Algorithm Partitioning and Scheduling for Adaptive Computers

by
Christopher Cornelius Doss

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Doctorate of Philosophy

Department of
Electrical and Computer Engineering

Raleigh, NC

May 2001

Computer Engineering

Approved By:

Chair, Dr. Clay S. Gloster	Member, Dr. Winser E. Alexander
Member, Dr. Thomas Conte	Member, Dr. Purush Iyer

Abstract

Doss, Christopher Cornelius. Algorithm Partitioning and Scheduling for Adaptive Computers *(Under the direction of Dr. Clay S. Gloster Jr. and Dr. Winser E. Alexander.)* Adaptive, or reconfigurable, computing has emerged as a viable computing alternative for computationally intense applications. (We use the terms *adaptive* and *reconfigurable* interchangeably). Here, an adaptive computer is a computing system that contains a general purpose processor attached to a programmable logic device such as a field programmable gate array (FPGA). These computing systems combine the flexibility of general purpose processors with the speed of application specific processors. The computer system designer can cater the hardware to a specific application by modifying the configuration of the FPGAs. The designer can reconfigure the FPGAs at some future time for other applications since the FPGAs do not have a fixed structure.

Several reconfigurable computers have been implemented to demonstrate the viability of reconfigurable processors [1, 2, 3, 4]. Applications mapped to these processors include pattern recognition in high-energy physics [5], statistical physics [6], and genetic optimization algorithms [7, 8]. In many cases [1, 9, 10], the reconfigurable computing implementation provided the highest performance, in terms of execution speed, published (at the respective time).

To achieve such performance, the application must effectively utilize the available resources. This presents a challenge for software designers, who are generally

used to mapping applications onto fixed computing systems. Generally, the designers examine the available hardware resources and modify their application accordingly. With reconfigurable computers, the available resources can be generated when needed. While it may seem that this flexibility would ease the mapping process, it actually introduces new problems, such as what components should be allocated, and how many of each component should be used to generate the best performance. With conventional hardware components, these questions were not an issue. In addition, software engineers are generally not adept at hardware design [11].

In this dissertation, we present a design methodology for systematically implementing computationally intense applications on reconfigurable computing systems. This methodology is based on concepts from compiler theory to ease automation.

In addition to the design methodology, we present *RAS*, a tool that implements a significant portion of the design methodology. *RAS* can be considered as a module generation tool for assisting the design process. Given a flow graph representing a loop nest, *RAS* allocates a set of resources, and schedules the nodes of the graph to the resources. *RAS* also generates an estimate of the amount of time it would take if the design implemented according to the schedule.

This dissertation also presents results of designs produced by *RAS*. Multiple tests were performed using three computationally intense algorithms. *RAS* mapped the algorithms to five configurations representing different sets of resource constraints. Two of the configurations were based on actual systems used in the research development, while the remaining three were hypothetical systems based on other components available in the market. Experimental results from *RAS* indicate that a significant amount of speedup is attainable using the allocated resources with the given schedule.

Biography

Christopher Cornelius Doss was born in Columbus, Ohio, on July 9, 1971 to Calvin Edward Doss and Queenie Lensua Brown. He was preceded by one brother, Surgret Urania Doss. He attended high school in both Brandon, Florida and Wiesbaden, Germany during the years 1985 to 1989. Upon graduation from high school, he attended the University of South Florida, where he received his Bachelor of Science degree in Computer Engineering in Spring, 1994. He began his graduate work at North Carolina State University in Fall, 1994. He received his Masters of Science degree in Computer Engineering in Spring, 1997. He then entered the Ph.D. program at North Carolina State University in Fall, 1997. He presented his Qualifying Review in Spring, 1999, and Preliminary Orals in Spring 2000.

Mr. Doss has extensive work experience in the field of computer engineering. His experience includes the following:

- He has worked several times as a teaching assistant for North Carolina State University, beginning in Fall, 1994. His duties included developing and grading lab assignments.
- He has worked as a computer programmer for the North Carolina Department of Transportation from 1997 to 1998. His duties included creating a database update system for use by field technicians.
- He worked as a software developer for International Business Machines from 1998 to 1999. His duties included program development and correction.

- He worked as a research assistant for the National Aeronautics and Space Administration from 1999 to 2000. His duties included the development of reconfigurable computing applications.
- He worked as an instructor at Wake Technical Community College from 2000 to 2001. His duties included course development, teaching, and grading.

Acknowledgments

I would like to thank God for giving me the strength and abilities needed to successfully complete my thesis. Without His strength, inspiration, and care, I would not have been able to focus on completing this work.

I would also like to thank Dr. Clay Gloster, Jr. for his support throughout my graduate education and the writings of both my Masters thesis and my doctoral dissertation. His keen insights, constructive suggestions, and availability have been greatly appreciated. I also thank Dr. Winser Alexander for assisting me throughout my graduate education. He has graciously given me insights and comments beneficial to my development as a researcher. I would also like to thank the other members of my committee, Dr. Thomas Conte and Dr. Purush Iyer, who have most generously offered their advice and help.

I also thank Dr. Kevin Bowyer, Dr. Joan Holmes, and Dr. Joy O'Shields for taking a special interest in me while obtaining my undergraduate degree. Without their encouragement, preparation, and assistance, I might not have pursued a graduate degree.

I would like to express my deepest appreciation and love to my parents, who have given me the necessary background, encouragement, and financial support to make pursuing this task possible. I also thank my brother for his loving guidance and words throughout my life.

Contents

List of Figures

List of Tables

Chapter 1

Dissertation Overview

This dissertation presents a design methodology for mapping computationally intense regions of an algorithm to reconfigurable computers, and a tool that automates a significant portion of the methodology. Section 1.1 presents an introduction to the field. Section 1.2 presents the contributions made in this dissertation. Definitions of terms used in this dissertation are presented in Section 1.3.

1.1 Introduction

Reconfigurable computers (R.C.) have been shown to perform extremely well with many computationally intensive applications [1, 5, 6, 7, 8, 9, 10]. These systems combine the flexibility of general purpose processors with the speed of application specific processors. Several reconfigurable computers have been implemented to demonstrate the viability of reconfigurable processors [1, 2, 3, 4]. Applications mapped to these processors include pattern recognition in high-energy physics [5], statistical physics [6], and genetic optimization algorithms [7, 8].

This research has shown that reconfigurable computers present a viable new option to the computer design paradigm. However, this new option has opened the door to a methodology that is unfamiliar to most computer designers. These implementations require the skills of both hardware designers and software programmers.

Because there are limited tools for implementing high-level languages on R.C. systems, mapping an application to a R.C. is primarily a manual process.

This motivated our research group to investigate the potential for developing a tool for automating this process. While there is a fair amount of research into the development of tools that assist in the implementation of R.C. applications [3, 11, 12, 13, 14], we decided to approach this research by first mapping an application to the R.C. system in our research lab. We anticipated discovering potential methodology steps that would be prime candidates for automation while implementing this candidate design.

The completion of this initial candidate application required several members of our research group to work for several months to complete a functional implementation. After successfully mapping the application to the R.C system, our R.C. implementation compared favorably to the general purpose processor implementation. However, we realized we were using an ad hoc approach for mapping the design to the R.C. system. We felt subsequent implementations of other applications would also have an ad hoc approach and require several months to obtain good results. We also felt it would be difficult for other members of our research group to duplicate our results.

Thus, we discovered the need for a design methodology that could be followed to consistently obtain good results in a reasonable time period. We determined this methodology should be developed in a manner that would facilitate automation, enabling the development of a tool for mapping algorithms to R.C. systems. Thus, this methodology should meet the following goals:

- Reduce development time for implementing an R.C. application.
- Enhance the performance of the R.C. system implementation in comparison to a typical general purpose processor system implementation.
- Be developed in a manner that can be easily automated.

The effectiveness of this methodology has been demonstrated through the design of three computationally intensive algorithms on R.C. systems. The basis for automation has been developed by automating a key portion of the design methodology.

This dissertation is organized as follows. The following chapter presents an overview of tools for mapping algorithms to hardware. Chapter 3 presents an overview of our design methodology. Chapter 4 presents a discussion of the techniques and concepts our tool is based on. We present the implementation of our tool in Chapter 5. Experiments are presented in Chapter 6. The final chapter presents conclusions and future research.

1.2 Major Contributions

The objective of this research is the development and automation of a design methodology for implementing applications on reconfigurable computers. The contributions of this research are:

1. A design methodology that can be utilized by a novice computer designer to produce R.C. implementations that are significantly faster than a general purpose processor implementation.

2. A tool that automates key portions of the design methodology. This tool can be considered a module generator for implementing loop nests on R.C. systems.

3. A foundation upon which to continue the development of a tool that automates the entire process for mapping applications to R.C. systems.

4. The R.C. designs of three widely used computationally intensive algorithms:
   [A.] Matrix multiplication;
   [B.] The red-black Gauss-Seidel relaxation;
   [C.] The calculation of the residual.

5. The demonstration of the feasibility of using floating point precision in R.C. systems.

1.3 Terminology

This section defines some important terms used throughout this dissertation.

- **Reuse:** - An array element whose value is used (fetched or stored) more than once in a loop, or is used in multiple iterations of the loop.
- **Reuse distance:** - Distance in memory locations between two memory references.
- **Spatial reuse:** - Two memory uses of nearby memory locations.
- **Temporal reuse:** - Two uses of of the same memory location at different times.
- **Sequential reuse:** - Spatial reuse with distance of one.
- **Locality:** - Memory location present in higher levels of memory hierarchy at time of reuse.
- **Spatial locality** - Use of two nearby memory locations, with elements in higher levels of memory at time of reuse.
- **Sequential locality** - Spatial locality in which the elements have a distance of one, and are both in higher levels of memory at time of reuse.
- **Temporal locality** - Reuse of an element at different times, with the element in higher levels of memory at time of reuse.
- **Stride:** - Reuse distance.

- **Stride - 1:** - Sequential reuse.
- **Dependence:** - When a program statement S_1 accesses (reads or writes) a value that another program statement S_2 accesses (reads or writes). This is denoted $S_1 \lhd S_2$.
- **Flow dependence** - When $S_1 \lhd S_2$, and S_1 sets a value that S_2 uses. This is sometimes denoted $S_1 \delta^f S_2$.
- **Anti-dependence** - When $S_1 \lhd S_2$, and S_2 sets a value that S_1 uses. This is sometimes denoted $S_1 \delta^a S_2$.
- **Output dependence** - When $S_1 \lhd S_2$, and both set the value of some variable. This is sometimes denoted $S_1 \delta^o S_2$.
- **Input dependence** - When $S_1 \lhd S_2$, and both read the value of some variable. This is sometimes denoted $S_1 \delta^i S_2$.

Chapter 2

An Overview of Reconfigurable Computing

The advancement of technology has resulted in the computer industry witnessing the development of many computationally intensive applications. Examples of these application are from the fields of digital signal processing, encryption, and pattern matching. As the programs become more complex, digital system designers must develop more efficient systems to ensure that the execution time for these applications is not excessive.

In the past, computer designers generally have had two choices to implement their system:

1. Design a system that is flexible.

2. Design a system for a specific purpose [15].

Flexible systems, typically including a general purpose processor, are capable of performing most common functions in an acceptable amount of time. Because the design of the system is not centered around a single function, or a small group of functions, they can be programmed to perform a large variety of applications. Example applications executed on general purpose processors are: word processors, computer automated design applications, databases, and games. Special purpose systems, typically

consisting of an application specific integrated circuit (ASIC), are usually designed to perform a small group of functions very efficiently. Because ASICs are designed in this manner, they are generally very fast at a certain type of application, but incapable of performing a wide variety of applications. For instance, a particular special purpose system could possibly display graphics quickly, but would be incapable of, or extremely slow at, performing scientific computations.

The recent advancements in technology, specifically with configurable hardware devices, have given designers a new option when designing a system. With these reconfigurable hardware devices, designers can configure the system to perform one set of tasks, specifically for one application, then configure the system to perform a different set of tasks for another application at a later time. The reconfigurability provides the ability to execute a large variety of applications. The hardware provides enhanced performance since it is configured to perform a specific set of tasks. Thus, reconfigurable computers provide a happy medium between two disparate design choices. The potential performance enhancements of R.C. systems has led to a significant amount of research in the area. The following section discusses a brief history of reconfigurable computing. Section 2.2 discusses some applications that have been mapped to R.C. systems. Section 2.3 discusses some methodologies for mapping algorithms to R.C. systems. Section 2.4 concludes the chapter with a discussion of several design automation tools for R.C. systems.

2.1 A Historical Overview of Reconfigurable Computing

The complexity of scientific algorithms has necessitated the need for more advanced processing techniques. Often, the performance of these applications on typical computer systems is unacceptable. This has resulted in research into advanced techniques

to reduce the execution time of these algorithms. There are three factors that determine the execution time for a computer application [16]:

1. The clock frequency.
2. The total number of instructions executed.
3. The average number of clock cycles needed to execute each instruction.

The most straight forward method of decreasing the execution time of an application is to simply increase the clock frequency. However, the highest speed at which an integrated circuit can operate is usually limited by the technology. Usually, running an integrated circuit at a speed higher than the rate for which it is designed can cause problems. For a given technology, this method is typically not a viable alternative.

The total number of instructions is based on the underlying algorithm, and predominantly independent of the hardware it is executed on. Thus, computer designers tend to attempt to reduce the average number of clock cycles needed to execute each instruction. This can be accomplished by altering the architecture of the system to reduce the number of clock cycles for each instruction, or by performing multiple instructions per clock cycle.

Computer architects have generally had two choices for addressing the need for increased performance. The primary option has been the use of the latest generation of general purpose processors due to their flexibility and increased clock rate. For applications that require even more computational power than obtained with typical processors, the parallelism embedded in the algorithm can be exploited by using multi-processor systems or advanced general purpose processors, such as super-scalar and very long instruction word processors. While multi-processor systems can theoretically be expanded to an infinite number of processors, they are often expensive and difficult to program [16].

The implementation of specialized co-processors, or more generally, special purpose processors, is the secondary option available to system designers. This option reduces the average number of clock cycles for each instruction. Co-processors and special purpose systems are developed to perform a small number of functions very quickly, and provide lower execution time than general purpose processors for these tasks [16]. Because they are designed in this manner, they are generally very fast at a certain class of applications, but incapable of performing a wide variety of applications.

The concept of specializing the architecture of a processor while maintaining flexibility first appeared in the early 1970's, with researchers investigating the potential for augmenting the microcode of processors for a given application. Here, the idea was to modify the instruction set at compile time to reduce the traffic to main memory [17]. The instruction set was modified dynamically to allow the usage of very complex instructions. These complex instructions reduced the number of instruction fetches and decodes, thus reducing the number of main memory accesses [17]. Since, at that time, the access to main memory was an order of magnitude slower than microcode memory, it was envisioned that these complex, multi-cycle instructions would lead to enhanced performance.

One implementation of an application specific instruction was the polynomial evaluation instruction for the DEC VAX. However, it is very difficult for compilers to recognize when to use such a complex instruction in high-level languages. Research at IBM showed that only a small subset of these complex instructions were actually being used in practice [18]. As a result, in the 1980's, computer architects focused on reducing the number of instructions available [16]. This led to the introduction of the Reduced Instruction Set Computer (RISC). This reduction in the number of available instructions allows for a faster clock cycle, translating to a reduction in execution time.

The concept of combining the speed of special purpose processors with the flexibility of general purpose processors has been revisited with the use of reconfigurable computing systems [15]. We define reconfigurable computing systems as computer systems containing at least one hardware component that is designed for its functionality to be modified by the end user (we do not include storage elements here). Operations implemented on R.C. systems can be viewed as complex instructions. However, instead of simply reducing interaction with main memory, these instructions take advantage of the inherent parallelism of the underlying algorithm. This enables a significant performance advantage over a typical general purpose processor implementation. Because the structure of the reconfigurable component is not fixed, it can be customized for other applications. This allows the component to be flexible, a quality not available in special purpose processors.

The key to reconfigurable computing systems is the capability of the reconfigurable component to implement any given logic function. The reason for this is that the logical functions, and the interconnections to them, are not fixed. This is in contrast to most hardware used in computing, which have fixed logical functions and interconnections.

Although the current interest in reconfigurable computing is only a few years old, reconfigurable computing components were first proposed over 30 years ago by Gerald Estrin of UCLA [19]. He proposed coupling a fixed general purpose processor with a configurable logic component. Unfortunately for Estrin, the technology for these components was not available at the time. Interest in this methodology did not surface until after the introduction of more advanced logical devices. These devices include such components as:

- **Programmable Array Logic (PAL[1])** - A programmable logic device, designed to implement two level sum-of-product (SOP) functions (where the

[1]PAL is a registered trademark of Monolithic Memories Inc.

first level consists of AND gates and the second level consists of OR gates) in which the first logic level is a programmable array the second logic level is a fixed array.

- **Programmable Logic Array (PLA)** - A programmable logic device designed to implement SOP functions in which both logic levels are programmable arrays.

- **Complex Programmable Logic Device (CPLD)** - A device that contains a number of PLA or PAL functions sharing a common programmable interconnection matrix.

- **Field Programmable Gate Array** - An array of functionally complete (capable of implementing any logic combination of its input) logic elements connected via interconnection network.

- **Dynamically Programmable Gate Arrays** - A FPGA with on-chip memory for storing configurations.

While there are many devices that qualify for components in a reconfigurable system, most are not seen in typical designs. As PALs and PLAs implement two-level logic, they are not flexible enough to implement the functions reconfigurable systems are foreseen to be used for [2, 9]. FPGAs, on the other hand, are capable of implementing any synchronous digital circuit [9].

When first introduced, the FPGA was typically viewed as a programmable gate array or programmable logic device. However, their method of configuration, with the use of volatile memory cells, was viewed as a liability. During the late 1980's and early 1990's, research groups began to realize this configuration method could open the door to a new method of designing applications. Because the configuration of the FPGA can be specified electrically, their functionality can be modified very

easily. While early FPGAs required several seconds for configuration, newer FPGAs can be configured in less than a millisecond [15]. This, along with their flexibility, has led to the widespread use of FPGAs in computing systems [2, 3, 16, 20, 21]. In fact, FPGAs are the most frequently used component for reconfigurable computing systems [22, 23]. Therefore, in this dissertation, we focus our attention on FPGAs.

The key to reconfigurable computing systems is the capability of the FPGA to implement any given logic function. The reason for this is that the logical functions, and the interconnections to them, are not fixed. This is in contrast to most hardware used in computing, which have fixed logical functions and interconnections. An FPGA consists of an array of logic blocks connected by a programmable network. The FPGA can be programmed at three levels [24]:

1. the logic block level,
2. the interconnection level,
3. the input/output level.

Figure 2.1 from [25] shows the architecture of a Xilinx Virtex chip. It consists of an array of configurable logic blocks (CLBs), surrounded by a programmable interconnection (VersaRing) to the input/output buffers (IOBs). It also contains 4K-bit block memories (BRAMs) and distributed clock signals (DLLs).

Programming at the logic block level consists of specifying the relationship between the inputs of a logic block and its output. An example would be the specification that the output is the addition of the inputs. Figure 2.2 shows the schematic of a configurable logic block (CLB). Each CLB contains two 4-input look-up tables (LUT). The LUTs are function generators, capable of implementing any logical function of the inputs.

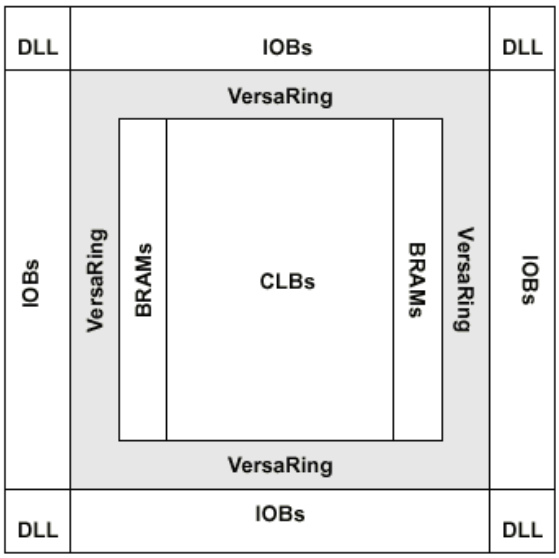

Figure 2.1: Architecture of a modern FPGA.

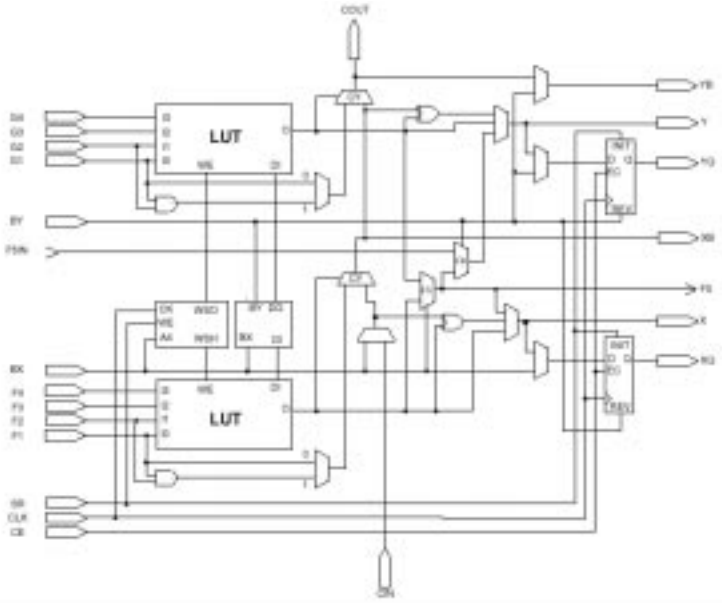

Figure 2.2: Schematic of a modern configurable logic block.

Programming at the interconnection level consists of specifying which logic blocks are connected, and in what manner. An example would be the connection of 16 logic blocks, each configured to perform the ADD operation. The overall structure is a 16-bit ADD function. Figure 2.3 from [25] shows the structure of a modern

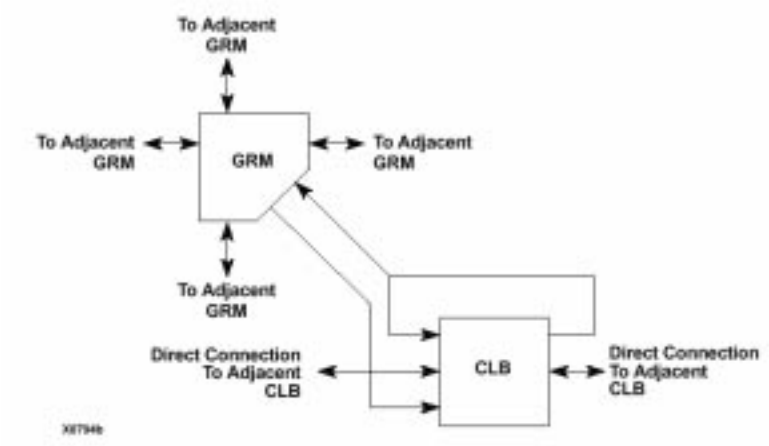

Figure 2.3: Schematic of a modern general routing matrix.

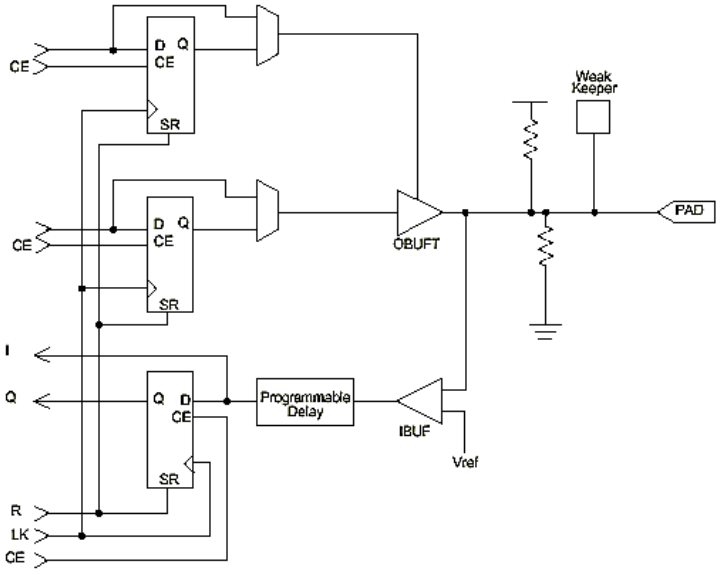

Figure 2.4: Schematic of a modern input/output buffer.

interconnection matrix. Horizontal and vertical resources can be connected via the GRM. CLBs are also connected via the GRM.

Programming at the input/output level consists of specifying how the circuit on the FPGA interacts with the outside world. The input/output pads on the FPGA can be configured to link to the circuit implemented on the FPGA in any way desired. An example would be linking 32 pads to the inputs of the 16-bit ADD circuit, and linking

16 other pads to the output of the 16-bit ADD circuit. Figure 2.4 from [25] shows the structure of a modern input/output buffer. The IOB contains storage elements that can act as flip-flops or latches. Control signals are used to enable the storage elements.

The reconfigurability of FPGAs has limited the device's ability to store designs as large and/or complex as its counterparts are. For example, while a special purpose system could perform all the functions necessary to manipulate graphics, an FPGA may only be capable of storing 50% of the functions (at one time). In addition, the use of logic blocks has resulted in a lower clock rate. However, newer FPGAs contain the equivalent of several million logic gates, and support clock rates of over 300MHz. In addition, they can take advantage of the fact that a large portion of execution time is spent in a small portion of the application code [26]. Thus, the kernel of the application could be coded onto the FPGA, resulting in a large impact on the overall execution time for the application. Also, the FPGA could be configured to perform other functions as needed.

2.2 Summary of Several Implementations of Applications on Reconfigurable Computing Systems

Several research groups have investigated the performance that can be achieved with reconfigurable systems [1, 9, 27, 28, 29]. An early example of a reconfigurable processor is the Splash machine, presented in [1]. This processor, consisting of a linear array of Xilinx 3000 FPGAs, was used to implement a genetic string matching algorithm. This processor was configured to execute the algorithm 200 times faster than the speed achievable on a supercomputer [30].

The researchers in [9] used a 4×4 array of FPGAs connected to a host processor to produce their reconfigurable computer. They refer to it as a programmable active

memory (PAM) because it is attached to the main bus of the host computer, and is read and written much like a RAM. Several scientific applications were implemented on this system. One such application was long integer multiplication [31], where the product $A \times B + S$ is calculated. Here, A is an n-bit long multiplier (up to 2000), B is an arbitrary sized multiplicand, and S is the summand. Their performance was 16 times faster than the same application on a Cray II and Cyber 170/750 [32]. In fact, their implementation was faster than all previously published benchmarks. RSA Cryptography [33] is another application in which their system was faster than those previously published [34]. Reconfigurable systems were shown to provide the highest performance available for other applications as well, including: modular multiplication [35], statistical physics [6, 36], region detection and labeling [37], and hidden Markov modeling [38].

With many of the implementations, there were no guidelines to follow for the implementation. Thus, future implementations would possibly not lead to significant performance enhancement. Thus, several research groups have developed environments and methodologies that facilitate the mapping of algorithms to a reconfigurable computing system [39, 40, 41, 42]. The following section discusses these methodologies.

2.3 Summary of Design Methodologies for Reconfigurable Computing Systems

In this section, we discuss several design methodologies for implementing algorithms on R.C. systems. The goal of these methodologies is to have a standard method for obtaining good results from the mapping of computationally intense applications on R.C. systems.

Hazelwood et. al. present a design environment for creating reconfigurable computing systems in [39]. The purpose of the Reconfigurable Computing Application

Development Environment (RCADE) is to facilitate novice and expert R.C. system designers to implement R.C. applications that provide enhanced performance over a software solution. RCADE includes a set of tools and a library of components that can be used by the designer.

RCADE designers represent algorithms with the *Algorithm Description Format* (ADF), comprised of *Design Units* (DU). A DU represents a single graph. The nodes contain information about the component selected for implementing a given operation in the algorithm. This information includes the number of input and output ports. The edges in the graph represent connections between ports.

RCADE has/will have several analysis tools to aid in the development process. These tools include:

- **Precision Analysis Tool:** A tool to help the user to specify the precision for the algorithm.
- **Throughput Analysis Tool:** A tool that examines the data production and consumption rates of the components, and reduces the number of components accordingly.
- **Pipeline Balancing Tool:** A tool that examines alternate paths of the design that merge to a common node, and inserts FIFOs in the path with the least delay to ensure data synchronization.
- **Placement Tool:** A tool for placing the components.

With RCADE, the creation of an R.C. application begins with the user first specifying the design as a graph. The nodes of this graph are abstract operations from the component library. The analysis tools are then used to further refine the design. The

component selector then binds the abstract operations to components in the component library. The placement tool is then used to partition the components among the available devices. The code generator then synthesizes the application.

GajjalaPurna et. al. [40] present a temporal partitioning design methodology, where tasks are separated into subtasks that are executed at different intervals in time. Here, applications are analyzed for their *spatial flexibilities*, tasks that can execute simultaneously, and *temporal constraints*, the amount of freedom for specifying the time interval a task must execute. This examination results in temporal partitions, which each represent a configuration of the FPGA. The temporal constraints specify the order in which the configurations are used. The spatial flexibility of each partition is determined by the parallelism available between different tasks.

The design process begins with an input design specification that is preprocessed into a data flow graph model. Each node of the data flow graph symbolizes a functional block of the application. Edges between the nodes represent the flow of data. Each node is then assigned an *ASAP level* [43]. The ASAP level represents the depth of a node with respect to the design inputs. The ASAP level indicates the available parallelism. Nodes with the same ASAP level can execute in parallel. The nodes are then partitioned using a methodology similar to that used for multiprocessors [44].

Dawood et. al. present their design methodology for implementing a FIR filter in [41]. This methodology focused on a specific implementation, which is in contrast to other methodologies [40, 45].

The design process began by first examining the available factors, such as the filter coefficients and the required performance. The next step was to determine the filter coefficients using Matlab. They then used the following design cycle to produce the actual hardware component:

1. **Design Entry:** A schematic diagram editor is used to produce a FPGA netlist.

2. **Design Implementation:** Place and route tools are used to convert the netlist is into a bitstream, or FPGA configuration file.

3. **Design Verification:** The design is simulated to ensure proper functionality.

4. **FPGA Configuration:** The FPGA is configured by loading the bitstream onto the FPGA.

Shrivastava and Jain present a design methodology for mapping R.C. applications to a rapid prototyping system in [42]. This system is a coarse grain reconfigurable platform, with only a small number of computational cells. Their system is based on two flexible cells:

1. **MA_PLUS:** An enhanced multiply-add cell, capable of performing several functions, including:

   The multiplication of two values, $Z = A \cdot B$.

   The addition or subtraction of two values, $Z = C \pm D$.

   The multiplication and addition of three values, $Z = A \cdot B + C$.

   The multiplication and accumulation of two values with a third, $Z = Z + A \cdot B$.

2. **UNL:** A universal nonlinear cell, capable of the performing several complex functions, including trigonometric functions, inverse operations, and square-root.

The design process involves a two step process. The first step involves selecting several MA_PLUS and UNL cells for the corresponding algorithm. The second step involves the interconnection of the cells to perform parallel processing.

As can be seen, several methodologies can emerge when attempting to implement applications on R.C. systems. The system development needs a fair amount

of experience to properly use these methodologies. (This is in contrast to our design methodology, which remains constant regardless of the application.) In addition, some are fairly complex to utilize. As a result, a vast amount of research is being performed to design tools that will assist in the design process [3, 11, 12, 13, 14]. In order for these tools to gain wide acceptance with computer designers, they must be capable of assisting the designer in mapping algorithms into hardware. The computer designers should be able to implement their algorithm in some high-level hardware description language similar to Verilog or VHDL, and have the tools perform the steps involved to produce a working implementation.

Of course, requiring a hardware description as input limits the use of the software tools to those adept in computer design. For the software to cater to the scientific community at large, the software should allow the user to implement their algorithm in a computer language more familiar to them, such as C, C++, or Java. The acceptance of a high-level language input would allow scientists who are not familiar with computer design to express their algorithms in a language they are familiar with. The software tool should then be capable of mapping these algorithms onto the reconfigurable computer, and executing the algorithm with a significant gain in performance. The following section presents several tools that address this issue.

2.4 Summary of Tools for Mapping Algorithms to Reconfigurable Computing Systems

Several reconfigurable computing systems utilize the FPGA as a high-performance substitute for standard computers or supercomputers [1, 9, 27, 28, 29]. The results indicate that the reconfigurable computing system provides very good performance. As the use of reconfigurable computing systems grows, many new applications will likely use the reconfigurable computing system in this manner. The designers of

these applications will largely be software programmers who are not familiar with hardware design. These programmers are used to writing their applications in high-level languages such as C, C++, or Java. These programmers would want to continue using these languages, while taking advantage of the full power of the reconfigurable system. Therefore, software tools must be capable of mapping applications written in a high-level language to an implementation in the target reconfigurable computing system.

Several research groups have written tools that transform programs written in high-level languages into a hardware implementation [46, 47, 48, 49, 50]. The supported languages include C [3, 12, 13, 14, 16, 20, 47, 51], C++ [48], Java [52, 53], Ada [54], Occam [55, 56], SmallTalk [57], Assembly [58, 59], and various hardware description languages [60, 61]. Typically, these tools accept some subset of the supported language as input. Data computations are translated into hardware operations. Control flow is recreated with the use of state machines, latches, and multiplexors.

In this section, we present an overview of several tools that map applications written in C [3, 12, 13, 14, 16, 20, 47, 51, 62] or Java [53] to reconfigurable computing systems. We also include an overview of a system that maps C programs to a nonprogrammable hardware accelerator [**?**]. We focus mainly on the compilation and mapping techniques used in the tools. These tools were chosen because of the wide use of the languages, and because our future research will likely map C applications to reconfigurable hardware. Also, some of our development tools focus on remote configuration via the internet. The Java Development Kit has a set of classes to ease the communication between objects over the internet. Therefore, we have included tools that map Java to reconfigurable hardware.

These papers represent a wide range of reconfigurable platforms. Tools that accept C programs as input include: PRISM, the proof-of-concept system [13]; PRISM-II [3, 12], an enhancement of PRISM; DISC [51], a microprocessor implemented with

FPGAs; the Transmogrifier C compiler [20], a tool that targets a specific FPGA family; RECON [16], a system consisting of a workstation connected to a FPGA; CoDe-X [62], a tool that targets a reconfigurable ALU; and tools that map C to multiple FPGAs [14, 47]. JVX [53] is a rapid prototyping system that maps Java to FPGAs.

PRISM-I (Processor Reconfiguration through Instruction Set Metamorphosis), the tool presented in [13], maps programs written in C to a host processor and an FPGA. The tool is a configuration compiler that first performs function identification and extraction. This process identifies segments of the input program as either candidates for hardware implementation or software implementation by imposing a minimum resolution function subroutine on potential segments. The hardware candidates are then transformed into a hardware description capable of reconfiguring the hardware device. The software candidates are redefined to access the newly synthesized functions.

The synthesis process first verifies the integrity of the input by passing the C hardware candidate(s) through cpp and lint, two standard Unix tools. Next, the parser converts the input into an internal representation, and creates software access-definitions that will link to the reconfigurable hardware. The internal representation is then used by the parser to build a data-flow graph (DFG). Nodes in the DFG are then used to instantiate hardware components from a library of structures. Optimizations are then performed to minimize the hardware image.

After the generation of the hardware image, the software candidates are combined with the access definitions to create a new program specification. Also, library functions that load and initialize the associated hardware image are inserted into the program. The new program is compiled to produce the final executable. When ran, the program loads and initializes the hardware image(s) automatically.

PRISM proves the viability of having compilers configure hardware in conjunction with software code. The resulting program attains substantial speedup compared to

a Sun Sparc workstation. However, since PRISM is a proof-of-concept system, it has several limitations, including: no global variable support; input arguments and return values are limited to 32-bits; the exit condition for loops must be independent of input arguments; no support for floating-point operations; no support for do-while or switch-case constructs.

Wazlowski et. al. present PRISM-II, an enhancement to PRISM-I, in [3] and [12]. The main differences between PRISM-I and II are the front-end and the number of restrictions on the input source program. PRISM-I uses cpp and lint as the front-end. PRISM-II uses the parsing and optimization stages of the GNU C compiler (GCC) as the front-end of the configuration compiler. While PRISM-I requires for loops to have a fixed loop count and does not allow switch-case statements, PRISM-II allows for loops with variable loop count as well as switch-case constructs.

In PRISM-II, the input is transformed into a control flow graph (CFG), where each node represents a basic block. The CFG is then transformed into an operator network, a directed graph where the nodes represent operations, and edges represent the flow of values. An initial controller is then developed from the CFG. Each basic block represents a different state in the CFG. The controller is then optimized. Finally, the operator network and controller are translated into a netlist for placement and routing.

PRISM-II has several limitations: it does not support memory address references; it does not support arrays, a structure utilized heavily in computationally intense applications; it does not support floating point operations; it does not account for resource constraints while developing the operator network.

The compiler presented in [16] compiles C to RECON, a system consisting of a Sun SparcStation host and a reconfigurable co-processor. Here, a gate array compiler automates the translation of a subset of the C programming language into a netlist. The compiler first performs lexical, syntactic, and semantic analysis. The compiler

then performs an extensive analysis to extract the maximal amount of parallelization inherent in the source function. The compiler uses this information to produce the netlist.

RECON extracts parallelism from a program by performing data dependency analysis on the input function. This approach keeps a list of variables in a table. Information is stored in the table when a variable is assigned. When the variable is encountered, its status is checked by retrieving its corresponding data from the table. This data indicates how the current statement depends on other statements. The compiler then classifies the statements into sequences of blocks where all statements can be executed concurrently.

Each block comprises a state in an asynchronous state machine (ASM). The compiler then converts the ASM into hardware logic by constructing a state transition table. This transition table contains the current state, next state, and condition for transition. The compiler uses combinational logic to construct the state controller.

The compiler then extracts the operations in the ASM and finds the corresponding logic gate/macro in the Xilinx database. The macros, along with the appropriate interconnects, are written to a netlist file. Placement and routing tools ensure the design will fit on the reconfigurable system.

The results indicate that the RECON implementation of an 8-bit multiplication program provides substantial speedup over the SparcStation implementation. However, the RECON compiler accepts only a subset of the C language as input. Multiplication, division, and modulus operations are not supported.

Galloway presents a new language, Transmogrifier C, and its compiler in [20]. Transmogrifier C is similar to regular C, with a minimum number of extensions. It has integer variables, constants, and expressions, as well as if statements, while loops, and function calls. It also supports C preprocessor directives. A `#pragma` statement allows the user to define the bit-length of declared variables. The `inputport`, `outputport`,

and `busport` function calls associate C variables inside the program with the input and output signals at the periphery of the final circuit.

The Transmogrifier C compiler maps Transmogrifier C to the Xilinx 4000 FPGA. The compiler uses lex and yacc, standard UNIX tools, to perform the lexical and semantic checking. It makes two passes on the input program. During the first pass, it creates a state machine and creates combinational expressions that represent the values of each identifier in each state. During the second pass, it performs optimizations on the circuit and produces the final netlist. The extensions to the C language aid in the translation process.

The compiler creates a sequential state machine during compilation. The compiler keeps track of which state corresponds to the code being compiled. The compiler attempts to pack as many assignment statements and if statements as possible into the current state. Assignment statements and boolean expressions in if statements are evaluated as combinational logic expressions. During program execution, the outputs of the combinational logic expressions are stored in flip flops which share a common clock. Execution of such constructs as while loops and function calls require states to determine which portion of the program is executing. The compiler creates two states for while loops: one for the top of the while loop, indicating the test succeeded and the loop is to be executed; and one for the exit of the while loop, indicating the loop test failed. The compiler creates complete state machines for functions. These state machines require at least two states, the initial state and the final state. During program execution, the program enters the initial state when the caller sets the function's initial state to true. The program enters the final state, known to the caller, when the function is finished.

Transmogrifier C is easier for a C programmer to learn than traditional high-level synthesis languages such as Verilog or VHDL. The timing model is easy for a programmer to understand, and allows the programmer to predict the timing of

the final circuit. However, the language severely limits the programmer's options. It forbids multiplication, division, pointers, arrays, data structures, and the use of recursion. Also, it does not allow floating point operations.

Clark and Hutchings present the programming environment for DISC (dynamic instruction set computer) in [51]. This environment consists of an ANSI-C compiler and a retargetable assembler. (It also has a debugger which we do not discuss here). The compiler supports all constructs of C, as well as custom hardware capabilities of FPGAs.

The programming environment uses the lcc [63] front-end to parse the input program, perform optimizations, and produce intermediate code. The modified lcc back-end produces code for the DISC processor [2]. The back-end uses a set of base instructions, stack protocols, register conventions, and other mnemonics that describe the processor. Using these rules, the back-end transforms the intermediate code produced by the front-end into assembly code.

The programming environment uses dasm, a retargetable assembler, to assemble the code produced by the lcc back-end. dasm accepts a header file and an assembly code file. The header file fully specifies the target machine, including word length, instruction mnemonics, and bit placement for each instruction.

The programming environment supports the entire ANSI-C language. However, the tool does not actually map C to hardware. Instead, the user must first partition the algorithm into hardware and software sections. Next, the user must design the actual hardware modules that will be mapped to the reconfigurable logic. The user then rewrites the software to use the hardware modules.

Peterson et. al. [14] propose an architecture independent tool that translates programs written in C to a data-flow representation. It also schedules and partitions the resulting graphs onto multi-FPGA custom computing machines (CCM). The design

of the tool allows it to target a wide variety of computing platforms. The user specifies the target by modifying a library file that describes the target CCM architecture.

The compiler accepts as input the source program and a specification of the target architecture of the FPGA processor. The compiler supports the entire ANSI-C language. The architecture specification describes the organization of the FPGA-based CCM.

Peterson uses the retargetable C compiler lcc [63] as the basis for their compiler. The lcc compiler is divided into a front-end and a back-end. The front-end performs lexical and syntactic analysis and produces directed acyclic graphs (DAG). Optimizations are performed while the DAGs are being generated. The back-end uses the DAGs produced by the front-end to generate target-specific code. For this research, the targeted code is GDL (Graph Description Language), a language that describes data-flow graphs.

After the original program is translated into GDL, the data-flow graphs are partitioned and scheduled to the FPGA architecture. The tool uses simulated annealing [64] to provide an optimal scheduling and partitioning of the data-flow graphs. A PERL script then translates the schedules into VHDL code for synthesis.

As mentioned before, the compiler supports the entire ANSI-C language, and is capable of targeting many FPGA architectures. Also, the compiler supports multi-FPGA platform. It also generates optimal partitions and schedules of the data-flow graphs. Unfortunately, this tool has not been implemented.

A major goal of the Program-In-Chip-Out (PICO) system [65, 66] is the automation of the design of nonprogrammable hardware accelerators (NPA). Schreiber et. al. present the NPA compiler in [66]. The system accepts C as input, and outputs VHDL code for an array of very-long instruction word (VLIW) processors. Included is the logic for the controller, local memory, and interface to the host processor.

The user provides the system with a loop nest, as well as a range of architectures to explore. The NPA compiler then performs several analyses, in which the compiler can determine any flow dependences, as well as possible iteration space transformations. The compiler then performs on the loop body, including reduction in bit widths for the operands. The compiler then binds operations to the available assets of the processor. The compiler finishes by producing and interconnecting multiple processors, and outputting the corresponding VHDL code and time estimates.

An important disadvantage of the PICO system is the fact that the user must supply the system with a set of potential architectures. Engineers and scientists that are not adept at computer architecture design may not be able to supply this input.

Yamauchi et. al. present the architecture and compiler for the reconfigurable massively parallel Sea of Processors (SOP) system. The mapping process consists of two stages. The tool compiles the input program during the first stage, and performs placement and routing during the second stage.

The SOP compiler supports a language similar to ordinary C, with a few extensions and restrictions. It analyzes the input program, generates the hardware netlist, and partitions the netlist into segments suitable for mapping onto the FPGA. It extracts the parallelism inherent in the application while analyzing the program. The compiler regards variables as nets, and functions and operations as nodes. It expands function calls according to the control data-flow.

The compiler generates data-paths and control circuits for loops. It generates registers for variables. Variables involved in loops are 'imported' if they are inherited from outside the loop. They are 'assigned' if they are assigned within the loop. A token synchronizer at the top of the loop waits for the arrival of tokens, or values of variables. The arrival of tokens triggers the condition circuit derived from the condition part of the loop. The condition circuit activates the loop's inner body

operations. When the condition circuit becomes false, the circuit issues a token that activates the circuitry after the loop.

The compiler produces condition circuits for if statements in a similar fashion. Again, variables involved in conditional statements are classified as either 'imported' or 'assigned'. When the condition circuit becomes true, the true block circuitry is activated. When the condition circuit becomes false, the circuitry following the if block is activated.

The compiler exploits several levels of parallelism using the data-flow representation of the circuit. It exploits inter-block parallelism in loops and if statements when those blocks have no data dependency.

Results indicate that the SOP compiler can achieve a significant amount of speedup compared to the MIPS R4400, up to 4.5 times. However, the compiler needs extensions to the C language. In addition, there are restrictions on the program's input.

Hartenstein et. al. present CoDe-X, a framework for scheduling data and tasks on an Xputer in [62]. Two partitioning levels and a GNU C compiler comprise the CoDe-X system. Data sequencers and a reconfigurable ALU array comprise the hardware. Here, we discuss the partitioners and schedulers. The reader is referred to [62] for details on the hardware.

The first level partitioner accepts X-C, a dialect of C, as input and produces a software image and a hardware image. The software image is input to the GNU C compiler. The software image consists of host tasks, dynamic structures, and operating system calls. The second level partitioner accepts the hardware image as input and produces sequential code for the data sequencer, and structural code for the reconfigurable ALU.

Data and task schedulers in the CoDe-X framework target the data-driven Xputer. CoDe-X creates a total execution graph (TEG) that represents the order of execution

for all tasks in the program. The TEG also represents all data dependencies. Concurrency update points divide the TEG into sequential synchronization time steps with parallel tasks in each step. CoDe-X then processes the first time step of the TEG by assigning all tasks that can be executed in parallel to different Xputer modules. CoDe-X ensures there are not more tasks than available Xputer modules. For the remaining time steps, CoDe-X considers only the tasks that begin in the corresponding time step.

Macketanz and Karl present the JVX (Java, VHDL, Xilinx) rapid prototyping system in [53]. The JVX system consists of a compiler, interpreter, and reconfigurable hardware device. The compiler generates VHDL code from a Java input program. The interpreter executes Java byte-code and triggers the hardware device. The hardware consists of a Xilinx FPGA board connected via PCI bus to a host processor. Here, we discuss the compilation method. The reader is referred to [53] for details of the interpreter and hardware.

The compilation process consists of three phases: input parsing, grammar matching, and byte-code and VHDL generation. Java functions, expressions, and control structures are translated into VHDL state machines. Expressions are partitioned into their corresponding atomic parts. (For example, the assignment expression $a = b + c * d$ would be partitioned into a multiply, addition, and assignment.) The compiler translates all atomic parts into individual states. The parsing grammatics enforce proper mathematical precedence rules. The compiler produces similar states for conditional expressions in if statements and loops. The result is evaluated at runtime to determine the next state.

An important disadvantage of the JVX system is the elimination of much of the object-oriented nature of the Java program. However, the JVX system is still in the early stages of development. Preliminary results for the JVX system are poor, due to the nature of the experiments. Two research topics that have emerged while

developing the current implementation include generating parallel VHDL processes for every independent subexpression, and generating a high-level VHDL behavioral description instead of the expression-for-expression method currently used.

Chapter 3

Design Methodology

3.1 Overview of the Proposed Design Methodology

One of the goals of this research is to develop a step-by-step process that reduces the time required for mapping an application to an R.C. system, and to develop a foundation upon which to build a compiler that is capable of automating the process of mapping software applications to a reconfigurable computer. These goals have been addressed by developing a methodology for mapping algorithms to R.C. systems, and automating a key portion of it. In this dissertation, the validity of this methodology is verified by applying the design methodology to several computationally intensive algorithms and developing the correlation between the steps in the design methodology and existing compiler techniques.

The design methodology we have developed is for implementing designs on a reconfigurable computer architecture, which consists of a general purpose processor connected to a FPGA board. The methodology assumes that the application has been described using Rebel, a format similar to typical assembly language code. This format can be obtained from C code using several tools readily available in the LEGO system developed by the Tinker Group at North Carolina State University.The methodology also requires a predefined library of arithmetic and logical hardware modules. The

library specification for each module includes: the function the module performs, the number of cycles required to complete the function, the number of inputs, amount of area, etc. The functions available in the library include adders, multipliers, dividers, and comparators. The performance goal of our research is an implementation that is significantly faster than a general purpose processor implementation of the same application.

Since we want this methodology to be incorporated into a future compiler, all of the steps of the process must either use techniques currently available in traditional compilers or can be implemented in software and integrated into the final system. We have automated several key steps of the methodology. We have also defined the method by which the steps can be incorporated into given compiler, such as the LEGO Compiler, developed by the Tinker Research Group. We feel this methodology is of value because a novice engineer can follow these steps and produce a design that is several times faster than a general purpose processor implementation.

Figure 3.1 shows the major steps of our design methodology. We have automated some portions of the program analysis step, shown in the highlighted box. What follows is an overview of the design methodology. We include a brief overview of some of the compilation techniques that can be incorporated into our methodology.

3.2 The Proposed Design Methodology

Step 1: Perform Analysis of Input Program

The first step in the mapping process is to perform an extensive analysis of the input program. This analysis includes determining the compute intensive regions of the program and determining the parallelism inherent in the program. Any additional opportunities for improving the program's execution time should be identified during this step.

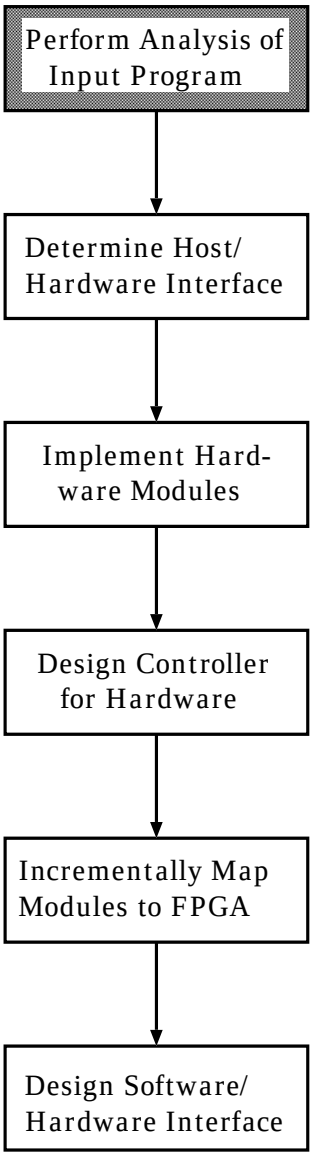

Figure 3.1: Methodology for mapping applications to R.C. systems.

We consider this step to be the most important part of the design methodology. This step involves determining the various options available for implementing the application on the R.C. system. After completing this step, the designer should have a general idea of how fast the resulting implementation will be. The tasks specified in this step can be viewed as a set of heuristics for reducing the searchable design space for R.C. implementations, which is NP-complete. The output of this step is of value because the designer will know how much speedup to expect before going through the

tedious process of creating, synthesizing, and debugging a hardware implementation.

This step is composed of the following tasks:

A.) Create loop block and superblock representation of the program.

B.) Perform dependence analysis of loops and superblocks.

C.) Perform timing analysis of loops and superblocks.

D.) Determine partitions for which superblocks should be implemented in hardware and software.

E.) Perform extensive analysis of the memory access patterns and determine loop and superblock input requirements.

F.) Apply software pipelining techniques to produce pipelined implementation of hardware loops and superblocks.

G.) Apply vectorization and software tiling techniques to produce parallel implementation of software blocks and reduce interaction with memory (if needed).

Step 1.A: Create Loop Block and Superblock Representation of Program

The first task in the analysis of the application is to partition the Rebel code into loops and superblocks. The loop representation is useful because many of the computationally intensive portions of our applications are contained inside of loops. In addition, the loop constructs may be used by the optimization techniques comprising the latter stages of this step.

The remainder of the program will be represented with superblocks. A superblock is a collection of instructions with a single entry point. There may be several exit points in a superblock. This representation can be used to partition entire regions of the program into hardware or software blocks. These constructs can be generated by LEGO.

For example, the following is the pseudo-code for the PNN algorithm [67]:

Algorithm: PNN Multispectral Image Classification

Input: $\mathbf{X}, n, d, \mathbf{W}, \mathbf{P}, k, \mathbf{K1}$, and $\mathbf{K2}$
where $\mathbf{X}$ is the set of n^2 image pixels, each containing d bands; $\mathbf{W}$ is the set of weights, $\mathbf{P}$ is the set of the number of weights for the given classes, k is the number of classes, and $\mathbf{K1}$ and $\mathbf{K2}$ are the sets of $K1$ and $K2$ constants, respectively

Output: New image vector, $\mathbf{X}'$, where each pixel of $\mathbf{X}'$ represents the classification of the corresponding pixel in $\mathbf{X}$

```
For pixel = 1 to n DO
    winner = −1;
    maxval = −1;
    For class = 1 to k DO
        c_sum = 0.0;
        For weight = 1 to P_k DO
            p_sum = 0.0;
            For band = 1 to d DO
                diff = X_{pixel,band} − W^{class}_{pixel,band};
                p_sum += diff^2;
            End For // band loop
            k2_mult = K2_class · p_sum;
            c_sum += e^{k2_mult};
        End For // weight loop
        c_sum *= K1_class;
```

```
        If c_sum > maxval
            maxval = c_sum;
            winner = class;
        End If
    End For // class loop
    X'_pixel = winner;
End For //pixel loop
```

Figure 3.2 shows a loop representation of the two inner loops (weight and band loop). Each block represents a basic block. Block 1 contains the instructions for initializing the weight loop and p_{sum}. Block 2 contains the instructions for calculating the variables $diff$ and p_{sum}. Please note that this block is performed four times for each iteration of the outer loop. Block 3 contains the instructions for calculating the variables $k2_{mult}$ and c_{sum}. This representation was obtained using the LEGO compiler.

Step 1.B: Perform Dependence Analysis of Loops and Superblocks

The second task is to determine the dependence between and within variables in the loops and superblocks. The dependence between blocks can be used to determine if it is possible to execute two blocks simultaneously, or shift their ordering. This is useful because it would possibly allow the host processor to execute some segments of code while the hardware executes other segments. The LEGO compiler has some support for dependence analysis.

Step 1.C: Perform Timing Analysis of Loops and Superblocks

The next task is to perform a timing analysis of the blocks. This analysis is an estimate of the number of cycles that is required to execute the respective blocks. This information will be used to determine the amount of time needed for the application

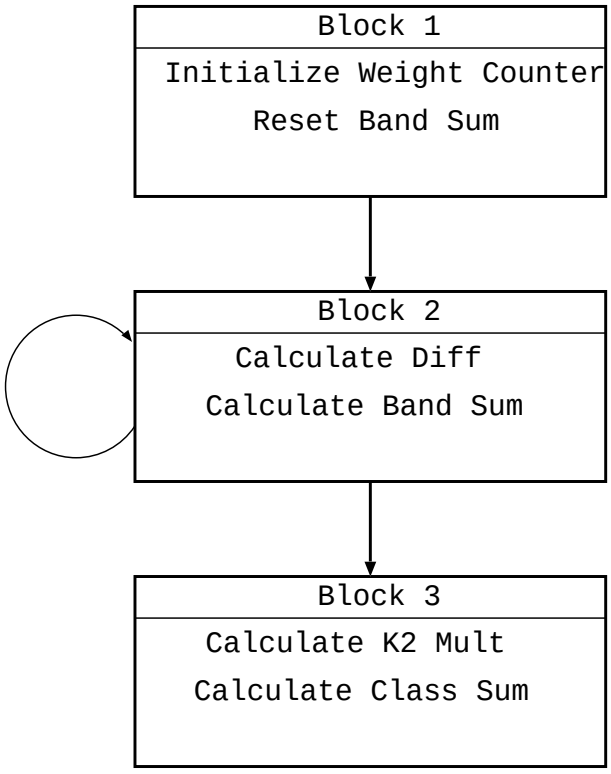

Figure 3.2: Loop block representation of inner loops for PNN.

to run on the host, and the amount of speedup we must attain to meet the user specified constraints. We define speedup as:

$$\frac{T_{GP}}{T_{RC}} \tag{3.1}$$

where T_{GP} is the execution time of the general purpose processor implementation, and T_{RC} is the execution time of the RC implementation.

Step 1.D: Partition Hardware and Software Components

The fourth task is to partition the blocks into hardware and software components. The techniques used in hardware/software co-design (HSC), such as simulated annealing or genetic algorithms, can be applied directly to the blocks [68] to determine which blocks should be implemented in hardware, and the respective timing constraint. HSC uses integer linear programming [69] to find solutions to a set of variables that meet certain constraints.

Step 1.E: Perform Analysis of Memory Access Patterns

The fifth task is to perform an analysis of the memory accesses. Here, the number of memory accesses required for each loop iteration and superblock should be determined. The extent of memory reuse should also be determined during this step to remove repeated loading and storage of data from/to memory. In addition, the validity of transformations for optimizing loop performance should be determined.

Step 1.F: Apply Software Pipelining Techniques

If timing constraints are met, the last step is the pipelining of loop bodies, which provides a large amount of speedup for scientific applications [67]. Given a hardware architecture, software pipelining techniques [70, 71, 72, 73, 74] can be used to determine a schedule that overlaps the execution of successive iterations of a loop with previous iterations. The hardware architectures used in this methodology are derived directly from the data-flow graph of the loop body.

Step 1.G: Apply Vectorization and Tiling Techniques

If pipelining does not provide enough improvement in execution time, vectorization and tiling techniques can be applied to exploit coarse grain parallelism in the program. Here, the pipeline developed in the previous step can be duplicated to perform the steps in parallel.

Step 2: Determine Method Host will Interface with Hardware

The second step of the mapping process is to determine how the host will interface with the hardware. The analysis performed in the first step should be used to determine which portions of the program should be mapped to hardware and which portions should remain in software. When this partitioning is performed, the designer must decide how the host program will feed inputs to and receive outputs from the hardware. In addition, the host may have to perform other tasks, such as initialization and synchronization.

Step 3: Implement Hardware Modules Required for Data-path

The hardware modules that will be used in the program should be designed during this step. This includes any frequently used components, such as adders and subtractors, as well as any modules specific to this design, such as counters, comparators, etc. Since a component library is used, the designer will only be required to produce a small number of components during this step.

The results of several of the previous steps are required to perform this step. The analysis performed in Step 1 should be used to determine the necessary hardware components. Note that the hardware modules are not connected during this step.

Step 4: Design Controller for Hardware

The next step in the design process is to design a controller for the hardware. The controller is responsible for directing the flow of operands through the data-path. The controller accomplishes this by implementing a finite state machine that determines when certain operations must be performed. The controller must also interact with the host processor for synchronization. The controller is produced directly from the schedule developed in step 1.F.

Step 5: Incrementally Map Hardware Modules to FPGA

During this step, the designer incrementally maps modules to the FPGA. Here, the designer starts with the controller and ensures that the host can access it, and that the controller synchronizes as planned. Then, the designer should map a few other components, ensuring the host and any previously mapped components can access them. This process is repeated until either the entire hardware section is mapped or the available FPGA resources are exhausted. The designer may need to repartition the input program and redesign the controller during this step.

Step 6: Design Software Interface to Hardware

The final step is to write the software interface that determines how the host program will access the hardware board. This interaction includes: initializing the board, loading the hardware image onto the board, loading inputs to the board, retrieving outputs from the board, closing the board, and any synchronizations and handshaking required. Often, there are software APIs shipped with the FPGA board that ease the interaction with the board. These APIs are often object libraries, with the source code inaccessible to the end user. The designer must learn how to use these libraries. The designer must also determine where in the program the interactions will take place, and must implement the function calls accordingly.

Chapter 4

Overview of Major Components for Lego RC-RAS Module

A key portion of the design methodology presented in Chapter 3 has been automated. The segments we chose to automate are steps 1.F and 1.G, the resource allocation and scheduling (RAS) portion of the design methodology. The RAS module can be considered a module generation component that is capable of generating hardware modules for a given set of nested loops. RAS can be plugged into a driver program that determines the space and time requirements for the loop nest.

Here, we present the major components needed in this automation. First, we discuss the use of flow analysis techniques in Section 4.1. We then present a brief overview of the Lego Compiler. We discuss the software pipelining technique in Section 4.3. Resource allocation strategies are presented in Section 4.4. Transformations that allow parallelism are presented in Section 4.5.

4.1 Flow and Dependence Analysis

RAS requires a knowledge of the flow of operands throughout the program, and how it uses the available resources. This information is represented to RAS in the form of flow graphs. This information can be obtained by performing flow analysis on the

input program. What follows is an overview of how this information is gathered in standard compilers.

During the flow analysis phase, a global understanding of the program, and its uses of available hardware resources, is developed [75]. Programs read by compilers are initially a sequence of characters, which are then turned into a sequence of tokens by the lexical analyzer. The parser then develops some further level of syntactic structure from the set of tokens. This can be in the form of some intermediate code such as a syntax tree or directed acyclic graph. This result still gives little information about what a program does or how it does it. Flow analysis is used to develop this understanding. Control-flow analysis is used to discover the hierarchical flow of control within each procedure or function. Data flow analysis is used to determine information about the use of data. Dependence analysis is used to determine the possible ordering of instructions to exploit optimizations such as parallelization and redundant load/store removal.

4.1.1 Control-Flow Analysis

Control-flow analysis is used to discover the control structure in a program. As an example, consider the simple vector addition program shown in Figure 4.1. By examining the source code, one can determine that it contains an if-then construct, as well as a loop. However, this structure may not be as obvious in the intermediate code. In addition, there are many ways to represent the program, as can be seen by the assembly code representation shown in Figure 4.2. Therefore, the methods used in control-flow analysis do prove to be useful.

A flowchart, or flow graph, is a graphical representation of a sequence of operations or computations. The nodes of the flowchart represent the operations, while the edges represent the flow of control. Figure 4.3 shows the flowchart for the vector addition programs.

```
vector vector_add(vector A, vector B, int size_a, int size_b){
   int i = 0;
   if (size_a != size_b){
        printf("Error");
        return A;
   }
   else
   {
       for(i = 0; i < size_a; i++){
             A[i] = A[i] + B[i];
             return A;
        }
   }
}
```

Figure 4.1: Example high-level language implementation of vector addition.

```
        Receive A,R1          ; Start of Vector A
        Receive B,R2          ; Start of Vector B
        Receive size_A, R3    ;Size of Vector A
        Receive size_B,R4     ;Size of Vector B
        MOVI R5,0             ; Initialize i
        SUB R4,R3,R6          ; Compare sizes
        BEQ   Loop            ; They're equal
        Output "Error"        ; Print error
        RET                   ; Return
Loop:   Sub R5,R4,R6
        BEQ End
        LOAD R7, (R1+R5)   ;Load A[i]
        LOAD R8, (R2,R5)   ;Load B[i]
        ADD R7,R8,R7       ;Add A[i] to B[i]
        STORE R7, (R1+R5) ;Store A[i]
        ADDI R5,1          ; Increment i
        BRA Loop
End:    Ret
```

Figure 4.2: Example assembly code implementation of vector addition.

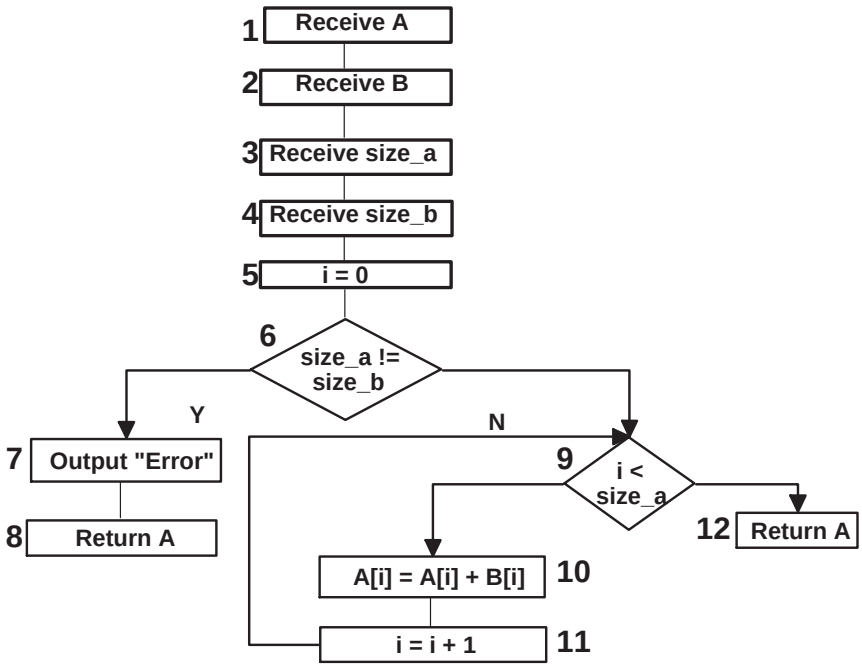

Figure 4.3: Example flowchart for vector addition.

A basic block is a segment of code in which the flow of control can enter only at the beginning and can only leave at the end, without possibility of halting or branching (except possibly at the end). To illustrate this, examine the vector addition flowchart shown in Figure 4.3. Blocks 1 through 6 represent one basic block, because this sequence of code can only be entered through block 1, and can only be exited through block 6. We'll call this basic block BB1. BB2 can be formed by combining blocks 7 and 8. BB3 consists of block 9 and BB4 consists of blocks 10 and 11. BB5 consists of block 12.

A simple approach to control-flow analysis is to discover loops in a program, and mark them for optimization. Loops usually represent code segments that are executed frequently. Since the largest potential improvement for the least amount of effort would be to improve the frequently used portions of the program [76], loops are generally prime candidates for optimization.

To determine loops in a program's flow graph, we first determine the set of dominator nodes. A block j is dominated by block i, denoted as $i\ dom\ j$, if all flows of control from the entry to block j go through block i. A node n post-dominates node m if all flows of control from m to the exit must go through node n. A basic approach to finding dominators in a flow graph uses the idea that node i dominates node j if and only if a.) $i = j$, b.) i is the unique immediate predecessor to j, or c.) j has multiple immediate predecessors such that all are dominated by i. A loop is found by identifying an edge whose head dominates its tail. The reader is referred to [75] for more uses and techniques on control-flow analysis.

4.1.2 Data-Flow Analysis

While control-flow analysis attempts to find control structures in a program's flowchart representation, data-flow analysis is used to determine how a procedure, or some other segment of code, manipulates its data. For example, the compiler may need to know that a definition, an assignment of a value to some variable, is valid later in the program. While this is obvious by examining the source code, it is much more difficult to determine when examining the intermediate code. In addition, when there are several loops or function calls, some possibly nested, it may not be as obvious. Therefore, the use of data-flow analysis does prove useful.

These analysis techniques provide insight into the sort of components needed in the system. The control-flow graph can be used to generate a behavioral description of the control unit, while the data-flow graph can be used to provide a behavioral description of the data unit.

4.1.3 Dependence Analysis

Dependence analysis is a tool used in instruction scheduling for determining parallelism and vectorization. It is used to determine the order of operations for a segment

of code that must be followed for correct execution of the program. This includes determining whether any two registers or memory references access overlapping areas of memory or share the same resource. This analysis enables the transformations of loops, which is needed for data-cache optimizations and for automatic vectorization and parallelization.

There are two main kinds of dependences, control and data. A control dependence exists between two instructions, S1 and S2, if the execution of instruction S2 is dependent upon the result of instruction S1. A data dependence exists when two instructions require the use of the same storage location for reading or writing. Given two instructions, S1 and S2, with S1 preceding S2, there are four types of data dependence:

1. Flow or True dependence: S1 writes to a storage location, and S2 reads from the storage location.

2. Anti-dependence: S1 reads from a storage location, and S2 writes to the storage location.

3. Output dependence: S1 and S2 write to the same storage location.

4. Input dependence: S1 and S2 read from the same storage location.

A dependence graph is used to represent a set of data dependencies. These graphs consist of a set of vertices, which represent operations, connected by edges, which represent the dependence between them. The edges are labeled with the type of dependence, as well as the number of cycles that must elapse between the two operations (latency).

4.1.4 Dependence Analysis for Loops

RAS focuses on developing modules for a set of nested loops. Therefore, it leverages some of the transformations available for optimizing loops. With a set of loop nests, we concern ourselves with the dependencies that occur due to the repetition of statements, as well as due to the ordering of the instructions. These dependencies are among operations involved with array variables inside of loops. We are interested only in loops that are in canonical form, meaning the subscript value goes from 1 to some value, n, by 1s. Figure 4.4 contains an example of a loop nest, and the corresponding iteration space. An iteration space comprises a finite discrete Cartesian space, and is defined by nested iterative loops [77].

Figure 4.4 shows an example of a rectangular iteration space. The points of the space are visited from left to right and from the top down. This is referred to as the iteration space traversal of the loop nest. The corresponding nested iterative loop is also shown. Note that the loop limits do not depend on the indices of other loops in the nest. The applications we plan to implement are required to have similar iteration spaces.

Careful inspection of the loop nest shown in Figure 4.4 shows that one can create a two element index vector of the subscript variables i and j as $\langle i, j \rangle$, which represents their corresponding values at a given instance in time. For example, the index vector $\langle 5, 4 \rangle$ represents the instance in time when $i = 5$ and $j = 4$. In general, there are m loops in the loop nest. The corresponding iteration space is the m-dimensional polyhedron, with index vector $\langle i_1, i_2, \cdots, i_m \rangle$, where $i_k, 1 \leq k \leq m$ are the subscripts of the loop nests, going from the outermost loop in to the innermost loop.

Given two index vectors, we refer to their ordering as the lexicographic ordering, with the "$\prec$" symbol. For example,

$$\langle i_{1:1}, i_{2:1}, \cdots, i_{m:1} \rangle \prec \langle i_{1:2}, i_{2:2}, \cdots, i_{m:2} \rangle \tag{4.1}$$

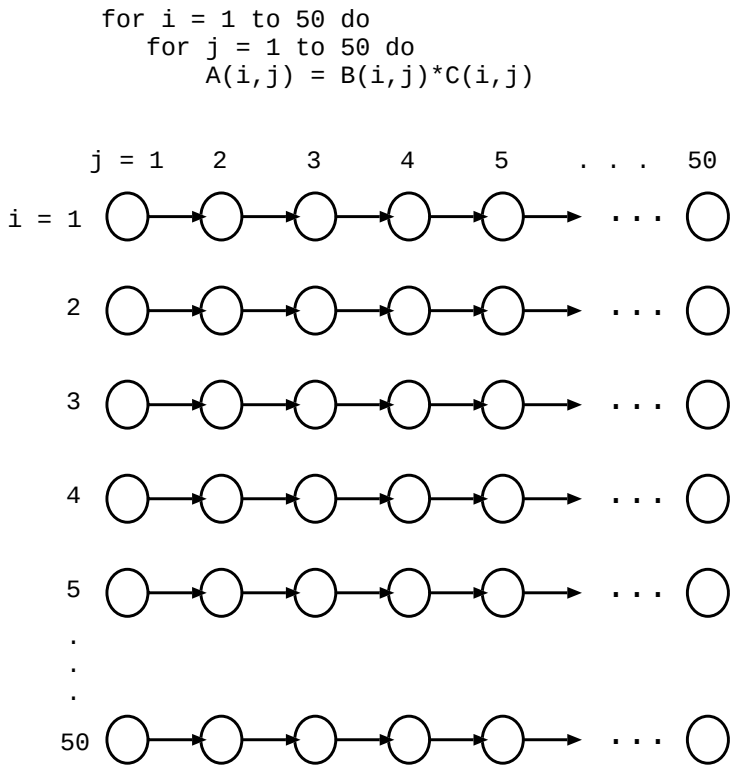

Figure 4.4: Sample nested loop and corresponding iteration space.

if and only if $\exists k, 1 \leq k \leq m$, such that $i_{1:1} = i_{1:2}$, $\cdots i_{k-1:1} = i_{k-1:2}$ and $i_{k:1} < i_{k:2}$. The lexicographic ordering corresponds to the relationship between when the two iterations represented by the given index vectors are visited in time. For example, two index vectors from Figure 4.4 would be $\vec{v_1} = \langle 2, 3 \rangle$ and $\vec{v_2} = \langle 2, 4 \rangle$. Index vector $\vec{v_1}$ is lexicographically less than $\vec{v_2}$, written $\vec{v_1} \prec \vec{v_2}$, because the first non equal indices of the vectors (the second one) is lower for $\vec{v_1}$ than for $\vec{v_2}$. This means that the iteration with $i = 2$, $j = 3$ is performed before the iteration with $i = 2$, $j = 4$.

Dependencies in loops are often expressed as a set of vectors, similar to the index vectors. The *distance* vector for an m-dimensional loop nest is the vector $\vec{dis} = \langle dis_1, dis_2, \cdots dis_m \rangle$. The values in the distance vector indicate that the iteration with index vector $\vec{i2} = \vec{i1} + \vec{dis} = \langle i_1 + dis_1, i_2 + dis_2, \cdots i_m + dis_m \rangle$ depends on the iteration with index vector $\vec{i1} = \langle i_1, i_2, \cdots i_m \rangle$.

To illustrate the use of dependence vectors, consider the following example, taken

from [75]:

```
   For i = 1 to 4 DO
      For j = 1 to 4 DO
S1       a[i,j] = b[i,j] + c[i,j]
S2       b[i,j] = a[i,j-1] + d[i+1,j]
      End For
   End For
```

There are two distance vectors associated with statements S1 and S2: $\vec{dis}_1 = \langle 0, 1 \rangle$, and $\vec{dis}_2 = \langle 0, 0 \rangle$. The distance vector, $\vec{dis}_1 = \langle 0, 1 \rangle$ represents the writing of an element of the array a in statement S1, and the reading of that element in statement S2 during the following iteration of the j loop. Thus, the iteration with index vector $\vec{v_2} = \langle 3, 2 \rangle$ depends on the iteration with index vector $\vec{v_1} = \langle 3, 1 \rangle$, since $\vec{v_2} = \vec{v_1} + \vec{dis}_1$. To see this, when $i = 3$ and $j = 1$, there is a write to element $a[3][1]$ in statement S1. When $i = 3$ and $j = 2$, there is a read of element $a[3][1]$ in statement S2. Similarly, the distance vector $\vec{dis}_2 = \langle 0, 0 \rangle$ represents the reading of an element of the array b in statement S1 and the writing to the same element of the array b in statement S2 during the same iteration.

A distance vector is associated with a given pair of statements inside the loop nest. The *direction* vector approximates one or more distance vectors. Each element of the direction vector indicates the sign of the corresponding element in the distance vector. The elements of the direction vector can be $=, <, >$ or $*$ to indicate the corresponding distance vector's element is in the range $[0, 0], [1, \infty], [-\infty, -1]$ or $[-\infty, \infty]$, respectively. The symbols $\neq, \leq$, and $\geq$ are used to represent the union of two sets. For example, the direction vector associated with vectors $\vec{dis}_1$ and $\vec{dis}_2$ above are $\vec{dir}_1$ equals $\langle =, < \rangle$ and $\vec{dir}_2$ equals $\langle =, = \rangle$, respectively. These vectors can

be combined into one vector by taking the union, resulting in the direction vector $\vec{dir}$ equals $\langle =, \leq \rangle$.

4.2 The Lego Compiler

RAS leverages some of the capabilities of the Lego Compiler, a part of a suite of tools developed by the Tinker Research Group at North Carolina State University. Lego is used to facilitate the development of a very long instruction word (VLIW) test bed for general purpose and embedded applications, one of the major research goals of the group. In this section, we discuss the language Lego recognizes. We then give a brief overview of the Lego components that can be utilized to produce the input for RAS.

4.2.1 Rebel Code

Rebel is a format similar to assembly language, designed for the Hewlett Packard PlayDoh architecture. In addition to machine level instructions, Rebel contains constructs that allow the programmer to extract other information from a program, including:

- Regions representing different levels of a program's structure.
- Source and destination operand type.
- Operation edge dependence type.
- Scheduled time for operation.

There are three primary regions represented in Rebel:

1. *Basic Block:* A region representing a set of operations with one entry point and one exit point (see Section 4.1).

2. *Procedure:* A region representing a set of basic blocks.

3. *Module:* The top level region, representing a group of procedures.

Operations typically contain a source and destination operand. The dependence type for edges between operations can be either flow, anti, or output.

The Tinker Group currently has tools for compiling C programs to Rebel. Therefore, RAS supports programs written in C. Other programming languages can be supported by developing a front-end to translate the given program into Rebel. The RAS module could be considered a back-end tool for compiling to reconfigurable architectures. This is a scheme typically used in compilers [76].

4.2.2 Lego Components Needed by RAS

Lego consists of a suite of C++ libraries that a developer can use and modify for implementing a variety of research projects. A RAS user will primarily utilize the following:

- Rebel reading/writing procedures.
- Region identification procedures.
- Data-flow graph creation procedures.

Before RAS can begin the resource allocation and scheduling phase, it must identify the control and data dependences between the operations in a given basic block. Standard Lego procedures can obtain this information for the basic blocks in the loop nest. This information is used to initialize RAS internal data structures before the hardware mapping can begin.

4.3 Software Pipelining: the RAS Scheduling Method

In this section, we give an overview of the concepts upon which we base the RAS scheduler. Details of the implementation are discussed in Chapter 5.

Due to the fact that programs spend most of their execution time in loops [75], the RAS scheduler uses software pipelining [70, 74], a scheduling process geared towards improving the execution time of loops. Software pipelining is a method for obtaining fine grain parallelism in regular loops, such as "for loops". Software pipelining restructures loops so that operations from various iterations are overlapped in time.

Figure 4.5, modified from [78], shows an example of a typical loop. Figure 4.5 (a) shows the loop in the C high-level language. Figure 4.5 (b) shows the corresponding assembly language code. The data dependence graph is shown in Figure 4.6. The labels on the edges represent the dependence type, iteration distance, and latency, respectively. For example, the label of $F, 0, 1$ from the LD instruction to the ADD instruction means there is a flow dependence, with an iteration distance of 0, and a latency of 1.

With normal scheduling techniques, the main instructions in the loop would take 4 cycles per iteration, assuming a 1 cycle latency for each instruction. Thus, n iterations would take approximately $4 * n$ cycles. With software pipelining, the iterations are overlapped, as in Figure 4.7. Here, the first iteration begins in cycle 1. The second iteration begins in cycle 2, the third in cycle 3, etc. This is contrasted with normal scheduling algorithms that would start the second iteration in cycle 5, the third in cycle 9, etc. Thus, n iterations would take approximately $n+3$ cycles (assuming there are enough resources to simultaneously execute the given instructions). For example, 4 iterations would take 7 cycles, assuming we have adequate resources.

The instructions for a given iteration are partitioned into pieces that execute in parallel with instructions from other iterations. Thus, they can be thought of as a

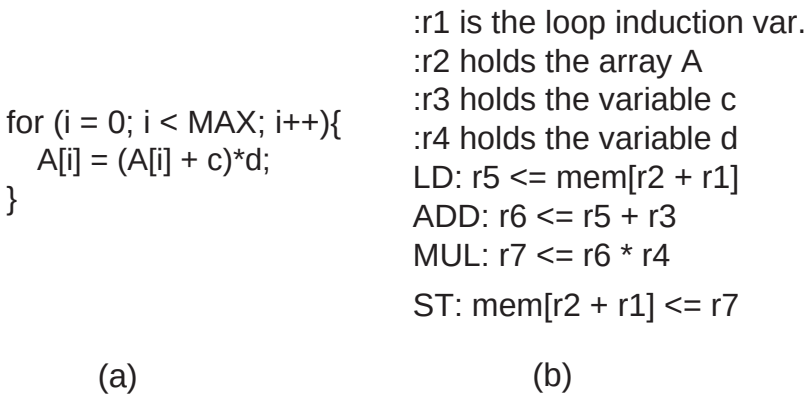

Figure 4.5: Example program loop. (a) C code. (b) Assembly code, without loop control instructions.

pipeline. With software pipelining, a new iteration is started after a given number of cycles. This is known as the *Initiation Interval* (II). Generally, one iteration finishes every II cycles. This is known as the *throughput*. The number of cycles required to finish the first iteration is known as the *pipeline latency*. Given N stages in a pipeline, the latency is equal to N. After several iterations, the program reaches a steady state in which a repeating pattern of executed instructions develops. These instructions are known as the *kernel*. The set of instructions that occur before reaching the kernel, which fill the pipeline, is known as the *prologue*. The *epilogue* is the set of instructions that empty the pipeline. The total execution time is $Te = l + (n \times II) - 1$. where l is the latency, and n is the number of iterations.

In Figure 4.7, $II = 1$. The throughput is one, which means one iteration completes every cycle. The latency is 4 cycles, since it takes 4 cycles for the first iteration to

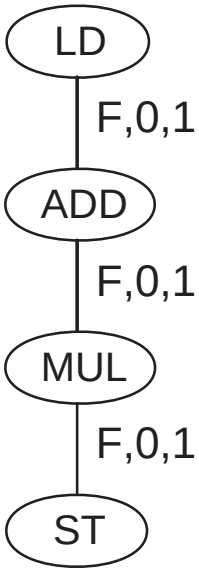

Figure 4.6: Data dependence graph for example program.

complete. Hence, for 5 iterations, the total execution time is $Te = 4 + (5 \times 1) - 1 = 8$. Here, the kernel instructions occur in cycles 4 and 5. The prologue and epilogue occur during cycles 1-3 and 6-8, respectively.

There are primarily two categories for software pipelining: modulo scheduling and kernel recognition. The technique used by our tool uses modulo scheduling, due to the fact there is no loop unrolling, where the loop is expanded for multiple iterations, involved. As a result, here, we only discuss modulo scheduling. The reader is referred to [74] and [79] for more information on other software pipelining scheduling techniques.

Modulo scheduling places operations so that the cyclic schedule is valid. Hence, when an operation O is scheduled, it cannot conflict with other operations in the current or subsequent iterations. The *flat schedule* is the schedule of the original iteration [74]. The locations are denoted $F_1, F_2, \cdots, F_f$, where f is the number of

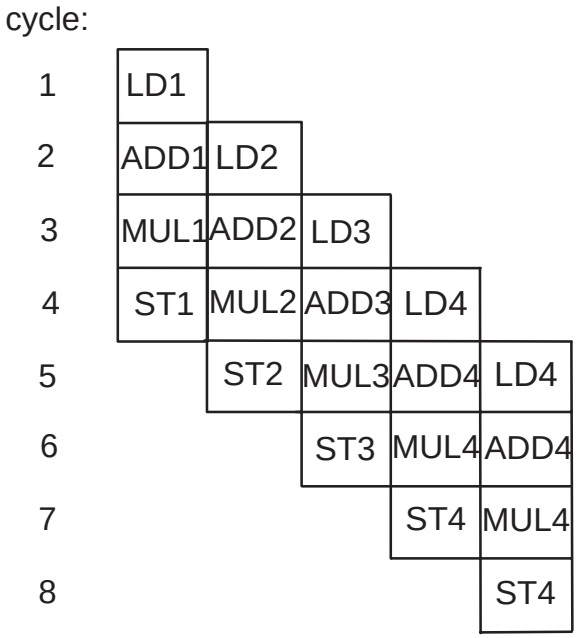

Figure 4.7: Overlapping loop iterations for example program, without loop control instructions.

cycles needed for the flat schedule. The loop is pipelined by overlapping successive iterations. Operations that have the same value modulo II execute simultaneously. For example, if $II = 2$, and $f = 6$, operations scheduled in F_1, F_3, and F_5 execute simultaneously ($F_i : i \ mod \ 2 = 1$). Likewise, operations scheduled in F_2, F_4, and F_6 execute simultaneously ($F_i : i \ mod \ 2 = 0$). Thus, operations with identical modulo locations in the flat schedule execute simultaneously, hence the name of the algorithm.

While constructing the schedule, operations are placed one at a time. The operations with higher placement difficulty are scheduled before the ones that are less difficult. The operation being scheduled must be valid in the schedule across the set of iterations. This means that an operation from all iterations is scheduled with previously scheduled operations from all iterations. This differs from other techniques that schedule an operation from a given iteration with previously scheduled operations

from particular iterations.

When constructing the flat schedule, we must account for any possible conflicts. Conflicts can occur due to resource constraints or dependence constraints. The problem is in the placement of operations, since successive iterations will have the same schedule. We can ensure that there are no conflicts by starting successive iterations at a valid offset, or II. Obviously, the initiation interval must be determined before scheduling. Because determining the optimal II is NP-complete [80, 81], we begin with an initial guess for II, i.e. 1, and increment until a solution is found.

There are two factors that constrain the value of a valid II: resources and dependences [74]. The number of resources constrain II because operations from successive iterations cannot be scheduled until the required resource is available. Since iterations are started every II cycles, there must be at least enough resources to meet the demands of one iteration. Thus, the resource constrained II, or $ResII$, is [81]:

$$ResII = max_k \lceil \frac{\sum^{v \in V, 0 \leq i \leq l(v)} u(i,k)}{R(k)} \rceil \tag{4.2}$$

where V is the set of nodes representing operations; $l(v)$ is the length, in cycles, of vertex v of the graph; $u(i,k)$ is the number of k resources used in step i; and $R(k)$ is the total number of k resources available. The pseudo-code for calculating $ResII$ is:

Algorithm: Computation of Resource Constrained Initiation Interval

Input: $1 \times K$ `vector` R `representing number of resources available` $N \times L \times K$ `matrix` V `representing the number of` K `resources used in the` L `steps of the` N `nodes`

Output: `Resource constrained II,` $ResII$

```
ResII = 1
For k = 1 to K DO
```

```
  Resources = 0;
  For n = 1 to N DO
    For l = 1 to L DO
      Resources = Resources + V[n,l,k]
    End For
  End For
  If ResII < ⌈Resources/R(k)⌉ then
    ResII = ⌈Resources/R(k)⌉
  End If
End For
```

Cyclic dependencies also constrain the II. A cycle is a path in the dependence graph, with at least one edge, from a node to itself [82]. A cyclic dependency is a dependence between an operation and another operation dis iterations later. This is also a loop-carried dependence. Let lat_{cycle} and dis_{cycle} be the total of the latencies and distances, respectively, of the edges in the cycle. The amount of time that elapses between an operation O and the same operation dis_{cycle} iterations later is $II * dis_{cycle}$. We must ensure that II is large enough so that the latter operation does not begin prematurely. Thus, it is required that $II * dis_{cycle} \geq lat_{cycle}$ for all cycles in the loop. Therefore, the dependence constrained II, or $DepII$ is [74]:

$$DepII = max_{(\forall\ cycles)} \lceil \frac{lat_{cycle}}{dis_{cycle}} \rceil \tag{4.3}$$

There are several algorithms for finding $DepII$ [74]. RAS uses the iterative shortest path technique [83]. This technique computes the $DepII$ using the following steps:

1. Estimate $DepII$.

2. Compute cost matrix for dependency graph.

3. Compute the transitive closure of the dependency graph for the current estimate of $DepII$.

4. Determine if estimate of $DepII$ is valid.

5. If $DepII$ is not valid, increase estimate and repeat steps 2-4. Otherwise, stop.

The entry $\langle i, j \rangle$ of the cost matrix indicates the number of cycles node j must follow node i in the flat schedule. We compute these values from the dependency graph using the formula, $cost[i][j] = lat_{i,j} - II \times dis_{i,j}$.

The *transitive closure* of a graph, G, is a graph, G_{tc}, with edges, E_{tc}, such that there is an edge $\langle i, j \rangle$ in E_{tc} if there is a path from nodes i and j in G. RAS uses a variant of Floyd's algorithm [74] for computing the transitive closure.

To determine whether a given $DepII$ is valid, we must examine the values of the cost matrix. The diagonals represent the cost of a cycle. If the value of a diagonal is positive, it would mean that $cost[i][i] > 0$, or $lat_{cycle} > II * dis_{cycle}$, which violates the dependency restraints for cycles. Thus, a given $DepII$ is valid only if the diagonal entries of the closure matrix are all non-positive. If any of the diagonal entries are positive, we must increase the value of $DepII$, and repeat steps 2 - 5 above.

The pseudo-code for calculating the dependence constrained initiation interval, $DepII$ is:

Algorithm: Computation of Dependence Constrained Initiation Interval

Input: $n \times n$ `matrix` G `representing directed data flow graph`

Output: `Dependence constrained II,` $DepII$

$Valid \leftarrow false$

$DepII \leftarrow 0$

```
While Valid is false Do
  DepII ← DepII + 1
  Cost =Compute Cost Matrix(G,DepII)
  Closure = Floyds(Cost)
  Valid = CheckDiagonal(Closure)
End While
```

The pseudo-code for computing the cost matrix is:

Algorithm: Computation of Cost Matrix

Input: $n \times n$ matrix representing directed graph G, and initiation interval, II

Output: Matrix G' whose i, j value is $lat_{ij} - dis_{ij} * II$, where i is the source node, and j is the destination

```
For i = 1 to n DO
  For j = 1 to n DO
    If (∃ edge_ij)
      G'[i][j] = lat_ij − dis_ij * II
    Else
      G'[i][j] = −∞
    End If
  End For
End For
```

The pseudo-code for Floyd's method is:

Algorithm: Floyd's Algorithm

Input: $n \times n$ matrix representing cost matrix, G

Output: Closure of directed graph

```
For k = 1 to n DO
  For i = 1 to n DO
    If (G[i][k] > -∞)
      For j = 1 to n DO
        t = G[i][k] + G[k][j]
        If(t > G[i][j])
          G[i][j] = t
        End If
      End For
    End If
  End For
End For
```

After computing the resource constrained initiation interval and the dependence constrained initiation interval, we can compute the minimum II, which is simply the maximum of $ResII$ and $DepII$. The II is then used to schedule the operands.

4.4 RAS Resource Allocation Method

In this section, we give an overview of the concepts upon which we base the RAS resource allocator. Details of the implementation are discussed in Chapter 5.

RAS uses architectural synthesis techniques to allocate the resources for the given loop nest. Architectural synthesis constructs a macroscopic implementation of a

digital circuit. The input for this method is a behavioral description, which can be obtained from data flow graphs. The outputs are a data path and a control unit.

Architectural optimization strategies attempt to optimize three objectives [84]:

1. *Latency*: The number of clock-cycles needed for the circuit.

2. *Cycle-time*: The time period for a cycle.

3. *Area*: The amount of space needed for the circuit.

The minimization of the first two objectives is not required at this stage in the process. RAS minimizes the latency through the scheduling process (see Section 4.3). With R.C. systems, the cycle-time is typically determined based on the target R.C. board used. Therefore, the RAS resource allocation strategy focuses on the third objective.

The resources needed for the application are determined during the resource allocation phase. The resource allocation process follows the ideals of library binding [84], in that we have a library of logic components, and we bind the functionality of the given application to the components of this library.

The component library is a set of elements capable of performing arithmetic and logical tasks. Associated with each component in our library is its function, input/output ports, area, and delay. Resource allocation binds resources to operations in the data flow graph.

4.5 RAS Transformations for Further Enhancing Speedup for Loop Nests

In this section, we give an overview of the transformations supported by RAS for further enhancing the performance of loop nests. Performance can be enhanced by

exploiting temporal reuse and spatial reuse inherent in a given loop nest. Strip-mining, vectorization, and tiling methods can be used to take advantage of the reuse and parallelism in the loop nest, and are supported in RAS. Since RAS only supports these transformations and does not detect their legality, we discuss the benefits of these transformations when applied to R.C. systems. The reader is referred to several sources for more information on these techniques [77, 85, 86, 87, 88].

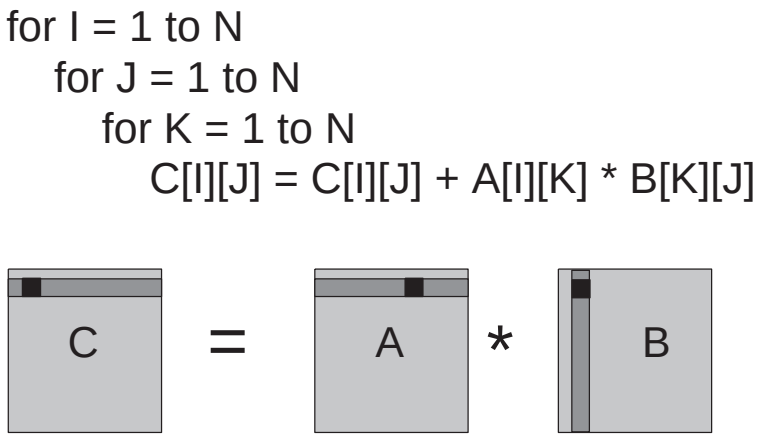

Figure 4.8: Example nested loop algorithm: matrix multiplication.

To illustrate the benefits of these transformations, consider the matrix multiplication algorithm shown in Figure 4.8. Each element of the **C** matrix is the dot product of the corresponding row of matrix **A** with the corresponding column of the matrix **B**.

Figure 4.9 shows the corresponding inner k loop in a format similar to Rebel, with $N = 100$. Instructions $I1$ through $I6$ perform element address calculations and

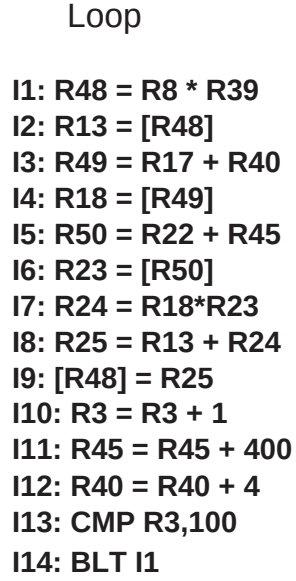

Figure 4.9: Inner k loop in assembly language format.

load the elements of the matrices into local registers. The dot product calculations are performed in instructions $I7$ and $I8$. Instruction $I9$ stores the result of the given **C** element in main memory. Instructions $I10$ through $I12$ update the iteration counter and the registers for calculating the address of the **A,B** values for the next iteration. Instructions $I13$ and $I14$ control when to cease loop execution. Assuming the following timing requirements for instructions:

- Loads and stores require 3 cycles;
- Register move instructions require 1 cycle;
- Integer additions and multiplications require 1 cycle;
- Floating point additions and multiplications require 4 cycles;
- Branch and compare instructions require 1 cycle;

each iteration requires 28 cycles. 100 iterations would require 2800 cycles. Removing the redundant load and store [75] of instructions $I2$ and $I9$ reduces the time for each iteration to 22 cycles. 100 iterations would require 2200 cycles. The i and j loops each execute 100 times. Thus, the total execution time for the inner k loop is 22 million cycles, ignoring the overhead in the i and j loops.

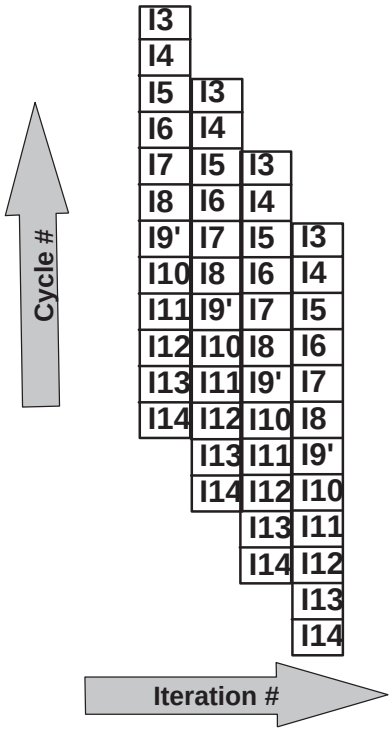

Figure 4.10: Pipelined schedule of inner k loop.

Overlapping iterations of the inner k loop, as is done in software pipelining, will lead to enhanced performance. Assuming pipelined units are used, Figure 4.10 shows the resulting schedule of the inner k loop. Here, instructions $I1$ and $I2$ are removed, and $I9'$ represents the corresponding copy of $FPR25$ to $FPR13$. Due to resource and dependency constraints, the minimum initiation interval is 2 cycles. The flat schedule has a latency of 22 cycles. Thus, 100 iterations of the inner k loop requires 221 cycles. The inner k loop is executed 10000 times. Therefore, the total execution

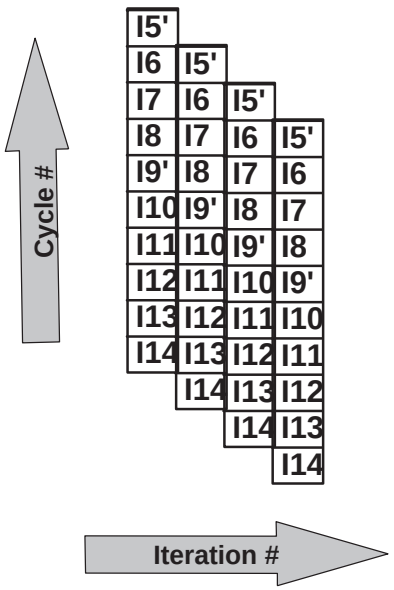

Figure 4.11: Pipelined schedule of inner k loop exploiting reuse.

time for the inner k loop is now 2.21 million cycles.

Since RAS focuses on loop nests, further enhancement of the performance can be obtained by reducing the number of instructions performed in each iteration and by reducing the total number of iterations required.

The number of instructions per iteration can be reduced by storing array operands that will be reused some time in the future. This is referred to as *temporal reuse*. The benefit is the reduction in the number of instructions since there are less memory accesses in the loop. The temporal reuse of scalars can be exploited when a memory access is invariant of a given loop. Tiling methods [77] can be used to exploit the reuse of entire arrays or portions of an array.

Temporal reuse can be exploited with the matrix multiplication example. Every iteration of the j loop requires loading a row of the matrix $\mathbf{A}$, and a column of the matrix $\mathbf{B}$. In addition, reuse occurs in the calculation of the address for the rows and

```
for I = 1 to N
   for JS = 1 to N by 2
      for J = JS to min(JS+1,N)
         for k = 1 to N
            C[I][J] = C[I][J] + A[I][K] * B[K][J]
```

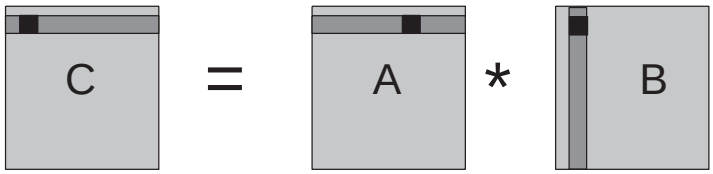

Figure 4.12: Strip-mined inner j loop.

columns. We can exploit this reuse by storing a row of the matrix **A**, and reducing the address calculations to one instruction. This results in the inner k schedule shown in Figure 4.11. Here, the instructions $I3$ and $I4$ have been removed. $I5'$ represents the new address calculation instruction. This reduces the memory accesses, removes the dependence cycle, and allows us to initiate a new iteration each cycle. As a result, 100 iterations of the inner k loop require 121 cycles. The total execution time for the inner k loop is now 1.21 million cycles.

The total execution time of the iterations can be further reduced by performing multiple iterations simultaneously. This can be achieved by exploiting any coarse-grained parallelism available in the loop nest. Vectorization, via strip-mining, and tiling techniques [85, 89] exploit this parallelism by executing several iterations simultaneously. Strip-mining techniques partition the iteration space of a loop into a set of iteration strips, traversed by two nested loops. The outer loop steps through the

```
for I = 1 to N
   for JS = 1 to N by 2
      for K = 1 to N
         C[I][JS] = C[I][JS] + A[I][K] * B[K][JS]
         C[I][JS+1] = C[I][JS+1] + A[I][K] * B[K][JS+1]
```

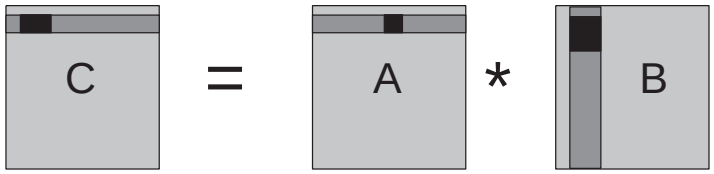

Figure 4.13: Vectorized inner j loop.

iteration strips, while the inner loop steps through the iterations. Figure 4.12 shows an example of strip-mining the inner j loop.

Vectorization techniques [89, 90] can be used for parallelization if the iterations are independent of each other. Figure 4.13 shows the result of vectorizing the inner j loop. Here, two iterations are essentially executed simultaneously on different processor nodes. Therefore, only half as many iterations of the inner j loop are executed on each node. As a result, the inner j loop is now executed 50 times. The inner k loop is now executed 5000 times on each node. The total execution time for the inner k loop is now .605 million cycles.

Applying strip-mining to a loop nest results in a set of tiles for the iteration space. Figure 4.14 shows the result of tiling the matrix multiplication loop nest. Here, the tiles are of size 2×2. If we have 4 processor nodes, and these tiles can execute independently, they can be executed simultaneously. Figure 4.15 shows the results

```
for IS = 1 to N by 2
   for I = IS to min(IS+1,N)
      for JS = 1 to N by 2
         for J = JS to min(JS+1,N)
            for k = 1 to N
               C[I][J] = C[I][J] + A[I][K] * B[K][J]
```

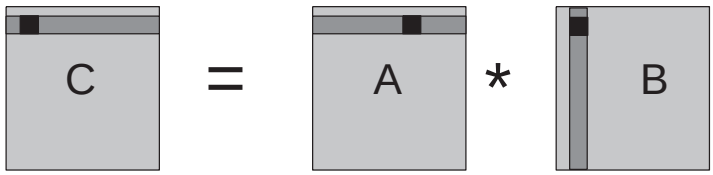

Figure 4.14: Tiled i and inner j loops.

of tiling and parallelizing the two outer loops. Here, two iterations of the i loop are executed with two iterations of the j loop. The number of iterations of the inner j loop is the same as for the vectorized loop in Figure 4.13. However, there are only 50 iterations of the IS loop (the new i loop). Thus, the inner k loop is now executed 2500 times. The total execution time for the inner k loop is 302,500 cycles.

RAS uses the above components to implement a portion of the design methodology presented in Chapter 3. RAS can be considered a module generation tool, which can be called by a driver program that partitions the overall application. Since we are primarily concerned with computationally intense algorithms, RAS focuses on enhancing the performance of loop nests. RAS leverages the development of several compilation techniques and circuit synthesis techniques to achieve this goal. The following chapter discusses the implementation of RAS.

```
for IS = 1 to N by 2
   for JS = 1 to N by 2
      for K = 1 to N
         C[IS][JS] = C[IS][JS] + A[IS][K] * B[K][JS]
         C[IS][JS+1] = C[IS][JS+1] + A[IS][K] * B[K][JS+1]
         C[IS+1][JS] = C[IS+1][JS] + A[IS+1][K] * B[K][JS]
         C[IS+1][JS+1] = C[IS+1][JS+1] + A[IS+1][K] * B[K][JS+1]
```

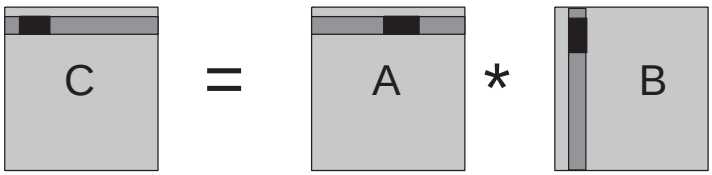

Figure 4.15: Parallel execution of loop tiles.

Chapter 5

RAS Implementation

This chapter presents the details of the implementation of the Resource Allocation and Scheduling module that is designed to be an extension to the Lego Compiler [91]. The chapter begins with an overview of the system inputs and outputs. Next, an overview of the scheduling and resource allocation methodology is presented. Section 5.3 presents details of the algorithms used in the scheduling component. Section 5.4 presents details of the algorithms for resource allocation and connection. The chapter concludes with a discussion of how the transformations discussed in Section 4.5 are incorporated. The matrix multiplication algorithm discussed in Section 4.5 will be used as a running example for this chapter.

5.1 Overview of RAS System

This section presents an overview of the RAS system. Figure 5.1 illustrates the inputs and outputs of RAS. RAS requires the following inputs:

1. Set of data-flow graphs representing the basic blocks of nested loops.
2. Set of available resources, including the following information:

   Function of corresponding resource.

   Latency for corresponding resource.

Size of corresponding resource.

3. Available space for implementation.

RAS outputs a schedule and set of resources required to implement the given algorithm.

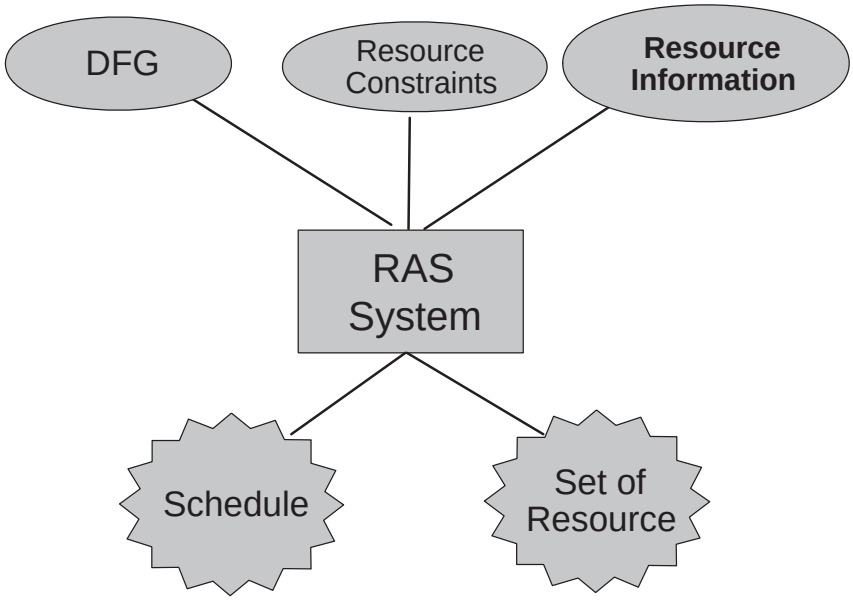

Figure 5.1: Overview of RAS system architecture.

RAS represents the set of data-flow graphs with an $n \times n$ matrix of 3-tuples, where n is the number of nodes in the data-flow graph. The i, j entry in the matrix represents the edge from node i to node j. Each tuple of the matrix contains the latency, distance, and dependence type for the corresponding edge.

Figure 5.2 shows the data-flow graph for the assembly language representation of the inner k loop shown in Figure 4.9 after the redundant load/store removal. The

graph contains 12 nodes. Associated with each node is an operation, a set of input edges and a set of output edges.

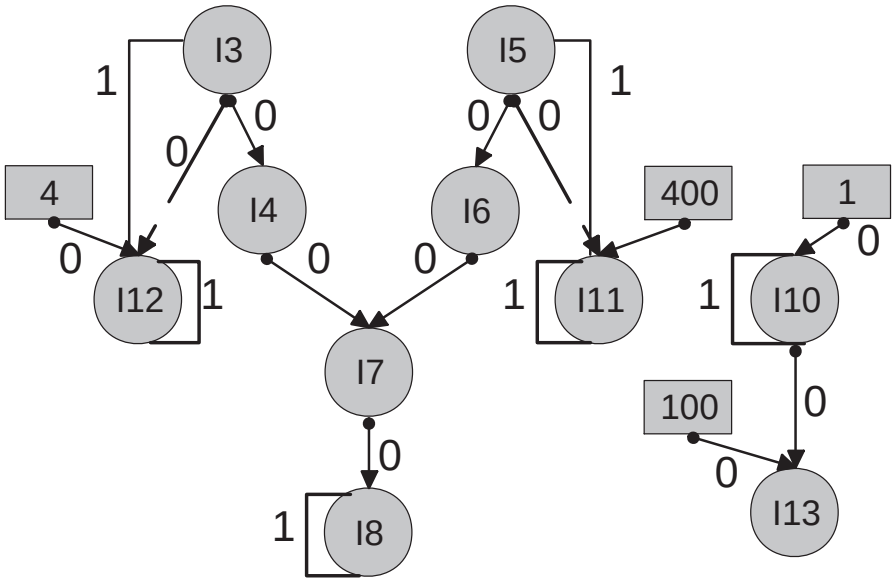

Figure 5.2: Data-flow graph of inner k loop for instruction level format.

RAS also requires a set of resources that are capable of implementing the operations contained in the data-flow graph. Associated with each resource is the function it implements, the latency, the area it requires, an identification number, the ID of the operations mapped to it, the ID of the resources connected to its inputs, and the ID of the resources connected to its output.

Figure 5.3 shows the set of resources available for implementing the matrix multiplication algorithm. The resources have a number associated with their type. Resource 0 is the integer adder; resource 1 is the floating point adder; resource 2 is the floating point multiplier; resource 3 is the memory access module; resource 4 is the branch unit. Initially, the resources have not been allocated, and are not associated

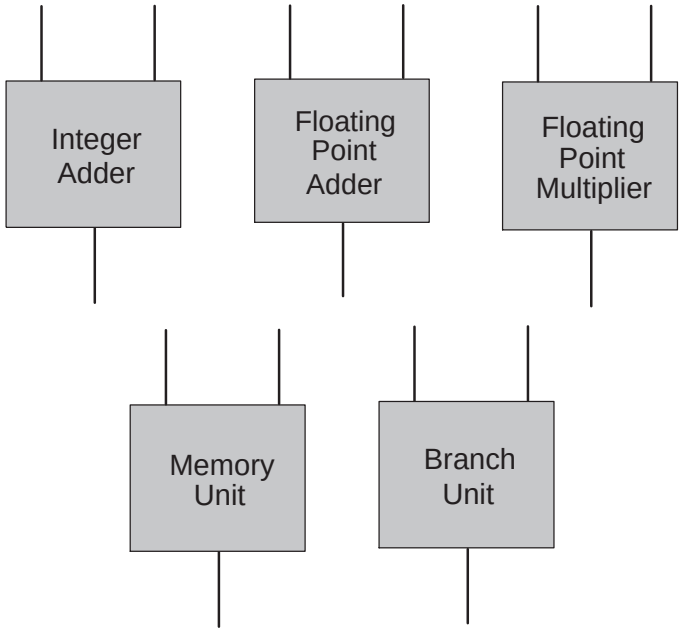

Figure 5.3: Resource set for inner k loop.

with any operations.

5.2 Overview of Resource Allocation and Scheduling

After the data-flow graph for the loop nest has been constructed, RAS allocates resources according to user constraints, and schedule the nodes to of the data-flow graph to the resources. RAS first schedules and allocates resources for the innermost loop of the loop nest. Outer loops are subsequently processed in order from the innermost loop outward. This strategy is used since the innermost loop is likely to require the most iterations and, therefore, consume the most amount of computer time. The pseudo-code for resource allocation and scheduling from the inner loop outward is:

Algorithm: Resource Allocate and Schedule Loop Nest

Input: *Loop*, structure representing entire loop nest.

Output: Flag reflecting whether resource allocation and scheduling of loop nest was successful.

```
If Loop.inner is not NULL
  Resource Allocate and Schedule Loop Nest(Loop.inner)
End If
Reduce Inner Loop For Body(Loop.Body, Loop.Inner)
Success = Resource Allocate and Schedule Loop Body(Loop.Body)
Return Success
```

The *ReduceBodyForInnerLoop* procedure reduces the inner loop to a single node for the outer loop. Once a loop has been selected for processing, RAS begins the resource allocation and scheduling process by determining the number of strongly connected components in the data-flow graph. RAS then initializes the set of instructions to correspond with the set of input nodes. Next, RAS determines the number of resources for the algorithm. RAS then allocates the resources and schedules the nodes to the resources. RAS finishes by calculating the time cost and area cost. The pseudo-code for the overall resource allocation and scheduling process for the loop body is:

Algorithm: Resource Allocate and Schedule Loop Body

Input: *Body*, structure representing body of a loop.

Output: Flag reflecting whether resource allocation and scheduling of loop body was successful.

```
NumberOfComponents = Strongly Connected Component(Body.DependenceGraph,
```

```
  Body.NodeList)
Initialize Instructions(Body.InstructionList,Body.NodeList)
II = Computation of Dependence Constrained Initiation Interval(
Body.DependenceGraph)
TotalResources = Find Number of Resources Needed(ResourceNeeds,
  Body.NodeList,II)
TotalResources = Allocate Resources(ResourceNeeds,II,Constraints)
If TotalResources is 0
  Return 0
End If
II = Schedule It All(II, Body.DependenceGraph,Body.NodeList,
  ResourceList, Body.InstructionList,G,TotalResources)
MuxesNeeded = Determine Number of Muxes Needed(ResourceList,
  TotalResources)
Link Resources Together(InstructionList,DependenceGraph,MuxesNeeded
  MuxList,DeMuxLIst,II, ResourceList,TotalResources)
Body.TimeCost = Determine Time of Pipelined Loop(Body.InstructionList,
  Body.LoopCount,II)
Body.II = II
Body.AreaCost = Determine Area Used By Resources (ResourceList,
  TotalResources)
Return 1
```

A *strongly connected component* is a set of nodes, N, such that for every pair of vertices a and b in N, there is a path from a to b and vice versa [82]. The algorithm for determining the strongly connected components can be found in [82].

The instruction structure holds the start and end time for the corresponding node in the dependence graph. It is also used for linking the nodes to a resource. The $InitializeInstructions$ procedure simply fills in the instruction data structure based on the corresponding node.

After initializing the instructions, RAS determines the dependence constrained initiation interval, $DepII$. The pseudo-code for determining this was presented in Section 4.3. After determining the dependence constrained initiation interval, RAS determines how many resources to request.

The *FindNoResources* procedure is responsible for calculating the number of each type of resource to request. RAS computes the number of resources to request by reversing Equation 4.2, solving for $R(k)$, the number of resources of type k, and substituting the $DepII$ for the $ResII$:

$$R(k) = \lceil \frac{\sum^{v \in V, 0 \leq i \leq l(v)} u(i,k)}{DepII} \rceil \tag{5.1}$$

The number of resources for $R(k)$ is calculated by first counting the number of times it is used in the flat schedule. This result is divided by the $DepII$ to yield the number of resources of type k requested.

After determining the number of each type of resource needed, RAS attempts to allocate the requested resources. RAS first calculates the amount of space needed to satisfy the requests from *FindNoResources*. If there is not enough space, RAS determines the minimum amount of area needed to use the methodology. The methodology requires at least one resource unit for each type needed. If there is not enough area to meet these requirements, the program exits.

If the user area constraints are met with the minimum area requirements, RAS modifies the requested number of resources. This process begins with setting the number of resources for each type initially requested to 1. Additional resources of a

given type are requested based on the following priority function:

$$Priority(k) = \frac{R'(k)}{R(k)} \tag{5.2}$$

where $R(k)$ is the original requested number of resources of type k, and $R'(k)$ is the number reserved so far.

Once RAS has established that the number of requested resources, either from the original request or by reducing the request, meets area constraints, RAS allocates the requested resources. RAS iterates through the resource types and allocates the number of resources requested for each type. The allocation includes initializing some parameters associated with the resource. The pseudo-code for allocating the resources is:

Algorithm: Allocate Resources

Input: $ResourceNeeds$, $1 \times NumberOfResourceTypes$ `vector representing number of resources needed; II, Constraints`

Output: `Allocation and initialization of each resource requested.`

$ResourceNumber \leftarrow 0$

$Area \leftarrow 0$

`For` $i = 1$ `to` $NumberOfResourceTypes$ `DO`

  $Area \leftarrow ResourceNeeds[i] \times ResourceArea[i]$

`End For`

`If` $Area > Constraints$

  $Area =$ `Base Allocation Space Requirements(`$ResourceNeeds$`)`

  `If` $Area > Constraints$

    `Return Failed`

  `Else`

```
    Reduce Number Of Resources(ResourceNeeds,Constraints)
  End If
End If
For i = 1 to NumberOfResourceTypes DO
  For j = 1 to ResourceNeeds[i] DO
    Initialize Resource[ResourceNumber]
    Resource[ResourceNumber].ResNum ← ResourceNumber
    Resource[ResourceNumber].latency ← ResourceType[i].Latency
    Resource[ResourceNumber].area ← ResourceType[i].Area
    Resource[ResourceNumber].type ← ResourceType[i].Operation
    ResourceNumber ← ResourceNumber + 1
  End For
End For
Return ResourceNumber
```

Referring back to the data-flow graph of the inner k loop, shown in Figure 5.2, the solid and dashed lines represent flow- and anti-dependencies, respectively. The number next to the lines represent the dependence distance between the corresponding nodes. The labels inside the nodes correspond to the operation number in Figure 4.10. Instruction $I9'$ has been removed because, with RC components, the output of the adder can be routed to the inputs. Hence, there is no need for the copy instruction.

The strongly connected components with multiple nodes are: nodes $I3, I12$ and nodes $I5, I11$. The other strongly connected components are the remaining individual nodes. For our example, $DepII$ is 2. The number of integer adders, resource 0, used is 6. Thus, the number of integer adders allocated, $R(0)$, would be $\frac{6}{2}$ or 3. Similarly, $R(1)$, $R(2)$, $R(3)$, and $R(4)$ are 1.

5.3 Implementation of the RAS Scheduling Unit

After the resources have been allocated, the scheduler binds operations to the given resources and the resources are linked together. The scheduler requires the following inputs:

- The minimum initiation interval, $DepII$.
- The array of nodes, $NodeList$.
- The array of resources, $ResourceList$.
- The array of instructions, $InstrList$.
- The matrix of dependences, $DepMatrix$.
- The matrix of cycle costs, $CostMatrix$.
- The number of strongly connected components.

The scheduling algorithm begins by attempting to schedule operations with an initiation interval of $DepII$. First, the loop independent graph is created. Next, the strongly connected components of the graph are scheduled independently. Then, the strongly connected components are scheduled together. RAS then calculates the heights of all the nodes. A node that is ready for scheduling is then selected and scheduled. The upper and lower time bound for the remaining unscheduled nodes is computed, and another ready node is selected and scheduled. This process is repeated until all nodes have been scheduled. If the scheduling fails at any step during this process, the initiation interval is increased by one, the cost matrix is recomputed, the instructions, nodes and resources are reinitialized, and the process is repeated. The pseudo-code for the overall scheduling routine is:

Algorithm: Schedule It All

Input: $NodeList$, $G: n \times n$ matrix representing directed graph
II, $ResourceList$, $InstructionList$, $NumberOfResources$, $Cost: n \times n$ matrix representing costs.

Output: Initiation interval at which nodes were scheduled.

```
LIG = Compute Loop Independent Graph
For i = minII to minII + numberOfElements DO
  For j = 1 to numberOfComponents DO
    success = Schedule Connected Component(LIG, G, Resource, Cost, i, j)
    If success is false
      break
    End If
  End For
  If success is false
    continue
  End If
  For j = 1 to numberOfElements Do
    NodeList[j].lowerbound = Compute Initial Lower Bound(Cost, j)
    NodeList[j].upperbound = VERYBIGNUMBER
  End For
  Find Node Heights( InstructionList, NodeList, LIG)
  readynode = Get Ready Node(NodeList, LIG)
  While readynode > -1 Do
    success = Schedule Node(NodeList, readynode, Resource, minII)
    If success is false
      break
    End If
    Recompute Upper and Lower Bounds(NodeList, Cost, readynode)
```

```
    readynode = Get Ready Node(NodeList, LIG)
  End While
  If success is true
    break
  Cost =Compute Cost Matrix(G,i + 1)
  Reset Nodes Instructions Resources(InstructionList,NodeList,
    ResourceList,NumberOfResources,i + 1)
End For
If i equals (minII + numberOfElements)
  return -1
Else
  return i
End If
```

The loop independent graph is the dependence graph that would result if the instructions were not performed in a loop. Thus, there are no dependences with a distance above 0. Figure 5.4 shows the loop independent graph for the matrix multiplication example.

The algorithm for scheduling the connected components is similar to that for the overall graph. Again, the scheduling is for a given initiation interval. First, the initial upper and lower time bounds for the nodes are computed. Then, a ready node for the component is grabbed and scheduled. The upper and lower bounds are then recomputed, another ready node obtained, and scheduled. This process continues until all the nodes in the component are scheduled. The procedure reports to the overall scheduler whether it was able to schedule the components for the given initiation interval. If the scheduling of a component node fails, the overall scheduler will have to increase the initiation interval.

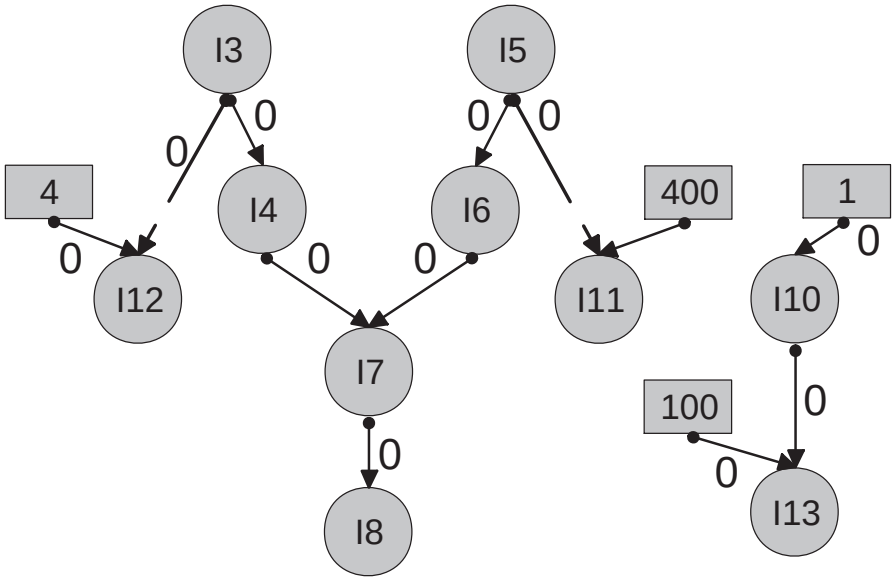

Figure 5.4: Loop independent data-flow graph of inner k loop.

After ensuring the strongly connected components can be scheduled with the given II, RAS schedules all the nodes together. RAS first determines the initial lower bound of the nodes using the cost matrix. The initial lower bound for node j is the maximum cycle cost to node j. Thus, $j_{lb} = max_{i \in G} cost^{II}(i, j)$.

RAS then computes the heights of all the nodes. The height of a node is the number of cycles from the node to the exit of the graph. RAS first puts a dummy sink node after all nodes with no successors. It then performs a breadth first search from the sink node to all other nodes. As each node is processed, the delay from the sink to the node is assigned as the corresponding node's height. If the node was previously assigned a height, the new height is compared with the old height. The node is given the larger height. The pseudo-code for the breadth first search algorithm can be found in [82].

RAS determines which node is ready for scheduling by iterating through the nodes in the graph. RAS first determines whether the current node is *Ready* or $NotReady$. If the given node, i, has already been scheduled, it is skipped. If the node i has not been scheduled, RAS examines whether the predecessors for node i have been scheduled. If they have, RAS marks the node *Ready* for scheduling. If any have not, RAS marks the node $NotReady$. RAS then determines if the current node i is the best candidate for scheduling by comparing its upper bound with the previous best node, $ReadyNode$. If the node i upper bound is lower, $ReadyNode$ is set to be the current node i. If the upper bounds are equal, and the height for the current node i is higher, $ReadyNode$ is set to be the current node i.

Once a ready node has been determined, the node is actually scheduled. First, the type of operation the node performs is determined. Then, the lower bound of the time slot for the node is calculated. Next, the list of resources is iterated over to find a valid resource with an available time slot. Each resource has II time slots available. If a resource that can perform the node's operation is available, the node is scheduled. If no time slots are available, the scheduling of the node fails, resulting in the overall scheduler being required to restart the scheduling process with a larger II. This results in each resource having a correspondingly larger number of available time slots during the next iteration of the overall scheduler.

After the node i has been scheduled, the upper and lower bounds on the available time slots for the remaining unscheduled nodes must be calculated. The new lower time bound for a node j, is $max(j_{lb}, i_{ScheduleTime} + (lat_{i,j} - II * dis_{i,j})$, where j_{lb} is the current lower time bound for node j and $i_{ScheduleTime}$ is the time slot node i was scheduled in. The new upper time bound for node j is $min(j_{ub}, i_{ScheduleTime} - (lat_{j,i} - II * dis_{j,i})$. where j_{lb} is the current upper time bound for node j.

The above scheduling process continues until all nodes are scheduled. If, at any time, a node cannot be scheduled, the II is increased by one, the cost matrix is

recomputed, the instructions, nodes and resources are reinitialized, and the process is repeated.

5.4 Implementation of the RAS Resource Allocation Unit

After the resources have been allocated and scheduled, they can be joined together to form the data-path for the given application. First, the number of multiplexors and de-multiplexers required is calculated. Then, the resources are linked together with the other resources, and bound to specific instructions.

During the scheduling phase, RAS specifies the number of instructions each given resource is bound to. If a resource must perform more than one instruction, it may need to link to multiple resources. RAS allocates muxes/de-muxes to facilitate this requirement when needed. The given resource's inputs must be linked to a multiplexor, and its output linked to a de-multiplexor. RAS determines the number of muxes and de-muxes needed by iterating through the resource list and calculating the number of resources that perform more than one instruction.

Once the number of muxes needed has been established, RAS links the resources together. RAS first initializes the mux and de-mux lists. RAS then iterates through the list of resources to perform the actual linkages. If the current resource, i, performs multiple instructions, the inputs of i are first linked to a mux and the output of i linked to a de-mux. Then, for each instruction, j, mapped to resource i, RAS first links the resources performing the instructions j depends on to one of the input multiplexors. RAS then links the resources performing the instructions depending on j to the output de-multiplexor.

If the current resource, i, performs a single instruction, j, RAS first links the instructions j depends on to the inputs of resource i. RAS then links the instructions dependent on j to the output of resource i.

5.5 Implementation of the RAS Transformation Unit

RAS supports some useful transformations that can be used after it has successfully allocated and scheduled the resources and created a data-path. In particular, RAS has routines for supporting the reuse and tiling optimizations presented in Section 4.5.

The reuse feature of RAS assumes that the reuse is for a set of array elements. To facilitate this feature, RAS needs to know the instruction that performs the loading of the array element. Also, RAS needs to know how many operands are being reused for subsequent iterations. RAS then moves the reused instruction outside the inner loop, and recalculates the remaining dependencies. RAS then reinitializes the loop, and calls the main resource allocation and scheduling procedure with the new loop nest.

The tiling feature of RAS also assumes the resources have been allocated and scheduled. There are two implementations for tiling in RAS. One version simply calculates the number of cycles required to implement the tiled loop nest. The other version requires the user to re-specify the amount of resources. RAS reduces the iteration count of the tiled loop nest, and reallocates and schedules the loop nest.

Chapter 6

Experimental Setup and Results

The goals of our experiments were to:

- verify the potency of the design methodology and RAS on a set of nested loops
- determine resources and schedule for several computationally intense algorithms
- measure speedup attainable using the basic design methodology and RAS
- measure speedup attainable using the design methodology optimizations
- measure amount of resources required to obtain significant speedup
- measure execution time of several algorithms on a general purpose processor

We used our design methodology and RAS to design the following computationally intense algorithms:

1. Matrix multiplication
2. Two subroutines for the Full Multi-grid Algorithm, including:

   Red-black Gauss-Seidel Relaxation

   Residual

To verify the accuracy of our schedule, we implemented a design for the matrix multiplication algorithm on a reconfigurable computing system. The remaining designs of the matrix multiplication, as well as the other algorithms were not implemented on the hardware system. However, the accuracy of the RAS estimates for the matrix multiplication indicate similar results would be attainable for the other implementations.

In the next section, we present an overview of the general purpose processors used in our comparisons, the hardware library, and the R.C. system and software tools we used to implement our design. The following section presents the RAS results for the matrix multiplication algorithm on the R.C. systems. The subsequent sections illustrate the RAS results for the relaxation and residual algorithms, respectively. These sections include: the pseudo-code for the algorithm; the corresponding assembly language code for the innermost loop; the required resources; the initiation interval; and timing estimates. Note that the timing estimates for the larger sized matrices do not reflect the overhead likely required to partition the data for the R.C. memory modules.

6.1 Overview of Computing Systems

We used three general purpose processors for comparison to the R.C. system. The first general purpose processor, *GP1* consists of a Dell with a 300 MHz Pentium II processor, 128MB of RAM, and 10 GB hard drive. *GP1* uses the Windows NT 4.0 operating system.

The next general purpose processor, *GP2*, consists of a Gateway computer with a 750 MHz Pentium III processor, 256MB of RAM, and 10GB hard drive. *GP2* uses the Windows 98 operating system.

The third general purpose processor, *GP3*, consists of a Gateway computer with

a 1400 MHz Pentium IV processor, 512MB of RAM, and 60GB hard drive. *GP3* uses the Windows 2000 Professional operating system.

In addition to the above general purpose processors, we used two actual reconfigurable computing systems. We had several hardware modules in our resource library. For our experiments, we assume the resources would require equivalent CLB usage for the given systems. In actual systems, the CLB usage would likely change for the different FPGA technologies. The available resources, as well as their latency and area requirements, are:

- *IADD*: integer adder, 1 cycle latency, 10 CLB area;
- *INC*: incrementor, 0 cycle latency, 10 CLB area;
- *IMULT*: integer multiplier, 6 cycle latency, 50 CLB area;
- *IDIV*: integer divider, 3 cycle latency, 100 CLB area;
- *FADD*: floating point adder/subtractor, 8 cycle latency, 300 CLB area;
- *FACC*: floating point accumulator, 1 cycle latency, 300 CLB area;
- *FMULT*: floating point multiplier, 8 cycle latency, 350 CLB area;
- *FDIV*: floating point divider, 8 cycle latency, 400 CLB area;
- *MEM*: load/store unit, 3 cycle latency, 5 CLB area;
- *SRAM*: register file, 1 cycle latency, $MemorySize * 10$ CLB area;

The incrementor is used for the calculation of memory addresses. We model it as having zero latency because in an actual system, the increment for the address calculation could take place simultaneously with the read. We still require the dependence graph to indicate the increment cannot take place until after the instruction using the memory address has read the value in the corresponding memory location.

Our first reconfigurable computing system, *RC1*, consists of an FPGA connected to a host processor. The FPGA is an Annapolis Microsystems (AMS) Wildforce board. It attaches five processing elements (PE_0 - PE_4) to the host via the PCI bus. Each PE consists of a Xilinx 4044xl FPGA attached to a 1 MB local memory module. For our experiments, we model the PEs as having a 1000 CLBs capacity. The PEs have a maximum frequency of 50 MHz. The Wildforce also has three FIFOs ($FIFO_0$, $FIFO_1$, and $FIFO_4$). $FIFO_0$ is connected to PE_0, $FIFO_1$ is connected to PE_1, and $FIFO_4$ is connected to PE_4. *GP1* serves as the host processor.

Our second reconfigurable computing system, *RC2*, consists of an FPGA connected to a host processor. The FPGA is an Annapolis Microsystems (AMS) Wildcard board. It attaches a processing element (PE) to the host via the PCI bus. The PE consists of a Xilinx Virtex 300 FPGA attached to two 256KB local memory modules. For our experiments, we model the PE as having a 5000 CLB capacity. The PE has a maximum frequency of 100 MHz. *GP2* serves as the host processor for *RC2*.

For both R.C. systems, the host processor communicates with the Wildforce board using the AMS application programmer interface (API) available with the respective FPGA board. The APIs consist of a set of C header file and libraries that can be included with the user program. These libraries contain procedures for performing several functions, including:

- Open/close the board.
- Load a hardware image onto the board.
- Reset the board.
- Read/write information from/to the PE memories.
- Set the clock frequency.

- Set the clock mode as free-run or stepped.
- Step the clock a given number of cycles.

The APIs for *RC1* also include procedures for accessing the FIFOs.

The hardware image for our verification implementation was modeled using VHDL. The VHDL code was synthesized into a netlist files using Synplicity. The netlists were placed and routed with Xilinx Foundation Series software.

In addition to the above actual R.C. systems, we used several hypothetical R.C. systems for our tests. The first hypothetical system, *HS1*, contains an adequate general purpose processor attached to a single node FPGA board. The processing element is an FPGA with a 700 CLB capacity, with a maximum frequency of 33 MHz. It contains one 1 MB memory module.

The second hypothetical system, *HS2*, contains an adequate general purpose processor attached to a 4 node FPGA board. Each processing element is an FPGA with a 30000 CLB capacity and maximum frequency of 400 MHz. Each PE is connected to two 8 MB memory modules. The final hypothetical system, *HS3*, is similar to *HS2*, except it contains 2 FPGA boards.

6.2 Matrix Multiplication

Matrix multiplication is used in several algorithms, including compression and image processing. This algorithm has the form $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$, where the $\mathbf{C}, \mathbf{A}$, and $\mathbf{B}$ matrices are of size $m \times p$, $m \times n$, and $n \times p$, respectively. The elements of $\mathbf{C}$ are computed as:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

$$i = 1, 2, \cdots, m$$

$$j = 1, 2, \cdots, p$$

$$k = 1, 2, \cdots, n$$

(6.1)

Our software implementation contains three tightly nested loops with a rectangular iteration space. Due to the flow of control, we feel this algorithm is well suited for implementation on our R.C. system. It has few conditionals and a large degree of independence between outputs. As a result, there is a fair amount of parallelism in this algorithm.

Various matrix sizes were used to thoroughly test the effectiveness of the computing platforms. The matrix sizes we used were: 128×128, 384×384, 512×512, 768×768, 1024×1024, 2048×2048, and 3072×3072.

Before comparing the general purpose implementation with the R.C. systems, we first made modifications to the matrix multiplication algorithm. We determined the ordering of the i, j, k loops that provided the most efficient design. We performed tests with the different matrix sizes with the six possible permutations of loop orderings:

1. i as outer loop, j as middle loop, k as inner loop;
2. i as outer loop, k as middle loop, j as inner loop;
3. j as outer loop, i as middle loop, k as inner loop;
4. j as outer loop, k as middle loop, i as inner loop;
5. k as outer loop, j as middle loop, i as inner loop;
6. k as outer loop, i as middle loop, j as inner loop.

For each given loop ordering, a scalar replaced the memory access that is reused for the corresponding inner loop. For example, when k was the inner loop, we replaced the memory reuse of $c_{i,j}$ with a scalar variable.

Table 6.1 shows the results of performing the matrix multiplication with the different permutations of the i, j, k loops using *GP1*. The first column shows the dimension size of the square matrices used in the experiment. The remaining columns show the results for a specific loop ordering. The heading shows the loop order starting from the outer loop. For example, the second column gives the results for the $i \cdot j \cdot k$ ordering. This implementation contained i as the outer loop, j as the middle loop, and k as the innermost loop. The body of the table shows the amount of time, in seconds, to multiply matrices of the given size, using the corresponding loop ordering. For example, it took .1586 seconds to perform matrix multiplication on a 128×128 matrix with the $i \cdot j \cdot k$ ordering using *GP1*. Tables 6.2 and 6.3 show the results for configurations *GP2* and *GP3*, respectively.

As can be seen from the tables, one matrix ordering did not consistently give the best results. This is likely due to variations in the usage of the memory hierarchy for the different orderings. Therefore, instead of using one ordering for our comparisons, we used the best time for each matrix size for our comparisons.

Table 6.1: Matrix multiplication execution times for different ordering of loops using *GP1*.

Matrix Dimension	$i \cdot j \cdot k$ Order	$i \cdot k \cdot j$ Order	$j \cdot i \cdot k$ Order	$j \cdot k \cdot i$ Order	$k \cdot j \cdot i$ Order	$k \cdot i \cdot j$ Order
128×128	.1586	.2017	.1509	.2201	.2235	.2025
384×384	6.84	20.61	6.96	26.72	26.78	6.21
512×512	18.72	15.64	33.13	31.87	32.497	15.36
1024×1024	177.05	125.05	178.35	278.51	300.12	137.24
2048×2048	2006.10	1004.94	2031.84	3636.75	3738.04	1004.15
3072×3072	6803.73	3478.88	6956.04	15096.37	15440.52	3481.9

After determining the best loop ordering for each general purpose processor implementation, we used RAS to allocate and schedule resources for our reconfigurable

Table 6.2: Matrix multiplication execution times for different ordering of loops using *GP2*.

Matrix Dimension	$i \cdot j \cdot k$ Order	$i \cdot k \cdot j$ Order	$j \cdot i \cdot k$ Order	$j \cdot k \cdot i$ Order	$k \cdot j \cdot i$ Order	$k \cdot i \cdot j$ Order
128×128	.109	.076	.065	.089	.086	.076
384×384	3.26	8.81	2.78	9.34	9.33	3.07
512×512	6.95	7.11	12.4	8.7	14.4	7.1
1024×1024	62.95	56.64	59.92	78.12	73.71	62.12
2048×2048	471.96	453.32	459.58	1133.23	1171.49	455.96
3072×3072	3651.43	1515.06	3670.97	10388.78	10493.87	1503.77

Table 6.3: Matrix multiplication execution times for different ordering of loops using *GP3*.

Matrix Dimension	$i \cdot j \cdot k$ Order	$i \cdot k \cdot j$ Order	$j \cdot i \cdot k$ Order	$j \cdot k \cdot i$ Order	$k \cdot j \cdot i$ Order	$k \cdot i \cdot j$ Order
128×128	.039	.041	.0417	.043	.042	.0406
384×384	1.82	4.21	1.62	5.22	5.23	1.15
512×512	7.43	5.6	3.99	8.77	8.86	5.63
1024×1024	38.43	22.99	39.62	132.12	132.3	23.08
2048×2048	748.71	157.24	750.96	1656.6	1655.31	161.22
3072×3072	3078.91	521.09	3081.45	5817.72	5813.9	520.63

computing systems. Our initial implementation, referred to here as the *regular* implementation, was the basic usage of Step 1.F (see Section 3.2). This involves examining the algorithm, determining the base set of resources, and scheduling the instructions to the resources.

Table 6.4 shows the RAS estimates, in seconds, for the regular implementation of matrix multiplication. The first column indicates the matrix size. The subsequent columns indicate the time estimate, in seconds, to implement matrix multiplication on the given configuration. *RC1* and *HS1* (group 1) required an initiation interval of 2 for the inner k loop, due to the two memory accesses. *RC2, HS2,* and *HS3* (group 2) can attain the dependence constrained initiation interval of one since they have two

memory busses. This, in addition to the clock speed, results in the difference in execution times for the two groups. Note that *HS2* and *HS3* have the same performance. This is because we have not yet taken advantage of the additional FPGAs available on *HS3*. RAS determined that only the inner loop would fit on *HS1*. Therefore, the results include some estimated overhead in having to perform the outer loops on the host system.

Table 6.4: Estimated matrix multiplication times using regular implementation for reconfigurable computing systems.

Matrix Dimension	*RC1* Configuration	*RC2* Configuration	*HS1* Configuration	*HS2* Configuration	*HS3* Configuration
128×128	.091	.024	.134	.006	.006
384×384	2.33	.593	3.49	.148	.148
512×512	5.48	1.39	8.24	.347	.347
1024×1024	43.39	10.93	65.52	2.73	2.73
2048×2048	345.36	86.66	522.38	21.66	21.66
3072×3072	1163.61	291.61	1761.04	72.9	72.9

Table 6.5 shows the number of each resource type allocated by RAS for the inner k loop and the entire loop nest. The first column shows the resource type. The remaining columns show the number of each resource type required for the configurations. The first number shows the number of the given resource type for the inner loop. The second number shows the total needed for the entire loop nest. For example, RAS allocated 2 *IADD* resources for the inner k loop with configuration *RC1*. RAS allocated 3 *IADD* units for the entire loop nest, including the two allocated for the inner k loop. A single 0 indicates no resources for both the inner k loop and the entire loop nest. Note the allocation of the two memory busses used for the inner loop with *RC2, HS2,* and em HS3. The bottom row shows the total space, in terms of CLB usage, needed for the entire loop nest.

Table 6.5: Resource requirements for regular implementation of matrix multiplication on reconfigurable computing systems.

Resource Type	*RC1* Configuration	*RC2* Configuration	*HS1* Configuration	*HS2* Configuration	*HS3* Configuration
IADD	2,3	3	2,2	2,2	2,2
INC	9	9	2,2	0	0
IMULT	0	0	0	0	0
IDIV	0	0	0	0	0
FADD	0	0	0	0	0
FACC	1,1	1,1	1,1	1,1	1,1
FMULT	1,1	1,1	1,1	1,1	1,1
FDIV	0	0	0	0	0
MEM	1,2	2,3	1,2	2,3	2,3
SRAM	0	0	0	0	0
Total Cost	780	785	695	785	785

RAS predicts that a substantial amount of speedup can be attained over the general purpose processor implementations discussed earlier. Tables 6.6, 6.7, and 6.8 give the estimated speedups of the various configurations over *GP1, GP2,* and *GP3*, respectively. The first column gives the matrix size. The other columns indicate the estimated speedup attainable by the reconfigurable computing systems. The information in the table is the speedup over the most efficient loop ordering for the given size. For example, the speedup of *RC2* over *GP2* for the 128×128 matrix is $\frac{.065}{.024} = 2.71$. The numerator is the best performance for *GP2* for the 128×128 sized matrix. Table 6.2 indicates this was the $j \cdot i \cdot k$ ordering. The denominator is the time RAS estimates *RC2* would require to perform the matrix multiplication.

In addition to the regular implementation, we used RAS to produce enhanced designs for matrix multiplication. One enhancement that can be made is to take advantage of the reuse of a given row of the matrix $\mathbf{A}$, or column of the matrix $\mathbf{B}$. This reduces the pressure on the memory bus due to the decrease in the number of

Table 6.6: Estimated speedup of R.C. systems over *GP1* for matrix multiplication.

Matrix Dimension	*RC1* Speedup	*RC2* Speedup	*HS1* Speedup	*HS2* Speedup	*HS3* Speedup
128×128	1.66	6.29	1.13	25.15	25.15
384×384	2.67	10.47	1.78	41.96	41.96
512×512	2.8	11.05	1.86	44.27	44.27
1024×1024	2.88	11.44	1.91	45.81	45.81
2048×2048	2.91	11.6	1.92	46.4	46.4
3072×3072	2.99	11.93	1.98	47.73	47.72

Table 6.7: Estimated speedup of R.C. systems over *GP2* for matrix multiplication.

Matrix Dimension	*RC1* Speedup	*RC2* Speedup	*HS1* Speedup	*HS2* Speedup	*HS3* Speedup
128×128	.714	2.71	.485	10.83	10.83
384×384	1.19	4.69	.797	18.78	18.78
512×512	1.27	5.0	.84	20.03	20.03
1024×1024	1.305	5.18	.864	20.77	10.77
2048×2048	1.313	5.23	.868	20.93	20.93
3072×3072	1.29	5.16	.85	20.63	20.63

Table 6.8: Estimated speedup of R.C. systems over *GP3* for matrix multiplication.

Matrix Dimension	*RC1* Speedup	*RC2* Speedup	*HS1* Speedup	*HS2* Speedup	*HS3* Speedup
128×128	.429	1.625	.291	6.5	6.5
384×384	.494	1.94	.33	7.77	7.77
512×512	.728	2.87	.484	11.5	11.5
1024×1024	.53	2.10	.351	8.42	8.42
2048×2048	.455	1.81	.301	7.26	7.26
3072×3072	.447	1.79	.296	7.14	7.14

memory accesses.

Table 6.9 gives the results for reusing a given row of the matrix **A**. Here, RAS tries to store the entire row at one time. The first column gives the matrix size. The subsequent columns give the time, in seconds, for the implementations on the corresponding configuration. Only *HS2* and *HS3* have the amount of resources needed to implement the design for this reuse version (except the 3072×3072 matrix). *RC2* can accommodate the smaller matrix sizes. Note that there was no improvement in the amount of time needed over the regular implementation. This is due to the fact that *RC2, HS2,* and *HS3* contain two memory busses. RAS used both busses for the regular implementation, achieving an initiation interval of one. Removing the memory accesses for the **A** matrix simply results in RAS using only one of the available memory busses.

Table 6.9: Estimated matrix multiplication times using reuse implementation for reconfigurable computing systems.

Matrix Dimension	*RC1* Configuration	*RC2* Configuration	*HS1* Configuration	*HS2* Configuration	*HS3* Configuration
128×128	FAIL	.024	FAIL	.006	.006
384×384	FAIL	.593	FAIL	.148	.148
512×512	FAIL	FAIL	FAIL	.347	.347
1024×1024	FAIL	FAIL	FAIL	2.73	2.73
2048×2048	FAIL	FAIL	FAIL	21.66	21.66
3072×3072	FAIL	FAIL	FAIL	FAIL	FAIL

Table 6.10 shows the resources RAS allocates for the reuse implementation. The first column shows the resource type. The other columns show the configuration. Since *RC1* and *HS1* were unable to utilize this form of reuse, they are not shown in the table. The results for the 128×128 sized matrix are shown since *RC2* could

accommodate the register requirements. *HS2* and *HS3* could accommodate reuse for the matrices up to 2048 × 2048.

Table 6.10: Resource requirements for reuse implementation of matrix multiplication on reconfigurable computing systems.

Resource Type	*RC2* Configuration	*HS2* Configuration	*HS3* Configuration
IADD	2	2	2
INC	8	8	8
IMULT	0	0	0
IDIV	0	0	0
FADD	0	0	0
FACC	1	1	1
FMULT	1	1	1
FDIV	0	0	0
MEM	2	2	2
SRAM	128	128	128
Total Cost	2040	2040	2040

After attempting to reuse entire rows of the matrix, we tried to reuse a portion of a row. Here, RAS first determines how much space remains to dedicate for reuse registers. It then doubles the number of registers, starting with 1, until resource constraints restrict further doubling. It then adds a value for modeling the overhead that would be incurred for this type of reuse, and schedules the loop nest.

Tables 6.11 and 6.12 show the results for the segmented reuse of rows. The first column shows the matrix size. The body of Table 6.11 shows the estimated execution time for the segmented reuse implementation. For example, RAS estimates the time for *RC1* would be 22.19 seconds for the 1024 × 1024 matrix. Note that the times are slower with this implementation on *RC2,HS2,* and *HS3* than for the corresponding regular implementations. This is due to the overhead. RAS estimates the time for

this implementation would be faster for *RC1* than for the regular implementation. This is because it can now achieve an initiation interval of one. *HS1* was unable to use any sized register file. Therefore, it is not included in the table.

The body of Table 6.12 shows the number of registers RAS determined could be allocated with the remaining space. For example, RAS determined 16 registers could be allocated for *RC1* with the remaining space after allocating the regular loop nest.

RAS also estimated the time for a tiled implementation for matrix multiplication. Here, the initial design is replicated, with each copy processing a portion of the iteration space. The FPGA board must have duplicate resources to benefit from this optimization. For example, the iteration space for *RC1* is split into 4 tiles, one per FPGA. Each FPGA has a copy of the earlier regular implementation. Since *HS1* only contains one board, it cannot be used for the tiled implementation. Although *RC2* also only contains one board, we can implement a tiled version on *RC2* by utilizing the reuse implementation discussed earlier. Here, we're splitting *RC2* into two identical halves, each with its own memory bus. Since the reuse implementation only required one bus during the inner loop, and *RC2* has enough space for two copies of the regular implementation, it can be used here. The iteration space is therefore split into 2 tiles, one per half. The iteration space is split into 4 and 8 tiles for *HS2* and *HS3*, respectively. The estimates for the tiled versions are shown in Table 6.13.

Table 6.11: Estimated matrix multiplication times for segmented reuse implementation for reconfigurable computing systems.

Matrix Dimension	*RC1* Configuration	*RC2* Configuration	*HS2* Configuration	*HS3* Configuration
128×128	.053	.067	.091	.091
384×384	1.23	.972	.907	.907
512×512	2.86	2.06	1.69	1.69
1024×1024	22.19	13.62	8.11	8.11
2048×2048	174.65	97.4	43.16	43.16
3072×3072	586.24	315.79	121.25	121.25

Table 6.12: Resgister file size for segmented reuse implementation of matrix multiplication on reconfigurable computing systems.

Matrix Dimension	*RC1* Configuration	*RC2* Configuration	*HS2* Configuration	*HS3* Configuration
128×128	16	128	128	128
384×384	16	256	256	256
512×512	16	256	512	512
1024×1024	16	256	1024	1024
2048×2048	16	256	2048	2048
3072×3072	16	256	2048	2048

Table 6.13: Estimated matrix multiplication times using tiled implementation for reconfigurable computing systems.

Matrix Dimension	*RC1* Configuration	*RC2* Configuration	*HS2* Configuration	*HS3* Configuration
128×128	.023	.012	.0015	.00077
384×384	.585	.297	.0373	.0187
512×512	1.38	.696	.0872	.0438
1024×1024	10.87	5.47	.684	.343
2048×2048	86.42	43.35	5.42	2.71
3072×3072	291.09	145.85	18.24	9.12

6.3 Full Multi-grid Algorithm

The Full Multi-grid Algorithm (FMG) is an algorithm for solving a set of partial order differential equations [92]. The FMG algorithm contains several computationally intense sub-algorithms, including the red-black Gauss-Seidel relaxation algorithm and an algorithm for determining the residual. These algorithms were chosen because of their usefulness and suitability for implementation on R.C. systems. Therefore, here we simply discuss the algorithms and the RAS results. The reader is referred to [92] for further information about the FMG algorithm.

6.3.1 Red-black Gauss-Seidel Relaxation

The Gauss-Seidel relaxation method is used as a smoothing operator in the FMG algorithm. As a result, this method is used frequently during the application of the FMG. A reduction in the execution time of the relaxation routine should result in a reduction in the execution time for the overall FMG algorithm.

Figure 6.1 shows the C code for the relaxation algorithm. It contains three loops. Figures 6.2 and 6.3 show the equivalent assembly code for the inner loop. Figure 6.4 shows the registers used for the variables. Here, we're assuming an unlimited number of registers. The inner loop for this algorithm translates to significantly more instructions than the inner loop for matrix multiplication.

Tables 6.15 through 6.19 show the results for relaxation procedure. Table 6.15 shows the results for executing the relaxation algorithm on the general purpose processors. The first column shows the matrix dimension size. The remaining columns show the results for the corresponding processor. The results are shown in seconds. (We do not show the results for the 3072×3072 matrix for *GP1* because, after several attempts, we were unable to obtain satisfactory results. We feel the experiments were

```
void relax2(double **u, double **rhs, int n)
{
   int i, ipass, isw, j, jsw = 1;
   double foh2, h, h2i, res;
   h=1.0/(n-1);
   h21 =1.0/(h*h);
   foh2 = -4.0*h2i;
   for (ipass = 1; ipass<=2; ipass++,jsw=3-jsw)
      isw=jsw;
      for (j=2;j<n;j++,isw=3-isw)
         for(i = isw+1;i<n;i+=2)
            res=h2i*(u[i+1][j] + u[i-1][j] +
            u[i][j+1] + u[i][j-1] - 4.0*u[i][j])
            + u[i][j]*u[i][j] - rhs[i][j];
}
```

Figure 6.1: C code for the red-black Gauss-Seidel relaxation algorithm.

Table 6.14: Execution times for performing Gauss-Seidel relaxation on general purpose processors.

Matrix Dimension	*GP1*	*GP2*	*GP3*
128×128	.006755	.003573	.001736
384×384	.1074	.05195	.02171
512×512	.1922	.0876	.04213
1024×1024	.8484	.3503	.31243
2048×2048	4.006	2.107	1.8668
3072×3072	-	8.91	4.418

skewed due to the memory size, which results in numerous page faults, and do not properly reflect what is attainable on the computing system.)

Tables 6.15 and 6.16 show the RAS results. Table 6.15 shows the RAS estimates for executing relaxation on the R.C. systems. The first column gives the matrix size.

```
R26 = R22 + R21       -- address for u[i+1][j]
R27 = [R26]           -- value of u[i+1][j]
R28 = R22 + R20       -- address for u[i-1][j]
R29 = [R28]           -- value of u[i-1][j]
R30 = R23 + R19       -- address for u[i][j+1]
R31 = [R31]           -- value of u[i][j+1]
R32 = R24 + R19       -- address for u[i][j-1]
R33 = [R32]           -- value of u[i][j-1]
R34 = R22 + R19       -- address for u[i][j]
R35 = [R34]           -- value of u[i][j]
R36 = R25 + R19       -- address for rhs[i][j]
R37 = [R36]           -- value of rhs[i][j]
R38 = R27 ADD R29     -- sum1 = u[i+1][j] + u[i-1][j]
R39 = R38 ADD R31     -- sum2 = sum1 + u[i][j+1]
```

Figure 6.2: First segment of assembly language of inner loop for relaxation routine.

The body of the table shows the estimates in seconds. The methodology fails for *RC1* and *HS1*. RAS was able to successfully allocate and schedule resources for the remaining R.C. systems. RAS specifies that an initiation interval of three is attainable for the inner loop on the systems. Table 6.16 shows the resources RAS allocated. RAS estimates that a total of 2800 CLBs would be needed to implement the relaxation algorithm.

Tables 6.17 through 6.19 show the RAS estimates for the speedup attainable with the corresponding R.C. system. Inspection of the tables indicate the results get better as the size of the matrices gets larger. This is likely due to a reduction in the efficiency of the general purpose processor for the larger matrix sizes. The larger matrix sizes lead to increased time, which allows for more interruptions by the operating system. The R.C. systems would not be hampered by this limitation.

```
R40 = R39 ADD R33 -- sum3 = sum2 + u[i][j-1]
R41 = 4.0 MULT R35 -- tmp = 4.0 * u[i][j]
R42 = R40 MINUS R41    -- sum4 = sum3 - tmp
R43 = R11 MULT R42- Prod1 = h2i*sum4
R44 = R35 MULT R35- Prod2 = u[i][j]*u[i][j]
R45 = R43 ADD R44 -- Sum5 = Prod1 + Prod2
R12 = R45 MINUS R37    -- res = sum5 - rhs[i][j]
R46 = 2.0 MULT R35 -- tmp = 2.0*u[i][j]
R47 = R9 ADD R46         -- tmp2 = foh2 + tmp
R48 = R12 DIV R47        -- tmp3 = res/tmp2
R35 = R35 MINUS R48    -- u[i][j] = u[i][j] - tmp3
[R34] = R35              -- store result in u[i][j]
R19 = R19 + R15      -- increment i offset
R16 = R20 + R15      -- increment i - 1 offset
R17 = R21 + R15      -- increment i + 1 offset
R4 = R4 + 1              -- increment i
CMP R5,R3                -- compare i with n
BLT BB5                  -- branch to top of loop
```

Figure 6.3: Second segment of assembly language of inner loop for relaxation routine.

Table 6.15: Estimated execution times for performing Gauss-Seidel relaxation on reconfigurable computing systems.

Matrix Dimension	*RC1* Configuration	*RC2* Configuration	*HS1* Configuration	*HS2* Configuration	*HS3* Configuration
128×128	FAIL	.00105	FAIL	.000263	.000263
384×384	FAIL	.00758	FAIL	.00189	.00189
512×512	FAIL	.0131	FAIL	.00326	.00326
1024×1024	FAIL	.0497	FAIL	.0124	.0124
2048×2048	FAIL	.194	FAIL	.0484	.0484
3072×3072	FAIL	.432	FAIL	.108	.108

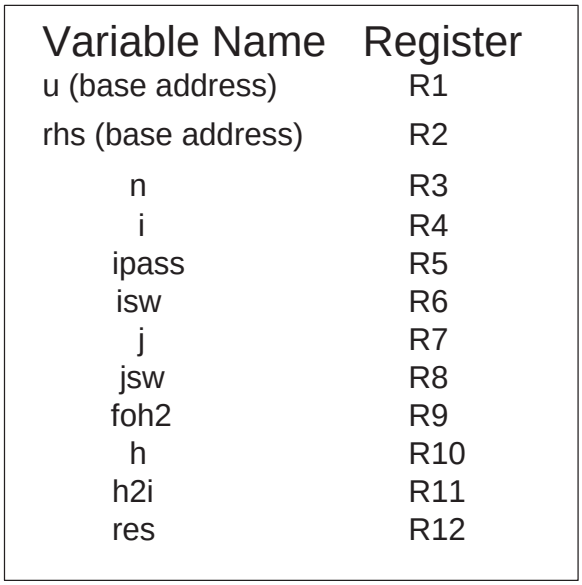

Figure 6.4: Registers used for variables in relaxation routine.

Table 6.16: Resource requirements for implementation of Gauss-Seidel relaxation on reconfigurable computing systems.

Resource Type	*RC2* Configuration	*HS2* Configuration	*HS3* Configuration
IADD	6	6	6
INC	6	6	6
IMULT	1	1	1
IDIV	0	0	0
FADD	3	3	3
FACC	0	0	0
FMULT	2	2	2
FDIV	1	1	1
MEM	3	3	3
SRAM	0	0	0
Total Cost	2800	2800	2800

Table 6.17: Estimated speedup of R.C. systems over *GP1* for Gauss-Seidel relaxation.

Matrix Dimension	*RC2* Speedup	*HS2* Speedup	*HS3* Speedup
128×128	6.43	25.68	25.68
384×384	14.17	56.83	56.83
512×512	14.67	58.96	58.96
1024×1024	17.07	68.42	68.42
2048×2048	20.65	82.77	82.77

Table 6.18: Estimated speedup of R.C. systems over *GP2* for Gauss-Seidel relaxation.

Matrix Dimension	*RC2* Speedup	*HS2* Speedup	*HS3* Speedup
128×128	3.4	13.59	13.59
384×384	6.85	27.49	27.49
512×512	6.69	26.87	26.87
1024×1024	7.05	28.25	28.25
2048×2048	10.86	43.53	43.53
3072×3072	20.63	82.5	82.5

Table 6.19: Estimated speedup of R.C. systems over *GP3* for Gauss-Seidel relaxation.

Matrix Dimension	*RC2* Speedup	*HS2* Speedup	*HS3* Speedup
128×128	1.65	6.6	6.6
384×384	2.86	11.49	11.49
512×512	3.22	12.92	12.92
1024×1024	6.29	25.2	25.2
2048×2048	9.62	38.57	38.57
3072×3072	10.23	40.91	40.91

6.3.2 Residual

The calculation of the residual is another sub-algorithm used frequently in the FMG. A reduction in the execution time of calculating the residual should result in a reduction in execution time for the overall FMG algorithm.

Figure 6.5 shows the C code for calculating the residual. The primary section is the first two nested loops. The bottom loop simply sets some values in the result matrix to zero. This sort of operation does not exploit the advantages provided by R.C. systems. Therefore, this loop was not implemented by RAS, and was not included in the timings for the general purpose processors. It is shown here for completeness.

```
void resid(double **res, double **u, double **rhs, int n)
{
    int i, j;
    double h, h2i;
    h=1.0/(n-1);
    h2i=1.0/(h*h);
    for ( j=2; j<n;j++)
        for (i=2;i<n;i++)
            res[i][j] = -h2i*(u[i+1][j] + u[i-1][j] + u[i][j+1] +
                u[i][j-1] - 4.0*u[i][j]) + rhs[i][j];
    for (i=1;i<=n;i++)
        res[i][1] = res[i][n] = res[1][i] = res[n][i] = 0.0;
}
```

Figure 6.5: C code for calculating the residual of a function.

Figures 6.6 and 6.7 show the equivalent assembly code for the inner loop. Figure 6.8 shows the registers used for the variables. Here, we're assuming an unlimited

number of registers. Again, the number and type of instructions is significantly more than for matrix multiplication.

```
R23 = R18 + R17       -- address for u[i+1][j]
R24 = [R23]           -- value of u[i+1][j]
R25 = R18 + R16       -- address for u[i-1][j]
R26 = [R25]           -- value of u[i-1][j]
R27 = R19 + R15       -- address for u[i][j+1]
R28 = [R27]           -- value of u[i][j+1]
R29 = R20 + R15       -- address for u[i][j-1]
R30 = [R29]           -- value of u[i][j-1]
R31 = R18 + R15       -- address for u[i][j]
R32 = [R31]           -- value of u[i][j]
R33 = R22 + R15       -- address for rhs[i][j]
R34 = [R33]           -- value of rhs[i][j]
R35 = R24 ADD R26     -- sum1 = u[i+1][j] + u[i-1][j]
R36 = R35 ADD R28     -- sum2 = sum1 + u[i][j+1]
```

Figure 6.6: First segment of assembly language of inner loop for calculating residual.

Tables 6.20 through 6.25 show the results for the residual procedure. Table 6.20 shows the results for executing the residual algorithm on the general purpose processors. The first column shows the matrix dimension size. The remaining columns show the results for the corresponding processor. The results are shown in seconds. (We do not show the results for the 3072×3072 matrix for *GP1* because, after several attempts, we were unable to obtain satisfactory results. We feel the experiments were skewed due to the memory size, which results in numerous page faults, and do not properly reflect what is attainable on the computing system.)

Tables 6.21 and 6.22 show the RAS results. Table 6.21 shows the RAS estimates for calculating the residual with the R.C. systems. The methodology fails for *HS1*.

```
R37 = R36 ADD R30    -- sum3 = sum2 + u[i][j-1]
R36 = 4.0 MULT R32   -- tmp = 4.0 * u[i][j]
R39 = R37 MINUS R38  -- sum4 = sum3 - tmp
R40 = R8 MULT R39    -- Prod = -h2i*sum4
R41 = R40 ADD R34    -- Total = Prod + rhs[i][j]
R42 = R21 + R15      -- address for res[i][j]
[R42] = R41          -- store total in res[i][j]
R15 = R15 + R11      -- increment i offset
R16 = R16 + R11      -- increment i - 1 offset
R17 = R17 + R11      -- increment i + 1 offset
R5 = R5 + 1          -- increment i
CMP R5,R4            -- compare i with n
BLT BB4              -- branch to top of loop
```

Figure 6.7: Second segment of assembly language of inner loop for calculating residual.

Table 6.20: Execution times for calculating the residual on general purpose processors.

Matrix Dimension	*GP1*	*GP2*	*GP3*
128×128	.003263	.002486	.000741
384×384	.047365	.02356	.005879
512×512	.084335	.043127	.009666
1024×1024	.3474	.175	.03922
2048×2048	1.399	.6949	.15625
3072×3072	-	1.555	.3482

RAS was able to allocate and schedule resources for *RC1,RC2,HS2,* and *H3*. RAS could only schedule the inner loop on *RC1*. It could only attain an initiation interval of seven. For the other systems, it could successfully schedule both loops, achieving an initiation interval of four. Table 6.22 shows the number of resources RAS allocated.

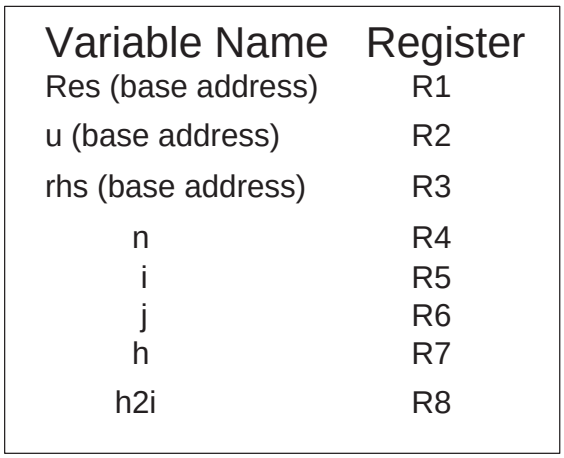

Figure 6.8: Registers used for variables in routine for calculating the residual.

RAS used all of the 1000 CLBs available on *RC1*. It used 1375 CLBs for *RC2,HS2,* and *HS3*.

Table 6.21: Estimated execution times for calculating the residual on reconfigurable computing systems.

Matrix Dimension	*RC1* Configuration	*RC2* Configuration	*HS1* Configuration	*HS2* Configuration	*HS3* Configuration
128×128	.00246	.000744	FAIL	.000186	.000186
384×384	.0212	.00616	FAIL	.00154	.00154
512×512	.0374	.0108	FAIL	.00271	.00271
1024×1024	.1482	.0427	FAIL	.0107	.0107
2048×2048	.5899	.169	FAIL	.0423	.0423
3072×3072	1.325	.3796	FAIL	.0949	.0949

Tables 6.23 through 6.25 show the RAS estimates for the speedup attainable with

Table 6.22: Resource requirements for implementation of residual algorithm on reconfigurable computing systems.

Resource Type	*RC1* Configuration	*RC2* Configuration	*HS2* Configuration	*HS3* Configuration
IADD	4	7	7	7
INC	0	4	4	4
IMULT	0	0	0	0
IDIV	0	0	0	0
FADD	1	2	2	2
FACC	0	0	0	0
FMULT	1	1	1	1
FDIV	0	0	0	0
MEM	1	2	2	2
SRAM	0	0	0	0
Total Cost	1000	1375	1375	1375

the corresponding R.C. systems. The results for the systems are not as good as for the other algorithms. This is likely due to the fact that the residual only contains two loops. This results in less repetition of the inner (pipelined) loop, which is where the major speed advantage occurs.

Table 6.23: Estimated speedup of R.C. systems over *GP1* for residual algorithm.

Matrix Dimension	*RC1* Speedup	*RC2* Speedup	*HS2* Speedup	*HS3* Speedup
128×128	1.33	4.39	17.5	17.5
384×384	2.23	7.69	30.76	30.76
512×512	2.25	7.81	31.12	31.12
1024×1024	2.34	8.14	32.47	32.47
2048×2048	2.37	8.28	33.07	33.07

Table 6.24: Estimated speedup of R.C. systems over *GP2* for residual algorithm.

Matrix Dimension	*RC1* Speedup	*RC2* Speedup	*HS2* Speedup	*HS3* Speedup
128×128	1.01	3.34	13.81	13.81
384×384	1.11	3.82	15.29	15.29
512×512	1.15	3.99	15.9	15.9
1024×1024	1.18	4.1	16.36	16.36
2048×2048	1.18	4.11	16.43	16.43
3072×3072	1.17	4.1	16.39	16.39

Table 6.25: Estimated speedup of R.C. systems over *GP3* for residual algorithm.

Matrix Dimension	*RC1* Speedup	*RC2* Speedup	*HS2* Speedup	*HS3* Speedup
128×128	.301	.996	3.98	3.98
384×384	.277	.95	3.82	3.82
512×512	.258	.895	3.57	3.57
1024×1024	.265	.919	3.67	3.67
2048×2048	.265	.925	3.69	3.69
3072×3072	.263	.917	3.67	3.67

Chapter 7

Conclusions and Future Research

This dissertation presented a design methodology for implementing computationally intense algorithms on reconfigurable computing systems. Reconfigurable computers present software programmers with a new option for implementing their applications. This new option is capable of providing much better performance than conventional general purpose processors and, in some cases, supercomputers. However, as of now, achieving good performance requires expertise in hardware design, a quality not typical of software programmers. The design methodology allows novice computer designers to implement applications on R.C. systems. Using this methodology, the designer can anticipate achieving significant speedup over a general purpose processor implementation. This methodology also significantly reduces the amount of time needed to implement an algorithm on an R.C. system.

This methodology also lays the foundation for the development of a tool that maps applications to R.C. systems. Several portions of the design methodology correspond closely to techniques used in current compilers. The resource allocation and scheduling steps, a major portion of the design methodology, was automated in the tool RAS. This tool was formed by utilizing techniques used in software pipelining and architectural synthesis. While RAS is primarily a module generation tool for loop nests, the overall design methodology is for larger algorithms.

RAS accepts as input a data-flow graph representing a set of nested loops, a set of available resources, and the resource constraints for the implementation. RAS outputs a set of resources, including their interconnections, and a schedule of when instructions are executed on the corresponding resources. The schedule includes the initiation interval at which the instructions are to be repeated.

Three benchmark computationally intense algorithms were designed with the design methodology and RAS. The validity of the designs was verified by simulating the matrix multiplication algorithm. Our results indicated that a significant performance advantage is attainable using our design methodology and RAS.

Several enhancements can be made for future research. The remaining portions of the program analysis phase of the design methodology should be automated. In addition, a front-end to RAS should be developed that can detect when optimizations such as reuse and tiling can be implemented. This front-end should also ease the transition from a high-level programming language to the flow graph format of RAS.

Bibliography

[1] M. Gokhale, B. Holmes, A. Kopser, D. Kunze, D. Lopresti, S. Lucas, R. Minnich, and P. Olsen. Splash: A Reconfigurable Linear Logic Array. In *International Conference on Parallel Processing*, pages 526–532, 1990.

[2] M. J. Wirthlin and B. L. Hutchings. DISC: A Dynamic Instruction Set Computer. In *Third IEEE Workshop on FPGAs for Custom Computing Machines*, 1995.

[3] L. Agarwal, M Wazlowski, and S. Ghosh. An Asynchronous Approach to Efficient Execution of Programs on Adaptive Architectures Utilizing FPGAs. In *Second IEEE Workshop on FPGAs for Custom Computing Machines*, pages 101–110, 1994.

[4] P. Bertin and Herve' Touati. PAM Programming Environments: Practice and Experience. In *Second IEEE Workshop on FPGAs for Custom Computing Machines*, pages 133–138, 1994.

[5] H. Hogl, A. Kugel, J. Ludvig, R. Manner, K. H. Noffz, and R. Zoz. Enable++: A Second Generation FPGA Processor. In *Third IEEE Workshop on FPGAs for Custom Computing Machines*, 1995.

[6] C. P. Cowen and S. Monaghan. A Reconfigurable Monte-Carlo Clustering Processor (MCCP). In *Second IEEE Workshop on FPGAs for Custom Computing Machines*, pages 59–65, 1994.

[7] S. D. Scott, A. Samal, and S. Seth. HGA: A Hardware-Based Genetic Algorithm. In *ACM/SIGDA Symposium on Field Programmable Gate Arrays*, pages 53–59, 1995.

[8] P. Graham and B. Nelson. Genetic Algorithms in Software and in Hardware. In *Fourth IEEE Workshop on FPGAs for Custom Computing Machines*, 1996.

[9] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touti, and P. Boucard. Programmable Active Memories: Reconfigurable Systems Come of Age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, 1996.

[10] N. Howard, A. Tyrrell, and N. Allinson. FPGA Acceleration of Electronic Design Automation Tasks. 1994.

[11] S. Hauck and A. Agarwal. Software Technologies for Reconfigurable Systems. *submitted to IEEE Transactions on Computers*.

[12] M Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh. PRISM-II Compiler and Architecture. In *First IEEE Workshop on FPGAs for Custom Computing Machines*, pages 9–16, 1993.

[13] P. M. Athanas and H. F. Silverman. Processor Reconfiguration Through Instruction-Set Metamorphosis. *IEEE Computer*, 26(3):11–18, 1993.

[14] J. B. Peterson, R. B. O'Connor, and P. M. Athanas. Scheduling and Partitioning ANSI-C Programs onto Multi-FPGA CCM Architectures. In *Fourth IEEE Workshop on FPGAs for Custom Computing Machines*, 1996.

[15] J. Villasenor and W. H. Mangione-Smith. Configurable Computing. *Scientific American*, June 1997.

[16] D. Wo and K. Forward. Compiling to the Gate Level for a Reconfigurable Co-Processor. In *1985 International Symposium On Circuits and Systems*, pages 2–4, 1985.

[17] A. K. Agerwala and T. G. Rauscher. *Foundations of Microprogramming Architecture, Software, and Applications*. Academic Press, New York, N.Y., 1976.

[18] D. A. Patterson and D. R. Ditzel. The Case for RISC. *Computer Architecture News*, 8(6):25–33, October 1980.

[19] G. Estrin et al. Parallel Processing in a Restructurable Computer System. In *IEEE Transactions on Electronic Computers*, pages 747–755, 1963.

[20] D. Galloway. The Transmogrifier C Hardware Description Language and Compiler for FPGAs. In *Third IEEE Workshop on FPGAs for Custom Computing Machines*, 1995.

[21] M. Weinhardt and W. Luk. Pipeline Vectorization for Reconfigurable Systems. In *Seventh IEEE Workshop on FPGAs for Custom Computing Machines*, pages 52–63, 1999.

[22] S. Hauck. The Roles of FPGAs in Reprogrammable Systems. *Proceedings of the IEEE*, 86(4):615–638, April 1998.

[23] W. H. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V. K. Prasanna, and H. Spaanenburg. Seeking Solutions in Configurable Computing. *Computer*, 30(12):38–43, December 1997.

[24] Pierre Marchal. Field Porgrammable Gate Arrays. *Communcations of the ACM*, 42(4):57–59, 1999.

[25] Xilinx. Virtex 2.5 V Field Programmable Gate Arrays, October 2000. Final Product Specification.

[26] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[27] X. P. Ling and H. Amano. WASMII: a Data Driven Computer on a Virtual Hardware. In *First IEEE Workshop on FPGAs for Custom Computing Machines*, pages 33–42, 1993.

[28] E. Mirsky and A. DeHon. MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources. In *Fourth IEEE Workshop on FPGAs for Custom Computing Machines*, 1996.

[29] A. DeHon. DPGA Utilization and Application. In *ACM/SIGDA Workshop on Field Programmable Gate Arrays*, pages 115–121, 1996.

[30] D. P. Lopresti91. Rapid Implementation of a Genetic Sequence Comparator Using Field-Programmable Logic Arrays. In *Advanced Research in VLSI 1991*, pages 139–152, 1991.

[31] R. F. Lyon. Two's Complement Pipeline Multipliers. *IEEE Transactions on Communications*, COM-24:418–425, 1976.

[32] D. A. Buell and R. L. Ward. A Multiprecise Integer Arighmetic Package. *The Journal of Supercomputing*, 3:89–107, 1989.

[33] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Communcations of the ACM*, 21(2):120–126, 1978.

[34] E. F. Brickell. A Survey of Hardware Implementations of RSA. *Lecture Notes in Computer Science Nr. 435*, pages 368–370, 1990.

[35] S. A. Cuccaro and C. F. Reese. The CM-2X: A Hybrid CM-2 / Xilinx Prototype. In *First IEEE Workshop on FPGAs for Custom Computing Machines*, pages 121–130, 1993.

[36] S. Monagham and C. P. Cowen. Reconfigurable Multi-Bit Processor for DSP Applications in Statistical Physics. In *First IEEE Workshop on FPGAs for Custom Computing Machines*, pages 103–110, 1993.

[37] R. V. Rachakonda. High-Speed Region Detection and Labeling Using an FPGA-based Custom Computing Platform. *Lecture Notes in Computer Science Nr. 975*, pages 86–93, 1995.

[38] H. Schmit and D. Thomas. Implementing Hidden Markov Modelling and Fuzzy Controllers in FPGAs. In *Third IEEE Workshop on FPGAs for Custom Computing Machines*, 1995.

[39] K. Hazelwood, W. B. Ligon III, G. Monn, N. Pothen, R. Sass, D. Stanzione, and K. D. Underwood. Creating Applications in RCADE. In *Proceedings of the 1999 Aerospace Conference*, volume 2, pages 337–349, 1999.

[40] K. M. GajjalaPurna and D. Bhatia. Partitioning in Time: A Paradigm for Reconfigurable Computing. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors*, pages 340–345, 1998.

[41] A. Dawood, Z. Asdani, and B. Bravo. FIR Filter Design and Implementation on Reconfigurable Computing Technology. In *Proceedings of the Fifth International Symposium on Signal Processing and its Applications*, pages 383–386, 1998.

[42] S. Shrivastava and V. K. Jain. Rapid System Prototyping for High Performance Reconfigurable Computing. In *IEEE International Workshop on Rapid System Prototyping*, pages 32–37, 1999.

[43] D. D. Gajski, N. D. Dutt, A. Wu, and S. Lin. *High-level Synthesis, Introduction to Chip and System Design*. Kluwer Academic Publishers, Boston, 1992.

[44] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, Mass., 1989.

[45] R. Maestre, M. Fernandez, R. Hermida, , and N. Bagherzadeh. A Framework for Scheduling and Context Allocation in Reconfigurable Computing. In *Proceedings of the 12th International Symposium on System Synthesis*, pages 134 – 140, 1999.

[46] T. Isshiki and W. W.-M. Dai. High-Level Bit-Serial Datapath Synthesis for Multi-FPGA Systesm. In *ACM/SIGDA Symposium on Field Programmable Gate Arrays*, pages 167–173, 1995.

[47] T. Yamauchi, S. Nakaya, and N. Kajihara. SOP: A Reconfigurable Massively Parallel System and Its Control-Data-Flow Based Compiling Method. In *Fourth IEEE Workshop on FPGAs for Custom Computing Machines*, 1996.

[48] C. Iseli and E. Sanchez. A C++ Compiler for FPGA Custom Execution Units Synthesis. In *Third IEEE Workshop on FPGAs for Custom Computing Machines*, 1995.

[49] M. Gokhale and R. Minnich. FPGA Programming in a Data Parallel C. In *First IEEE Workshop on FPGAs for Custom Computing Machines*, pages 94–102, 1993.

[50] S. A. Guccione and M. J. Gonzalez. A Data-Parallel Programming Model for Reconfigurable Architectures. In *First IEEE Workshop on FPGAs for Custom Computing Machines*, pages 79–87, 1993.

[51] D. A. Clark and B. L. Hutchings. Supporting FPGA Microprocessors through Retargetable Software Tools. *Fourth IEEE Workshop on FPGAs for Custom Computing Machines*, pages 195–205, 1996.

[52] Rainer Kress and Andreas Pyttel. High-Level Synthesis for Dynamically Reconfigurable Hardware/Software Systems. In *Field-Programmable Logic and Applications*, pages 288–297, 1997.

[53] Robert Macketanz and Wolfgang Karl. JVX - A Rapid Prototyping System Based on Java and FPGAs. In *Field-Programmable Logic and Applications*, pages 99–108, 1998.

[54] M. F. Dossis, J. M. Noras, and G. J. Porter. Custom Co-processor Compilation. 1994.

[55] I. Page and W. Luk. Compiling Occam into FPGAs. pages 271–283, 1991.

[56] W. Luk, D. Ferguson, and I. Page. Structured Hardware Compilation of Parallel Programs. 1994.

[57] B. Pottier and J. Llopis. Revisiting Smalltalk-80: A logic Generator for FPGAs. In *Fourth IEEE Workshop on FPGAs for Custom Computing Machines*, 1996.

[58] R. Razdan. *PRISC: Programmable Reduced Instruction Set Computers.* PhD thesis, Harvard University, Cambridge, MA, 1994.

[59] R. Razdan and M. D. Smith. A High-Performance Microarchitecture with Hardware-Programmable Functional Units. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 172–180, 1994.

[60] P. Bertin and Herve' Touati. PAM Programming Environments: Practice and Experience. In *Second IEEE Workshop on FPGAs for Custom Computing Machines*, pages 49–58, 1994.

[61] G. Brown and A. Wenban. A Software Development System for FPGA-Based Data Acquisition Systems. In *Fourth IEEE Workshop on FPGAs for Custom Computing Machines*, 1996.

[62] R. W. Hartenstein, J. Becker, M. Herz, and U. Nageldinger. Data Scheduling to Increase Performance of Parallel Accelerators. In *Field-Programmable Logic and Applications*, pages 294–303, 1997.

[63] Christopher Fraser and David Hanson. *A Retargetable C Compiler: Design and Implementation.* Addison-Wesley Publishing Co., Reading, Massachusetts, 1995.

[64] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in Pascal–The Art of Scientific Computing.* Cambridge University Press, New York, 1990.

[65] V. K. S. Aditya and B. R. Rau. Automatic Architecture Synthesis of VLIW and EPIC Processors. In *Proceedings of the 12th International Symposium on System Synthesis*, pages 107–113, Nov. 1999.

[66] R. Schreiber, S. Aditya, B. R. Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider. High-level synthesis of nonprogrammable hardware accelerators . In *IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, pages 113–124, 2000.

[67] M. Figueiredo and C. Gloster. Implementation of a Probabilistic Neural Network for Multispectral Image Classification on an FPGA Based Custom Computing Machine. In *5th Symposium on Neural Networks*, Belo Horizonte, Brazil, December 1998.

[68] R. Niemann. *Hardware/Software Co-Design for Data Flow Dominated Embedded Systems*. Kluwer Academic Publishers, Boston, M.A., 1998.

[69] Jr. W. R. Smythe and L. A. Johnson. *Introduction to Integer Linear Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1966.

[70] B. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 63–74, 1994.

[71] M. Lee, P. Tirumalai, and T-F Ngai. Software Pipelining and Superblock Scheduling: Compilation Techniques for VLIW Machines. In *Proceedings of the 26nd Hawaii International Conference on System Sciences*, 1993.

[72] J. Wang and G.R. Gao. Pipelining-Dovetailing: A Transformation to Enhance Software Pipelining for Nested Loops. In *Proceedings of the 6th International Conference on Compiler Construction*, pages 1–17, 1996.

[73] J. Wang and B. Su. Software Pipelining of Nested Loops for Real-time DSP Applications. In *Proceedings of the IEE International Conference on Acoustics, Speech and Signal Processing*, pages 3065–8, 1998.

[74] V.H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software Pipelining. *ACM Computing Surveys*, 27(3):367–432, 1995.

[75] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, California, 1997.

[76] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers - Principles, Techniques, and Tools*. Addison-Wesley Publishing Co., Reading, Massachusetts, 1986.

[77] M. Wolfe. More Iteration Space Tiling. In *Proceedings of the 1989 Conference on Supercomputing*, pages 655–664, 1989.

[78] N. J. Warter. *Modulo Scheduling with Isomorphic Control Transformations*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, September 1994.

[79] B. R. Rau and J. A. Fisher. Instruction-Level Parallel Processing: History, Overview, and Perspective. *The Journal of Supercomputing*, 7:9–50, 1993.

[80] P. Y. T. Hsu. *Highly Concurrent Scalar Procesing.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, September 1986.

[81] M. S. Lam. *A Systolic Array Optimizing Compiler.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1987.

[82] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, Mass., 1990.

[83] A. M. Zaky. *Efficient Static Scheduling of Loops on Synchronous Multiprocessors.* PhD thesis, Ohio State University, Columbus, OH, 1989.

[84] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, Inc., New York, 1994.

[85] Michael Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Publishing Company, New York, NY, 1995.

[86] Uptal Banerjee. *Dependence Analysis.* Kluwer Academic Publishers, Boston, MA, 1997.

[87] Uptal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, Boston, MA, 1993.

[88] M. Wolfe. Experiences with Data Dependence Analysis. In *Proceedings of the 1991 International Conference on Supercomputing*, pages 321–329, 1991.

[89] C. Eisenbeis, W. Jalby, and A. Lichnewsky. Squeezing More CPU Performance Out of a Cray-2 by Vector Block Scheduling. In *Proceedings of the 1988 Conference on Supercomputing*, pages 237–246, 1988.

[90] D. Loveman. Program Improvement by Source-to-Source Transformation. *Journal of the ACM*, 20(1):121–145, 1977.

[91] Tinker Research Group. The Lego Compiler. World Wide Web address: http://www.tinker.ncsu.edu/LEGO/.

[92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C.* Cambridge University Press, New York, 1992.