead and can lead to a much higher overhead than a correct decision. Non-uniform configuration latency and the small number of configurations that reside simultaneously on the chip increase the complexity of this decision. Both frequency and latency factors of configurations need to be considered to assure the highest reconfiguration overhead reduction. Specifically, in certain situations retaining configurations with high latency is better than keeping frequently required configurations that have lower latency. In other situations, keeping

configurations with high latency and ignoring the frequency factor results in thrashing between other frequently required configurations because they cannot fit in the remaining area. This switching causes reconfiguration overhead that would not have occurred if the configurations with high latency but low frequency were unloaded.

In addition, the different features of various reconfigurable models add complexity to configuration caching. The specific architecture and control structure of each reconfigurable model require unique caching algorithms.

5.2 Reconfigurable Models Review

In order to explore the best configuration architecture, we now evaluate five reconfigurable models discussed in Chapter 2.

For a Single Context FPGA, the whole chip area must be reconfigured during each reconfiguration. Even if only a small portion of the chip needs to reconfigure, the whole chip is rewritten during the reconfiguration. Configuration caching for the Single Context model allocates multiple configurations that are likely to be accessed near in time into a single context to minimize switching of contexts. By caching configurations in this way, the reconfiguration latency is amortized over the configurations in a context. Since the reconfiguration latency for a Single Context FPGA is fixed (based on the total amount of configuration memory), minimizing the number of times the chip is reconfigured will minimize the reconfiguration overhead.

The configuration mechanism of the Multi-Context model is similar to that of the Single Context FPGA. However, instead of having one configuration stored in the FPGA, multiple complete configurations are stored. Each complete configuration can be viewed as multiple configuration memory planes contained within the FPGA. For the Multi-Context FPGA, the configuration can be loaded into any of the contexts. When needed, the context containing the required configuration is switched to control the logic and interconnect plane. Compared to the configuration loading latency, the single cycle

configuration switching latency is negligible. Because every SRAM context can be viewed as a Single Context FPGA, the methods for allocating configurations onto contexts for the Single Context FPGA could be applied.

For the Partial Run-Time Reconfigurable (PRTR) FPGA, the area that is reconfigured is just the portion required by the new configuration, while the rest of the chip remains intact. Unlike the configuration caching for the Single Context FPGA, where multiple configurations are loaded to amortize the fixed reconfiguration latency, the configuration caching method for the PRTR is to load and retain configurations that are required rather than to reconfigure the whole chip. The overall reconfiguration overhead is the summation of the reconfiguration latency of the individual reconfigurations. Compared to the Single Context FPGA, the PRTR FPGA provides greater flexibility for performing reconfiguration.

Current PRTR systems are likely to suffer from "thrashing problems" when two or more frequently used configurations occupy overlapping locations in the array. Simply increasing the size of the chip will not alleviate this problem. However, the Relocation model [Compton00], which dynamically allocates the position of a configuration on the FPGA at run-time instead of at compile time, can minimize its impact.

The Relocation + Defragmentation model (R+D model) can significantly improve hardware utilization by collecting the small unused fragments into a single large one. This allows more configurations to be retained on the chip, increasing the hardware utilization and thus reducing the reconfiguration overhead. For example, Figure 5.1 shows three configurations currently on-chip with two small fragments. Without defragmentation, one of the three configurations has to be replaced when Configuration 4 is loaded. However, as shown in the right side of Figure 5.1, by pushing Configurations 2 and 3 upward, the defragmentor produces one single fragment that is large enough to hold Configuration 4. Notice that the previous three configurations are

still present, and therefore the reconfiguration overhead caused by reloading a replaced configuration can be avoided.

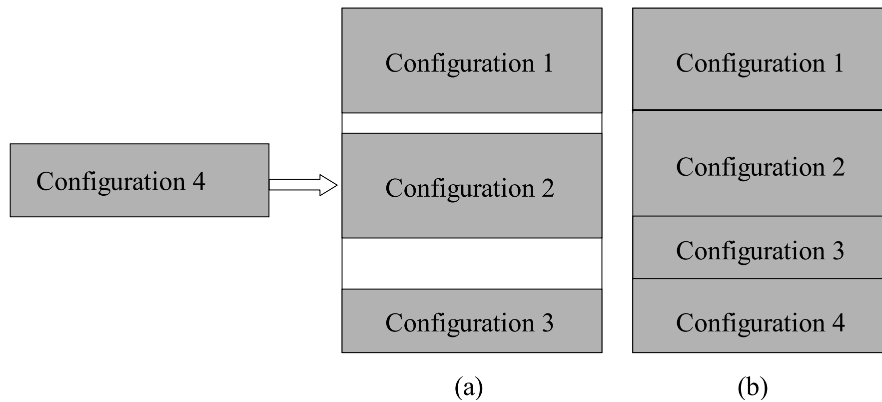


Figure 5.1: An example illustrating the effect of defragmentation. (a) The two small fragments are located between configurations, and neither of them is large enough to hold Configuration 4. (b) After defragmentation, Configuration 4 can be loaded without replacing any of the three other configurations.

5.3 Experimental Setup

In order to investigate the performance of configuration caching for the five different models presented above, we developed a set of caching algorithms for each model. A fixed amount of hardware resources (in the form of overall area) is allocated to each model. To conduct the evaluation, we must perform three steps. First, for each model, the capacity equation must be derived for a given architecture model and a given area. Since the number of programming bits represent the maximum amount of the configuration information that a model can retain, the number of programming bits is calculated to represent the capacity of each model. Second, we test the performance of the algorithms for each model by generating a sequence of configuration accesses from an execution profile of each benchmark. Third, for each model, caching algorithms are executed on the configuration access sequence, and the configuration overhead for each algorithm is measured.

5.4. Capacity Analysis

Layout for the programming structure of each reconfigurable model is required to carry out capacity analysis. Our fellow graduate student Katherine Compton created layouts of all five reconfigurable models using the Magic tool. The area models are based on the layout of tileable structures that composed the necessary memory system, and the sizes (in lambda^2) are obtained for the tiles [Compton00].

The Single Context FPGA model is built from shift chains or RAM structures. The PRTR FPGA, however, requires more complex hardware. The programming bits are held in five-transistor SRAM cells that form a memory array similar to traditional RAM structures. Row decoders and column decoders are necessary to selectively write to the SRAM cells. Large output tri-state drivers are also required near the column decoder to magnify the weak signals provided by the SRAM cells when reading the configuration data off the array. The Multi-Context FPGA is based on the information found in [Trimberger97], where each context is similar to a single plane of a PRTR FPGA. A few extra transistors and a latch per active programming bit are required to select between the four contexts for programming and execution. Additionally, a context decoder must be added to determine which of those transistors should be enabled.

The PRTR design forms the basis of the PRTR with Relocation FPGA. A small adder and a small register, both equal in width to the number of address bits for the row address of the memory array, were added for the new design. This allows all configurations to be generated so that the "uppermost" address is 0. Relocating the configuration is therefore as simple as loading an offset into the offset register, and adding this offset to the addresses supplied when loading a configuration. Finally, the R+D model is similar to the PRTR with Relocation, with the addition of a row-sized set of SRAM cells that forms a buffer between the input of the programming information and the memory array itself.

In order to account for the size of the logic and interconnect in these FPGAs, we assume that the programming layer of a Single Context FPGA uses approximately 25% of the area of the chip. All other architectures are assumed to require this same logic and interconnect area per bit of configuration (or active configuration in the case of a Multi-Context device). See [Compton00] for calculation details.

As mentioned before, all models are given the same total silicon. However, due to the differences in the hardware structures, the number of programming bits, and thus the capacity of the device, vary among models. For example, according to [Compton00], a Multi-Context model with one megabit of active configuration information and three megabits of inactive information has same area as a PRTR with 2.4 megabits of configuration information. Thus, the PRTR devices have 2.4 times as many logic blocks as the Multi-Context device, but require 40% less total configuration space.

5.5 Configuration Sequence Generation

We use two sets of benchmarks to evaluate the caching algorithms for FPGA models. One set of benchmarks was compiled and mapped to the Garp architecture [Hauser97], where the computational intensive loops of C programs are extracted automatically for acceleration on a tightly-coupled dynamically reconfigurable coprocessor [Callahan99]. The other set of benchmarks was created for the Chimera architecture [Hauck97]. In this system, portions of the code that can accelerate computation are mapped to the reconfigurable coprocessor [Hauck98]. In order to evaluate the algorithms for different FPGA models, we need to create an RFUOP access trace for each benchmark, which is similar to a memory access string used for memory evaluation.

The RFUOP sequence for each benchmark was generated by simulating the execution of the benchmark. During the simulated execution, the RFUOP ID is output when an RFUOP is encountered. Once the execution ends, an ordered sequence of the execution of RFUOPs is created. In the Garp architecture, each RFUOP in the benchmark

programs has size information in term of number of rows occupied. For Chimaera, we assume that the size of an RFUOP is proportional to the number of instructions mapped to that RFUOP.

5.6 Configuration Caching Algorithms

For each FPGA model, we develop realistic algorithms that can significantly reduce the reconfiguration latencies. In order to evaluate the performance of these realistic algorithms, we also attempt to develop tight lower-bound algorithms by using complete application execution information. For the models where true lower-bound algorithms are unavailable, we develop algorithms that we believe are near optimal.

We divide our algorithms into three categories: *run-time algorithms*, *general off-line algorithms*, and *complete prediction algorithms*. The classification of the algorithms depends on the time complexity and input information needed for each algorithm.

The run-time algorithms use only basic information on the execution of the program up to that point, and thus must make guesses as to the future behavior of the program. This is similar to run-time cache management algorithms such as LRU. Because of the limited information at run-time, a prediction of keeping a configuration or replacing a configuration may not be correct, and can even cause higher reconfiguration overhead. Therefore, we believe that these realistic algorithms will provide an upper-bound on reconfiguration overhead, and for some domains better predictors could be developed that improve over these results.

The complete prediction algorithms use complete, exact execution information of the application, and can use computationally expensive approaches. These algorithms attempt to search the whole execution stream to lower configuration overhead. They provide the optimal (lower-bound) or near optimal solutions. In some cases, these algorithms relax restrictions on system behavior in order to make the algorithm a true (but unachievable) lower-bound.

The general off-line algorithms use profile information of each application, with computationally tractable algorithms. They represent realistic algorithms for the case where static execution information is available, or approximate algorithms where highly accurate execution predictions can be developed. These algorithms will typically perform between the run-time and complete prediction algorithms in terms of quality, and are realistic algorithms for some situations.

5.7 Single Context Algorithms

In the next two sub-sections, we present a near lower-bound algorithm (based on simulated annealing), and a more realistic general off-line algorithm, which uses more restricted information. Note that since there are no run-time decisions in a single context device (if a needed configuration is not loaded the only possible behavior is to overwrite all currently loaded configurations with the required configuration), we do not present a run-time algorithm.

5.7.1 Simulated Annealing Algorithm for Single Context FPGAs

When a reconfiguration occurs in a Single Context FPGA, even if only a portion of the chip needs to be reconfigured, the entire configuration memory store will be rewritten. Because of this property, multiple RFUOPs should be configured together onto the chip. In this manner, during a reconfiguration a group (context) that contains the currently required RFUOP, as well as possibly one or more later required RFUOPs, is loaded. This amortizes the configuration time over all of the configurations grouped into a context. Minimizing the number of group (context) loadings will minimize the overall reconfiguration overhead.

It is obvious that the method used for grouping has a great impact on latency reduction; the overall reconfiguration overhead resulting from a good grouping could be much smaller than that resulting from a bad one. For example, suppose there are four

RFUOPs with equal size and equal configuration latency for a computation, and the RFUOP sequence is *1 2 3 4 3 4 2 1*, where *1, 2, 3*, and *4* are RFUOP IDs. Given a Single Context FPGA that has the capacity to hold two RFUOPs, the number of context loads is three if RFUOPs *1* and *2* are placed in the same group (context), and RFUOPs *3* and *4* are placed in another. However, if RFUOPs *1* and *3* are placed in the same group (context) and RFUOPs *2* and *4* are placed in the other, the number of context loads increases to seven.

In order to create the optimal solution for grouping, one simple method is to create all combinations of configurations and then compute the reconfiguration latency for all possible groupings, from which an optimal solution can be found. However, this method has exponential time complexity and is therefore not applicable for real applications. In this work, we instead use a Simulated Annealing algorithm to acquire a near optimal solution. For the Simulated Annealing algorithm, we use the exact reconfiguration overhead for a given grouping as our cost function, and the moves consist of shuffling the different RFUOPs between contexts. Specifically, at each step an RFUOP is randomly picked to move to a randomly selected group; if there is insufficient room in that group to hold the RFUOP, RFUOPs in that group are randomly chosen to move to other groups. Once finished, the reconfiguration overhead of the grouping is computed by applying the complete RFUOP sequence. The steps below outline the complete algorithm:

1. While the current temperature is greater than the terminating temperature:
    - 1.1. While the number of iterations is greater than 0:
        - 1.1.1. A candidate RFUOP is randomly chosen along with a randomly selected destination group to which the candidate will be moved.
        - 1.1.2. After the move, if the total size of the RFUOPs in the destination group exceeds the size of the context, a new candidate RFUOP in the

destination group is randomly selected. This RFUOP is then moved to any group that can hold it. This step is repeated until all groups satisfy the size constraint.

1.1.3. Execute the newly generated grouping on the RFUOP execution sequence and calculate the number of times reconfiguration is performed. The reconfiguration overhead, used as the cost function of this version of simulated annealing, can be calculated by multiplying the number of context switches by the loading latency of a context.

1.1.4. The new and the old cost are compared to determine if the move is allowed, then the number of iterations is decreased by one.

1.2. Decrease the current temperature.

5.7.2 General Off-line Algorithm for Single Context FPGA

Although the Simulated Annealing approach can generate a near optimal solution, its high computation complexity and the requirement of knowledge of the exact execution sequences make this solution unreasonable for most real applications. We therefore propose an algorithm better suited to general-purpose use. The Single Context FPGA requires that the whole configuration memory will be rewritten if a demanded RFUOP is not currently on the chip. Therefore, if two consecutive RFUOPs are not allocated to the same group, a reconfiguration will result. Our algorithm computes the likelihood of RFUOPs following one another in sequence and use this knowledge to minimize the number of reconfigurations required. Before we discuss this algorithm further, we first present the definition of a "correlate" as used in the algorithm:

Definition 5.1: Given two RFUOPs and an RFUOP sequence, RFUOP A is said to correlate to RFUOP B if in the RFUOP sequence there exists any consecutive appearance of A and B.

For the Single Context FPGA, highly correlated RFUOPs should be allocated to the same group. Therefore, the number of times a context is loaded is greatly decreased, minimizing the reconfiguration overhead. In our algorithm, we first build an adjacency matrix of RFUOPs. Instead of using 0 or 1 as a general adjacency matrix does, the degree of correlation of each RFUOP pair (the number of times two RFUOPs are adjacent) is recorded. These correlations can be estimated from expected behavior or determined via profiling. The details of our grouping algorithm are as follows:

1. Create COR, where COR[I, J]= number of times RFUOP I correlates to J.
2. While any A[I, J] > 0, do:
   - 2.1. Find I, J such that COR[I, J] + COR[J, I] is maximized;
   - 2.2. If SIZE[I] + SIZE[J] <= Maximum Context Size;
     - 2.2.1. Merge group I and group J and add their sizes;
     - 2.2.2. For each group K other than I and J:
       - 2.2.2.1. A[I, K] += A[J, K]; A[K, I] += A[K, J];
       - 2.2.2.2. A[J, K] = 0; A[K, J] = 0;
   - 2.3. A[I, J] = 0; A[J, I] = 0;

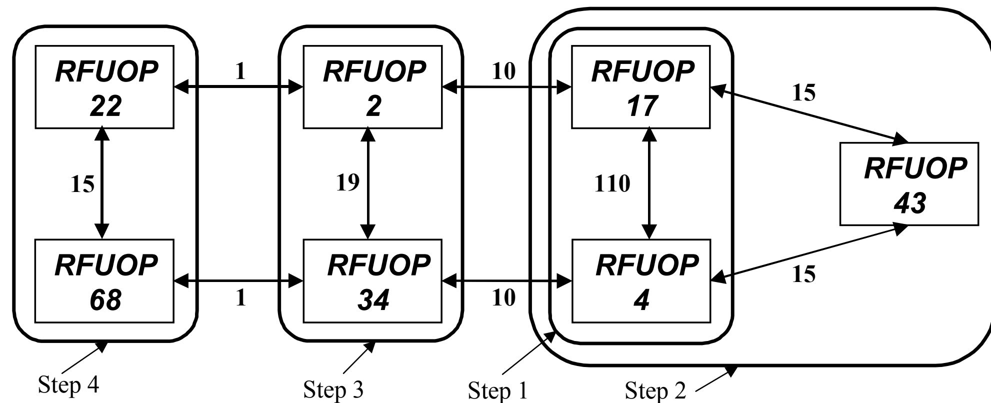


Figure 5.2: An example to illustrate the General Off-line algorithm for Single Context FPGAs.

Figure 5.2 illustrates an example of the General Off-line algorithm. Each line connects a pair of correlated RFUOPs and the number next to each line indicates the degree of the correlation. As presented in the algorithm, we merge the highly correlated groups together under the size constraints of the target architecture. In this example, assume that the chip can retain only a maximum of 3 RFUOPs at a time. In the first grouping step we place RFUOP *17* and RFUOP *4* together. In the second step we add RFUOP *43* into the group formed in Step 1, since it has a correlation of 30 (15+15) to that group. We then group RFUOP *2* and RFUOP *34* together in Step 3, and they cannot be merged with the previous group because of the size restriction. Finally, in the fourth step RFUOP *22* and RFUOP *68* are grouped together.

5.8. Multi-Context Algorithms

In this section we present algorithms for multi-context devices. This includes a Complete Prediction algorithm that represents a near lower-bound, and a General Offline algorithm that couples the Single Context General Offline algorithm with a run-time replacement policy.

5.8.1 Complete Prediction Algorithm for Multi-Context FPGAs

A Multi-Context FPGA can be regarded as multiple Single Context FPGAs, since the atomic unit that must be transferred from the host processor to the FPGA is a full context. During a reconfiguration, one of the inactive contexts is replaced. In order to reduce the reconfiguration overhead, the number of reconfigurations must be reduced. The factors that could affect the number of reconfigurations are the configuration grouping method and the context replacement policies.

We have discussed the importance of the grouping method for the Single Context FPGA, where an incorrect grouping may incur significantly larger overhead than a good grouping. This is also true for the Multi-Context FPGA, where a context (i.e. a group of configurations) remains the atomic reconfiguration data transfer unit. The

reconfiguration overhead caused by the incorrect grouping remains very high even though the flexibility provided by the Multi-Context FPGA can reduce part of the overhead.

As mentioned previously, even the perfect grouping will not minimize reconfiguration overhead if the policies used for context replacement are not considered. A context replacement policy specifies which context should be replaced once a demanded configuration is not present. As in the general caching problem, where frequently used blocks should remain in the cache, the contexts that are frequently used should be kept configured on the chip. Furthermore, if the atomic configuration unit (context) is considered as a data block, we can view the Multi-Context FPGA as a general cache and apply standard cache algorithms.

There is an existing optimal replacement algorithm, called the Belady [Belady66] algorithm for the Multi-Context FPGA. The Belady algorithm is well known in the operating systems and computer architecture fields. It states that the fewest number of replacements can be achieved provided the memory access sequence is known. This algorithm is based on the idea that a data item is most likely to be replaced if it is least likely to be accessed in the near future. For a Multi-Context FPGA, the optimal context replacement can be achieved as long as the context access string is available. Since the RFUOP sequence is known, it is trivial to create the context access string by transforming the RFUOP sequence.

We integrate the Belady algorithm into the simulated annealing grouping method used in the Single Context model to achieve the near optimal solution. Specifically, for each grouping generated, the number of the context replacements determined by the Belady algorithm is calculated as the cost function of the Simulated Annealing algorithm. The steps below outline the Complete Prediction algorithm for the Multi-Context model:

1. Traverse the RFUOP sequence, and for each RFUOP appearing, change the RFUOP ID to the corresponding group ID. This will result in a context access sequence.
2. Initially assign each RFUOP to a group so that, for each group, the total size of all RFUOPs is smaller than or equal to the size of the context. Set up parameters of initial temperature and the number of iterations under each temperature.
3. While the current temperature is greater than the terminating temperature:
   - 3.1. While the number of iterations is greater than 0:
     - 3.1.1. A candidate RFUOP is randomly chosen along with a randomly selected destination group to which the candidate will be moved.
     - 3.1.2. After the move, if the total size of the RFUOPs in the destination group exceeds the size of the context, a new candidate RFUOP in the destination group is randomly selected. This RFUOP is then moved to any group that can hold it. This step is repeated until all groups satisfy the size constraint.
     - 3.1.3. Apply the Belady algorithm to the context access string. Increase the total number of context loads by one if a replacement occurs. This creates the new cost of the simulated annealing.
     - 3.1.4. Compare the new cost to the old cost to determine if the move is allowed, then decrease the number of iterations by one.
   - 3.2. Decrease the current temperature.

The reconfiguration overhead for a Multi-Context FPGA is therefore the number of context loads multiplied by the configuration latency for a single context. As mentioned above, the factors that can affect the performance of configuration caching for the Multi-Context FPGA are the configuration grouping and the replacement policies. Since the optimal replacement algorithm is integrated into the simulated annealing approach, this algorithm will provide the near optimal solution. We consider this to be a complete prediction algorithm.

5.8.2 Least Recently Used (LRU) Algorithm

The LRU algorithm is a widely used memory replacement algorithm in operating system and architecture fields. Unlike the Belady algorithm, the LRU algorithm does not require future information to make a replacement decision. Because of the similarity between the configuration caching and the data caching, we can apply the LRU algorithm for the Multi-Context FPGA model. The LRU is more realistic than the Belady algorithm, but the reconfiguration overhead incurred is higher. Its basic steps are outlined below:

1. Apply the Single Context General Off-line algorithm to acquire a final grouping of RFUOPs into contexts, and give each group formed its own ID.
2. Traverse the RFUOP sequence, and for each RFUOP appearing, change the RFUOP ID to the corresponding group ID. This generates a context access sequence.
3. Apply the LRU algorithm to the context access string. Increase the total number of context loads by one when a replacement occurs.

5.9 Algorithms for the PRTR FPGAs

Compared to the Single Context FPGA, an advantage of the PRTR FPGA is its flexibility of loading and retaining configurations. Any time a reconfiguration occurs, instead of loading the whole group only a portion of the chip is reconfigured, while the other RFUOPs located elsewhere on the chip remain intact. The basic idea of configuration caching for PRTR is to find the optimal location for each RFUOP. This is to avoid the thrashing problem, which could be caused by the overlap of frequently used RFUOPs. In order to reduce reconfiguration overhead for the PRTR FPGA, we need to consider two major factors: the reconfiguration frequency and the latency of each RFUOP. Any algorithm that attempts to lower only one factor will fail to produce an optimal solution because the reconfiguration overhead is the product of the two. A

Complete Prediction algorithm that can achieve a near optimal solution is presented below.

5.9.1 A Simulated Annealing Algorithm for the PRTR FPGA

The purpose of annealing for the PRTR FPGA is to find the optimal mapping for each configuration such that the reconfiguration overhead is minimized. For each step, a randomly selected RFUOP is assigned to a random position on-chip, and the exact reconfiguration overhead is then computed. Before presenting the full Simulated Annealing algorithm, we first define "conflict" as used in our discussion.

Definition 2: Given two configurations and their positions on the FPGA, RFUOP A is said to be in conflict with RFUOP B if any part of A overlaps with any part of B.

We now present our Full Simulated Annealing algorithm for the PRTR FPGA.

1. Assign a random position to each RFUOP. Set up the parameters of initial temperature, number of iterations under each temperature, and terminating temperature.
2. While the current temperature is greater than the terminating temperature:
   - 2.1. While the number of iterations is greater than 0:
     - 2.1.1. A randomly selected RFUOP is moved to a random location on-chip.
     - 2.1.2. Traverse the RFUOP sequence. If the demanded RFUOP is not currently on the chip, load the RFUOP to the specified location, and increase the overall reconfiguration latency by the loading latency of the RFUOP. If the newly loaded RFUOP conflicts with any other RFUOPs on the chip, those conflicted RFUOPs are removed from the chip.
     - 2.1.3. Let the new cost be equal to the overall RFUOP overhead and determine whether the move is allowed. Decrease the number of iterations by one.

2.2. Decrease the current temperature.

Finding the location for each RFUOP is similar to the placement problem in physical design, where the simulated annealing algorithm usually provides good performance. Therefore, our Full Simulated Annealing algorithm should create a near optimal solution.

5.9.2 An Alternate Annealing Algorithm for the PRTR FPGA

In the Full Simulated Annealing algorithm presented in the last section, the computation complexity is very high, since the RFUOP sequence must be traversed to compute the overall reconfiguration overhead after every move. Obviously, a better algorithm is needed to reduce the running time. Again, as for the Single Context FPGA, an adjacency matrix of size $N \times N$ is built, where N is the number of RFUOPs. The main purpose of the matrix is to record the possible conflicts between RFUOPs. In order to reduce the reconfiguration overhead, the conflicts that create larger configuration loading latency are distributed to non-overlapped locations. This is done by modifying the cost computation step of the previous algorithm. To clarify, we present the full algorithm:

1. Create an $N \times N$ matrix, where N is the number of RFUOPs. All values of $A[i, j]$ are set to be 0, where $0 \leq i, j \leq N\text{-}1$.
2. Traverse the RFUOP sequence. For any RFUOP j that appears between two consecutive appearances of an RFUOP i, $A[i, j]$ is increased by 1. Notice that multiple appearances of an RFUOP j only count once between two consecutive appearances of an RFUOP.
3. Assign a random position for each RFUOP. Set up parameters of initial temperature, the number of iterations under each temperature, and terminating

temperature. An $N \times N$ adjacency matrix B is created. All values of $B[i, j]$ are set to be 0, where $0 \leq i, j \leq N\text{-}1$.

4. While the current temperature is greater than the terminating temperature:
   4.1. While the number of iterations is greater than 0:
      4.1.1. A randomly selected RFUOP is reallocated to a random location on-chip. After the move, if two RFUOPs i and j conflict, set $B[i, j]$ and $B[j, i]$ to be 1.
      4.1.2. For any $B[i, j]=1$, multiply the value of $A[i, j]$ by the RFUOP loading latency of j. The new cost is computed as the summation of the results of all the products.
      4.1.3. Determine whether the new move is allowed and decrease the number of iterations by one.
   4.2. Decrease the current temperature.

Generally, the total number of RFUOPs is much less than the length of the RFUOP sequence. Therefore, by looking up the conflict matrices instead of the whole configuration sequence, the time complexity can be greatly decreased. Still, one final concern is the quality of the algorithm because the matrix of potential conflicts derived from the sequence is used rather than using the complete configuration sequence. Even the matrix may not represent the conflicts exactly; however, it gives an estimate of the potential conflicts between any two configurations.

5.10 Algorithms for the PRTR R+D FPGAs

For the PRTR R+D FPGA, the replacement policies have a great impact on reducing the reconfiguration overhead. This is due to the high flexibility available for choosing victim RFUOPs when a reconfiguration is required. With relocation, an RFUOP can be dynamically remapped and loaded to an arbitrary position. With defragmentation, a

demanded RFUOP can be loaded as long as there is enough room on the chip, since the small fragments on-chip can be merged. In the next sub-sections we present the algorithms of Relocation + Defragmentation, which include a Lower-bound algorithm that relaxes restrictions in the system, a General Off-line algorithm integrating the Belady algorithm, and two Run-time algorithms using different approaches.

5.10.1 A Lower-bound Algorithm for the PRTR R+D FPGAs

The major problems that prevent us from acquiring an optimal solution to configuration caching are the different sizes and loading latencies of RFUOPs. Generally, the loading latency of an RFUOP is proportional to the size of the configuration.

The Belady algorithm gives the optimal replacement when the memory access sequence is known and the data transfer unit is uniform. Given the RFUOP sequence for the PRTR R+D model, we can achieve a lower-bound for our problem if we assume that a portion of any RFUOP can be transferred. Under this assumption, when a reconfiguration occurs, only a portion of an RFUOP might be replaced while the other portion is still retained on-chip. Once the removed RFUOP is needed again, only the missing portion (possibly the whole RFUOP) is loaded instead of loading the entire RFUOP. We present the Lower-bound algorithm as follows:

1. If a required RFUOP is not on the chip, do the following:
   1.1. Find the missing portion of the RFUOP. While the missing portion is greater than the free space on the chip:
      1.1.1. For all RFUOPs that are currently on the chip, identify a victim RFUOP whose next appearance is later than the appearances of all others.
      1.1.2. Let R = the size of the victim + the size of the free space – the missing portion.

1.1.3. If R is greater than 0, a portion of the victim that equals R is retained on-chip while the other portion is replaced and added to the free space. Otherwise, add the space occupied by the victim to the free space.

1.2. Load the missing portion of the demanded RFUOP into the free space. Increase the reconfiguration overhead by the loading latency of the missing portion.

5.10.2 A General Off-line Algorithm for the PRTR R+D FPGAs

Since the Belady algorithm can provide a lower-bound for the fixed size problem, some ideas can be transferred into a more realistic off-line algorithm. As in the Belady algorithm, for all RFUOPs that are currently on-chip, we identify the one that will not appear in the RFUOP sequence until all others have appeared. But instead of replacing that RFUOP, as in the Belady algorithm, the victim configuration is selected by considering the factors of size and loading latency. Before we discuss the algorithms further, we first define a "reappearance window" as used in our algorithms.

Definition 5.3: A reappearance window W is the shortest subsequence of the reconfiguration stream, starting at the current reconfiguration, that contains an occurrence of all currently loaded configurations. If a configuration does not occur again, the reappearance window is the entire remaining reconfiguration stream.

We now present our General Off-line algorithm for the PRTR R+D FPGA:

1. If a demanded RFUOP is not currently on the chip:

    1.1. While there is not enough room to load the RFUOP, do the following:

        1.1.1. Find the reappearance window W.

        1.1.2. For each RFUOP, calculate the total number of appearances in W

        1.1.3. For each RFUOP, multiply the loading latency and the number of appearances; replace the RFUOP with the smallest value.

1.2. Load the demanded RFUOP. Increase the overall latency by the loading latency of the RFUOP.

Steps 1.1.1 – 1.1.3 specify the rules to select the victim RFUOP. Counting the number of appearances of each RFUOP obtains the frequency of the use of the RFUOP to in the near future. As mentioned, this is not adequate to determine a victim RFUOP, because an RFUOP with lower frequency may have much higher configuration latency. Therefore, by multiplying the latency and the frequency, we can find the possible overall latency in the near future if the RFUOP is replaced.

5.10.3 LRU Algorithm for the R+D FPGAs

Since the PRTR R+D can be viewed as a general memory model, we can use an LRU algorithm for our reconfiguration problem. Here, we traverse the RFUOP sequence, and when a demanded RFUOP is not on-chip and there is insufficient room to load the RFUOP, the least recently used on-chip RFUOP is selected to be removed. Although simple to implement, this algorithm may display poor quality because it ignores the sizes of the RFUOPs.

5.10.4 Penalty-oriented Algorithm for the PRTR R+D FPGAs

Since the non-uniform size of RFUOPs is not considered a factor in LRU algorithm, a high reconfiguration overhead could potentially result. For example, consider an RFUOP sequence *1 2 3 1 2 3 1 2 3 …*; RFUOPs *1, 2* and *3* have sizes of 1000, 10 and 10 programming bits respectively. Suppose also that the size of the chip is 1010 programming bits. According to the LRU algorithm, the RFUOPs are replaced in same order of the RFUOP sequence. It is obvious that configuration overhead will be much smaller if RFUOP 1 is always kept on the chip.

This does not suggest that we always want to keep larger RFUOPs on the chip as keeping larger configurations with low reload frequency may not reduce the

reconfiguration overhead. Instead, both size and frequency factors should be considered in the algorithm. Therefore, we use a variable "credit" to determine the victim [Young94]. Every time an RFUOP is loaded onto the chip, its credit is set to its size. When a replacement occurs, the RFUOP with the smallest credit is evicted from the chip, and the credit of all other RFUOPs on-chip is decreased by the credit of the victim. To make this clearer, we present the algorithm as follows:

1. If a demanded RFUOP is currently on the chip, set its credit equal to its size. Else do the following:
    - 1.1. While there is not enough room to load the required RFUOP:
        - 1.1.1. For all RFUOPs on-chip, replace the one with the smallest credit and decrease the credit of all other RFUOPs by that value.
    - 1.2. Load the demanded RFUOP and set its credit equal to its size.

5.11 A General Off-line Algorithm for the Relocation FPGAs

One major advantage that the PRTR R+D FPGA has over the PRTR with Relocation is the ability to have higher utilization of the space on the reconfigurable hardware. Any small fragments can contribute to one larger area, so that an RFUOP could possibly be loaded without forcing a replacement. However, for PRTR with only Relocation, those fragments could be wasted. This could cause an RFUOP that is currently on chip to be replaced and thus may result in extra overhead if the replaced RFUOP is demanded again very soon. In order to reduce the reconfiguration overhead for this model, the utilization of the fragments must be improved. We present the algorithm as follows:

1. If a demanded RFUOP is not currently on the chip, do the following.
    - 1.1. While there is not enough room to load the RFUOP, do the following:
        - 1.1.1. Find the reappearance window W.
        - 1.1.2. For each RFUOP, calculate the total number of appearances in W.

1.1.3. For each RFUOP, multiply the loading latency by the number of appearances, producing a cost.

1.1.4. For each RFUOP on-chip, assume that it is to be the candidate victim, and identify the adjacent configurations that must also be removed to make room for the demanded RFUOP. Total the costs of all potential victims.

1.1.5. Identify the smallest sum of each RFUOP. The victim that produces the smallest costs is replaced.

1.2. Load the demanded RFUOP. Increase the overall latency by the loading latency of the configuration.

The General Off-line heuristic that applied to the R+D FPGA is also implemented in this algorithm. The major difference for this algorithm is to consider the geometric positions of the RFUOPs. Since the R+D FPGA model has the ability to collect the fragments, the RFUOPs are replaced in the increasing order of their costs (load latency times appearance in the reappearance window). However, this scheme does not work for the PRTR with Relocation if the victim RFUOPs are separated by other non-victim RFUOPs because the system cannot merge the non-adjacent spaces. Therefore, when multiple RFUOPs are to be replaced in the PRTR FPGA with Relocation, these RFUOPs must be adjacent or separated only by empty fragments. Considering this geometric factor, the victims to be replaced are adjacent RFUOPs (or those separated by fragments) that produce the smallest overall cost.

5.12 Simulation Results and Discussion

All algorithms are implemented in C++ on a Sun Sparc-20 workstation. Figure 5.3 demonstrates the bounds of Single Context, PRTR, and Multi-Context models. For each benchmark, we first normalize the reconfiguration penalty for each algorithm, then we calculate the average for each algorithm.

As can be seen in Figure 5.3, the reconfiguration penalties of the PRTR are much smaller (64% to 85% smaller) than for the Single Context model. This is because with almost the same capacity, the PRTR model can significantly reduce the average reconfiguration latency of the Single Context model without incurring a much larger number of reconfigurations. The Multi-Context model has smaller reconfiguration overhead (20% to 40% smaller) than the PRTR when the chip silicon is small. With a small silicon area, the Multi-Context model wins because of its much larger configuration area. When the silicon area becomes larger, the number of conflicts incurred in the PRTR model is greatly reduced, and thus the PRTR has almost the same reconfiguration penalty as the Multi-Context model. In fact, the PRTR performs even better than the Multi-Context model in some cases. The Multi-Context device must reload a complete context each time, making the per reconfiguration penalty in large chips much higher than in the PRTR model. Since the number of conflicts is small, the overall reconfiguration overhead of the PRTR FPGA is smaller than that of the Multi-Context FPGA.

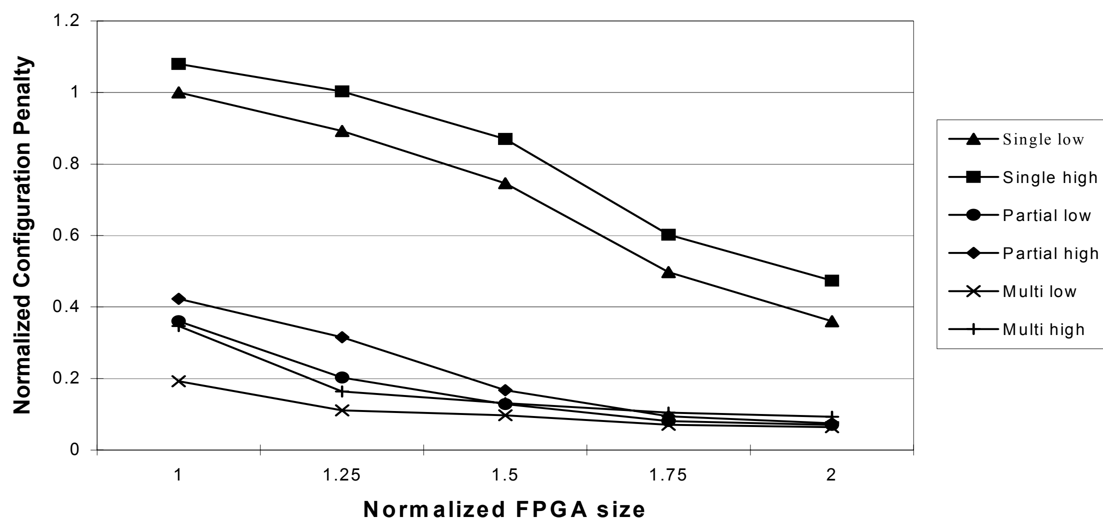


Figure 5.3. Reconfiguration overheads of the Single Context FPGA, the PRTR, and the Multi-Context models. The “low” represents the lower-bound or near optimal solution for each model, and the “high” represents the upper-bound.

Figure 5.4 shows the reconfiguration overheads of the Relocation model and the R+D model. For the R+D FPGA, the General Off-line algorithm performs almost as well as the Lower-bound algorithm in the reconfiguration overhead reduction, especially when the chip silicon becomes larger. Note that the Lower-bound algorithm relaxes the PRTR model restrictions by allowing portion of the RFUOPs can be replaced and loaded. As can be seen in Figure 5.4, future information is very important, as the General Off-line algorithm for the PRTR with Relocation performs better than both the LRU and the Penalty-oriented algorithms for the R+D FPGA. By only considering the frequency factor while ignoring load latency, the LRU algorithm has worse performance than the penalty oriented algorithm.

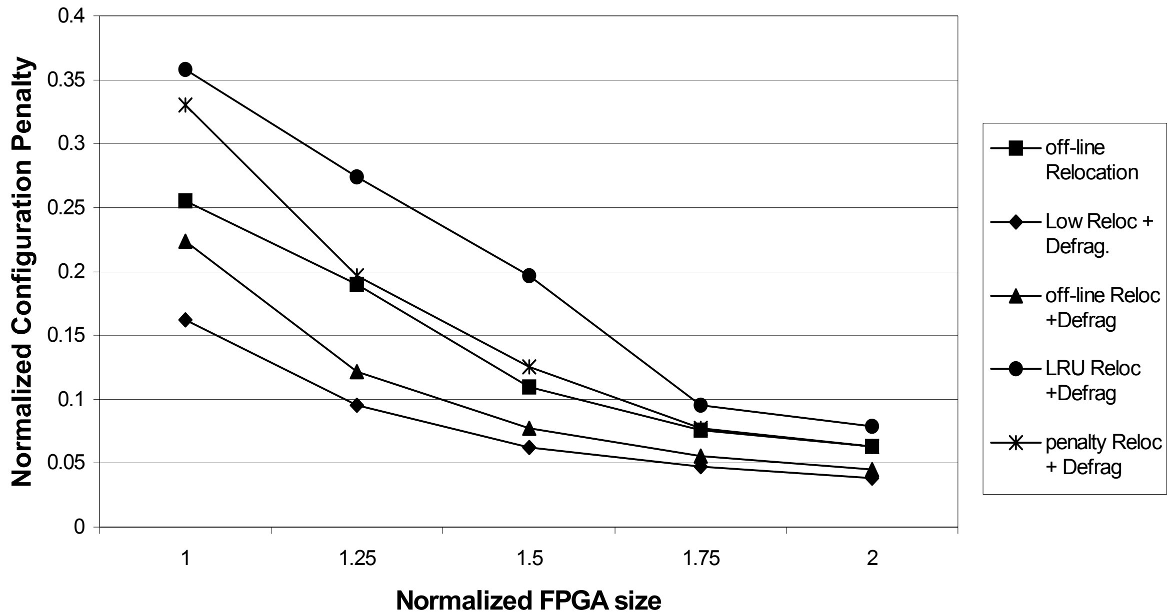


Figure 5.4: Reconfiguration overheads for the Relocation and the PRTR R+D FPGA. The "Low Reloc + Defrag" represents the lower-bound algorithm for the R+D FPGA.

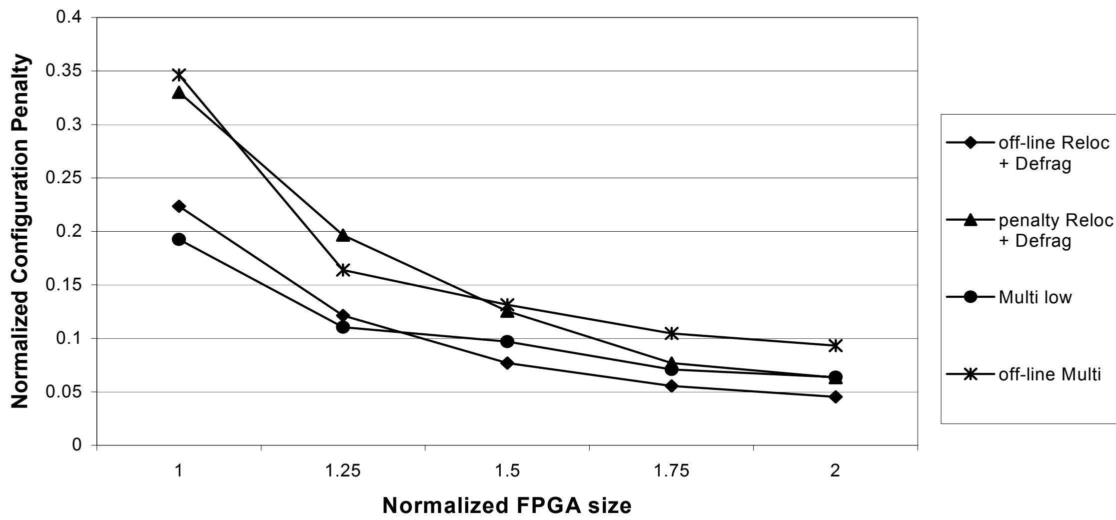


Figure 5.5: Comparison between the PRTR with Relocation + Defragmentation model and the Multi-Context model.

Figure 5.5 compares the PRTR R+D and the Multi-Context models. As we can see, when the chip silicon is small, the Complete Prediction algorithm for the Multi-Context FPGA performs better than the General Off-line algorithm for the R+D FPGA. However, as the chip silicon increases, the General Off-line algorithm for the R+D FPGA has almost the same ability to reduce the reconfiguration overhead as the Complete Prediction algorithm for the Multi-Context FPGA. In addition, the Penalty-oriented algorithm (run-time algorithm) for the R+D FPGA performs slightly better than the General Off-line algorithm for the Multi-Context FPGA.

5.13 Summary

In this chapter, we have presented the first cache management algorithms for reconfigurable computing systems. We have developed new caching algorithms targeted at a number of different FPGA models, as well as creating lower-bounds to quantify the maximum achievable reconfiguration reductions possible. For each model, we have implemented a set of algorithms to reduce the reconfiguration overhead. The simulation results demonstrate that the reconfiguration overhead of the PRTR is 85%

smaller than for the Single Context model. The Multi-Context model has smaller reconfiguration overhead (20% to 40% smaller) than the PRTR when chip silicon is small. When the chip area becomes larger, the PRTR has almost the same reconfiguration overhead as the Multi-Context model. Since the PRTR R+D provide higher hardware utilization, the reconfiguration overhead of the PRTR R+D model is about factor of 2-3 smaller than the PRTR model. The reconfiguration overhead of the PRTR R+D is slightly smaller than the Multi-Context model.

Chapter 6

Configuration Prefetching

As demonstrated in Chapter 5, an FPGA can be viewed as a cache of configurations. Prefetching configurations on an FPGA, which is similar to prefetching in a general memory system, overlaps the reconfigurations with computation to hide the reconfiguration latency. In this chapter, we present configuration prefetching techniques for various reconfigurable models.

6.1 Prefetching Overview

Prefetching for standard processor caches has been extensively studied. Research is normally split into data and instruction prefetching. In data prefetching, the organization of a data structure and the measured or predicted access pattern are exploited to determine which portions of a data structure are likely to be accessed next. The simplest case is array accesses with a fixed stride, where the access to memory location N is followed by accesses to (N+stride), (N+2*stride), etc. Techniques can be used to determine the stride and issue prefetches for locations one or more strides away [Mowry92, Santhanam97, Zucker98, Callahan91]. For more irregular, pointer-based structures, techniques have been developed to prefetch locations likely to be accessed in the near future, either by using the previous reference pattern, by prefetching all children of a node, or by regularizing data structures [Luk96].

Techniques have also been developed for instruction prefetching. The simplest is to prefetch the cache line directly after the line currently being executed [Smith78, Hsu98], since this is the next line needed unless a jump or branch intervenes. To handle such branches information can be kept on all previous successors to this block [Kim93], or

the most recent successor ("target prefetch") [Hsu98], and prefetch these lines. Alternatively, a look-ahead PC can use branch prediction to race ahead of the standard program counter, prefetching along the likely execution path [Chen94, Chen97]. However, information must be maintained to determine when the actual execution has diverged from the lookahead PC's path, and then restart the lookahead along the correct path.

Unfortunately, many standard prefetching techniques are not appropriate for FPGA configurations because of differences between configurations and single instruction or data block. Before we discuss the details of configuration prefetching, we first reexamine the factors important to the effectiveness of prefetching for a general-purpose system:

1) *Accuracy*. This is the ratio of the executed prefetched instructions or data to the overall prefetched instructions or data. Prefetching accuracy estimates the quality of a prefetching technique as a fraction of all prefetches that are useful. Based on the profile or run-time information, the system must be able to make accurate predictions on the instructions or data that will be used and fetch them in advance.

2) *Coverage*. This is the fraction of cache misses eliminated by the effectiveness of a prefetching technique. An accurate prefetch technique will not significantly reduce the latency without a high coverage.

3) *Pollution*. One side-effect that prefetching techniques produce is that the cache lines that would have been used in the future will be replaced by some prefetched instructions or data that may not be used. This is known as cache pollution.

These issues are even more critical to the performance of configuration prefetching. In general-purpose systems the atomic data transfer unit is a cache block. Cache studies consistently show that the average access time will likely drop when the block size increases until it reaches a certain value (usually fewer than 128 bytes), the access time

will then increases as the block size continue to increase since a very large block will result in an enormous penalty for every cache miss. The atomic data transfer unit in the configuration caching or configuration prefetching domain, rather than a block, is the configuration itself, which normally is significantly larger than a block. Therefore the system suffers severely if a demanded configuration is not present on chip. In order for a system to minimize this huge latency accurate prediction of the next required configurations is highly desired.

As bad as it could be in general-memory systems, cache pollution plays a much more malicious role in the configuration prefetching domain. As demonstrated in Chapter 5, due to the large configuration size and relatively small on-chip memory, very few configurations can be stored on reconfigurable hardware. As the result, an incorrect prefetch will be very likely to cause a required configuration to be replaced, and significant overhead will be generated when the required configuration is brought back later. Thus, rather than reducing the overall configuration overhead, a poor prefetching approach can actually significantly increase overhead.

In addition, the correct prefetch of a configuration needs to be performed much earlier than it will be required; otherwise, the large configuration loading latency cannot be entirely or mostly hidden. A correct prediction must be made as early as possible to make prefetching an effective approach. Also, in contrast to the fixed block size for general memory system the sizes of different configurations could vary drastically. As shown in Chapter 5, the variable configuration sizes make it more difficult to determine which configurations should be unloaded to make room for the required configuration.

In general, prefetching algorithms can be divided into three categories: *static prefetching*, *dynamic prefetching* and *hybrid prefetching*. A compiler-controlled approach, static prefetching inserts prefetch instructions after performing control flow or data flow analysis based on profile information and data access patterns. One major advantage of static prefetching is that it requires very little additional hardware.

However, since a significant amount of access information is unknown at compile time, the static approach is limited by the lack of run-time data access information. Dynamic prefetching determines and dispatches prefetches at run-time without compiler intervention. With the help of the extra hardware, dynamic prefetching uses more data access information to make accurate predictions. Hybrid prefetching tries to combine the strong points of both approaches—it utilizes both compile-time and run-time information to become a more accurate and efficient approach.

6.2 Factors Affecting Configuration Prefetching

In order to better discuss the factors that will affect prefetching performance, we first make the following definitions.

- L_k*: The latency of loading a configuration k.*
- S_k*: The size of the configuration k.*
- D_{ik}*: The distance (in instructions executed) between an operation i and the execution of the configuration k.*
- P_{ik}*: The probability that configuration k is the next executed RFUOP after instruction i.*
- $Cost_{ik}$*: The potential minimum cost if k is prefetched at instruction i.*
- *C: The capacity of the chip.*

The probability factor could have a significant impact on prefetching accuracy. The combination of S_k and C determines whether there is enough space on the chip to load the configuration k. The combination of L_k and D_{ik} determines the amount of the latency the configuration k that can be hidden if it is prefetched at i. It is obvious that there will

not be a significant reduction if D_{ik} is too short, since most of the latency of configuration k cannot be eliminated if it is prefetched at i.

In addition, the non-uniform configuration latency will affect the order of the prefetches that need to be performed. Specifically, we might want to perform out-of-order prefetch (prefetch configuration j before configuration k even if k is required earlier than j) for some situations. For example, suppose we have three configurations *1, 2*, and *3* to be executed in that order. Given that $S_3 >> S_1 >> S_2$, $S_1 + S_3 < C < S_1 + S_2 + S_3$, and $D_{12} >> L_3 >> L_2 > D_{23}$, prefetching configuration *3* before configuration *2* when *1* is executed results in a penalty of L_2 at most. This is because the latency of configuration *3* can be completely hidden and configuration *2* can be either demanded fetched (penalty of L_2) or prefetched once the execution of the configuration *1* completes. However, if the in-order prefetches are performed the overall penalty is calculated as $L_3 - D_{23}$, which is much larger than L_2.

6.3 Configuration Prefetching Techniques

Two reconfigurable models are considered in this work. The Single Context reconfigurable model is chosen mainly because of its architecture simplicity. Additionally, since only one configuration can be retained on-chip at any time, only in-order prefetching needs to be considered. This will simplify the techniques we use for configuration prefetching.

Partial R+D FPGA is the other reconfigurable model we selected because of its high hardware utilization demonstrated in Chapter 5. Note that Partial R+D and Multi-Context models show similar hardware utilization. We chose Partial R+D model over Multi-Context model based on the following reasons. First, various current commercial FPGAs are partial reconfigurable and the Partial R+D model can be built from them without significantly increasing hardware [Compton00]. Second, as mentioned in

Chapter 2, power consumption during context switches for Multi-Context remains a major concern.

6.4 Configuration Prefetching for the Single Context FPGAs

The initial prefetching algorithm for the Single Context model [Hauck98] was developed by professor Scott Hauck, advisor for this dissertation. A static prefetching technique, the algorithm applies a shortest path approach to identify the next candidate to prefetch. Since the probability factor is not considered in the algorithm, malicious prefetches can significantly lower the effectiveness of the prefetching. In order to avoid this, a post-processing pruning approach is repetitively executed to reduce malicious prefetches, causing a longer running time. Furthermore, though this approach can reduce malicious prefetches, it cannot add potentially helpful prefetches to take advantage of the distances left by the removal of malicious prefetches. In this work, we seek to improve the effectiveness of the static prefetching algorithm as well as to remove the pruning process.

Given the control flow graph of an application, our goal is to explore a technique that can automatically insert prefetch instructions at compile-time. These prefetch instructions are executed just like any other instructions, occupying a single slot in the processor's pipeline. The prefetch instruction specifies the ID of a specific configuration that should be loaded into the coprocessor. If the desired configuration is already loaded, or is in the process of being loaded by some other prefetch instruction, this prefetch instruction becomes a NO-OP. If the specified configuration is not present, the coprocessor trashes the current configuration and begins loading the specified one. At this point the host processor is free to perform other computations, overlapping the reconfiguration of the coprocessor with other useful work. Once the next call to the coprocessor occurs it can take advantage of the loading performed by the prefetch instruction.

6.4.1 Experiment Setup

In order to evaluate the different configuration prefetching algorithms we must perform the following steps. First, some method must be developed to choose which portions of the software algorithms should be mapped to the reconfigurable coprocessor. In this work, we apply the approach presented in [Hauck98] (these mappings of the portions of the source code will be referred to as RFUOPs). Second, a simulator of the reconfigurable system must be employed to measure the performance of the prefetching algorithms. Our simulator is developed from SHADE [Cmelik93]. It allows us to track the cycle-by-cycle operation of the system, and get exact cycle counts. We will compare the performance of the prefetching algorithms as well as the performance assuming no prefetch occurs at all. Also, we measure the impact of the techniques on reconfiguration time, which can be reasonably measured in this system, as opposed to overall speedup, which cannot accurately be measured due to uncertainties in future system architectures and exact program features.

To make our analysis clearer in the control flow graph, we use circles to represent the instruction nodes and squares to represent the RFUOPs. Since in a single entry and single exit path, the prefetch should be executed at the top, but not other nodes contained in the path, only the top node is considered as the candidate where prefetch instructions can be inserted. Therefore, we simplify the control graph by packing other nodes in the path, with a length that represents the number of nodes contained in the path.

Since all configurations for a Single Context FPGA have equal size, the only factors that determine prefetches are distance, probability and configuration latency. From our experiments and analysis, we found that prefetching the closest configuration could work well on applications whose reconfiguration latency is relatively small. In these cases, the prefetches are not required to be determined very far in advance to hide the entire latency, and the prefetch of the closest configuration will result in the correct

decision most of the time. Also, the prefetch of the closest configuration will lead to more direct benefit than the prefetch of other configurations, whose latency can be overlapped by later prefetches.

However, for systems with very large reconfiguration latencies, prefetching the closest configuration will not lead to the desired solution. This is because another factor, the probabilities of reaching a different configuration from the current instruction, can have a more significant effect. With large reconfiguration delays, the insertion of prefetches to load one configuration means that the reconfiguration latency of other configurations cannot be entirely hidden. Consider the example in Figure 6.1, where the "Length" and the "Probability" next to an arrow represents the number of instruction cycles and the probability to execute that path, respectively. If the closest configuration is prefetched, a prefetch instruction for configuration *1* will be inserted at instruction *I* and a prefetch instruction for configuration *2* will be inserted at instruction *L*. All reconfiguration latency will be eliminated in a system where it takes 10 cycles to load each configuration because the path from instruction *L* to configuration *2* is long enough to hide the reconfiguration latency of configuration *2*. However, if the reconfiguration latency of each configuration is 100 cycles, then the majority of the latency for configuration *2* still remains. Furthermore, although 90% of the time configuration *2* will be executed once instruction *I* is reached, the long path from instruction *I* to instruction *J* cannot be utilized to hide the reconfiguration cost 90% of the time. However, if configuration *2* were prefetched at instruction *I*, the rate of correct prediction of the next configuration improves from 10% to 90%. Thus the path from instruction *I* to instruction *J* can provide more benefit in reducing the overall reconfiguration cost.

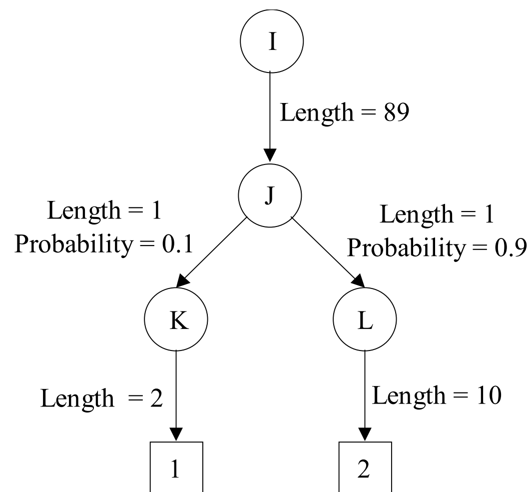


Figure 6.1: An example for illustrating the ineffectiveness of the directed shortest-path algorithm

As can be seen in Figure 6.1, the reconfiguration latency is also a factor that affects the prediction of the next required configuration. With the reconfiguration latency changes from 10 cycles to 100 cycles, the determination of the next required configuration at instruction *I* has changed. To correctly predict the next required configuration all three factors must be considered. Failing to do so will lower the efficiency of configuration prefetching.

6.4.2 Cost Function

For paths where only one configuration can be reached the determination of the next required configuration is trivial, since simply inserting a prefetch of the configuration at the top of the path is the obvious solution. The problem becomes more complex for paths where multiple configurations can be reached. We call these shared paths. Inserting a prefetch of any single configuration at the top of a shared path will keep this path from being used to hide the reconfiguration latency of any other configurations.

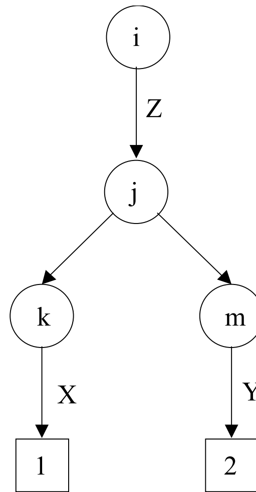


Figure 6.2: The control flow graph for illustrating the cost calculation.

We first start our cost calculation on the basic case that is shown in Figure 6.2. There are two ways to insert prefetches in this example. One is to prefetch configuration *1* at instruction *i* and prefetch configuration *2* at instruction *m*. The other is to prefetch configuration *2* at instruction *i* and prefetch configuration *1* at instruction *k.* The decision is made depending on the calculation of the overall reconfiguration cost resulting from each prefetching sequence, which is affected by the factors of the probability, the distance and the reconfiguration latency. The reconfiguration cost of each prefetching sequence is calculated as follows:

$$Cost_{i1} = P_{i1} \times (L - D_{i1}) + P_{i2} \times (L - D_{m2}) = P_{i1} \times (L - Z - X - 2) + P_{i2} \times (L - Y)$$

$$Cost_{i2} = P_{i2} \times (L - D_{i2}) + P_{i1} \times (L - D_{k1}) = P_{i2} \times (L - Z - Y - 2) + P_{i1} \times (L - X)$$

Since the cost of each prefetch cannot be negative, we modify the functions to be:

$$Cost_{i1} = P_{i1} \times max\ (0, (L - Z - X - 2)) + P_{12} \times max\ (0, (L - Y)) \quad (6.4.1)$$

$$Cost_{i2} = P_{i2} \times max\ (0, (L - Z - Y - 2)) + P_{11} \times max\ (0, (L - X)) \quad (6.4.2)$$

$Cost_{i1}$ and $Cost_{i2}$ are the potential minimum costs at instruction i with different prefetches performed. As can be seen from *(6.4.1)* and *(6.4.2),* the costs of upper nodes can be calculated by using the costs of lower nodes. Therefore, for more complex control graphs we can apply a bottom-up scheme to calculate the potential minimum costs at each node, with upper level nodes using the results calculated at the lower-level nodes.

6.4.3 The Bottom-up Algorithm for Prefetching

The Bottom-up algorithm works on the control flow graph when the edges from RFUOPs to their successors have been removed. We also eliminate loops, as discussed in Section 6.4.4. The algorithm starts from the configuration nodes, calculating the potential minimum costs at each instruction node once the costs of children nodes are available. This scheme continues until all nodes are processed. Once finished, the top-most nodes will contain a series of costs reflecting the different prefetch sequences. The sequence with the minimum cost represents the prefetch sequence that has the best potential to hide reconfiguration overheads. Before presenting the algorithm, we first discuss the information that must be calculated and retained at each instruction node.

From *(6.4.1)* and *(6.4.2),* we can see that the length of the paths are important to the generation of the best prefetch sequences. If the shared path in Figure 6.2 is not long enough to hide the remaining latency of both configurations, then by subtracting *(6.4.2)* from *(6.4.1)* we will have:

$$C_{i1} - C_{i2} = P_{i1} \times (L - Z - X - 2) + P_{i2} \times (L - Y) - (P_{i2} \times (L - Z - Y - 2) + P_{i1} \times (L - X))$$

$$= P_{i2} \times (Z + 2) - P_{i1} \times (Z + 2) \quad (6.4.3)$$

As can be seen in *(6.4.3),* for this case the configuration with the largest probability should be prefetched. However, this may not be true when the shared path is long enough to hide the remaining reconfiguration latency of at least one configuration. Suppose in Figure 6.2 that X were much longer than Y, and by prefetching configuration

1 at instruction *i* the entire reconfiguration latency of configuration *1* could be eliminated. Then by subtracting *(6.4.2)* from *(6.4.1),* we will have:

$$C_{i1} - C_{i2} = P_{i2} \times (Z + 2) - P_{i1} \times (L - X) \quad (6.4.4)$$

As can be seen in *(6.4.4),* probability is not the only factor in deciding the prefetches, since the length of each path for an instruction to reach a configuration also affects the way to insert prefetches. The difference in forms between equation *(6.4.3)* and equation *(6.4.4)* raises an important issue: Given two nodes in a shared path (such as *i* and *j*), the best configuration to prefetch at the close node may differ from the best RFUOP to prefetch at the more distant node. This is because the close node may be affected only by the difference in branch probabilities (as in equation *(6.4.3)*), while the far node will also be affected by the path lengths (as in equation *(6.4.4)*). The interested reader can verify this in Figure 6.2 by assuming that X, Y, Z, L, P_{11}, P_{12} equal 90, 10, 80, 100, 0.6, and 0.4, respectively. In this scenario, the best configuration to prefetch at instruction *j* is configuration *1*, while the configuration to prefetch at instruction *i* is 2. To deal with this discrepancy, we lean towards the decision made at the more distant node, since this provides the greatest opportunity to hide latency.

During the Bottom-up algorithm, the cost to prefetch each reachable configuration is calculated at each instruction node. The costs will be used to determine the prefetch at the current node and the cost calculation at the parent nodes. The basic steps of the Bottom-up scheme are outlined below:

For each instruction node *i*, set C_{ij}, P_{ij}, and D_{ij} to 0 for all *i* and set num_children_searched to 0.

1. For each configuration node *i*, set C_{ii}, P_{ii} and D_{ii} to 1. Place configuration nodes into a queue.
2. While the queue is not empty, do:

2.1. Remove a node k from the queue. If it is not a configuration node, do:

2.1.1. CALCULATE_COST(k).

2.2. For each parent node, if it is not a configuration node do:

2.2.1. Increase num_children_searched by 1, if num_children_searched equal the number of children of that node, insert the parent node into the queue.

Before we present the details of subroutine CALCULATE_COST(k), we must define some terms:

- B_{ij}: *the branch probability for instruction i to reach instruction j.*
- *Min_cost(i): Min* C_{ij} *for all configurations j reachable from i.*

Now we outline the basic steps of the subroutine CALCULATE_COST(k):

1. For each configuration j that can be reach by k, do:

1.1. $Temp_length = 0$.

1.2. For each child node i that can reach the configuration j, do:

1.2.1. $Temp_probability = B_{ki} \times P_{ij.}$

1.2.2. $P_{kj} = P_{kj} + Temp_probability.$

1.2.3. $Temp_length = Temp_length + min(Latency, D_{ij} + D_{ki}) \times Temp_probability.$

1.2.4. $Temp_cost \mathrel{+}= C_{ij} - Temp_probility \times max(0, Latency - D_{ij} - D_{ki}).$

1.3. $D_{kj} = Temp_length / P_{kj.}$

1.4. $C_{kj} = C_{kj} + Temp_cost.$

1.5. For each child node i that cannot reach the configuration j, do:

1.5.1. $C_{kj} = C_{kj} + Min_cost(i).$

The function call CALCULATE_COST(k) at instruction k is to calculate probability, distance, and cost of k to prefetch each reachable configuration j. The probability is calculated based on the probabilities of its children to reach j and the branch probability of k to reach each child node. Since the control flow graph can be very complex, there may exist several different paths for an instruction to reach a configuration. Therefore, we calculate the weighted average length of these paths. This could cause some inaccuracy in the cost calculation, but in most of the cases the inaccuracy is tolerable because the path with the high probability dominates the weighted average length. The cost calculation considers both the probability and length for k to reach configuration j, as well as the costs computed at the children nodes.

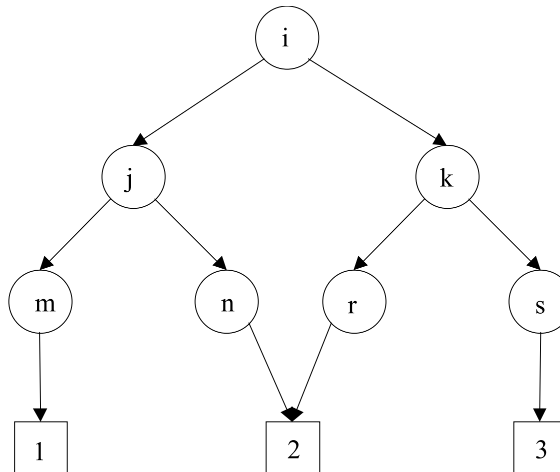


Figure 6.3: An example of multiple children nodes reaching the same configuration.

For example assume we are computing C_{i3} in Figure 6.3. C_{i3} includes the costs of configuration *3* at each of its children, as well as the latency hiding achieved (if any) during the execution of edges *(i, j)* and *(i, k)*. Since k is the only node to reach *3*, we will be unable to hide any latency on *(i, j)*. Also, we will incur the cost *min_cost(j)* at node j, which represents performing the best prefetch possible at node j. In cases where multiple children can reach the same configuration, the cost of the parent includes the reduced cost from the edges to each of those children. Consider the example in Figure

6.3, where both node *j* and node *k* can reach configuration *2*. C_{i2} includes the costs of configuration *2* at *j* and *k*, as well as the latency hiding achieved during edges *(i, j)* and *(i, k)*.

6.4.4 Loop Detection and Conversion

Our Bottom-up algorithm operates on acyclic graphs, and thus requires loop detection and conversion. If loop detection were not performed, two problems would result. First, our simple bottom-up processing would deadlock, since a node in a loop is its own descendant. Second, the insertion of prefetches into loops that do not contain RFUOPs can cause excessive overhead, as the same prefetch operation is called multiple times, wasting an execute slot each time it is encountered. For example, in Figure 6.4 left, our basic prefetching algorithm might decide that the best configuration to prefetch at *j* is *1*, and at *m* is *2*. This could insert two prefetch instructions into the loop, wasting two execute slots per iteration. After converting the loop as shown in Figure 6.4 right, the prefetch algorithm would insert a prefetch of *1* at *i*, and a prefetch of *2* at *n*, reducing the overhead of potentially redundant prefetches.

To solve these problems, we use a *strongly connected components* algorithm to identify the loops existing in the control flow and then convert the loops into dummy nodes in the graph. The strongly connected components algorithm is a standard method to identify a set of nodes in the graph so that every node in the set can reach every other node. In our model, a nested loop can be viewed as a set of strongly connected components, in which every node in the loop can reach every other. Furthermore, to perform the Bottom-up algorithm, appropriate values for *P*, *C*, and *D* must be calculated for each dummy node for each reachable configuration. The basic steps of loop detection and conversion are as following:

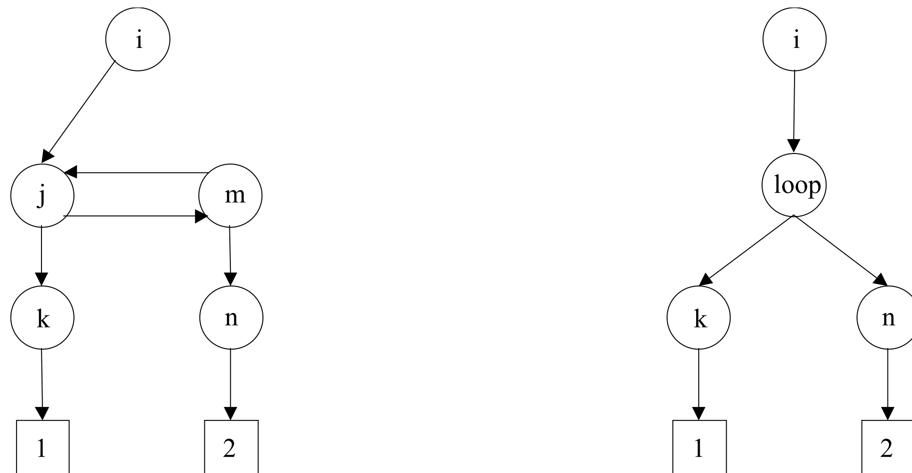


Figure 6.4. Loop conversion. The original control flow graph with a loop containing nodes j and m is shown on the left, while at right they are replaced by a dummy loop node.

1. Run the strongly connected component algorithm.
2. For each component, do:
   2.1. Compute the total number of executions of all nodes in the component.
   2.2. Calculate the total number of executions of paths exiting the loop.
   2.3. Divide the value calculated from 2.1 by the value of 2.2, producing the average length of the loop.
   2.4. For each path exiting the loop, do:
      2.4.1. The branch probability of the path is calculated as the execution of the path divided by the total number of executions of paths exiting the loop.

The execution information used in the algorithm can be gathered from the profile information provided by our simulator. Step 2.2 calculates the total control flow that exits from the loop. By dividing this number by the total number of executions of all loop nodes, the average length of the loop nodes can be calculated.

6.4.5 Prefetch Insertion

Once the bottom-up process is complete, we have discovered a potential way to perform prefetch. The prefetch instructions must be inserted to execute discovered prefetches. Inserting prefetch instructions represents additional overhead of the systems, therefore, no redundant prefetch instruction should be allowed.

The prefetch insertion is performed in a top-down style starting from the nodes that have no instruction parents. The prefetch of each of these nodes is determined by the minimum cost calculated. Once the prefetch instruction is inserted, each node passes the prefetch information to its children. Upon receiving the prefetch information, each child checks whether it can reach the configuration that is prefetched by its parents. If so, no new prefetch is inserted and the prefetch information is passed down. For each of the nodes that cannot reach the configuration prefetched by the parents, the prefetch to the configuration that was calculated with the minimum cost is inserted and this information is passed to its children. Note that the prefetch instruction will not be inserted if the RFUOP to prefetch is also the ancestor of the instruction node. This top-down method continues until all the instruction nodes are traversed.

6.4.6 Results and Analysis

The simulation results are shown in Table 6.1. Each benchmark is tested at four different per-reconfiguration delay values. Note that different reconfiguration delays results in different RFUOP being picked by the simulator. The "No Prefetch", "Optimal Prefetch" and "Prefetch" columns report the total number of cycles spent stalling the processor while the coprocessor is reconfigured, plus the number of cycles spent on prefetch opcodes. The "Opt/No", "Pre/NO" and "Pre/Opt" columns list the ratios of Optimal Prefetching delays to no prefetching delays, of delays of our prefetching algorithm to no prefetching and of delays of our prefetching algorithm to optimal prefetching. The cumulative rows compare the total reconfiguration penalties across the entire benchmark suite.

As can be seen in Table 6.1, the prefetching algorithm provides an overall 41%-65% reduction in reconfiguration overhead when compared to the base case of no prefetching. While this is not nearly as good as the 69%-84% improvement suggested by the optimal prefetch technique, it is important to realize that the optimal prefetch numbers may not be achievable by any static configuration prefetch algorithm. For example, if there exists a very long shared path for an instruction to reach multiple configurations, then static prefetch approaches will not gain as much as optimal prefetch since the prefetch instruction inserted on the top of the shared path will only reduce the overhead of one configuration. If the probability to reach one configuration is significantly greater than the combined probability to reach the rest of the configurations, the static approaches may not suffer too much compared to optimal prefetch. However, if the probabilities for an instruction to reach different configurations are almost identical, the performance of static prefetch approaches will be much worse than that of the optimal prefetch, which can dynamically issue different prefetch targets at the top of the shared paths.

Consider the benchmark "Perl" with a latency of 10,000, where only two prefetches are needed for the configurations by using Optimal Prefetch, which issues one at the beginning of the execution and another one after the first RFUOP is finished. However, since the two configurations have a long shared path, no static approach can perform as well as optimal Prefetch. Furthermore, as can be seen in Table 6.1, the shared path is more critical when the Latency is high. When the latency is small, the shared path may not be necessary to hide reconfiguration latency for most configurations. With the increase of the latency, it is more likely that the shared path is necessary to hide the overhead.

Table 6.1: Results of the prefetching algorithm.

Benchmark	Latency	No Prefetching	Optimal Prefetch	(Opt/No)	Prefetch	(Pre/No)	(Pre/Opt)
Go	10	6,239,090	2,072,560	33.2%	2,453,389	39.32%	118.37%
	100	6,860,700	1,031,739	15.0%	2,834,309	41.31%	274.71%
	1,000	2,520,000	225,588	9.0%	889,736	35.31%	394.41%
	10,000	1,030,000	314,329	30.5%	533,987	51.84%	169.88%
Compress	10	344,840	63,403	18.4%	83,876	24.32%	132.29%
	100	127,100	46,972	37.0%	80,998	63.73%	172.44%
	1,000	358,000	289,216	80.8%	253,981	70.94%	87.82%
	10,000	520,000	12,535	2.4%	252,673	48.59%	2015.74%
Li	10	6,455,840	958,890	14.9%	1,877,365	29.08%	195.79%
	100	4,998,800	66,463	1.3%	2,131,882	42.65%	3207.62%
	1,000	55,000	21,325	38.8%	41,102	74.73%	192.74%
	10,000	330,000	43,092	13.1%	110,392	33.45%	256.18%
Perl	10	4,369,880	656,210	15.0%	1,598,984	36.59%	243.67%
	100	3,937,600	398,493	10.1%	1,800,337	45.72%	451.79%
	1,000	3,419,000	9,801	0.3%	1,899,644	55.56%	19382.14%
	10,000	20,000	2	0.0%	5,714	28.57%	285700.00%
Fpppp	10	2,626,180	1,415,924	53.9%	1,667,678	63.50%	117.78%
	100	11,707,000	6,927,877	59.2%	6,982,352	59.64%	100.79%
	1,000	19,875,000	5,674,064	28.5%	6,269,614	31.55%	110.50%
	10,000	370,000	4,485	1.2%	349,870	94.56%	7800.89%
Swim	10	600,700	265,648	44.2%	311,092	51.79%	117.11%
	100	10,200	4,852	47.6%	6,092	59.73%	125.56%
	1,000	91,000	79,905	87.8%	83,822	92.11%	104.90%
	10,000	330,000	43,019	13.0%	55,267	16.75%	128.47%
Cumulative	10	20,636,530	5,432,635	26.3%	7,075,757	34.29%	130.25%
	100	27,641,400	8,476,396	30.7%	13,835,970	50.06%	163.23%
	1,000	26,318,000	6,299,899	23.9%	9,347,899	35.52%	148.38%
	10,000	2,600,000	417,462	16.1%	1,307,903	50.30%	313.30%

6.5 Configuration Prefetching for Partial R+D FPGA

In this section, we present efficient prefetching techniques for reconfigurable systems containing a Partial R+D FPGA. We have developed algorithms that apply the different configuration prefetching techniques. Based on the available access information and the additional hardware required, our configuration prefetching algorithms can be divided

into 3 categories: *Static Configuration Prefetching, Dynamic Configuration Prefetching*, and *Hybrid Configuration Prefetching*.

6.5.1. Static Configuration Prefetching

Unlike the Single Context FPGA a Partial R+D FPGA can hold more than one configuration, and at a given point multiple RFUOPs may need to be prefetched. Thus, the method to effectively specify the IDs of the RFUOPs to prefetch becomes an issue. One intuitive approach is to pack the IDs into one single instruction. However, since the number of IDs need to be specified could be different for each prefetch instruction, it is not possible to generate prefetch instructions with equal length. Another option is to use a sequence of prefetch instructions when multiple prefetching operations need to be performed. However, to make it an effective approach a method that can terminate previously issued prefetches is required. This is because during the execution certain previous unfinished prefetch instructions may become obsolete and useless cancelled these unwanted prefetching operations will significantly damage performance.

In Figure 6.5, for example, *1, 2, 3,* and *4* are RFUOPs and *P1, P2, P3*, and *P4* are prefetching instructions. It is obvious that when *P1* is executed, configurations *3* and *4* will not be reached, and the prefetches of *3* and *4* are wasted. This waste may be negligible for a general-purpose system since the load latency of an instruction or a data block is very small. However, because of the large configuration latency in reconfigurable systems, it is likely that the prefetch of configuration *3* has not completed or even not started when *P1* is reached. As a consequence, if we use the same approach used in general-purpose systems, letting the prefetches of *P3* and *P4* complete before prefetching *P1* and *P2*, the effectiveness of the prefetches of *P1* and *P2* will be severely damaged since they cannot completely or mostly hide the load latencies of configurations *1* and *2*. Therefore, we must find a way to terminate previously issued prefetches if they become unwanted.

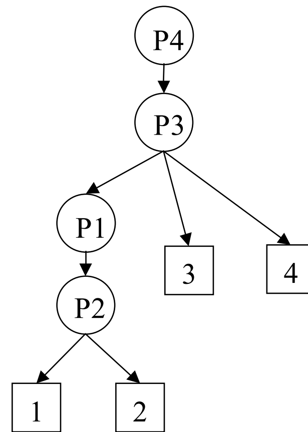


Figure 6.5: Example of prefetching operation control.

A simple approach used in this work to solve this problem is to insert termination instructions when necessary. The format of a termination instruction resembles any other instruction, consuming a single slot in the processor's pipeline. Once a termination instruction is encountered the processor will terminate all previously issued prefetches so the new prefetches can start immediately. For the example in Figure 6.5, a termination instruction will be inserted immediately before *P1* to eliminate the unwanted prefetches of *P4* and *P3*.

Now that we have demonstrated how to handle the prefetches, the remaining problem is to determine where the prefetch instructions will be placed given the RFUOPs and the control flow graph of the application. Since the algorithm used in the previous section demonstrated high-quality results for Single Context reconfigurable systems, we will extend it for systems containing a Partial R+D.

The algorithm we used to determine the prefetches contains three stages:

1) *Penalty calculation*. In this stage the algorithm computes the potential penalties for a set of prefetches at each instruction node of the control flow graph.

2) *Prefetch scheduling and generation*. In this stage the algorithm determines the configurations that need to be prefetched at each instruction node based on the penalties calculated in Stage 1. Prefetches are generated under the restriction of the size of the chip.

3) *Prefetch reduction*. In this stage the algorithm trims the redundant prefetches generated in the previous stage. In addition, it inserts termination instructions.

In the previous section, a bottom-up approach was applied to calculate the penalties using the probability and distance factors. Since the probability is dominant in deciding the penalties, and the average prefetching distance is mostly greater than the latencies of RFUOPs, it is adequate to use probability to represent penalty.

Given the simplified control flow graph and the branch probabilities, we will use a bottom-up approach to calculate the potential probabilities for an instruction node to reach a set of RFUOPs. The basic steps of the Bottom-up algorithm are outlined below:

1. For each instruction, initialize the probability of each reachable configuration to 0, and set the num_children_searched to be 0.
2. Set the probability of the configuration nodes to 1. Place the configuration nodes into a queue.
3. While the queue is not empty, do
    - 3.1. Remove a node k from the queue. If it is not a configuration node, do:
        - 3.1.1. $P_{kj} = \sum(P_{ki} \times P_{ij})$, for all children i of node k.
    - 3.2. For each parent node, if it is not a configuration node, do
        - 3.2.1. Increase num_children_searched by 1. If num_children_searched equals the total number of children of that node, insert the parent node into the queue.

The left side of Figure 6.6 shows a control flow graph with branch probability on each edge. Table 6.2 shows the probabilities calculated at the instruction nodes with more than one reachable RFUOP.

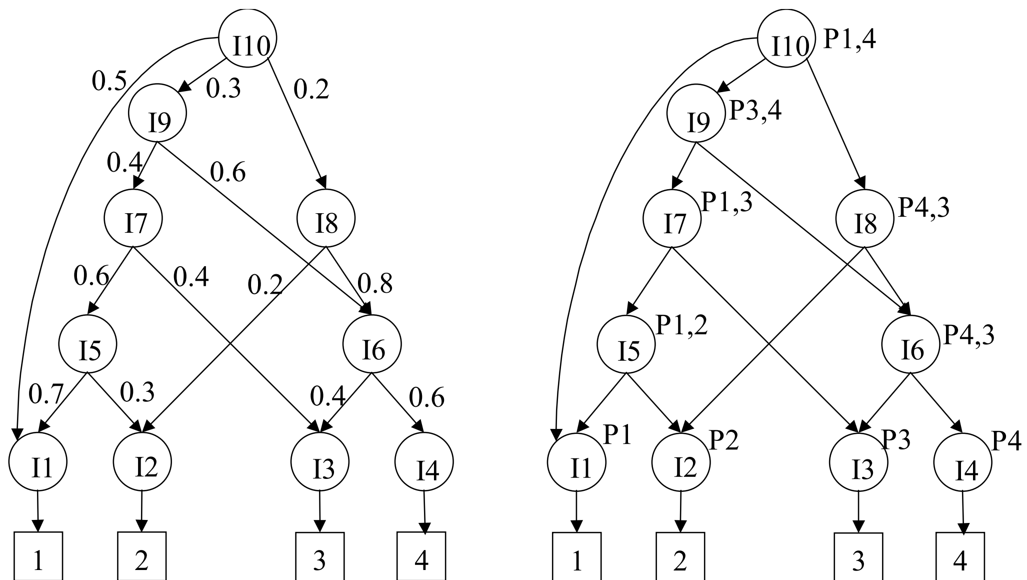


Figure 6.6: An example of prefetch scheduling and generation after the probability calculation.

Table 6.2: Probability calculation for Figure 6.6.

	1	2	3	4
I5	0.7	0.3	0	0
I6	0	0	0.4	0.6
I7	0.42	0.18	0.4	0
I8	0	0.2	0.32	0.48
I9	0.168	0.072	0.4	0.36
I10	0.5504	0.0616	0.184	0.204

Once this stage is complete, each instruction contains the probabilities of the reachable RFUOPs. We use the results to schedule necessary prefetches.

The prefetch scheduling is quite trivial once the probabilities have been calculated. Based on the decreasing order the probabilities, a sequence of prefetches could be generated for each instruction node. Since the aggregate size of the reachable RFUOPs for a certain instruction could exceed the capacity of the chip, the algorithm only generate prefetches under the size restriction of the chip. The rest of the reachable RFUOPs are ignored. For example, in Figure 6.6 we assume that the chip can at most hold 2 RFUOPs at a time. The generated prefetches at each instruction are shown on the right side of Figure 6.6.

The prefetches generated at a child node are considered to be redundant if they match the beginning sub-sequence generated at its parents. Our algorithm will check and eliminate these redundant prefetches. One may argue that the prefetches at a child node should be considered as redundant if they are a sub-sequence, but not necessary the beginning sub-sequence, of its parents, because the parents represent a superset of prefetches. However, by not issuing the prefetch instructions at the child node, the desired the prefetches cannot start immediately, since the unwanted prefetches at the parents might have not completed. For example, in Figure 6.6, *P2* at the instruction *I2* cannot be eliminated even though it is a sub-sequence of *P1,2* because when *I2* is reached *P2* may not be able to start if *P1* has not completed. In Figure 6.6, the prefetches at instructions *I1*, *I4*, and *I6* can be eliminated.

Once the prefetch reduction is complete, for each instruction node where prefetches must be performed a termination instruction is inserted, followed by a sequence of prefetch instructions. Note that a termination instruction does not flush the entire FPGA, but merely clears the prefetch queue. RFUOPs that have already been loaded by the preceding prefetches are often retained.

6.5.2 Dynamic Configuration Prefetching

Among the various dynamic prefetching techniques available for general-purpose computing systems, Markov prefetching [Joseph97] is a very unique approach. Markov prefetching does not tie itself to particular data structure accesses and is capable of prefetching both instructions and data.

As the name implies, Markov prefetching uses a Markov model to determine which blocks should be brought in from the higher-level memory. A Markov process is a stochastic system for which the occurrence of a future state depends on the immediately preceding state, and only on it. A Markov process can be represented as a directed graph, with probabilities associated with each vertex. Each node represents a specific state, and a state transition is described by traversing an edge from the current node to a new node. The graph is built and updated dynamically using the available access information. As an example, the access string *A B C D C C C A B D E* results in the Markov model shown in Figure 6.7. Using the Markov graph multiple prefetches with different priorities can be issued.

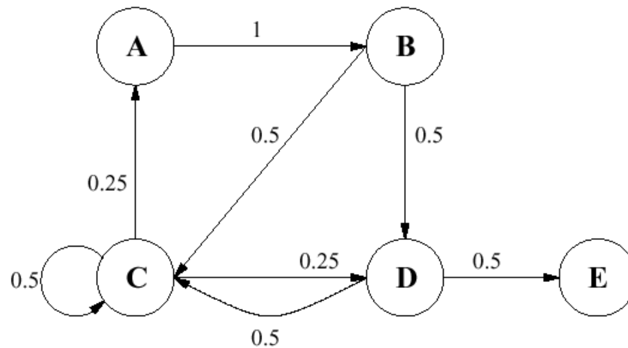


Figure 6.7: The Markov model generated from access string A B C D C C C A B D E. Each node represents a specific state and each edge represents a transition from one state to another. The number on an edge represents the probability of the transition occurring.

Markov prefetching can be extended to handle the configuration prefetching for reconfigurable systems. Specifically, the RFUOPs can be represented as the vertices in the Markov graph and the transitions can be built and updated using the RFUOP access sequence. However, the Markov prefetching needs to be modified because of the differences between general-purpose systems and reconfigurable systems.

Note that only a few RFUOPs can be retained on-chip at any given time because of the large size of each RFUOP. Therefore, it is very unlikely that all the transitions from the current RFUOP node can be executed. This feature requires the system to make accurate predictions to guarantee that only the highly probable RFUOPs are prefetched. In order to find good candidates to prefetch, Markov prefetching keeps updating the probability of each transition using the currently available access information. This probability represents the overall likelihood this transition could happen during the course of the execution and may work well for a general-purpose system which a large number of instructions or data blocks with small loading latency can be stored in a cache. However, without emphasizing the recent history, Markov prefetching could make poor predictions during a given period of the execution. Due to the features of the reconfigurable systems mentioned above, we believe the probability calculated based on the recent history is more important than the overall probability.

In this work, a weighed probability in which recent accesses are given higher weight in probability calculation is used as a metric for candidate selection and prefetching order determination. The weighted probability of each transition is continually updated as the RFUOP access sequence progresses. Specifically, probabilities out of a node u will be updated once a transition (u, v) is executed:

For each transition starting from u:

$$P_{u,w} = P_{u,w} \; / \; (1 + C) \text{ if } w \neq v;$$

$$P_{u,v} = (P_{u,v} + C) \; / \; (1 + C);$$

where C is a weight coefficient.

For general-purpose systems, the prefetching unit generally operates separately from the cache management unit. Specifically, the prefetching unit picks candidates and then sends prefetch requests into the prefetching buffer (usually a FIFO). Working separately, the cache management unit will vacate the requests one by one from the buffer and load the instructions or data blocks into the cache. If there is not enough space in the cache, the cache management unit will apply a certain replacement policy to select victims and evict them to make room for the loading instructions or data.

Though this works well for general-purpose systems, this separated approach may not be efficient for configuration prefetching because of the large size and latency of each RFUOP. For example, at a certain point 3 equal-size RFUOPs A, B, C are stored on chip, and equal-size RFUOPs D, A, B are required in sequence with very a short distance between each other. Suppose there is no room on chip and a FIFO replacement policy is used. The system will evict A first to make room for D, then replace B with A and C with B in order. It is obvious that the overall latency will not be significantly reduced because of the short distance between D, A, B. This situation can be improved by combining prefetching with caching techniques. For the example above, we can simply replace C with D and retain A and B on chip to eliminate the latencies of loading them. The intuition behind this approach is to use the prefetching unit as predictor as well as a cache manager. The dynamic prefetching algorithm can be described as following:

Upon the completion of each execution of RFUOP *k*, do:

1. Sort the weighted probabilities in decreasing order of all transitions starting from *k* in the Markov graph.
2. Terminate all previously issued prefetches. Select *k* as the first candidate.

3. Select the rest of the candidates in sorted order under the size constraint of the chip. Issue prefetch requests for each candidate that is not currently on chip.
4. Update the weighted probability of each transition starting from *j*, where *j* is the RFUOP executed just before *k*.

Though the replacement is not presented in the algorithm, it is carried out indirectly. Specifically, any RFUOPs that are currently on chip will be marked for eviction if they are not selected as candidates. One last point to mention is that the RFUOP just executed is treated as the top candidate automatically since generally each RFUOP is contained in a loop and likely to be repeated multiple times. Correspondingly, the self-loop transitions in the Markov graph are ignored.

6.5.3 Hardware Requirements of Dynamic Prefetching

A data structure is required to maintain and continually update the Markov graph as execution progresses. A table, as shown in Figure 6.8, can be used to represent the Markov graph. Each node (RFUOP) of the Markov graph occupies a single row of the table. The first column of each row is the ID of an RFUOP, and the rest of the columns are the RFUOPs it can reach. Since under chip size constraint only the high probability RFUOPs out of each node are used for the prefetching algorithm, keeping all transitions out of a node will simply waste precious hardware resources. As can be seen in Figure 6.8, the number of transitions retained is limited to K.

RFUOP 1	Next 1	Next 2	…	Next K
RFUOP 2	Next 1	Next 2	…	Next K
…	…	…	…	…
RFUOP M	Next 1	Next 2	…	Next K

Figure 6.8: A table is used to represent the Markov graph. The first column of each row is the ID of a RFUOP and the rest of the columns are k-reachable RFUOPs from this row's RFUOP with highest probability.

In addition, a small FIFO buffer is required to store the prefetch requests. The configuration management unit will take the requests from the buffer and load the corresponding RFUOPs. Note that the buffer will be flushed to terminate previous prefetches before the new prefetches are sent to the buffer. Furthermore, the configuration management unit can be interrupted to stop the current loading if an RFUOP not currently loaded is invoked.

In order for the host processor to save execution time in updating the probabilities, the weight coefficient C is set to 1. This means that when a transition needs to be updated, the host processor will simply right shift the register retaining the probability by one bit. Then the most significant bit of the register representing the currently occurring transition is set to 1. An N-bit register representing any transition (u, v) will become 0 if (u, v) does not occur for N executions of u. In that sense, only the most recent N accesses of u are used to compute the probability for each transition starting from u. As a consequence, the number of transitions that needs to be stored in the table representing the Markov graph can be set to N (i.e. K = N). To balance the hardware cost and retain enough history, we use 8-bit registers in this work.

6.5.4 Hybrid Configuration Prefetching

Dynamic prefetching using recent history works well for the transitions occurring within a loop. However, this approach will not be able to make accurate predictions for transitions jumping out a loop. For example, on the left side of Figure 6.9, we assume only one RFUOP can be store on chip at any given point. By applying dynamic prefetching approach, RFUOP 2 is always prefetched after RFUOP 1 assuming the inner loop will always be taken for several iterations. Thus, the reconfiguration penalty for RFUOP 3 can never be hidden due to the wrong prediction.

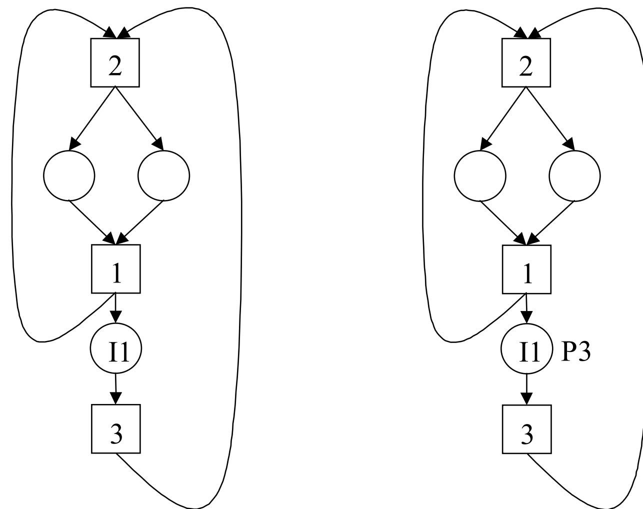


Figure 6.9: An example illustrates the ineffectiveness of the dynamic prefetching.

This misprediction can be avoided if static prefetching approach can be integrated with the dynamic approach. More specifically, before reaching RFUOP3 a normal instruction node will likely be encountered and the static prefetches determined at that instruction node can be used to correct the wrong predictions determined by the dynamic prefetching. As illustrated on the right side of the Figure 6.9, a normal instruction I1 will be encountered before RFUOP 3 is reached and our static prefetching will correctly predict 3 will be the next required RFUOP. As the consequence, the wrong prefetch of RFUOP 2 determined by our dynamic prefetching can be corrected at I1.

The goal of combining the dynamic configuration prefetching with the static configuration prefetching is to take advantage of the recent access history without exaggerating it. Specifically, dynamic prefetching using the recent history will make accurate predictions within the loops while static prefetching using the global history will make accurate predictions between the loops. The challenge of integrating dynamic prefetching with static prefetching is to coordinate the prefetches such that the wrong

prefetches are minimized. When the prefetches determined by the dynamic prefetching do not agree those determined by the static prefetching a decision must be made.

The basic idea we use to determine the beneficial prefetches for our hybrid prefetching is to penalize the wrong prefetches. We add a per-RFUOP flag bit to indicate the correctness of the prefetch made by previous static prefetching. When the prefetches determined by static prefetching conflict with those determined by dynamic prefetching, the statically predicted prefetch of a RFUOP is issued only if the flag bit for that RFUOP was set to 1. The flag bit of a RFUOP is set to 0 once the static prefetch of the RFUOP is issued, and will remain 0 until the RFUOP is executed. As the consequence, statically predicted prefetches, especially those made within the loops, are ignored if they have not been correctly predicting. On the other hand, those correctly predicted static prefetches, especially those made between the loops, are chosen to replace the wrong prefetches made by the dynamic prefetching. The basic steps of the hybrid prefetching are outlined as following:

1. Perform the Static Configuration Prefetching algorithm. Set the flag bit of each RFUOP to 1. An empty priority queue is created.
2. Upon the finish of a RFUOP execution, perform the Dynamic Prefetching algorithm. Set the flag bit of the RFUOP to 1. Clear the priority queue first, then place the IDs of the dynamically predicted RFUOPs into the queue.
3. When a static prefetch of a RFUOP is encountered and the flag bit of the RFUOP is 1, terminate current loading. Set the flag bit of the RFUOP to 0. Give the highest priority to this RFUOP and insert its ID into the priority queue. The RFUOPs with lower priorities are replaced or ignored to make room for the new RFUOP.
4. Load the RFUOPs from the priority queue.

6.5.5 Results and Analysis

All algorithms are implemented in C++ on a Sun Sparc-20 workstation and are simulated with the SHADE simulator [Cmelik93]. We choose to use the SPEC95 benchmark suite to test the performance of our prefetching algorithms. Note that these applications have not been optimized for reconfigurable systems, and may not be as accurate in predicting exact performance as would real applications for reconfigurable systems. However, such real applications are not in general available for experimentation. In addition, the performance of the prefetching techniques will be compared against the previous caching techniques, which also used the SPEC95 benchmark suite.

As can be seen in Figure 6.10, five algorithms are compared: Least Recently Used (LRU) Caching, Off-line Caching, Static Prefetching, Dynamic Prefetching, and Hybrid Prefetching. The LRU algorithm chooses victims to be replaced based on run-time information, while the Off-line algorithm takes into consideration future access patterns to make more accurate decisions. Note that our static prefetching uses the Off-line algorithm to pick victims. Since the cache replacement is integrated into dynamic prefetching and hybrid prefetching, no additional replacement algorithms are used for both prefetching algorithms.

Clearly, all prefetching techniques substantially outperformed caching-only techniques, especially when cache size is small. As cache size grows, the chip is able to hold more RFUOPs and cache misses are reduced. However, the prefetching distance is not changed. As the consequence, the performance due to prefetching will not significantly improve as the cache size grows. Among the prefetching techniques, dynamic prefetching performs consistently better than static prefetching because it can use the RFUOP access information. Hybrid prefetching performs slightly better than dynamic prefetching, because of its ability to correct some wrong prediction made by dynamic

prefetching. However, the advantage of hybrid prefetching becomes negligible as the cache size grows.

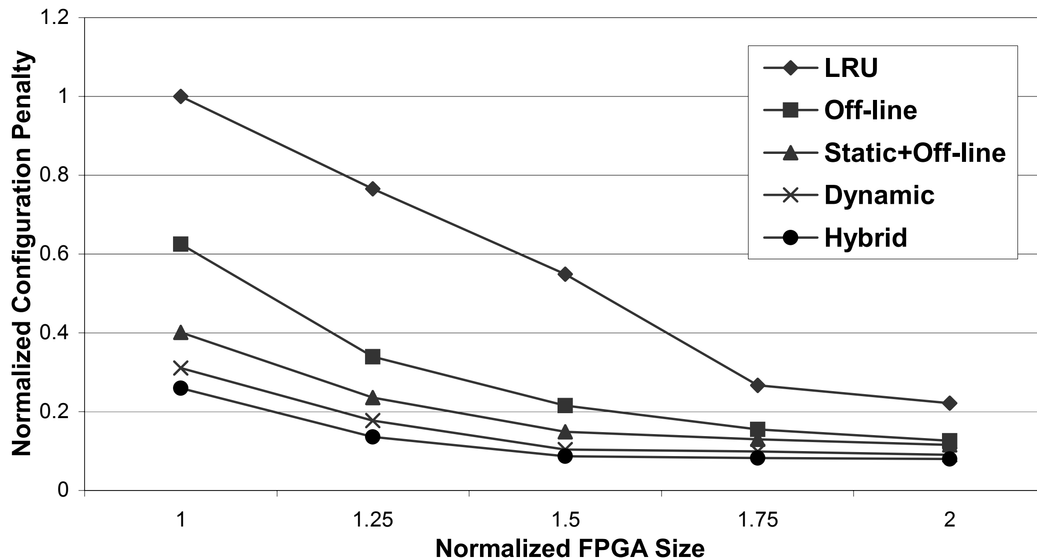


Figure 6.10: Reconfiguration overhead comparison. Five algorithms are compared: Least Recently Used (LRU) Caching, Off-line Caching, Static Prefetching, Dynamic Prefetching, and Hybrid Prefetching. The configuration penalty of each algorithm is normalized to the penalty of the LRU algorithm with a normalized FPGA size 1.

Figure 6.11 demonstrates the effect of the different replacement algorithms that used for the static prefetching. Since the static prefetching algorithm requires an augment replacement algorithm, the overall reconfiguration overhead reduction can be affected not only by the prefetching, but also by the replacement approach that is chosen. However, as can be seen in Figure 6.11, the prefetching is the more dominant factor in overall overhead reduction, as the off-line replacement algorithm (the optimum realistic replacement) with more complete information on performs just slightly better than the LRU.

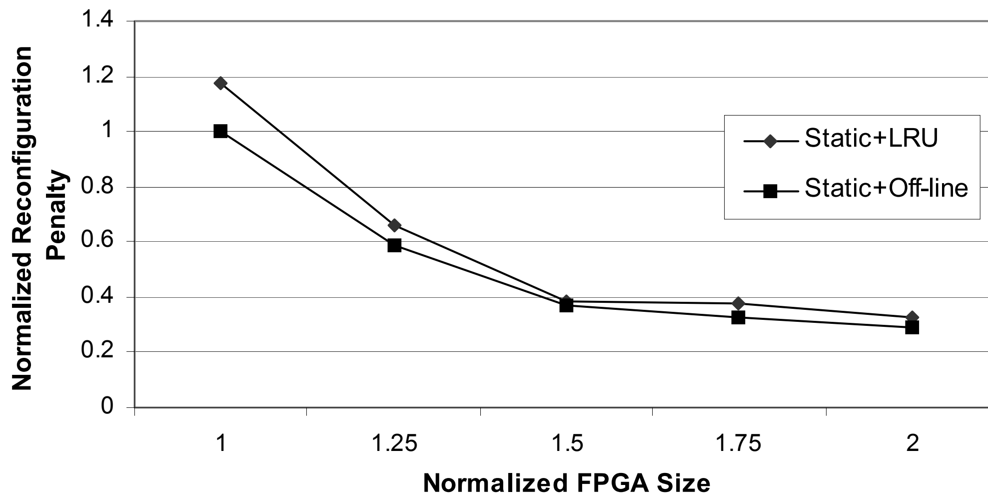


Figure 6.11: Effect of the replacement algorithms for the static prefetching.

6.6 Summary

In this chapter we have introduced the concept of configuration prefetching for reconfigurable systems. Our prefetching techniques target two reconfigurable models, Single Context and PRTR R+D. For the Single Context model, we have developed a static prefetching algorithm that can automatically determine the placement of these prefetch operations, avoiding burdening the user with the potentially difficult task of placing these operations by hand. Without using additional hardware, this approach can reduce the reconfiguration latency by more than a factor of two.

We have also developed efficient prefetching techniques for reconfigurable systems containing a PRTR R+D. We have developed algorithms applying the different configuration prefetching techniques. Based on the available access information and the additional hardware required, our configuration caching algorithms can be divided into three categories: static configuration prefetching, dynamic configuration prefetching, and hybrid configuration prefetching. Compare to the caching techniques presented in Chapter 5, our prefetching algorithms can further reduce reconfiguration overhead by a factor of 3.

Chapter 7

Conclusions

Reconfigurable computing is becoming an important part of research in computer architectures and software systems. By placing the computationally intense portions of an application onto the reconfigurable hardware, that application can be greatly accelerated. Gains are realized because reconfigurable computing combines the benefits of both software and ASIC implementations. As suggested by many applications, reconfigurable computing systems greatly improve performance over general-purpose computing systems.

Run-time reconfiguration provides additional opportunities for computation specialization that is not available within static configurable systems. In such systems, hardware configuration may change frequently at run-time to reuse silicon resources for several different parts of a computation. Such systems have demonstrated high hardware efficiency by specializing circuits at run-time.

However, the advantages of reconfigurable computing do not come without a cost. By requiring multiple reconfigurations to complete a computation, the time it takes to reconfigure the FPGA becomes a significant overhead. This overhead not only has a major negative impact on performance of reconfigurable systems, it can also limit the applications that can be executed on such systems.

In this work, we have developed a complete configuration management system to attack this problem.

7.1 Summary of Contributions

The main contributions presented in this thesis include:

- An investigation of the strategy for reducing the reconfiguration overhead for reconfigurable computing.

- An investigation of the most efficient reconfiguration model for reconfigurable computing. We have quantified the reconfiguration overhead for various reconfiguration models.

- An exploration of configuration compression techniques to reduce the size of the configuration bit-streams. Configuration compression takes advantage of regularities and repetitions within the original configuration data. However, using the existing lossless compression approaches cannot significantly reduce the size of configuration bit-streams because of several fundamental differences between data and configuration compression. The unique regularities and on-chip run-time decompression require distinct compression algorithms for different architectures.

- In this work, we have investigated configuration compression techniques for the Xilinx 6200 FPGAs and the Xilinx Virtex FPGAs. Taking advantage of the on-chip Wildcard Registers, our Wildcard algorithm can achieve a factor of 3.8 compression ratio for the Xilinx 6200 FPGAs without adding extra hardware. A number of compression algorithms are investigated for Virtex FPGA. These algorithms can significantly reduce the amount of data that needs to be transferred with minimum modification to hardware. In order to explore the best compression algorithm we have extensively researched existing compression techniques, including Huffman coding, Arithmetic coding and LZ coding. Simulation results demonstrate that a compression factor of 4 can be achieved.

- An exploration of the Don't Care discovery approach to further improve the compression ratios. Realizing that Don't Cares within the bit-streams increase regularities, our Don't Care discovery technique backtraces important locations, starting from the outputs, generating a new configuration. By combining this technique with our lossless techniques, compression ratio is improved from a factor of 4 to a factor of 7.

- An examination of configuration caching techniques to increase the likelihood of the required configuration presented on chip. We have developed new caching algorithms targeted at a number of different FPGA models, as well as creating lower-bounds to quantify the maximum achievable reconfiguration reductions possible. For each model, we have implemented a set of algorithms to reduce the reconfiguration overhead. The simulation results proved that the Partial Run-Time Reconfigurable FPGA and the Multi-Context FPGA are significantly better caching models than the traditional Single Context FPGA.

- An investigation of various configuration prefetching techniques that overlap the transfer of configuration bit-streams with useful computation. Our prefetching techniques target two reconfigurable models, the Single Context model and the PRTR R+D model. For Single Context model, we have developed a static prefetching algorithm that can automatically determine the placement prefetch operations, avoiding burdening the user with the potentially difficult task of placing these operations by hand. Without using additional hardware, this approach can reduce reconfiguration latency by more than a factor of 2. We have also developed efficient prefetching techniques for reconfigurable systems containing a PRTR R+D because of its high hardware utilization. We have developed algorithms applying the different configuration prefetching techniques that can significantly reduce reconfiguration overhead by a factor of 3.

Each of these techniques attacks different perspective of the reconfiguration bottleneck. Therefore, the gain from one technique will not be overlapped or hidden by others. Using all techniques together provides a complete system that virtually eliminates reconfiguration overhead. For a system that applies these techniques, compression reduces the size of configuration bit-streams by a factor of 2 to 7 at compile time. During execution, configuration caching reduces off-chip traffic by a factor of 2.5 to 10. Prefetching techniques can further improve caching by at least a factor of 2. Combining these techniques together represents a factor of 10 to 150 overhead reduction.

7.2 Future Work

All told, the results presented in this thesis indicate considerable promise for integrated techniques that improve the performance of reconfigurable computing systems by virtually eliminating reconfiguration overheads. Nevertheless, these techniques must be refined for future reconfigurable systems. The lack of benchmarks presents a great barrier to not only overhead reduction technique development, but also run-time reconfigurable system architecture research. We look forward to future research that develops adequate benchmark suites to address the impact of these techniques for different application domains.

The current reconfigurable systems are divided by three categories – *fine grain, medium grain, and coarse grain* – depending on applications they attempt to attack. To evaluate overhead reduction techniques or compare various architectures, a well-designed benchmark suite for each category is necessary. This requires research efforts to analyze a set of applications, discovering the parallelism and granularity.

In addition, we look forward to discover the trade-off of logic and configuration cache. As mentioned, hardware density of run-time reconfigurable system is very sensitive to reconfiguration overhead. Caching techniques demonstrate great promise in reducing reconfiguration overhead, and thus improve hardware density. Logic and interconnect

of configurable devices consume much more hardware resources than configuration cache. Consequently, converting hardware consumed by logic and interconnect into fast configuration caches can effectively improve hardware density. Specifically, larger configuration caches can significantly reduce reconfiguration overhead. Though logic and interconnect are reduced, more portions of applications can be executed on-chip due to the overhead reduction, effectively improving hardware density.

However, converting too much logic and interconnect into configuration cache can greatly reduce computation power of the configurable device. Although a very large cache can virtually eliminate reconfiguration overhead, it will use up hardware necessary for computation, limiting the system's utility. Therefore, it is hoped that future research will explore the trade-off of logic, and interconnect, and configuration cache.

References

[Altera99] Altera Inc.. Altera Mega Core Functions, *http://www.altera.com/html/tools/megacore.html*, San Jose, CA, 1999.

[Babb99] J. Babb, M. Rinard, C. A. Moritz, W. Lee, M. Frank, R. Barua, S. Amarasinghe. Parallelizing Applications into Silicon, *IEEE Symposium on Field-Programmable Custom Computing Machines*,1999.

[Belady66] L. A. Belady. A Study of Replacement Algorithms for Virtual Storage Computers. *IBM Systems Journal* 5, 2, 78-101, 1966.

[Betz99] V. Betz, J. Rose. FPGA Routing Architecture: Segmentation and Buffering to Optimize Speed and Density. *ACM/SIGDA International Symposium on FPGAs*, pp. 59-68, 1999.

[Bolotski94] M. Bolotski, A. DeHon, T. F. Knight Jr.. Unifying FPGAs and SIMD Arrays. *2nd International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, 1994.

[Brasen98] D. R. Brasen, G. Saucier. Using Cone Structures for Circuit Partitioning into FPGA Packages. *IEEE Transactions on CAD of Integrated Circuits and Systems*, Vol. 17, No.7, pp. 592-600, July 1998.

[Brayton84] R. K. Brayton, G. D. Hachtel, C. T. McMullen and A. L. Sangiovanni-Vincentelli. Logic Minimization Algorithms for VLSI Synthesis. *Kluwer Academic Publishers*, 1984.

[Brown92] S. D. Brown, R. J. Francis, J. Rose, Z. G. Vranesic. *Field-Programmable Gate Arrays*, Boston, Mass: Kluwer Academic Publishers, 1992.

[Burns97] J. Burns, A. Donlin, J. Hogg, S. Singh, M. de Wit. A Dynamic Reconfiguration Run-Time System, *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1997.

[Cadambi98] S. Cadambi, J. Weener, S. C. Goldstein, H. Schmit, D. E. Thomas. Managing Pipeline-Reconfigurable FPGAs. *ACM/SIGDA International Symposium on FPGAs*, pp. 55-64, 1998.

[Callahan91] D. Callahan, K. Kennedy, A. Porterfield. Software Prefetching. *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 40-52, 1991.

[Callahan98] T. J. Callahan, P. Chong, A. DeHon, J. Wawrzynek. Fast Module Mapping and Placement for Datapaths in FPGAs. *ACM/SIGDA International Symposium on FPGAs*, pp. 123-132, 1998.

[Cmelik93] R. F. Cmelik. Introduction to Shade. *Sun Microsystems Laboratories*, Inc., February, 1993.

[Compton00] K. Compton, J. Cooley, S. Knol, S. Hauck. Abstract: Configuration Relocation and Defragmentation for FPGAs. *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.

[Dandalis01] Andreas Dandalis, Viktor Prasanna. Configuration Compression for FPGA-based Embedded Systems. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2001.

[DeHon94] Andre DeHon. DPGA-coupled microprocessors: Commodity ICs for the early 21st century. In Duncan A. Buell and Kenneth L. Pocek, editors, *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, pp 31--39, April 1994.

[Deshpande99] D. Deshpande, A. Somani. Configuration Caching Vs Data Caching for Striped FPGA. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 206-214, 1999.

[Ebeling96] C. Ebeling, D.C. Cronquist, P. Franklin. RaPiD – Reconfigurable Pipelined Datapath. *Lecture Notes in Computer Science 1142—Field-programmable Logic: Smart Applications, New Paradigms and Compilers*. R. W. Hartenstein, M. Glesner, Eds. Berlin, Germany: Springer-Verlag, pp. 126-135, 1996.

[Elbirt00] A. J. Elbirt, C. Paar. An FPGA Implementation and Performance Evaluation of the Serpent Block Cipher. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 33-40, 2000.

[Estrin63] G. Estrin et al.. Parallel Processing in a Restructurable Computer System. *IEEE Trans. Electronic Computers*, pp. 747-755, 1963

[Garey79] M. R. Garey, D. S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, 1979.

[Hauck97] S. Hauck, T. Fry, M. Hosler, J. Kao. The Chimaera Reconfigurable Functional Unit. *IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.

[Hauck98] S. Hauck. Configuration Prefetch for Single Context Reconfigurable Coprocessors. *ACM/SIGDA International Symposium on Field-Programmable Gate Array*s, pp. 65-74, 1998.

[Hauck99] S. Hauck, W Wilson. Abstract: Runlength Compression Techniques for FPGA Configuration. *IEEE Symposium on FPGAs for Custom Computing Machines*, 1999.

[Heron99] J. Heron, R. Woods. Accelerating run-time reconfiguration on FCCMs. *IEEE Symposium on FPGAs for Custom Computing Machines,* pp. 131-140, 1999.

[Huffman52] D. A. Huffman, A Method for the Construction of Minimum Redundancy Codes. *Proceedings of the Institute of Radio Engineers 40*, pp 1098—1101, 1952.

[Joseph97] Doug Joseph, Dirk Grunwald. Prefetching Using Markov Predictors. *Proceedings of the 24th International Symposium on Computer Architecture*, 1997.

[Lawler] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys. The Traveling Salesman Problem. *John Wiley and Sons*, New York, 1985.

[Leung00] K. Leung, K. Ma, W. Wong, P. Leong. FPGA Implementation of a Microcoded Elliptic Curve Cryptographic Processor. *IEEE Symposium on FPGAs for Custom Computing Machines,* pp. 192-201, 2000.

[Li00] Z.Li, K. Compton, Scott Hauck. Configuration Caching Management Techniques for Reconfigurable Computing". *IEEE Symposium on FPGAs for Custom Computing Machines,* pp. 87-96, 2000.

[Li99] Z. Li, S. Hauck, Don't Care Discovery for FPGA Configuration Compression. *ACM/SIGDAInternational Symposium on Field-Programmable Gate Arrays*, pp. 91-100, 1999.

[Lucent98] Lucent Technology Inc.. FPGA Data Book, 1998.

[Luk96] C.-K. Luk, T. C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 222-233, 1996.

[Mowry92] T. C. Mowry, M. S. Lam, A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 62-73, 1992.

[National 93] National Semiconductor. Configurable Logic Array (CLAy) Data Sheet, December 1993.

[Nelson95] Mark Nelson, Jean-loup Gailly. The Data Compression Book. *M&T Books,* 1995

[Park99] S.R. Park, W. Burleson. Configuration Cloning: Exploiting Regularity in Dynamic DSP Architecture. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 81-90, 1999.

[Razdan94] R. Razdan, *PRISC: Programmable Reduced Instruction Set Computers*, Ph.D. Thesis, Harvard University, Division of Applied Sciences, 1994.

[Rencher97] M. Rencher, B. Hutchings. Automated Target Recognition on SPLASH2. *IEEE Symposium on FPGAs for Custom Computing Machines,* pp. 87-96, 1997.

[Richmond01] Melany Richmond. A Lemple-Ziv based Configuration Management Architecture for Reconfigurable Computing. Master's Thesis, University of Washington, Dept. of EE, 2001.

[Sanchez99] E. Sanchez, M. Sipper, J.O. Haenni, P.U. Andres, Static and dynamic configurable systems, *IEEE Transaction on Computers,* 48 (6) 556-564, 1999.

[Santhanam97] V. Santhanam, E. H. Gornish, W.-C. Hsu. Data Prefetching on the HP PA-8000. *International Symposium on Computer Architecture*, pp. 264-273, 1997.

[Schmit97] H. Schmit. Incremental Reconfiguration for Pipelined Applications. *IEEE Symposium on FPGAs for Custom Computing Machines,* pp. 47-55, 1997.

[Spec95] SPEC CPU95 Benchmark Suite. Standard Performance Evaluation Corp., Manassas, VA, 1995.

[Storer82] J.A. Storer, T. G. Syzmanski. Data Compression via Textual Substitution. *Journal of the ACM*, 29:928-951, 1982.

[Trimberger97] Steve Trimberger, Khue Duong, and Bob Conn. Architecture issues and solutions for a high-capacity FPGA. *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp 3--9, February 1997.

[Welch84] T. Welch. A technique for high-performance data compression. *IEEE Computer*, pp 8-19, June 1984.

[Wirthlin95] M. J. Wirthlin, Brad L. Hutchings. A dynamic instruction set computer. *In Peter Athanas and Kenneth L. Pocek, editors, Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pp 99--107, April 1995.

[Wirthlin96] M. Wirthlin, B. Hutchings. Sequencing Run-time Reconfigured Hardware with Software. *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 122-128, 1996.

[Witten87] I. H. Witten, R. M. Neal, J. G. Cleary. Arithmetic Coding for Data Compression. Communications of the ACM, vol. 30, pp. 520-540, 1987.

[Wittig96] R. Wittig, P. Chow. OneChip: An FPGA Processor with Reconfigurable Logic. *IEEE Symposium on FPGAs for Custom Computing Machines*, 1996.

[Xilinx 94] Xilinx Inc.. The Programmable Logic Data Book, 1994.

[Xilinx97] Xilinx, Inc.. XC6200 Field Programmable Gate Arrays Product Description. April 1997.

[Xilinx99] Xilinx, Inc. Virtex Configuration Architecture Advanced Users' Guide. June, 1999.

[Xilinx00] Xilinx, Inc. Virtex II Configuration Architecture Advanced Users' Guide. March, 2000.

[Young94] Neal E. Young. The k-server dual and loose competitiveness for paging. *Algorithmica, 11(6),* 535-541, June 1994

[Ziv77] J. Ziv, and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, pp 337-343, May 1977.

[Ziv78] J. Ziv, and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, pages 530-536, September 1978.

Appendix NP-Completeness of the Wildcard Compression

In the standard two-level logic minimization problem, the goal is to find the minimum number of cubes that cover the ON set of a function, while covering none of the OFF set. Wildcard compression seeks the fewest wildcard-augmented writes that will set the memory to the proper state.

The two problems can be formally defined as follows:

TWO-LEVEL-LOGIC: Given a Boolean function X, specified as an ON-set and an OFF-set with a total of n terms, and a value j, is there a set of at most j cubes that covers precisely those minterms in the ON-set of the given function?

WILDCARD-WRITES: Given a configuration Y with n total addresses and a value k, is there a sequence of at most k wildcard writes that implements the given configuration?

TWO-LEVEL-LOGIC is known to be NP-complete, by a reduction from the MINIMUM-DNF problem [Garey79]. In the following, NP-completeness of WILDCARD-WRITES is established.

Theorem: WILDCARD-WRITES is NP-complete.

Proof: First, we observe that WILDCARD-WRITES is in NP, since if we are given a configuration Y, a value k, and a sequence of wildcard writes, it is easy to verify in polynomial time that the sequence contains at most k wildcard writes and implements the configuration Y.

To show that WILDCARD-WRITES is NP-hard, we proceed by reduction from the TWO-LEVEL-LOGIC problem.

Let an instance *(X, j)* of TWO-LEVEL-LOGIC with *n* terms be given. Each of the *n* terms is in either the ON or the OFF set of the function *X*. We will construct a corresponding instance *(Y, k)* of WILDCARD-WRITES as follows: For each address in the configuration *Y*, set the address to “1” if the corresponding term is in the ON set of *X*, and to “Don't Touch” if the corresponding term is in the OFF set of *X*. Set $k = j$.

Now, we observe that there is a one-to-one correspondence between a cube that covers a collection of minterms in the ON set of *X* and a wildcard write that sets the values of the corresponding addresses in *Y* to “1”. It follows from this observation that the ON set of *X* can be precisely covered by at most *j* cubes if and only if *Y* can be implemented with at most *j* (equivalently, *k*) wildcard writes. Since it is clear the construction of *(Y, k)* from *(X, j)* can be performed in polynomial time, it follows that that TWO-LEVEL-LOGIC is polynomial-time reducible to WILDCARD-WRITES.

From the reducibility of TWO-LEVEL-LOGIC to WILDCARD-WRITES, it follows that WILDCARD-WRITES is NP-hard. Since we have already established that it is in NP, it follows that WILDCARD-WRITES is NP-complete.

-

5 -