UNIVERSITY OF CALIFORNIA

Los Angeles

Synthesis Techniques and Optimizations for Reconfigurable Systems

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Ryan Kastner

2002

The dissertation of Ryan Kastner is approved.

Jason Cong

William Mangione-Smith

Miodrag Potkonjak

Majid Sarrafzadeh, Committee Chair

University of California, Los Angeles

2002

To everyone that got me this far…

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

VITA

Born August 17, 1977	Greensburg, PA
June, 1999	B.S., Electrical Engineering
	B.S., Computer Engineering
	Northwestern University
	Evanston, IL
1999 – 2000	Graduate Research Assistant
	Department of Electrical and Computer Engineering
	Northwestern University
	Evanston, IL
August, 2000	M.S., Computer Engineering
	Northwestern University
	Evanston, IL
2000 – 2002	Graduate Research Assistant
	Computer Science Department
	University of California, Los Angeles
	Los Angeles, CA

PUBLICATIONS

Ryan Kastner, Adam Kaplan, Seda Ogrenci Memik, Elaheh Bozorgzadeh, "**Instruction Generation for Hybrid Reconfigurable Systems**", *ACM Transactions on Design Automation of Embedded Systems (TODAES)*, October 2002

Ryan Kastner, Elaheh Bozorgzadeh and Majid Sarrafzadeh, "**Pattern Routing: Use and Theory for Increasing Predictability and Avoiding Coupling**", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* July 2002

Xiaojian Yang, Ryan Kastner and Majid Sarrafzadeh, "**Congestion Estimation During Top-down Placement**", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, January 2002

Ankur Srivastava, Ryan Kastner and Majid Sarrafzadeh, "**On the Complexity of Gate Duplication**", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, September 2001

Kiarash Bazargan, Ryan Kastner and Majid Sarrafzadeh, "**3-D Floorplanning: Simulated Annealing and Greedy Placement Methods for Reconfigurable Computing Systems**", *Design Automation for Embedded Systems (DAfES) - RSP'99 Special Issue*, August 2000

Kiarash Bazargan, Ryan Kastner and Majid Sarrafzadeh, "**Fast Template Placement for Reconfigurable Computing Systems**", *IEEE Design and Test - Special Issue on Reconfigurable Computing,* January - March 2000

Elaheh Bozorgzadeh, Adam Kaplan, Ryan Kastner, Seda Ogrenci Memik and Majid Sarrafzadeh, "**Multi-level Optimization in Reconfigurable Computing Systems**", *Multi-level Optimization and VLSI CAD*, Kluwer Press, October 2002

Elaheh Bozorgzadeh, Ryan Kastner, Seda Ogrenci Memik and Majid Sarrafzadeh, "**Strategically Programmable Systems**", *The Computer Engineering Handbook*, CRC Press, December 2001

Philip Brisk, Adam Kaplan, Ryan Kastner and Majid Sarrafzadeh, "**Instruction Generation and Regularity Extraction For Reconfigurable Processors**", *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES),* October, 2002

Ryan Kastner, Christina Hsieh, Miodrag Potkonjak and Majid Sarrafzadeh, "**On the Sensitivity of Incremental Algorithms for Combinatorial Auctions**", *IEEE International Workshop on Advanced Issues of E-Commerce & Web-Based Information Systems (WECWIS),* June 2002

Elaheh Bozorgzadeh, Seda Ogrenci Memik, Ryan Kastner and Majid Sarrafzadeh, "**Pattern Selection: Customized Block Allocation for Domain-Specific Programmable Systems**", *International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA),* June 2002

Ryan Kastner, Seda Ogrenci Memik, Elaheh Bozorgzadeh and Majid Sarrafzadeh, "**Instruction Generation for Hybrid Reconfigurable Systems**", *International Conference on Computer-Aided Design (ICCAD)*, November, 2001

Seda Ogrenci Memik, Elaheh Bozorgzadeh, Ryan Kastner and Majid Sarrafzadeh, "**A Super-Scheduler for Embedded Reconfigurable Systems**", *International Conference on Computer-Aided Design (ICCAD),* November 2001

Xiaojian Yang, Ryan Kastner and Majid Sarrafzadeh, "**Congestion Reduction During Placement Based on Integer Programming**", *International Conference on Computer-Aided Design (ICCAD),* November 2001

Andrew B. Kahng, Ryan Kastner, Stefanus Mantik, Majid Sarrafzadeh and Xiaojian Yang, "**Studies of Timing Structural Properties for Early Evaluation of Circuit Design**", *Workshop on Synthesis and System Integration of Mixed Technologies (SASIMI)*, October 2001

Elaheh Bozorgzadeh, Ryan Kastner and Majid Sarrafzadeh, "**Creating and Exploiting Flexibility in Steiner Trees**", *Design Automation Conference (DAC),* June 2001

Ryan Kastner, Elaheh Bozorgzadeh and Majid Sarrafzadeh, "**An Exact Algorithm for Coupling-Free Routing**", *International Symposium on Physical Design (ISPD)*, April 2001

Xiaojian Yang, Ryan Kastner and Majid Sarrafzadeh, "**Congestion Estimation during Top-down Placement**", *International Symposium on Physical Design (ISPD)*, April 2001

Majid Sarrafzadeh, Elaheh Bozorgzadeh, Ryan Kastner and Ankur Srivastava, "**Design and Analysis of Physical Design Algorithms**", *International Symposium on Physical Design (ISPD)*, April 2001

Seda Ogrenci Memik, Elaheh Bozorgzadeh, Ryan Kastner and Majid Sarrafzadeh, "**Strategically Programmable Systems**", *Reconfigurable Architecture Workshop (RAW)*, April 2001

Ryan Kastner, Elaheh Bozorgzadeh and Majid Sarrafzadeh, "**Predictable Routing**", *International Conference on Computer-Aided Design (ICCAD)*, November 2000

Ankur Srivastava, Ryan Kastner and Majid Sarrafzadeh, "**Timing Driven Gate Duplication: Complexity Issues and Algorithms**", *International Conference on Computer-Aided Design (ICCAD)*, November 2000

Ryan Kastner, Elaheh Bozorgzadeh and Majid Sarrafzadeh, "**Coupling Aware Routing**", *International ASIC/SOC Conference*, September 2000

Ankur Srivastava, Ryan Kastner and Majid Sarrafzadeh, "**Complexity Issues in Gate Duplication**", *International Workshop on Logic Synthesis (IWLS)*, June 2000

Kiarash Bazargan, Ryan Kastner, Seda Ogrenci and Majid Sarrafzadeh, "**A C to Hardware/Software Compiler**", *Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2000

Ryan Kastner, Kiarash Bazargan and Majid Sarrafzadeh, "**Physical Design for Reconfigurable Computing Systems using Firm Templates**", *Workshop on Reconfigurable Computing (WoRC)*, October 1999

Kiarash Bazargan, Ryan Kastner and Majid Sarrafzadeh, "**3-D Floorplanning: Simulated Annealing and Greedy Placement Methods for Reconfigurable Computing Systems**", *International Workshop on Rapid System Prototyping (RSP)*, June 1999

ABSTRACT OF THE DISSERTATION

Synthesis Techniques and Optimizations for Reconfigurable Systems

by

Ryan Kastner

Doctor of Philosophy in Computer Science

University of California, Los Angeles

Professor Majid Sarrafzadeh, Chair

Computing devices are becoming smaller and more complex due to the sheer number of transistors that we can fit on a single chip. The exponential scaling predicted by Moore's Law will allow even more complex system-on-chip devices into the foreseeable future. In order to cope with the complexities of system design, we must work at a higher level of abstraction. This will allow designers to automatically synthesis applications that are written in an application-level description. In addition, it will allow designers to explore the tradeoffs between different system architectures. This work looks at system synthesis techniques for a particular class of system architectures - reconfigurable systems.

Reconfigurable systems allow post fabrication configurability enabling customizable spatial computing. Furthermore, it permits the system to transform the underlying hardware to adapt to the temporal requirements of an application. Reconfigurable architectures can differ in their reconfiguration granularity. An FPGA has a bit-level reconfiguration (datapath width). This gives the system great *flexibility* – the capacity to implement a large number of applications at the cost of reconfiguration time and performance. A reconfigurable architecture with larger datapath width (e.g. byte or word level) loses flexibility but can gain in reconfiguration time and system performance. We discuss many aspects of these tradeoffs in this work.

The heart of this work is a framework for the synthesis for of reconfigurable systems. The framework is built on top of the SUIF compiler and interfaces with Synopsys Behavioral Compiler. We focus on optimizations for two particular system synthesis problems – instruction generation and data communication. Instruction generation is the process of automatically generating customized instructions in a system. Data communication is the amount of data transferred between various components of a system. We detail both of these problems and give methodologies and algorithms to solve each problem. Furthermore, we study methods to increase the amount of parallelism when mapping an application onto a reconfigurable system.

CHAPTER 1   Introduction

1.1 Overview of Dissertation

The invention of the transistor in 1947 is arguably the most important discovery of the 20th century. Without it, Jack Kilby and Robert Noyce could not have created the integrated circuit (IC) in the late 50s. Within the span of 40 years, the IC has permeated our world in almost every possible aspect. It is almost inconceivable to go through the day without the assistance of an IC. Computing devices play a part in delivering the basic human needs; they enable our transportation, allow communication and play a large role in the food preparation.

There are many factors that play a role in delivering computing devices to the world. Of course, no computing system could be made without an understanding of the underlying physics behind the transistors and other circuitry that provide the basic functionality of the IC. Another significant area for developing computing devices is the design and analysis of the organization of the computing device. The organization of computing systems is especially necessary as the complexity of the devices has exponentially grown since the development of the first IC. It is inconceivable to design a present-day computing device using the transistor as your basic element. Therefore, abstraction is needed to tame the complexity.

Abstraction is underlying view or model that a designer uses to create a computing device. As the size of computing systems has grown, so has the level of abstraction. Each step up in abstraction brings about the need to develop automated methods for synthesizing (the process of moving from one level of abstraction to a lower level of abstraction) the computing device. We have reached the point where synthesis techniques have matured to the point of the system level of abstraction. In order to move to successfully move to yet an even higher level of abstraction, we must enhance the synthesis techniques at the system level of abstraction. My thesis focuses on this realm of abstraction, taking into account a special type of computing system – reconfigurable systems.

Reconfigurable systems initially emerged as fast prototyping device for ASIC designers. It allowed the designer to "compile" the application to reconfigurable hardware (e.g. FPGA) to determine that the application exhibited the correct functionality. The prototyping removed the costly step of fabricating the application, especially when fabrication yielded a device it exhibited incorrect functionality. Additionally, the rapid prototyping lessened the need for intense simulation to verify correctness. If the application functioned correctly in the environment when compiled to an FPGA, it would retain the same correctness once it was fabricated. The main drawback of the FPGA was the performance. An FPGA was far behind the ASIC in terms of important performance aspects, like latency, power consumption, etc. In essence, the application was synthesized to the "static" nature of an ASIC and was not taking into account the dynamic reconfigurability allowed by the FPGA; in this sense, the performance of the FPGA can never overcome that of an ASIC.

The power of reconfigurable systems lays the immense amount of flexibility that it provides. The flexibility allows run time reconfiguration and on-the-fly reorganization of the circuit based on the input parameters. Due to the ability to customize to the input data, many applications show speedups when implementing them on reconfigurable systems [1, 2]. In addition, many reconfigurable machines have achieved 100x speedups and 100x performance gain per unit silicon as compared to a similar microprocessor [3-7].

Reconfigurable devices have emerged as flexible, high-performance component in computing systems. We are seeing various examples of computing systems that are fully reconfigurable at the logic level as well as devices that are reconfigurable at an architectural-level. In addition, we are seeing the increased use of reconfigurable cores as components in embedded systems – microprocessors coupled with a reconfigurable component as well as ASICs coupled with a reconfigurable component.

1.2 Organization of Dissertation

This work deals with the synthesis of reconfigurable systems. The thesis is organized in the following manner: First, we discuss the relevant background material. We start with a discussion on reconfigurable systems. In particular, we focus on the Strategically Programmable System to show the tradeoffs between the levels of reconfigurability with respect to circuit performance, area, and reconfiguration time. The following chapter presents an overview of synthesis techniques for digital systems. We focus on system synthesis and architectural synthesis as the remaining chapters lie between these two realms of abstraction.

CHAPTER 4 details our method of synthesis. We utilize two complex software environments – the SUIF compiler and Synopsys Behavioral Compiler. The analysis of our optimizations relies on these two software systems to compile an application written in a high level language all the way into a hardware realization. The remaining part of the thesis discusses optimizations done within the framework to improve the quality of the final hardware implementation.

CHAPTER 5 focuses on the problem of instruction generation. It relates the problem to several types of architectures – including microprocessors – but focuses on the use and impact of instruction generation to reconfigurable architectures. The following chapter looks at the problem of data communication within a reconfigurable system. We utilize the compiler technique of static single assignment to minimize the amount of data that must be communicated between various components of our design. In addition, we give several methods to minimize the data communication and hence yield a design with smaller area.

CHAPTER 7 looks into the important problem of increasing the amount of parallelism in the design. It delves into two EPIC (Explicitly Parallel Instruction Computing) compiler techniques, superblock formation and trace scheduling, which are known to increase the amount of available parallelism in the application. Also, we discuss the side effects of these optimizations on other circuit parameters and especially focus on the effect of these techniques on the interconnect area. Furthermore, we give a formulation that combines hardware partitioning with the increase in parallelism. The final chapter gives some ideas for future research directions in the realm of synthesis for reconfigurable systems. Most of these directions follow directly from the work that we present in the thesis.

CHAPTER 2 Reconfigurable Systems

2.1 Overview

Due to recent advances in the semiconductor industry, computing devices are moving from highly tailored application specific processors (ASICs) and general purpose microprocessors to systems on a chip (SOC). A system on a chip device can integrate many differing components e.g. microprocessors, digital signal processing (DSP) functional units, video processing components, etc. onto one die. Often, these devices require a large amount of flexibility in order to handle a variety of different tasks the system may encounter over its lifetime.

Flexibility is particularly important for embedded systems. Normally, an embedded system is deployed once. Ever changing standards may require different algorithms over the lifetime of the system. For example, the communication standards for cell phones change from year to year, country to country. Furthermore, more efficient, lower power algorithms are frequently developed. A flexible system allows functionality of the device to change over time. The operation of an adaptable device can change on demand, whether the standards change or better algorithms surface.

In the past decade, substantial evidence was provided by the research community, as well as commercial devices, regarding the advantages of reconfigurable systems. Logic reconfigurability was first available in standalone chips. These devices (FPGAs) – developed by companies like Xilinx™, Altera™ and Actel™ -- are 100% programmable. Recently, reconfigurable fabric was integrated into SOCs forming hybrid reconfigurable systems. *Hybrid reconfigurable systems* contain some kind of computational unit, e.g., ALUs, Intellectual Property units (IPs) or even traditional general-purpose processors, embedded into a reconfigurable fabric.

One type of hybrid reconfigurable architecture embeds reconfigurable cores as a coprocessor to a general-purpose microprocessor e.g. Garp [8] and Chimaera [9]. Another direction of new architectures considers integration of highly optimized hard cores and hardwired blocks with reconfigurable fabric. The main goal here is to utilize the optimized blocks to improve the system performance. Such programmable devices are targeted for a specific *context* – a class of similar applications, such as DSP, data communications (Xilinx Platform Series) or networking (Lucent's ORCA®). The embedded fixed blocks are tailored for the critical operations common to the application class. In essence, the programmable logic is supported with the high-density high-performance cores. The cores can be applied at various levels, such as the functional block level, e.g., Fast Fourier Transform (FFT) units, or at the level of basic arithmetic operations (multipliers).

2.2 Strategically Programmable Systems

Presently, a context-specific architecture is painstakingly developed by hand. The Strategically Programmable System (SPS) explores an automated framework, where a systematic method generates context-specific programmable architectures.

The basic building blocks of the SPS architecture are parameterized functional blocks called *Versatile Parameterizable Blocks (VPBs)*. They are pre-placed within a fully reconfigurable fabric. When implementing an application, operations can be performed on the VPBs or mapped onto the fully reconfigurable portion of the chip. An instance of our SPS architecture is generated for a given set of applications (specified by C or Fortran code). The functionality of the VPBs is tailored towards implementing those applications efficiently. The VPBs are customized and fixed on the chip; they do not require configuration, hence there are considerably less configuration bits to program as compared to the implementation of the same design on a traditional FPGA.

The motivation is to automate the process of developing hybrid reconfigurable architectures that target a set of applications. These architectures would contain VPBs that specially suit the needs of the particular family of applications. Yet, the adaptable nature of our architecture should not be severely restricted. The SPS remains flexible enough to implement a very broad range of applications due to the reconfigurable resources. These powerful features help the architecture maintain its tight relation to its predecessors, traditional FPGAs. At the same time the SPS is forming one of the first efforts in the direction of context-specific programmable devices.

In general, there are two aspects of the SPS system. The first area involves generating a context-specific architecture given a set of target applications. Once there is a context-specific architecture, one must also be able to map any application to the architecture.

2.2.1 Overview of SPS

2.2.1.1. Versatile Parameterizable Blocks

The main components of SPS are the *Versatile Parameterizable Blocks (VPBs)*. The VPBs are embedded in a sea of fine-grain programmable logic blocks. We consider a *Look-Up Table (LUT)* based logic blocks commonly referred to as *Combinatorial Logic Blocks (CLBs)*, though it is possible to envision other types of fine-grain logic blocks, e.g. PLA-based blocks.

Essentially, VPBs are hard-wired ASIC blocks that perform a complex function. Since the VPB is fixed resource, it requires little reconfiguration time when switching the functionality of the chip[1]. Therefore, SPS is not limited by large reconfiguration times like current FPGAs. But, the system must strike a balance between flexibility and reconfiguration time. The system should not consist mainly of VPBs, as it will not be able to handle a wide range of functionality.

There is a considerable range of functionality for the VPBs. It ranges from high-level, intensive tasks like FFT to a "simple" arithmetic task like addition or multiplication. Obviously, there is a large range of complexity between these two extremes. Since we are automating the architecture generation process, we wish to

[1] By functionality, we mean the application of the chip can change entirely, e.g. from image detection to image restoration, or part of the application can change, e.g. a different image detection algorithm.

extract common functionality for the given context (set of applications). The functionality should be as complex as possible while still serving a purpose to the applications in the context. An extremely complex function that is used frequently in only one application of the context would be wasted space when another application is performed on that architecture.

The past decade has brought about extensive research as to the architecture of FPGAs. As we previously discussed, many researchers have spent copious amounts of time analyzing the size and components of the LUT and the routing architecture. Instead of developing a new FPGA architecture, the SPS leverages the abundant body of FPGA architecture knowledge for our system. Embedding the VPBs into the programmable logic is the most important task for the SPS architecture generation.

2.2.2 SPS Framework

In this section, we discuss the tools and algorithms that actually generate strategically programmable systems and perform the mapping of applications on the architecture. The architecture formation phase and the architecture configuration phase are the two major parts of the framework. The SPS framework is summarized in FIGURE 1.

Architecture Formation

This task can be described as making the decision on the versatile programmable blocks to place on the SPS chip along with the placement of fine-grain reconfigurable portion and memory elements, given an application or class of applications. In this phase, SPS architecture is customized from scratch given certain directives. This process requires a detailed description of the target application as an input to the formation process. A tool will analyze these directives and generate the resulting architecture. Again the relative placement of these blocks on the SPS chip along with the memory blocks and the fully reconfigurable portions need to be done by the architecture formation tool.

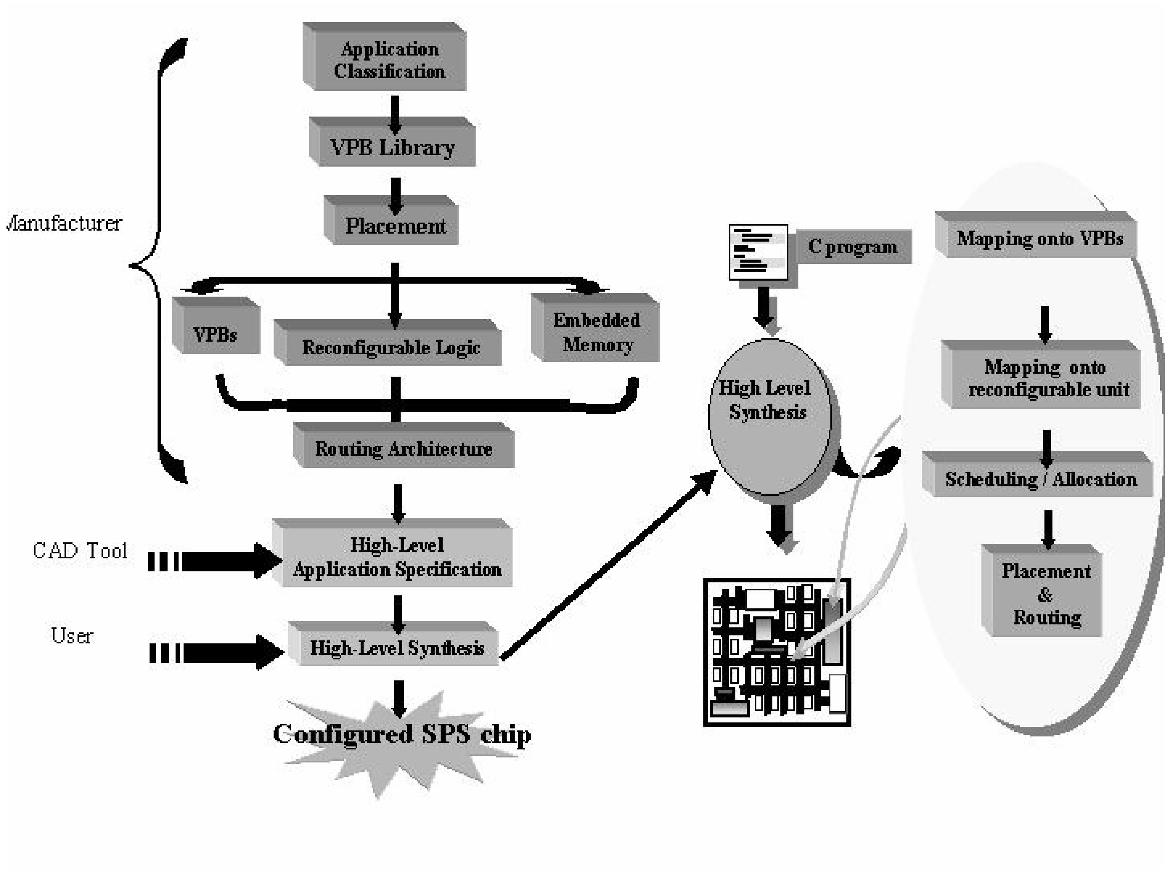


FIGURE 1 A detailed flow for configuring a Strategically Programmable System

Unlike a conventional fine-grain reconfigurable architecture, a uniform distribution of configurable logic blocks does not exist on the SPS. Hence for an efficient use of the chip area as well as high performance, the placement of VPBs on the chip and the distribution of configurable logic block arrays and memory arrays among those are critical. The routing architecture supports such hybrid architecture is equally important and requires special consideration. If the routing architecture cannot provide sufficient routing resources between the VPBs and the configurable blocks, the hardware resources will be wasted. The type and number of routing tracks and switches need to be decided such that the resulting routing architecture can support this novel architecture most efficiently. The most important issues here are the routability of the architecture and the delay in the connections.

2.2.2.1. Fixed Architecture Configuration

Another case that we are considering in our work is mapping an application onto a given architecture. At this point we need a compiler tailored for our SPS architecture. This SPS compiler is responsible for three major tasks:

The compiler has to identify the operations, groups of operations or even functions in the given description of the input algorithm that are going to be performed by the fixed blocks. These portions will be mapped onto the VPBs and the information regarding the setting of the parameters of the VPBs will be sent to the SPS chip.

Secondly, the SPS compiler has to decide how to use the fully reconfigurable portion of the SPS. Based on the information on the available resources on the chip and the architecture, mapping of suitable functions on the fine-grain reconfigurable logic will be performed. Combining these two tasks the compiler will generate the scheduling of selected operations on either type of logic.

Finally, memory and register allocation need to be done. An efficient utilization of the available RAM blocks on the SPS has to be realized.

CHAPTER 3 Synthesis of Digital Systems

3.1 Overview

There have been several fundamental levels of abstraction in hardware design. In the early days of hardware design, the entire circuit was mapped by hand. Soon, low-level standard cells (at the Boolean functional level of complexity) were used as basic building blocks. CAD algorithms for placement and routing are used to layout the circuit. As the number of transistors on a chip continued to increase, the level of abstraction moved to the logic level (logic synthesis), microarchitectural level (high-level synthesis) and the architectural level (behavioral synthesis). I believe that the increasing complexity of computing systems brings about a need for automation/exploration algorithms at an even higher level of abstraction – the system level.

In order to understand the stages of the synthesis of digital systems, we must understand the relationship between the different levels of abstraction. Gajski and Kuhn [10] introduced the Y-chart to describe such relationships. There are three different types of representations – behavioral, structural and physical. Each of these representations is a "branch" of the Y. The circles denote the level of abstraction. By transitioning from a circle (level) to one of its inner circles (levels), we perform a step called *refinement*. Refinement gives more details about the digital circuit. If we go the opposite direction, the process is called *abstraction*. Abstraction gives a high view of the circuit, allowing more dramatic changes to occur.

We can move from the various representations. *Synthesis* is the transformation from a behavioral representation (at any level) to a structural representation of the same level. The reverse process – moving from a structural to behavioral representation – is called *analysis*. Going from a structural representation to a geometric representation is called *generation*. *Extraction* is the reverse of generation – creating a structural representation from a geometric representation.

It is also possible to modify the representation while staying at the same level. Such modifications are called optimizations. There are many possible optimizations that can be done at the various levels and representations. Optimizations are done in order to improve some aspect of the circuit. For example, we may wish to optimize the circuit to reduce the power consumption, increase the throughput or minimize the area. Optimizations are the heart of digital circuit synthesis. Determining the best area, power, and/or throughput for a circuit is a difficult problem. This is the problem of design space exploration.

The innermost circle is the physical level; this is the lowest level of abstraction. The geometric representation of a physical level is a set of polygons. These polygons correspond to metal wires, transistor contacts and channels, positive and negative doped regions of silicon and all other physical materials needed to realize transistors on a mask. Differential equations describe the behavioral representation at the physical level. For example, the relationship between the inductance, voltage and capacitance of the circuit elements can be described in a behavioral manner using differential equations. As a structural representation at the physical level, we have transistors or we could even describe things using resistors and capacitors.

The next level of abstraction is the logic level. In this level, the behavior is described using Boolean equations and Boolean logic. The basic element for the structural representation is gate and a flip-flop. The gates represent some (usually simple) Boolean function such as a 2-input “and” gate, a 4-input “or” gate, etc.. The geometrical representation uses standard cells, which are usually modeled as rectangles. With each standard cell, there is an associated physical representation. For example, a 2-input “and” gate will have the transistor and/or polygonal representation.

By going one level higher in abstraction, we reach the microarchitecture level; this level is sometimes called register-transfer (RT) level or behavioral level. The basic structural elements are data path (e.g. ALUs, adders, multipliers), memory elements (e.g. registers), and steering logic (e.g. multiplexors (MUXs)). The behavioral representation of the microarchitecture level is a register transfer specification. Macro cells are used to model the geometric representation.

The architecture level is the next level of abstraction. An algorithm is used as the behavioral representation. The algorithm allows common operators such as add, multiply, shift and control flow constructs like "for" and "while" loops and if-then-else branching. A processor is an example element as a structural representation at the architecture level. The geometric representation is on the order of blocks or even chips.

The highest abstraction is the system level. The system level uses some model of computation (MOC) as the behavioral representation. We will go into more details on MOCs in a following section. The structural representation has processing elements like a CPU; the processing elements are usually easily programmable. The system level geometric representation is on the level of chip or even a PCB board.

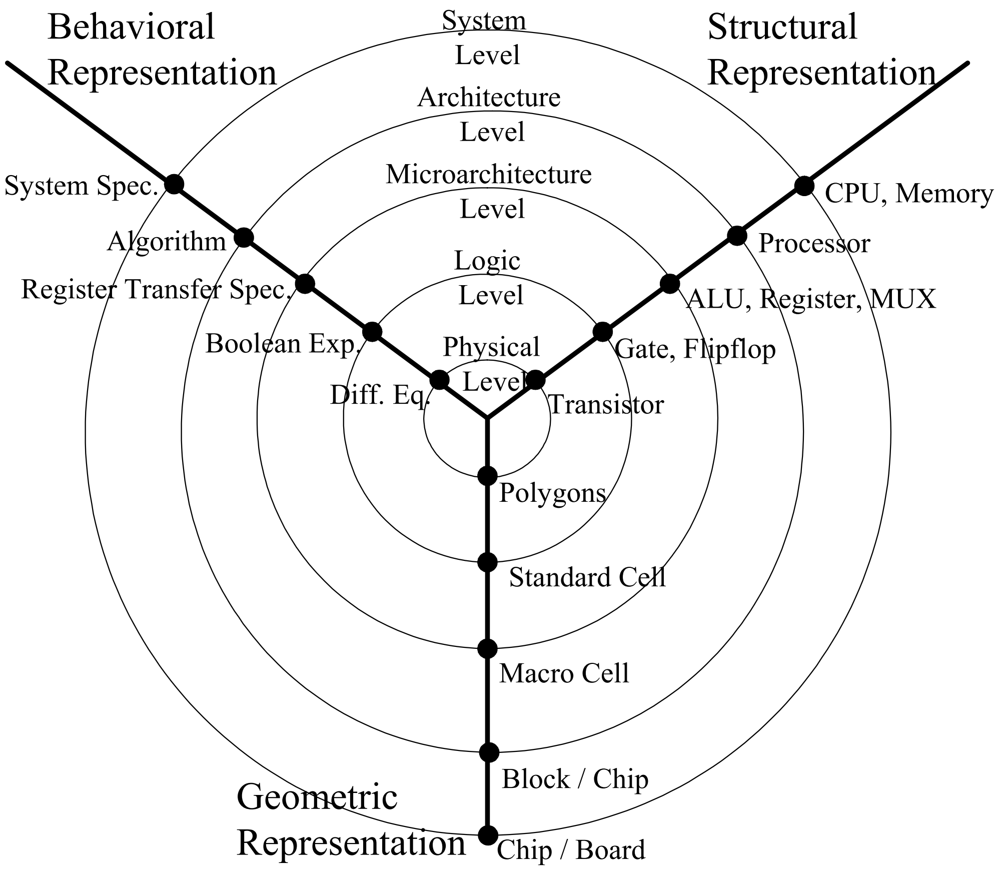


FIGURE 2 Gajski and Kuhn's Y-chart. It describes the relationship between the three representations of digital systems and the levels of description.

3.2 System Synthesis

Computing devices are permeating every portion of our world. From cell phones to laptops, embedded systems in planes, trains and automobiles and even high-performance supercomputers, computing devices are everywhere. The rate of growth in the number of computing devices in the world continues to scale faster than the number of humans. We have begun to expect low cost, high-performance, specialized computing systems. Our cell phones must be light, have long battery, and good reception. Our personal digital assistants (PDAs) are becoming more and more like tiny desktop computers. Present day desktop systems and supercomputers run tens to thousands of processes at a time. With every new generation of computers, we expect better runtime, power, energy, size, etc., etc. With such a wide variety of computing systems (with even more likely to emerge as the years go by), we need methods to explore the enormous system design space.

System synthesis is the methodologies and optimizations to implement an application as a computing system. An application is defined by a *specification*. A specification contains only the necessary requirement, constraints and functionality for the application. The system could be prespecified as a number of computational resources and the communication protocols between them or we may wish to completely derive and specify the system. Computing devices are becoming more integrated as the number of transistors per die doubles exponentially every 18 months. Computation elements such as microprocessors can now fit on a tiny corner of a chip, allowing memory (e.g. on chip cache), additional (possibly different) microprocessors, video processing units, reconfigurable computing devices and a bevy of other system resources on a single die. As a result of the on-die communication, the transmission of data between the devices is faster. For example, a system bus can operate at frequencies close to the rest of the computing devices, instead of at an extremely slower rate. This increases the number of possibilities – equivalently the size of the design space – in which a system can be tailored to an application.

The notion of system synthesis can encompass a broad area of system design. We could imagine a system that is comprised of various components – digital computing devices, analog devices, MEMS components, sensors, actuators, etc. The integration between these various components is often done in an ad hoc manner, most often using a system-by-system approach. We limit the scope of this work to digital systems.

We increasingly narrow our focus to the synthesis of a platform based computing system. User demands for high performance, lightweight and low power computing devices have created a market for application specific computing systems. However, the design of a custom computing system takes considerable amount of time and a tremendous amount of resources. We must explore a middle ground between time to market and system performance.

A *platform based computing system* [11] is a middle point between these two extremes. A platform based computing system consists of a number of programmable computing devices. For example, a platform will consists of a RISC processor, VLIW processor and an FPGA (see FIGURE 3). Each of the devices in the platform allows different design tradeoffs when it is programmed. For example, the maximum amount of parallelism of an FPGA is much larger than that of the VLIW processor, which in turn is larger than the RISC processor. On the other hand, an FPGA takes much longer to program as opposed to the processor counterparts.

A platform may be designed for a specific set of applications. For example, an audio platform may be designed for applications that predominately process audio e.g. an MP3 player. This platform would undoubtedly have dedicated (ASIC) analog to digital and digital to analog converters. Additionally, it would have a variety of general-purpose processing devices like a DSP processor and reconfigurable device. The general-purpose elements add flexibility to the platform. Flexibility allows the platform to be used towards a variety of different applications. The aforementioned audio platform could be used in a MP3 player, cell phone, video game console, etc., depending on the amount of flexibility in the platform.

As a platform becomes more flexible, it can handle a larger number of applications making it more financially viable to fabricate. The non-recurring engineering (NRE) costs of fabrications grow substantially with every process change. The NRE costs alone keep many companies out of designing and fabricating application specific computing systems as system on chip (SOC). Yet, the only reason they would wish to fabricate an SOC is if a current platform does not suffice to their needs for performance, power, weight, etc.

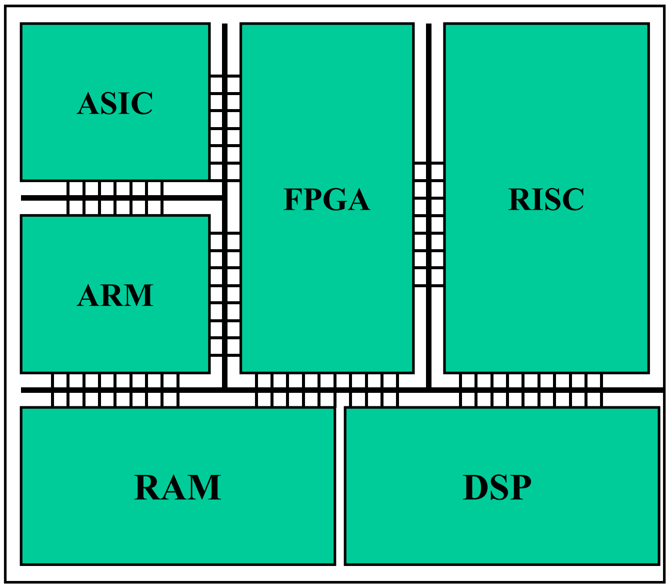


FIGURE 3 An example of a platform based computing system. There are 6 components to this platform – three microprocessors (RISC, ARM and DSP), an ASIC and a RAM. This platform uses a simple method of communication through a system bus.

Currently the design of platforms is done by hand in an ad hoc manner. We must look into efficient CAD methods for the design of platforms. Furthermore, we must develop automated methods for mapping a specific application to a platform. A translator from the application specification to hardware description is a necessary component for both the design of a platform and the mapping of an application to a platform.

3.2.1 Models of Computation

The language to describe an application greatly affects process of system synthesis. The more a language exposes the underlying hardware, the better the application programmer can customize the application to the system. Yet, the additional semantics of the language can limit the productivity of the programmer, as they must learn exactly what the semantics mean to the implementation. In addition, the semantics may enable the programmer to limit the design space; a design specified with less restrictive semantics may result in better system performance if the system synthesis engine is highly optimized. There is a huge tradeoff between the various semantics allowed by the application programming language. The underlying model of the application specification is called the *model of computation (MOC).*

There is a plethora of research in the development of models of computation. There are many different MOCs. Synchronous data flow (SDF), communicating sequential processes (CSP), process networks and discrete event (DE) are just some of the many MOCs; each has benefits and drawbacks. An emerging trend is heterogeneous MOCs, which allows different MOCs to be embedded within each other in a hierarchical manner. Ideally, this allows applications written in any MOC to be embedded into each other. Also, this enables the use of different MOCs to describe distinct portions of the application.

We focus on the control data flow graph (CDFG) as a MOC.

A data flow graph $G_{dfg}(V,E)$ is a directed graph that represents a set of operations O and their dependencies. For each operation $o_i \in O$ there is a vertex $v_i \in V$ i.e. there is a one to one correspondence between the operations and the vertices. Each operation has set of input operands and produces a set of output operands. If an output operand of operation o_i (vertex v_i) is in the set of input operands of operation o_j (vertex v_j), there is a directed edge $e(i,j)$. The behavioral description has arithmetic and logical operations.

A *control data flow graph (CDFG)* is a type of the data flow graph that contains information related to branching (if-else constructs) and repetition (while loops). There are many different models for CDFGs [12-14]. We choose to represent the CDFG as a two-level hierarchical sequence graph. A sequence graph is a hierarchy of data flow graphs [14]. The highest level in the hierarchy represents the flow of control in the application. The vertices of the highest level are associated with a data flow graph. The edges correspond to a possible flow of control in the application. Consider FIGURE 4, an if-then-else construct. The top level CDFG of the hierarchy has four vertices. Each vertex contains a data flow graph that is executed when the control flow reaches that vertex. The control edges model the flow of control of the if-then-else construct.

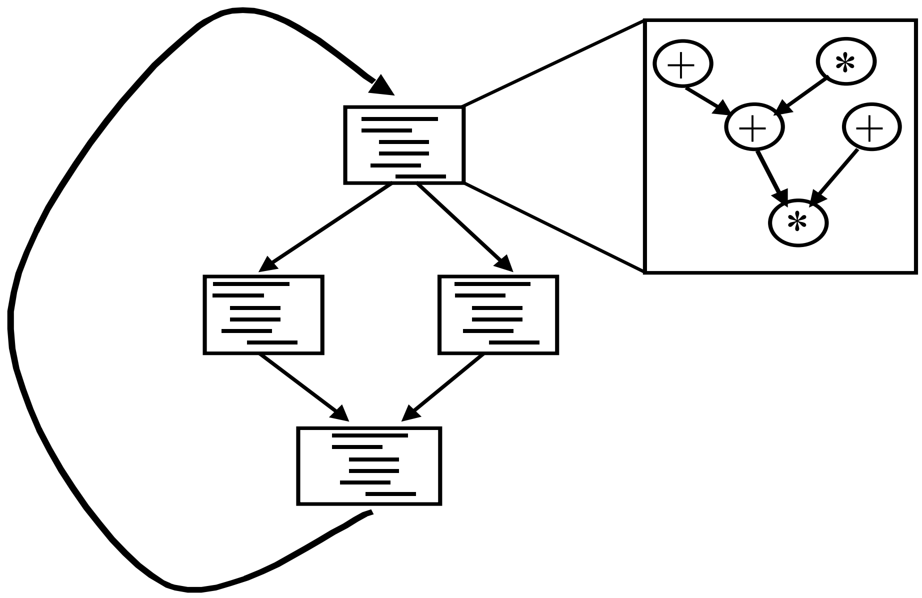


FIGURE 4 A Control Data Flow Graph

The CDFG offers several advantages over other models of computation. Most compilers have an internal representation (IR) that can easily be transformed into a CDFG. Therefore, this allows us to use the back-end of a compiler to generate code for a variety of processors. Furthermore, the techniques of data flow analysis (e.g. reaching definitions, live variables, constant propagation, etc.) can be applied directly to CDFGs. Finally, many high-level programming languages (Fortran, C/C++) can be compiled into CDFGs with slight modifications to pre-existing compilers; a pass converting a typical high-level IR into control flow graphs and subsequently CDFGs is possible with minimal modification. Most importantly, we believe that the CDFG can be mapped to a variety of different microarchitectures. All of these reasons make CDFGs a good MOC for investigating the performance of mapping different parts of the application across a wide variety of SOC components.

On the other hand, CDFGs only have the ability to describe instruction level parallelism. In order to specify a higher level of parallelism, another MOC must be used. But, we could embed CDFG into another MOC – one that can describe a higher level of parallelism. For example, we could embed CDFGs into finite state machines (FSM). Edward Lee's (UC Berkeley) *charts do something similar; they embed SDF graphs into FSM.

3.2.2 Hardware/Software System Partitioning

A popular means of accelerating an application's execution is to extract portions of it for realization in hardware. In the most ideal performance situation, the entire application would be implemented as hardware logic rather than as a software product, as hardware execution is many times faster than software. Unfortunately, the cost and size constraints of practical hardware systems make it impossible to realize large applications as hardware alone. Therefore, designers frequently consider solutions comprised of both software and custom hardware, working together to meet a set of performance constraints. The exploration of the system design space formed by combinations of hardware and software components is referred to as *hardware/software codesign*.

The foundations of codesign lie in embedded system design research, in which designers have often augmented a microprocessor with custom hardware to achieve boosts in system performance. The proven performance potential of these hardware/software systems, combined with the rapid evolution of powerful reconfigurable logic platforms, has sparked recent interest in architectures which combine a general-purpose processor with a reconfigurable coprocessor. Several industrial [15-17] and academic [8, 18, 19] platforms have been built combining microprocessor and reconfigurable logic on a single chip.

The ability to quickly and easily program hybrid hardware/software systems is of key importance if they are to be widely adopted. Ideally, hybrid application design should approximate the ease of application design for conventional microprocessor platforms. Without simple design methodologies, system performance would have to be considered against an unfavorably long time-to-market, as programmers would be bogged down in overly complicated implementation and verification cycles. Therefore, platform designers typically seek to enable specification of hardware/software systems from a single description, usually written in a high-level programming language [20-24].

The division of a system specification into hardware and software components is referred to as *hardware/software partitioning*. A partitioning of a system description impacts many aspects of the solution's quality, including but not limited to system execution time, total hardware area and cost, and power consumption. Unfortunately, as system partitioning typically occurs at a very high-level in the design flow, system characteristics resulting from a particular partitioning decision can only be estimated. The estimated characteristics of a partitioning are used to evaluate a cost function, which rates the overall quality of a given partitioning. Thus, the design space is explored in the quest for the highest quality system partitioning (i.e. the hardware and software partition that minimizes a cost function for an arbitrary set of constraints).

The rest of this section will provide an overview of hardware/software partitioning, and its connection to embedded and reconfigurable system research. We start by formally expressing the classic partitioning problem and then formulate Hardware/Software Partitioning problem mathematically, and then discuss the solutions and problem refinements that have been composed over the last decade.

3.2.2.1. The General Partitioning Problem

Partitioning is a fundamental CAD optimization problem. It is used at almost every level of abstraction during the synthesis of a digital system. The partitioning problem attempts to take a set of connected modules and group them to satisfy a set of constraints and optimize a set of design variables. During physical synthesis, partitioning is used during the floorplanning and placement tasks. In this case, the modules are gates that are connected by nets. We wish to create a partitioning such that highly connected gates are in the same partition. This reduces the amount of interconnect delay caused by the wires between the gates. As we move to higher levels of abstraction, the modules become larger; they move from standard cells to macro cells (logic level) to blocks (architecture level). Partitioning at higher levels of abstraction will impact the system performance in a more drastic way. A bad partitioning at the architecture level is hard to correct at lower levels of abstraction. Furthermore, the interconnect delay at the higher levels of abstraction is more pronounced. The delay between two blocks is orders of magnitude larger than the delay between two standard cells. Hardware/software partitioning lies at one of the highest level of abstraction for the design of digital system. A good system partitioning is essential for the overall quality of the circuit.

A *k-way partitioning problem* is the basis for any variant of the partitioning problem. Given a set of modules $M = \{m_1, m_2, \dots m_n\}$, the k-way partitioning problem finds a set of clusters $P = \{p_1, p_2, \dots p_k\}$ such that $p_i \subseteq M$ for $i \rightarrow 1$ to k, $\bigcup_{i=1}^{k} p_i = M$ $p_i \cap p_j = \varnothing$ for $i \rightarrow 1$ to k, $j \rightarrow 1$ to k and $i \neq j$ (i.e. the clusters are mutually disjoint). The partitioning solution must satisfy a set of constraints and optimize an objective function. The objective function and constraints are subject to the type of problem we are trying to optimize. When $k = 2$, the problem is called *bipartitioning*.

The partitioning must be evaluated using some sort of estimation function. The estimation function $E(P) = DP(dp_1, dp_2, \dots, dp_q)$ takes a partition P and returns a set of design parameters DP. The design parameters are properties of the circuit such as the area, power, throughput, latency, etc. The estimation function is used to tell us if the constraints are met and also to evaluate the optimization function. The estimation function is extremely important, especially at the system level. The ideal estimation function would run through the entire synthesis flow. Unfortunately, this is an extremely time consuming task. A large design can take days to fully synthesize. Obviously, this is unfeasible, especially if we wish to evaluate a large number of partitions. Therefore, the quality of the partitioning directly depends on the precision of the evaluation function.

There are many methods and heuristics to solve the partitioning problem. Alpert and Kahng [25] give an excellent survey on formulations and heuristics to solve the netlist partitioning problem. The next section gives a detailed formulation of the Hardware/Software Partitioning problem.

3.2.2.2. Formulation of the Hardware/Software Partitioning Problem

One of the clearest mathematical formulations of the Hardware/Software Partitioning problem was presented in [26]. The following is a slight modification of the problem as presented in their work.

Given: a set of functions $F = \{f_1, f_2, \dots f_n\}$ that comprise the complete functionality of the system, as well as a set of performance constraints $C = \{C_1, C_2, \dots C_m\}$, where $C_i = \{G_i, V_i\}$, $G_i \subset F$, and $V_i \in Real > 0$. (V_i is a time constraint on the maximum execution time allowed for all functions in function group G_i. Each function f_i can represent system functionality at nearly any granularity: as coarse-grained as an entire program task or as fine-grained as a single micro-operation on a processor.)

Define: a hardware/software partition as $P = \{H, S\}$, where $H \subset F$ and $S \subset F$, and $H \cup S = F$, and $H \cap S = \varnothing$. The hardware size of H is defined as the area needed to implement the functions in set H in hardware. The performance of the system is defined as the total execution time of a set of functions G is defined as the total execution time of the functions in G, or

$$Performance(G) = \sum_{f_i \in G} execution\_time(f_i)\,.$$

A *performance-satisfying partition* is a P such that $Performance(G_i) \le V_i$ for all $i = 1 \dots m$.

Using the inputs and definitions provided, the original Hardware/Software Partitioning problem is to find a performance-satisfying partition $P = \{H, S\}$ such that the hardware size of H is minimal. With such a partition, the system would be able to satisfy its performance constraints (for which additional hardware beyond the microprocessor was required) with a minimal amount of coprocessor area. The problem is a subset of the General Partitioning problem for a system containing one or more additional ASIC or FPGA resources (as the set H could be subdivided into functionality across many pieces of hardware).

Over time, the goals of the Hardware/Software Partitioning problem have changed depending on the design situation. For instance, a general computing system (for which there are few or no performance constraints) might contain a RISC microprocessor and a fixed-size FPGA coprocessor. On such a system, the nature of the problem would be different, as the design constraint would be hardware size, and the goal would be to minimize execution time of F. We define an *area-satisfying partition* to be one in which the hardware size of $H \leq A$, where A is a given maximum hardware size. The Modified Hardware/Software Partitioning problem would be to find an area-satisfying partition $P = \{H, S\}$ such that *Performance(F)* is minimal. Other systems seek to minimize other characteristics, such as power consumption of the system [27]. These modifications are usually represented as straightforward modifications of the mathematical formulation presented here.

The Hardware/Software Partitioning problem is NP-complete, as it requires the exploration of a design space that is exponential in size (relative to the number of functions in F). (The Hardware/Software Partitioning problem can be restricted to the Precedence Constrained Scheduling problem (PCS), by setting the amount of time of execution of hardware and software tasks to be the same. PCS finds an m-processor schedule for an ordered set of tasks subject to precedence constraints such that deadline D is met. PCS is NP-complete, and therefore so is Hardware/Software Partitioning [28, 29].) The time of determining a partitioning solution is therefore proportional to $|F|$, and therefore the granularity at which functions are considered will largely impact the performance of a partitioning heuristic. However, as function granularity becomes coarser and the design space is decreased, design quality is potentially sacrificed. In an extremely coarse-grained case, where only two very large tasks are considered ($|F| = 2$), the partitioning of the system would be very easy. However, considering the partitioning of subdivisions of each task might have led to higher quality system.

The hardware/software partitioning must be performed early in the design cycle, as the functionality residing on a piece of hardware must be determined before the hardware can be synthesized. As a result, the design characteristics (including hardware size and performance of a partition P) must be estimated at the earliest stages of design. The accuracy of a given partitioning is as only as good as the estimation of the design parameters for that partitioning. The estimations provide insight into the quality of a solution. They must be fast, to keep the execution time of the partitioning heuristic within reasonable bounds.

Therefore, the quality and performance of the partitioning solution depend largely on the granularity at which functions are considered, the performance and accuracy of design estimators, and the performance and effectiveness of the search heuristic utilized. These factors make the Hardware/Software Partitioning problem very complex, and difficult to solve for a general system.

3.2.2.3. Initial Models and Solutions for Hardware/Software Partitioning

Early investigations into codesign for embedded systems frequently mentioned characteristics that a good system partitioning would possess, although the researchers made no attempt to automate the solution to the partitioning problem. Researchers at Carnegie Mellon University [30] described a type of system codesign that extracted portions of application software for hardware realization, directed at acceleration of the application. They developed a model for system-level simulation and synthesis, including a transformation capability allowing the generation of design options. Their proposed partitioning would take place at a coarse-grained task level rather than at the operation level, although no specific algorithm was describe in that work. They characterized the quality of a task-level partitioning by three criteria: the impact of the partition on total system execution time, the difference in execution time between a given task in hardware as opposed to software, and the total cost of the custom hardware required to realize a task. The Ptolemy group at UC Berkeley also developed an early codesign methodology and framework [31], but in this work the designer was required to partition the design manually. (Subsequently, Kalavade and Lee have developed techniques to automate design partitioning, as well as to solve an extended form of the problem [32].)

Gupta and DeMicheli developed one of the earliest algorithms to automate a search of the design space for a hardware/software partitioning [33]. Their algorithm operated on an operation-level granularity, and was greedy in nature. Starting with an initial solution where all functionality was implemented in hardware ($H = F$, $S = \varnothing$), their partitioning heuristic selected operations for movement into software based on the cost of communication overhead. Movements were not taken if they did not improve the cost of the current system partition, or if they violated the given set of imposed timing constraints. The algorithm iteratively improved the partition until no cost-improving move could be found. The decision to start with an all-hardware partition was chosen to ensure that no infeasible design was considered. (An initial all-hardware solution is assumed to satisfy the performance constraints of the system. From this point, only moves which continue to satisfy performance constraints are accepted.) An assumption of the Gupta/DeMicheli algorithm is that most inter-function communication will happen between successive operations. Therefore if an operation is moved from hardware to software, its successor operations are given priority consideration as possible movement candidates.

Due to the greedy nature of the Gupta/DeMicheli algorithm, and its resulting inability to make short-term increases in system cost to achieve long-term cost minimization, it was easily trapped in a local minimum. Therefore, sub-optimal partitioning solutions were typically arrived at. Additionally, the algorithm frequently created very costly hardware consuming many resources, due to the initial partition being an all-hardware solution. (If cost-reducing movements from hardware to software were exhausted, the remaining design would be accepted. Typically the algorithm completed before the hardware area was reduced to a pragmatic quantity.) To overcome the limitations of this initial development, hill-climbing heuristics such as simulated annealing were subsequently explored.

Ernst and Henkel developed a hill-climbing partitioning heuristic that sought to minimize the amount of hardware used, while meeting a set of performance constraints [34]. Their work operated on the basic block level of functional granularity, slightly coarser than the work of Gupta and DeMicheli. Like the Gupta/DeMicheli algorithm, they started with an initial partitioning that was improved on subsequent iterations. However, to escape convergence to a local minimum, they utilized simulated annealing to explore design cost. Unlike greedy heuristics, simulated annealing often accepts changes which decrease the quality of a design, in hopes of achieving a more optimal final design. The process of simulated annealing is controlled by a temperature parameter, which begins at a high value and decreases as the system "cools" and stabilizes. Initially, during high temperature, moves which increase system quality are always accepted, and moves which decrease design quality are accepted randomly. As the temperature approaches zero, only moves which decrease system costs are accepted. Ernst and Henkel began the process with an all-software partition, seeking to minimize hardware costs by starting with less hardware. Obviously, their initial partition generally violated the performance constraints of the system. In order to prevent annealing before a performance-satisfying partition has been reached, they used a heavily weighted cost function that provided high penalties for violating runtime constraints. This choice proved effective in minimizing hardware costs. To provide performance enhancement estimates for hardware implementation, Ernst and Henkel utilized simulation and profiling information to determine the most frequently executed and computationally intensive regions of functionality.

Like Ernst and Henkel, the early work of Peng and Kuchcinski [35] also utilized simulated annealing in order to approach a more optimal partitioning. Their algorithm provided a more general multi-way partitioning (as opposed to the two-way partitions of other works), which enabled a decomposition of system functionality into a number of clusters (each resulting in a different hardware or software component). They modeled the system functionality as a Petri Net, which was then used to construct a graph representing the components and their communication. The weight of each node represents the cost of building this component, and the weight of an edge between two nodes represents the cost of implementing the data connection. The goal of their partitioning was to decompose the graphs into a set of sub-graphs such that the sum of the weights of all cut edges is minimized, while the total weights of the sub-graphs are balanced. Their work, although promising in its application to general system partitioning, lacked any direct representation of system performance, and also described no specific hardware minimization technique.

Vahid, Gong, and Gajski [26] built upon the work of their peers by developing a technique that partitioned a set of very coarse granularity tasks into software and a minimal amount of hardware. The task granularity was on the order of statement-level control constructs, such as loops and function bodies. Large variables were also considered as "tasks" to be partitioned. In this work, the goal of the partitioning is closely aligned with the formulation of the Hardware/Software Partitioning problem mentioned earlier. They describe a partitioning algorithm named *PartAlg*, based on a hill-climbing heuristic (much like the work of [34, 35]), which takes an input set of performance constraints and a maximum hardware size C_{size}. Every design possibility is weighed with a cost function equal to the weighted sum of runtime performance and hardware size violations. In other words, every violation of a performance constraint, or any violation of input maximum hardware size C_{size} contributes to the cost of a design. The assumption is that there exists a threshold input hardware size C_{size} at and above which the design cost of *PartAlg*'s solution will be equal to zero (as performance constraints will be met, and the hardware area constraint will be large enough that the design will satisfy it). Under this assumption, their solution is a nested loop algorithm. The outer loop is a binary search of the solution cost of the inner loop's *PartAlg* for various values of input area constraint C_{size}. The idea is to find the smallest C_{size} at which the partitioning returns a zero-cost design; this C_{size} will be the minimal hardware partition satisfying the performance requirements of the system. Although their heuristic reduced total hardware area compared to the Gupta/DiMicheli and hill-climbing approaches, the execution time of the algorithm suffers due to the execution of an iterative improvement algorithm roughly *log(n)* times. The coarse granularity of the operations considered helps reduce the execution time, but may also lead to less optimal design results.

3.2.2.4. Refined and Enhanced Heuristics

Subsequent rigorous heuristics to model and solve the Hardware/Software Partitioning problem are typically refinements of the previously mentioned approaches. In this section, we briefly outline some of the more recent improvements to the techniques described above.

The translation of the Hardware/Software Partitioning problem into a set of integer programming (IP) constraints was described in the work of [36]. The extraction and creation of coprocessing hardware is broken into two stages. The first phase solved the traditional Hardware/Software Partitioning problem by estimating the schedule time for each functional node, and the second phase generated a correct scheduling for hardware/software-mapped nodes. The IP model devised by the authors is one of the most flexible partitioning heuristics presented, as it accounts for multi-coprocessor technologies as well as hardware sharing and interface costs. Additionally, a high-level synthesis schedule is produced for the functional nodes, guaranteeing system execution-time constraints. This is one of the most optimal and complete solutions proposed, and the algorithmic execution remains reasonable due to the utilization of better metrics (resulting in fewer iterations).

Kalavade and Lee, whose previously cited work on the Ptolemy project contained no automated partitioning heuristic, introduced the Global Criticality/Local Phase (GCLP) algorithm to solve the two-way partitioning problem [32] for tasks of moderate to large granularity. The authors note that two possible objective functions could be used to map a task to hardware or software: minimization of the execution time of that node, and minimization of the solution size (hardware or software area) of the node's implementation. To this end, the authors devise a global criticality measurement, which is re-evaluated at each step of the algorithm to determine whether time or area is more critical in the design. As the list of functional tasks is traversed, the global criticality measurement is checked to determine the current design requirement. If time is critical, the mapping minimizes the finish time of the task; otherwise the resource consumption of the task is minimized. In addition to the global system requirements, local optimality is sought by classifying each task as either an extremity (meaning it consumes an extreme amount of resources), a repeller (meaning the task is intrinsically preferred to either have a software or hardware implementation), or a normal task. This classification of each task, and its weighty consideration in the choice of hardware or software mapping, represents the local phase of a given task. The running time of the GCLP algorithm is extremely efficient ($O|N|^2$), and the partitions it determines are no more than 30% larger than the optimal solution (as determined by an ILP formulation). (The work of Kalavade and Lee also formulates and heuristically solves an extension of the Hardware/Software Partitioning problem, which simultaneously determines whether a task should be implemented in hardware or software, as well as the best speed/area hardware implementation tradeoff for the task. Since the focus of this work is largely an extension of GCLP to hardware synthesis of a high-level task, it is beyond the scope of this section.)

Just as Kalavade and Lee incorporated a dynamic performance metric into their partitioning decision, Henkel and Ernst extended their previous work on the COSYMA environment [34] to incorporate a dynamic functional granularity [37]. Whereas their earlier work had been limited to basic block granularity, the authors' new partitioning method allowed the dynamic clustering of fine-grain tasks (at the basic block or instruction level) into larger units of operation (as large as the procedure/subroutine level). The rationalization for having a flexible functional granularity are that large partitioning objects should contain whole control constructs (in the form of loop bodies or procedures), and that only a few moves should be necessary (between hardware and software) to determine a good partition. The authors' innovation amounts to a hierarchical search of the design space, and the fast retrieval of a good solution by applying this modification to their earlier simulated annealing approach.

As an alternative to the simulated annealing-based iterative-improvement heuristics explored by their contemporaries, Vahid and Le extended the Kernighan-Lin (KL) circuit partitioning heuristic to explore the design space of Hardware/Software functional partitioning [38]. The chief advantage of the KL heuristic is its ability to overcome local minima without making excessive numbers of moves. The basic strategy of KL is to make the least costly swap of two nodes in different partitions, and then to lock those nodes. This continues until all nodes are locked. The best partition *bestp* is selected from this set. All nodes are subsequently unlocked, and the previous *bestp* becomes the starting point for the next set of node swaps. This swapping, locking, selection of *bestp*, and subsequent unlocking and looping continues until no subsequent improvement over the former *bestp* exists. (Usually this takes approximately five passes.) Vahid and Le extend the KL heuristic by replacing its cost function with a combined execution-time/area/communication metric, by redefining a move as a movement of a functional node across partitions (rather than a swap of nodes), and by reducing the running time of "next move selection" to be constant. Via these means, the authors are able to achieve nearly equal-quality partitions to simulated annealing in an order of magnitude less time. The running time of the algorithm is accelerated by consideration of task nodes at subroutine-level granularity.

3.2.2.5. Partitioning for Reconfigurable Hardware/Software Systems

The development of reconfigurable hardware and its realization in field programmable gate array (FPGA) technology has motivated continuing interest in hardware/software codesign platforms and frameworks. The evolution of FPGAs from small custom hardware to low-cost, powerful platforms able to store more than a million gates of logic has led to their adoption as coprocessor hardware. The flexibility and programmability of configurable devices allow them to be personalized to enhance the performance of a given application, or set of applications. Exciting research has investigated the potential performance boosts for real-time embedded systems, as well as general-purpose personal computer applications. A large number of processor/reconfigurable hardware architectures and design frameworks have been devised as a result.

The circumstances upon which a programmable device is reconfigured shape the potential performance gains realizable via its use, and mold the partitioning strategy that should be used to realize an effective co-design. In this work we differentiate between run-time reconfigurable (RTR) devices, whose functionality is changed during application execution, and semi-static reconfigurable devices. Unlike RTR devices, semi-static reconfigurable devices cannot be reprogrammed during an application's execution. However, a semi-static reconfigurable device might be modified between executions of different applications, or before subsequent executions of the same application, in order to provide the maximum hardware support for an application's current environment. (Note that this classification of device reconfiguration is potentially independent of the actual type of FPGA hardware used. Indeed, a single FPGA might be used as either a semi-static reconfigurable device or an RTR device, or each at different times.)

For semi-static reconfigurable devices, the flavor of the Hardware/Software Partitioning problem is not very different from the original formulation described previously. For each application execution, a different partitioning into hardware and software components might be configured onto the system. However, the decomposition of an application into software and reconfigurable hardware implementations could be devised and implemented using any of the aforementioned heuristics.

However, for architectures consisting of a processor and one or more fully dynamic RTR devices, the nature of the partitioning problem changes, as a spatial as well as temporal partitioning must be performed. The hardware/software partition must satisfy area constraints (the size of the FPGA configuration required at any given time) as well as performance constraints (where system execution time now includes the latency of every reconfiguration, as well as data/configuration communication and hardware execution time). Any violation of the hardware area constraint would lead to a subsequent reconfiguration, which might lead to a violation of system performance constraints. Therefore, the added dimension of time increases the complexity of partition design and evaluation.

Many reconfigurable architectures and design platforms leave the actual hardware/software partitioning choice to the system designer, or allow the designer to interactively explore the design space of partitioning options. Celoxica's DK-1 compiler interactively profiles the code and annotates the code with area and delay estimates, enabling the designer to make an informed partitioning decision [39]. The Napa C compiler, which targets National Semiconductor's NAPA1000 chip, uses user annotations in the application source code to guide system partitioning [21]. TOSCA's co-design evolution is driven by its Exploration Manager (EM) tool, which maintains a complete history of design alternatives explored by the user [40]. TOSCA's framework allows both direct intervention of the user in selecting a partition, and also an automatic partition selection guided by built-in evaluation criteria. Other design frameworks incorporating manual design partitioning include CASTLE [41] and POLIS [42].

PRISM (an acronym for Processor Reconfiguration Through Instruction Set Metamorphosis) was one of the earliest design environments seeking to augment the performance of a microprocessor via the synthesis of new operations [19]. The key innovation of this work was the choice to implement only instructions in the cooperating hardware logic. For instance, on a RISC-based system with a frequently executed subroutine *foo(a, b)*, the function *foo* could potentially become an architecture level instruction acting on two input registers (whose inner details are synthesized on the supporting hardware). Rather than begin with the ISA of the given processor, the PRISM compiler would initialize the application instruction set to be null. At compile time, the PRISM compiler would identify a set of operations that best represent the needs of the application. Any chosen operation not already supported by the microprocessor would be synthesized on the coprocessor. (As PRISM was a proof-of-concept implementation, the automation of this operation selection was never described.)

Another new system partitioning technique favored by reconfigurable architecture researchers is the utilization of accepted compiler techniques to find parallel regions of code, and then implement these regions in reconfigurable fabric. The search for reconfigurable candidate code begins with the identification of loop bodies that are frequently executed as well as computation-intensive. These loop bodies can be identified via static analysis or with a profiling tool. The rest of the partitioning involves pruning the list of candidate loops to satisfy system feasibility requirements and also maximize performance.

The Nimble compiler [43] extracts candidate loops as kernels for possible hardware implementation on RTR coprocessors, and applies hardware-oriented compiler optimizations (loop unrolling, fusion, pipelining) to generate multiple optimized versions of each loop. In determining which loops to synthesize, as well as which hardware implementation to use for each loop, the compiler is guided by a global cost function incorporating execution times on hardware and software as well as hardware reconfiguration time. The area constraints of the FPGA are considered, as violation would result in further reconfiguration. On Nimble's platform, only one loop can be synthesized in hardware at a time, and loops are executed purely sequentially, even if realized on the reconfigurable fabric.

DEFACTO, an end-to-end design environment, identifies loop candidates for realization in hardware via existing array data dependence analysis, privatization, and reduction recognition techniques [20]. Like Nimble, the DEFACTO compiler generates many specialized hardware versions of a given loop body. Working in tandem with an estimation tool, the design manager uses locality analysis to create a feasible partition that minimizes data communication and synchronization. DEFACTO specifically seeks to minimize the number of costly hardware reconfigurations, aiming to create hardware exhibiting semi-static behavior. Unlike Nimble, DEFACTO can produce a multi-way partitioning, supporting a single processor, multiple FPGA architecture.

Berkeley's BRASS project compiles to the Garp platform, which is composed of a MIPS microprocessor and a datapath-optimized reconfigurable coprocessor [23, 44]. Their compiler identifies candidate loops for reconfigurable acceleration. For each accelerated loop, they form a hyperblock containing the most frequently executed control paths of the loop body. This hyperblock will be the portion of the loop body that becomes implemented on the reconfigurable coprocessor. The predicated execution model of the hyperblock is realized in hardware via multiplexors, which act as the select instructions.

Markus Weinhardt and Wayne Luk of Imperial College also investigate loop synthesis techniques for reconfigurable coprocessors [45]. They refer to their approach as pipeline vectorization. Like the previous loop-identification approaches, compiler techniques are utilized to discover good candidates and subsequently resize them for high-performance FPGA implementation. Weinhardt and Luk only generate hardware from innermost loops with regular true loop-carried dependences (non loop-carried dependences, as well as anti- and output dependences, do not pose a threat to loop pipelining in hardware, and are thus not considered). Therefore, only loops whose current values are dependent upon previously written values (or in which no dependence on previous iterations exists) are considered. This dependence must be regular, meaning it spans the same distance (in iterations). A pipelined version of the loop, able to execute multiple iterations concurrently, is then synthesized on the hardware (containing circuit feedback loops to explicitly handle true loop-carried dependences). Realizing that this loop-investigation technique has limited applicability, Luk and Weinhardt apply classical loop transformations such as unrolling, tiling, and merging to reshape innermost loops into suitable hardware candidates. Therefore, the applicability of their technique is widened to innermost loops that would previously not have been considered, as they were too large or too small for hardware synthesis.

3.3 Architectural Synthesis

Architecture synthesis is the process of creating a structural microarchitectural-level representation from a behavioral architectural description of an application. A *structural representation* defines an exact interconnection between a set of architectural resources. An *architectural resource* is a memory, data path, or steering logic. A memory component is a method of storing the state of the circuit. A register is an example of memory component. A data path component performs an arithmetic or logic operation e.g. an ALU, multiplier, shifter, etc. Steering logic is used to route data. For example, a multiplexor propagates a particular piece of data (correspondingly a set of signals) depending on a condition. A control unit (controller) issues control signals to the direct the resources.

Architectural synthesis can be performed using any number of different methods. Additionally, a designer can add additional constraints or optimization objectives. For example, we may want to produce a circuit that requires the least amount of area. In this case, our objective function would be to minimize the area. Many other constraints have been considered during architectural synthesis. Throughput, power, clock frequency, and latency are some of the more popular optimization variables.

The architectural synthesis problem can be defined in the following manner: given a CDFG, a set of fully characterized architectural resources and a set of constraints and an optimization function, determine a fully connected set of resources (a structural representation) conforms to the given constraints and minimizes the objective function. The architectural synthesis problem can be split into two sub-problems: scheduling and resource binding.

Scheduling determines the temporal ordering of the vertices in the CDFG. Given a set of operations with execution delays and a partial ordering, the scheduling problem determines the start time for each operation. The start times must follow the precedence constraints as specified in the CDFG. Additional restrictions such as timing and area constraints may be added to the problem, depending on the architecture one is targeting.

Two important scheduling algorithms are the *as soon as possible (ASAP)* and the *as late as possible (ALAP)* algorithms. The ASAP algorithm has run time $O(|V||E|)$. The vertices are topologically sorted. This insures that when a vertex is visited, all of its predecessors have been visited. The start time of a vertex earliest possible time the operation can be started without violating the precedent constraints. The ASAP algorithm optimally solves the unconstrained scheduling problem. Just as the ASAP algorithm gives the earliest possible time that an operation can be scheduled, the ALAP algorithm gives the latest possible time that an operation can be scheduled. Given a maximum latency, the ALAP algorithm starts by performing a reverse topological sort of the vertices of the CDFG. The start time of a vertex is determined by finding the minimum start time of all of the operations successors.

The *slack* of an operation is the difference in between the start time of the operation scheduled by the *as late as possible* (ALAP) algorithm and the *as soon as possible* (ASAP) algorithm. The slack gives an idea as to how much freedom the operation has to be scheduled.

The scheduling problems becomes NP-Complete [28] when resource constraints are added. There are many different kinds of constraints that we can put on the scheduling problem. We could schedule for minimum latency given a set of resource constraints. Or we could consider the dual of that problem, which attempts to minimize the number of resources under latency constraints. When we assume that a single type of resource can be used for any operation, the problem becomes a precedence-constrained multiprocessor scheduling problem. This problem is intractable and remains intractable even when all the resources have unit delay.

Hu [46] provided a lower bound on the number of resources for the precedence-constrained multiprocessor scheduling problem. Hu also describes a greedy algorithm that gives an optimal result when the CDFG is a tree. It starts at the first time step and schedule the vertices on the longest path first. It continues until all of the operations are scheduled.

Unfortunately, the CDFG is not often a tree and no exact polynomial time algorithm exists to solve the general scheduling problem. As a result, many heuristics have been proposed to solve the problem.

One popular scheduling heuristic is list scheduling. The list scheduling algorithms use a priority queue to hold the unscheduled nodes. At each step, an operation is popped from the queue and scheduled. This continues until all of the operations have been scheduled. The priority queue is sorted based on some measure of urgency to schedule the operation. A common sorting function is based on sorting the vertices corresponding to the distance to the sink. The list scheduling algorithm runs in linear time, making it a quite attractive method of scheduling.

Force directed scheduling is another class of scheduling heuristic. It was first proposed by Paulin and Knight [47] in 1989. The force directed algorithm works in a constructive manner. They proposed two algorithms for force directed scheduling. One algorithm minimizes latency under resource constraints and the other minimizes resources under latency constraints. The algorithms can be differentiated by the order in which they choose to schedule the operations. First consider scheduling under latency constraints. The algorithm schedules all of the available resources at each time step during an iteration. At each time step, the resources are matched with operations that have the largest force. The minimum resource algorithm schedules the operations based on the calculation of the force on the operations. At each step, it takes the operation with the lowest force and schedules it. Essentially, it tries to schedule operations to minimize the local concurrency.

Other scheduling algorithms have been proposed that employ an iterative improvement technique. The technique starts with an initial scheduling and transforms the schedule by changing the start time of some of the operations. Trace scheduling [48] and percolation scheduling [49] are two examples of iterative improvement scheduling algorithms.

Resource binding is the assignment of hardware resources to one or more operations; it is an explicit mapping between operations and resources. The goal of resource binding is to minimize the area by allowing multiple operations to share a common resource. Resource binding may be done before or after scheduling. It is most often done after scheduling, so we will focus on that methodology. When it is done after scheduling, the scheduling will give some limitations on the possible resource bindings. For example, operations that are scheduled to execute during the same time step cannot share the same resource. To be more precise, any two operations can be bound to the same resource if they are not concurrent, i.e. are not scheduled in overlapping time steps, and the operations can be implemented using the same resource. For example, if a resource is an ALU, both an addition and subtraction operation can be bound to an ALU resource.

The resource binding can greatly affect the area and latency of the circuit as it dictates the amount of steering logic and memory components of the circuit. We first consider the slightly simplified problem of resource binding for circuits that are dominated by the arithmetic units.

When the circuit is arithmetically bounded, the delay and area of steering and memory components are miniscule when compared to the arithmetic units. One method of modeling the problem is through a resource compatibility graph. Two operations are *compatible* if they can be scheduled using the same resource. The *resource compatibility graph* has a one-to-one correspondence between the operations and the resources of the graph. There exists an edge between two vertices if the corresponding resources are compatible.

Since we are focusing on the arithmetically bounded problem, an optimum resource binding is one that minimizes the number of resources needed to bind the operations. This is equivalent to finding the minimum number of cliques in the resource compatibility graph. A conflict graph is the complement of a compatibility graph. Minimizing the number resources can be cast as a minimum clique partitioning (each partition must be a clique) problem of the compatibility graph or into a minimum coloring problem of the conflict graph [14]. The resources can be functional units (resource allocation), registers (register allocation), etc. The graph coloring problem for the conflict graph of an interval graph can be solved in polynomial time via the left_edge algorithm [50].

CHAPTER 4 A Framework for System Synthesis

4.1 Introduction

The framework that we are investigating lies between the realm of a traditional microprocessor compiler and architectural synthesis. A compiler allows a number of application languages to be compiled into an *intermediate representation (IR)*. An IR is representation of the application [51]. There are different forms of IRs each of which lie somewhere between the original application specification and the representation used to program a computing device of a platform.

The SUIF compiler [52] is a publicly available compiler infrastructure from Stanford University. It provides a high-level IR (HIR). Constructs like "for" loops, "while" loops, and if-then-else clauses remain visible in the HIR. In addition, arrays and array accesses are explicitly available. Due to the availability of the array information, memory layout transformations are possible on the HIR. In addition, loop transformations like unrolling [51, 53-55] and software pipelining [56-58] are often done at this level.

Machine SUIF [59] is a backend for the SUIF infrastructure. The backend takes a machine independent representation and creates machine level representation. Machine SUIF starts by transforming the HIR from SUIF to a medium-level IR (MIR). The MIR is in the form of instruction lists and *control flow graphs (CFG)*. A CFG $G_{cfg}(V_{cfg}, E_{cfg})$ represents the control relationships between the set of basic blocks. For each basic block there is a unique vertex $v \in V_{cfg}$. An edge $e(u,v) \in E_{cfg}$ means that a control can be transferred from basic block u to basic block v. Each basic block contains a set of instructions that are executed atomically. Each basic block contains at most one *control transfer instruction (CTI)* that dictates the flow of control after the instructions of the basic block have executed i.e. it determines the appropriate control edge to follow. Two examples of a CTI are a conditional branch and a jump.

The instruction list of a basic block can be modeled by a *data flow graph (DFG)*. A DFG models the interdependencies of the instructions of the basic block. A DFG

$G_{dfg}(V_{dfg},E_{dfg})$ has a vertex $v \in V_{dfg}$ for every instruction in the basic block. If a output or result of instruction i is used as an input of instruction j, there is a directed edge $e(i, j)$. A *Control Data Flow Graph (CDFG)* is a CFG with the instructions of the basic blocks expressed as a DFG. Therefore, each vertex of a CDFG (corresponding to a specific basic block) has a DFG modeling the dependencies of the instructions in the basic block. The CDFG is a common input representation for architectural synthesis.

Architectural synthesis is the transformation from a behavioral description to a structural hardware description. A behavioral description is a description of the flow of an application through a set of computations. A structural hardware description is an explicit description between a set of hardware entities. A hardware entity is an exact description of the workings of the entity [60]. Architectural synthesis is part of a larger flow that can be used to produce a mask to manufacture an ASIC, provide a bit stream to program an FPGA, and produce an output to “program” other devices that are considered hardware.

The framework that we developed produces a synthesizable behavioral hardware description that can be used as input to a behavioral synthesis tool. Specifically, we produce an output in behavioral VHDL using IEEE specification 1076.3 [61]. This allows the use of many commercial behavioral synthesis tools such as Synopsys' Behavioral Compiler and Monet from Mentor Graphics.

In the next section, we discuss the details of the framework. Additionally we give some optimization problems that can be applied at the compiler level to improve the hardware description. Furthermore, we discuss the different architecture decisions that we must make during the translation to behavioral HDL.

4.2 Hardware Compilation

We start this section by introducing the basic ideas of our method of hardware compilation. *Hardware compilation* is the process of transforming an application level language to a form that can be synthesized onto a hardware component. A hardware component is any device that can be programmed by using an architecture-level behavioral description. The word “hardware” is an overloaded term; we define hardware as any device that can be completely programmed or fabricated from a hardware description language. There are many well-developed paths to take an architectural-level behavioral HDL description and use it to create a mask – the blueprints to fabricate an ASIC. Additionally, we would consider an FPGA hardware because it also has a set of tools to transform an HDL into a bitstream to program the component.

We call any optimization unit a *region*. A region is piece of the IR that we wish to focus our attention. We wish to transform a region into an *entity*. An entity is a set of input and output ports and a description of the relationship between the ports. An entity is a high-level entry point into the hardware synthesis flow. The *interface* of an entity is the input and output relationship of the entity to the outside world. A VHDL *architectural body* describes the relationship between the ports. It gives the functionality of the entity. Every entity is associated with at least one architectural body.

The input and output ports that can serve a variety of purposes. The main use of ports is for transferring data between entities, but we can also use ports to dictate the flow of control, a clock for the circuit, a reset, etc.

The *data input ports* of an entity corresponding to a region would be any data that is consumed in the region and generated by an instruction outside of the region. Data is *generated* if it is the output of some instruction. If some data *d* it is an input to an instruction *i*, we say that *d* is *consumed* by *i*. A *data output port* is needed for any data that is generated by some instruction in the region and consumed by at least one instruction outside of the region.

An entity can have many different architecture bodies. This allows different implementations of an entity. In addition, the architecture body can be modeled using a structural representation, behavioral representation or a mix of the two representations. We use a behavioral representation to model the entity describing a basic block region. The interconnections between the basic block regions are described using a structural representation. In effect, we are using a two level hierarchy for synthesis. The first (highest) level is a structural representation modeling the data and control interconnections between the basic block regions. The second (lowest) level is the modeling of the basic block regions as a basic block entity. The architecture body of a basic block entity is described using a behavioral representation. Each basic block entity is then synthesized using Synopsys Behavioral Compiler to yield a structural representation of the basic block. After the every basic block entity is synthesized, the entire design is realized as a two level hierarchical structural representation. We can then feed the design to a high level synthesis engine – we use Synopsys Design Compiler – to get to a logic level structural (gate level) representation. Then, we can hand off the design to any physical design tool to realize the final implementation of the application. FIGURE 5 shows a high level view of the entire flow, from application specification through system, architectural, logic and physical synthesis. The subsequent sections give in depth details on the entire flow.

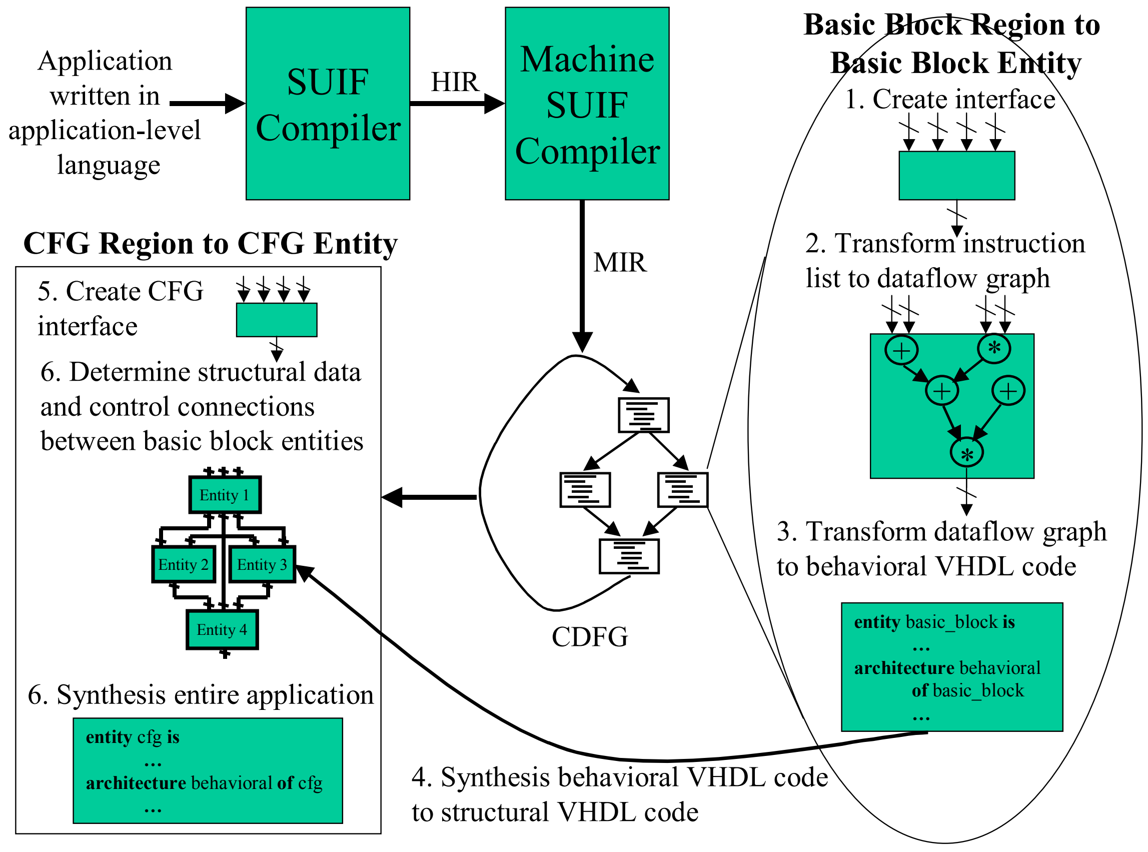

Application written in application-level language
SUIF Compiler
HIR
Machine SUIF Compiler
Basic Block Region to Basic Block Entity
1. Create interface
2. Transform instruction list to dataflow graph
3. Transform dataflow graph to behavioral VHDL code
entity basic_block is … architecture behavioral of basic_block …
4. Synthesis behavioral VHDL code to structural VHDL code
MIR
CDFG
CFG Region to CFG Entity
5. Create CFG interface
6. Determine structural data and control connections between basic block entities
Entity 1
Entity 2
Entity 3
Entity 4
6. Synthesis entire application
entity cfg is … architecture behavioral of cfg …

FIGURE 5 The flow from application specification to its hardware realization.

4.3 Basic Block Synthesis

We start the discussion of synthesizing the Machine SUIF IR [62] by describing the translation of a basic block to an entity. Equivalently, we describe the translation of a basic block region to a basic block entity. The CfgNode is the Machine SUIF description of basic block. The operations of the basic block are held in a linked list. The linked list enforces an ordering on the instructions. The first instruction in the linked list is executed before the second instruction, which is executed before the third instruction, and so on. There are many possible correct orderings of the instructions. In addition, many instructions can be executed in parallel. We will discuss the dependencies between the instructions in a later section. The instruction itself has an opcode that describes its functionality, a set of input operands and an output operand. We use the architecture independent opcodes from the SUIF Virtual Machine (VM) [63].

The operands are in the form of a variable symbol, a virtual register or an immediate operand. A *virtual register (VR)* (sometimes called a pseudo-register) is the notion of assigning a unique register to every point where data is generated. When assigning virtual registers, we have no regard for the "hard" registers of the final target microarchitecture. We assume that we have an infinite number of registers. The exact number and use of registers is decided during architectural synthesis. Similarly, a back-end for a processor microarchitecture (e.g. RISC) assigns the VRs to the hard registers in the register banks during the register allocation stage. By using VRs, the operands of the instructions show only the true dependencies – a read after write (RAW) dependencies. A *variable symbol* is a source generated variable or a compiler generated temporary variable [64]. An *immediate operand* represents a signed or unsigned integer immediate value.

Each instruction can be categorized as an arithmetic instruction, a control transfer instruction (CTI) or a memory instruction. An *arithmetic instruction* takes input operands and produces output operand(s). The opcode of the instruction dictates the functionality of that instruction. Some common arithmetic instructions are add, multiply, divide, shift, etc. The Machine-SUIF SUIFvm Library [63] describes all of the opcodes for the machine independent virtual machine used in our framework. All of the arithmetic opcodes from the SUIF VM have corresponding functionality in IEEE 1076.3 VHDL standard.

A control instruction determines the flow of the program. A jump instruction and a conditional branch are two examples of a CTI. In a microprocessor architecture, a CTI is any instruction that modifies the program counter. There is no analogue to a program counter when we synthesize to hardware (unless we choose to synthesis a controller that uses a program counter). The flow of control is dictated by a set of controllers. We will describe different methods for synthesizing control in a later section.

A *memory instruction* is any instruction that moves data between local memory and main memory. Local memory could be a register, register bank or scratch pad memory. The access latency for main memory can greatly affect the performance of the system. There are techniques (e.g. caching, prefetching) to reduce the effective latency of the main memory accesses. We assume that the read/write access times for main memory are on the order of a few cycles. A nice future study would be the effects of different memory access latencies on the design; we discuss this study in more detail in our future work section (CHAPTER 8). Additionally, we assume that the main memory is a coherent, memory architecture i.e. every piece of data that we access from memory is the correct data and any data that we write to memory will be properly updated throughout the system.

The translation to VHDL is done from the CFG representation. As described in the previous section, each CfgNode of the CFG contains a list of instructions. First, we will describe how we translate such a list. The instructions we encounter will be of the type arithmetic instruction or memory instruction. There is at most one CTI instruction per CfgNode. We will describe how to handle the CTI instruction separately.

The entity for an instruction list and correspondingly, a CfgNode, has a set of input ports and output ports. The input ports are any VR operands that are consumed within the instruction list but not generated by an instruction in the instruction list. Additionally, if there is variable symbol that is consumed in the instruction list, there is an input port for that variable. There is an output port for a variable symbol if it is generated or, similarly, there is an instruction that has a destination operand that is a variable symbol. If a variable symbol is consumed and generated, there is an input/output port. Each of the variable symbol ports is directly connected to a single register that always holds the data for a variable symbol across the CFG. Additionally, there is an output port for every VR that is generated within the instruction list and consumed outside of the list.

If there exists a memory instruction, then we must add input and output ports corresponding to the interface of the main memory. The main memory can be any type of memory, (e.g. SRAM, DRAM, etc.) and located anywhere within the platform. The memory may be realized as an embedded RAM in an FPGA, hard core (ASIC) on a SOC, chip on a PCB platform or as any fully specified memory component of a platform library. Presently, we assume that the main memory interface is that of a scratch pad memory[2] of the Synopsys DesignWare library. Our framework allows for different memory interfaces with only minimal changes. The actual amount of main memory depends on the amount of data storage needed. The data storage would be the memory necessary to program the device, the appropriate input data, and storage for the output data. It is easy to envision that another component of the platform (e.g. a microprocessor) will use program this reconfigurable component and control the overall flow of the application on the platform. The reconfigurable component that we are synthesizing could be realized to a functional unit of some VLIW processor. We believe that design space exploration for main memory is an extremely interesting research topic, however it is out of the scope of this work.

We have defined the entity interface for the instructions in the basic block. Our next task is the creation of a behavioral VHDL architecture corresponding to the entity. We define a VHDL variable for every VR that is local. A VR is *local* if it is generated by an instruction in the basic block and never consumed by an instruction outside of the

[2] We use the synchronous write-port, asynchronous read-port RAM (DW_ram_r_w_s_dff) from the DesignWare library.

basic block. We now have defined every operand as either an input, output and input/output ports of the entity or as a variable in the architectural description.

The arithmetic instructions are translated directly to the IEEE 1076.3 specification. The VM "add" instruction is translated to "+". Likewise, the various shift, rotate, multiple VM instructions are translated into there IEEE 1076.3 counterparts. Not all of the VM instructions can be implemented using Synopsys Behavioral Compiler. Presently, Behavioral Compiler can only synthesize a subset of the 1076.3 specification. For example, Behavioral Compiler cannot synthesis an arbitrary division. Therefore, we change every division to a multiply instruction. This does not yield correct functionality of the compiled application, but we imagine that Behavioral Compiler will be able to synthesis the entire 1076.3 in the near future.

The memory instructions are handled by specifying the main memory interface. We define two macros, *main_memory_write* and *main_memory_read*. The different memory instructions use these macros during the translation to VHDL. For example, the STR (store) operand from the SUIF VM has two input operands. One input operand specifies the data that is written to main memory. The second input operand gives the memory address or the location in memory to store the data. This instruction would use the main_memory_write macro. Additionally, I could associate the latency of the memory with different types of shared memory schemes.

Now that we have described a method for synthesizing a basic block, we must give ways to control the flow of execution through the basic blocks and the best manner to pass data between basic blocks.

4.4 Controlling the Flow of Execution

Now that we have translated every instruction in the basic block to its equivalent VHDL construction, we must provide a methodology to handle the flow of control.

To handle the control flow, we add an additional input port to every entity. We call this input port the EXECUTE port. When the EXECUTE port is set, the entity runs. When it is not set, the entity is in an idle state. It is easy to imagine that the idle state would put the hardware in low power mode. There are some interesting low power hardware design methodologies, but any further discussion is out of the scope of the work. We must add a mechanism to direct the control flow i.e. a controller. We focus on two types of control – distributed control and centralized control.

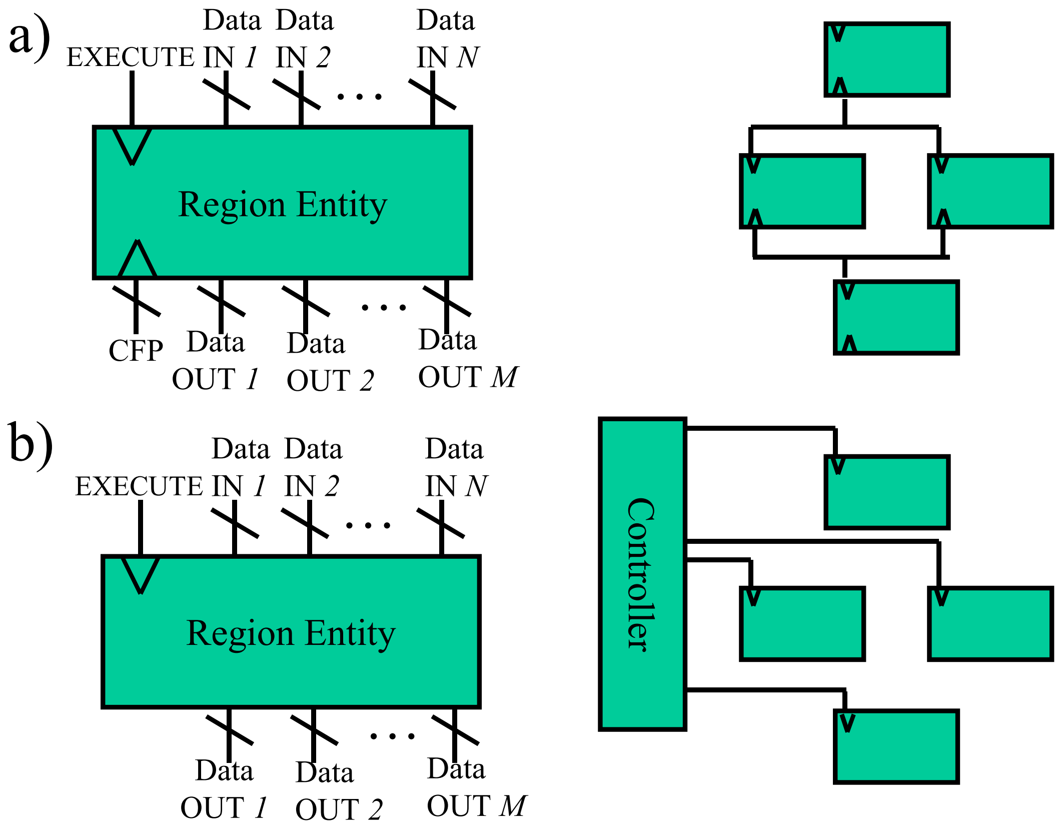


FIGURE 6 a) Distributed Control b) Centralized Control

Distributed control has many different entities that control the path of execution. Each entity has a local controller that determines the next control node in the execution sequence. Therefore, there are direct connections between control nodes. Additionally, each control node has a set of *control flow ports (CFP)*. There is a CFP for each of the different control nodes that may follow this node in execution. Equivalently, there is a CFP for each control edge of the CFG. A CFP connects to the execute port of other entities. FIGURE 6 a) illustrates a simple example of distributed control for an if-then-else clause.

Centralized control has one controller that determines the (possibly multiple) entities that execute at any given instant. As with distributed control, each control node has an execute port that initiates the execution of the VHDL architecture of the entity. Unlike distributed control, every execute port of control node is connected to the controller. Centralized control closely resembles the separation of control flow and data flow assumed by most high-level synthesis engines for data path dominated circuits. FIGURE 6 b) gives an example of centralized control

Often, the control flow depends on the result of the computations local to the currently executing control node. For example, the condition for control flow may depend if a local variable is greater than zero. In this case, the control node must transmit the condition to the centralized controller. Then, the controller will determine the next control node in the execution sequence.

Of course, one could imagine many other control schemes. For example, there could be many distributed controllers, each of which controls a small number of nodes. A single centralized controller could control these distributed controllers. Hybrid local/global control schemes are an interesting area of research but are out of the scope of this work.

4.5 Data Communication Schemes

In addition to determining the type of control for the CDFG, we must determine the method of data communication between the regions. Once again, there is a centralized and distributed method of data communication. A *centralized data communication scheme* passes the data through a centralized storage area such as a register bank or RAM block, depending on the amount of data. This allows a memory hierarchy scheme where data can be cached and large amount of data can be accessed by the CDFG.

A *distributed data communication* scheme passes the output data from the currently executing entity directly to the inputs of the control node(s) that might need the data later. The output of the entity may connect to multiple other entities including itself (in the case of loops).

As with most of engineering decisions, there are benefits and drawbacks to consider for each of the communication schemes. The distributed data communication scheme is simple to implement, as you do not have to worry about interfacing to bus and memory protocols. Additionally, the distributed scheme has direct connections, meaning that the communication between control nodes will occur quicker compared to the centralized scheme; data passing does not involve writing to and reading from a central memory bank. Yet, the centralized scheme allows a sharing of resources. The distributed scheme will have more connections (interconnect), many of which will not be active at a particular time leading to a waste of communication resources. Furthermore, the increased connectivity between control nodes may have a negative impact on the circuit's area.

We study different methods to minimize data communication in CHAPTER 6. In our framework, we have implemented a locally distributed, globally centralized data communication architecture. The values that are generated and used by the CFG are stored in local registers. This allows quick read and write access for the values. Any value that is generated outside of the CFG and used by the CFG is stored in a global memory. In such a case, the CFG will make a memory call to a global memory component. In addition, any value that is generated in the CFG and used outside of the CFG will be written to global memory. Global memory could consist of many main memory components. Then, we would need to determine the best way to distribute the data across the memory components. Multiple memory components may enhance memory performance, though this would require additional optimizations to determine the best manner to distribute the data. We make the simplifying assumption that there is only one global memory component. Hence, we use the terms global memory and main memory interchangeably.

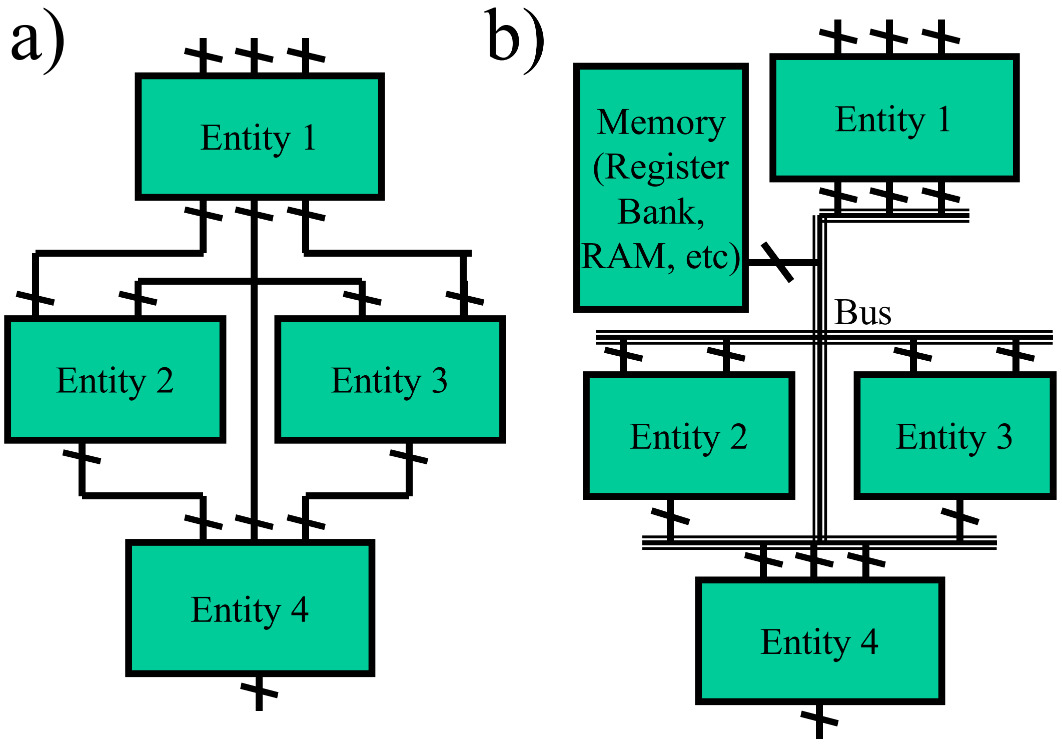


FIGURE 7 a) Distributed data communication b) Centralized data communication

Our scheme requires that we determine the *local values* – values that are both generated and used locally to the CFG and the *global values* – any value that is generated outside of the CFG and used within the CFG or generated within the CFG and used outside of it. We can leverage Machine-SUIF to determine the local and global values.

A global value is any memory access instruction as translated by Machine-SUIF into the Machine-SUIF SUIFvm architecture. This allows the use of pointers, though we assume that the global memory will handle any arbitrary address that we may give it. A formal analysis of the possible pointer values in order to restrict the address space would be quite beneficial, as it would reduce the size of the memory giving greater efficiency in terms of silicon usage as well as reducing the memory latency. We refer an interested reader to the many recent works of Rinard et al. [65-73] for a techniques and other references for pointer and alias analysis. Machine-SUIF will also translate any array accesses as global values. It is possible to treat the entire array as a set of local values, though our framework treats all arrays as global values.

The local values are translated to virtual registers (VR) by Machine-SUIF. In addition, we treat parameters and static variables as local values. We create a register for each variable and assume that the system will load the initial values into those registers before executing the hardware that we synthesize. Since we are synthesizing each basic block separately, we must determine what local values that the basic block generates and uses. In addition, we must create a way to transfer the local values between basic blocks. A simple method for handling the transfers would create a register for each local value. Then, a basic block that uses the value would be connected to the register that holds the values and retrieve the value from the corresponding register. Also, any basic block that generates a value would be connected to a register. With the reuse of values and different paths of control, it is possible that a register is connected to a basic block that will never use the value. This would require increase the number of the number of multiplexors and control logic needed to handle accesses to a register. In turn, this increases the area and latency of the registers. We explain a scheme to handle this problem in CHAPTER 6. We defer the exact determination of passing data through the CFG to that chapter.

In this chapter, we attempt to give the reader an understanding of the underlying framework that we assume throughout the rest of the work. The framework is not complete. We have tried to convey the shortcomings and the assumptions that we make. We have attempted to make a standard framework that can be extended as well as enhanced using the SUIF infrastructure. The framework conforms to the SUIF 2 methodology of creating independent passes so that they can be replaced or interleaved to suit the user. The framework is still in the research stage; it is constantly be improved by a group of graduate students at UCLA and UCSB.

CHAPTER 5 Instruction Generation

5.1 Introduction

Computational devices are becoming more complex. The number of transistors on a single die – dictated by Moore's Law – is increasing exponentially. This allows a *system-on-chip* – a variety of different computing devices interacting as a complete system on a single die. An additional benefit of Moore's Law is the decreasing cost of computations leading to a ubiquity of embedded systems. Much like a system-on-chip, these embedded systems are a complex interaction of many different computational devices, embedded within a larger entity e.g. a car, telephone, building, etc.

With the proliferation of computing systems comes a need for specialization; each use of the system is tailored for a specific set of applications. For example, a digital system embedded within a cellular phone will encounter DSP-type applications. It most likely needs to perform operations like analog to digital conversions (and vice-versa), FFT, and filtering. Therefore, if we customize the embedded system to such operations, we gain increased performance, power/energy reduction and smaller silicon footprint. This tends towards the use of ASICs for such systems.

Yet, an ASIC is extremely inflexible; once the device is fabricated, the functionality cannot be changed. For example, if a new cellular communication standard emerges, we must throw away our phone and buy a new one customized for the new standard. However, if a new standard appears and the embedded system is flexible, we can change the functionality of the system to handle the migration from one standard to another. Another increasingly important trend is time to market. The initial market share of a product that is released first is inherently larger than that of a product that is released much later. In fact, we are seeing that time to market is becoming vital to the success of the product. The public accepts new products at a fanatical pace. Products once took 10+ years to gain consumer acceptance (e.g. television, radio). Now, products permeate the market in under a year (e.g. DVD, MP3 players). Both of these trends accentuate the need for flexible devices like a general purpose processor. The more applications/standards that the device can service, the more companies will use it, as they want to reach the market quickly and service as many markets as possible.

We have conflicting forces pushing digital systems in two seemingly separate directions. On one hand, computing systems must be specialized to meet the performance, power, energy, and area constraints. In this regard, ASICs are the answer. On the other hand, time to market and flexibility constraints push for general purpose systems. Obviously, there is a tradeoff between the flexibility (general purpose) and performance (application specific) of the system. We need methods to allow us to perform the tradeoffs between these two metrics. We must be able to customize a system towards the tasks that it will most likely perform and give it the flexibility to adapt over the course of its lifetime.

Customized instructions are one way to explore the tradeoff between customization and flexibility. They approach the problem through the customization of a general purpose device. If we know the *context* – a set of applications that will likely run on a system, we can look for commonly occurring, computational sequences within the various applications of the context. These customized instructions can be optimized for high-performance, low energy/power consumption and/or small area. For example, it is well known that the computational sequence, multiply-accumulate (MAC), occurs frequently in DSP applications. The MAC unit would be an ideal candidate for a customized instruction when the context is DSP applications. In this paper, we develop a systematic method for customized instruction generation and explore the theoretic aspects of *instruction generation* – the process of finding commonly occurring computational patterns within a context.

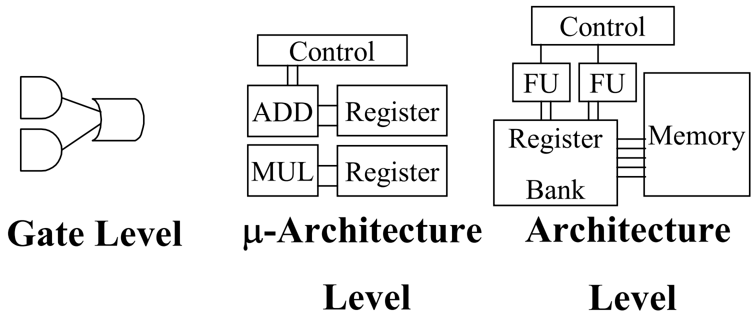
Customized instructions can serve to optimize two general purpose devices – the processor and the FPGA. *Application specific instruction set processors (ASIPs)* take a small processor core and add customized instructions to service the specific context of the applications. The PICO project [74] aims to automatically generate the customized instructions based a specific application. They use a VLIW core and generate nonprogrammable hardware accelerators (NPA), which are akin to systolic arrays. The interface between the NPAs and the processor core is automatically synthesized. The user is responsible for identifying the customized instructions in the form of loop nests. Another product, Tensilica's Xtensa processor [75], takes the customized instructions as an input in the form of a *hardware description language (HDL)* called the Tensilica Instruction Extension (TIE) Language. It incorporates the instruction into a compiler allowing the user to execute the instruction through an intrinsic function. Our instruction generation algorithms can serve the Xtensa and PICO frameworks to automatically find the customized instructions.

The FPGA is another general purpose computing device, albeit quite different from a general purpose processor. The FPGA has the benefit of adapting its architecture directly to the application that it implements. Data intensive applications running on an FPGA can achieve up to 100x increased performance as compared to the same application running on a processor [76-79]. This mainly comes from the ability to customize at the architecture level. The architecture of an application running on an FPGA is completely flexible, whereas the processor architecture is fixed.

Yet, the architecture of an FPGA is still tailored for the general case. Adding *macros* to the architecture could customize the FPGA. Macros are hard or soft reconfigurable computational sequences. A *hard macro* is a fixed ASIC core embedded into the fabric of the FPGA. The embedded multipliers of the Xilinx Virtex series are an example of a hard macro. A *soft reconfigurable macro* is a sequence of computations that are implemented as a fixed entity on the FPGA fabric. Examples of soft reconfigurable macros are the components of the Xilinx CoreGen library.

In this sense, reconfigurable architectures are moving away from reconfiguration exclusively at the gate level and moving towards a *hybrid reconfigurable architecture*. Hybrid reconfigurable architectures contain reconfigurability at multiple levels of the *computational hierarchy* (see FIGURE 8). The computational hierarchy is the level of abstraction that computations may be implemented. One level of the computational hierarchy is the gate or Boolean level. At this level, every computation is built up from the Boolean (gate) level computations. The FPGA is an example of a device that functions at this level. A device can also be reconfigured at the microarchitecture or architecture level. These levels of computational hierarchy have coarser basic computational units. A basic unit at the microarchitecture level is on the level of an arithmetic function. PipeRench [80] and RaPiD [81] are examples of this type of reconfigurable system. At the architecture level, the basic unit of computation is more coarse grained, for example the RAW project [82].

Reconfigurability at the various levels of the computational hierarchy gives many tradeoffs in terms of flexibility, reconfiguration time, performance, area and power/energy consumption. A fine-grained reconfigurable device (gate level) is extremely flexible; it can implement any application. But, the flexibility comes at a cost. The routing architecture must allow a connection from any part of the chip to any other part of the chip. Switchboxes are used to enable this sort of flexibility. The switchboxes are composed of many transistors to enable a flexible routing. Compared to a direct connection, it is apparent that switch boxes add much overhead to the area, delay (performance), and power/energy consumption. Furthermore, the implementation of an arithmetic unit (e.g. an adder) on a fine-grained reconfigurable device consists of programming each gate of the arithmetic unit. Since the gates are designed to be extremely flexible – you can implement any Boolean function on any gate – the arithmetic unit will be large, slow and power/energy hungry as opposed to the same arithmetic unit that is designed specifically for implementing that function, as is the case for a device that is reconfigurable at the microarchitecture level. On the other hand, the area, delay, power/energy consumption of implementing a Boolean function favors a gate level reconfigurable device. If we implement a one bit "and" function on a device that is reconfigurable at the architectural level, the function will be over designed. It will implement an "and" instruction and 31, 63 or 127 "and" operations will be unnecessarily performed, depending on the size of the instruction. As you can see, there are benefits for reconfigurability at various levels of the computational hierarchy. A hybrid reconfigurable architecture allows us to mix and match these levels to tailor to the applications at hand.



Reconfigurability	Bit	Byte	Instruction (32 –128 bits)
Basic Unit of Computation	Boolean Operation (and, or, xor)	Arithmetic Operation (add, multiply)	Functional Operation (ALU, MAC)
Communication	Connections through switchboxes	Bundles of wires, registers	Bus, memory

FIGURE 8 A comparison between three levels of the computational hierarchy. The gate level is the most flexible with the architecture level being the least flexible. But, the architecture level has the fastest reconfiguration time. The performance, area and power/energy consumption depend on the type of operation being implemented.

Examples of hybrid reconfigurable systems include Garp [83], which couples a MIPS-II processor with a fine-grained FPGA coprocessor on the same die. The *Strategically Programmable System (SPS)* architecture [84] combines memory blocks, *Versatile Programmable Blocks (VPBs)* – embedded ASIC blocks that perform complex instructions – into a LUT-based fabric. Many other academic projects can be called a hybrid reconfigurable system, for example Dynamically Programmable Gate Array (DPGA) [85] and Chimaera [9].

In addition, several industrial projects fall into the category of hybrid reconfigurable systems. One example is the Virtex-II devices from the new Xilinx Platform FPGAs, which embed high-speed multipliers into their traditional LUT-based FPGAs. Also, the CS2112 Reconfigurable Communications Processor (RCP) from Chameleon Systems, Inc. contains reconfigurable fabric organized in slices, each of which can be independently reconfigured.

To understand how instruction generation works in hybrid reconfigurable systems, we consider the SPS project. SPS consists of VPBs embedded into a LUT-based fabric. It is targeted towards a specific context. An example of a specific SPS architecture is shown in FIGURE 9. Because the VPBs are hard macros, implementing an operation on them as opposed to on the reconfigurable fabric will give lower power and energy consumption. Additionally, the VPBs require no time to program. Hence, the SPS architecture can be reconfigured faster than an FPGA. Furthermore, the performance of the operation on the VPB will be much better than the performance of the same operation on the reconfigurable fabric. But, we must carefully consider the type of functionality for the VPB. If the applications of the context never use the VPBs, then they are wasting space on the chip.

A designer of an SPS architecture can specify the functionality of the VPBs towards the targeted context. The designer must consider the types of operations that make up the applications of the context. The operations must occur frequently. Furthermore, the operations must give performance or other benefits (e.g. reduced power consumption) when implemented as VPB as opposed to on the reconfigurable fabric. There are many tradeoffs to consider when choosing the functionality of the VPB. A tool to determine these tradeoffs would help the designer pick the functionality. Ideally, the tool would automatically determine the functionality of the VPBs based on the given context. This is exactly the problem of instruction generation.

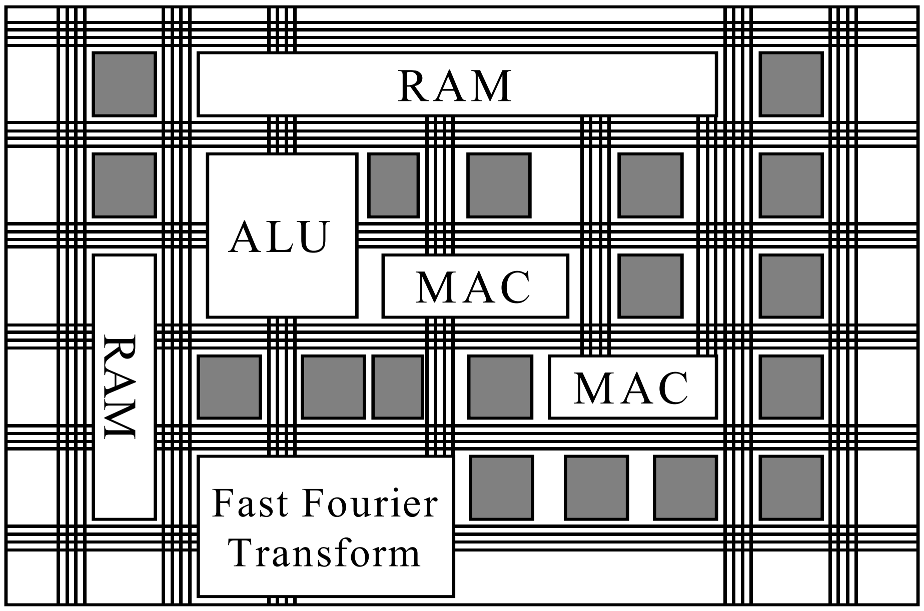


FIGURE 9 Example of the Strategically Programmable System (SPS) – a hybrid reconfigurable system. Functional units are embedded within a reconfigurable fabric. This SPS architecture would be specific to the DSP context.

Instruction generation is also useful for the generation of soft reconfigurable macros. Soft reconfigurable macros act as a "black box" to a designer. The input, output and functionality of the soft reconfigurable macro are given to the designer, but the actual implementation of the macro on any specific reconfigurable architecture is abstracted away. In this sense, the terms soft reconfigurable macro and soft reconfigurable IP (intellectual property) are synonymous. A soft reconfigurable macro is customized for each reconfigurable architecture. For example, the Xilinx CoreGen library has IPs like decoders, filters, memories, etc. Each of these functions is customized for the architecture on which it runs. The CoreGen components are generated based on what Xilinx believes their customers will use. Also, the functionalities of the CoreGen IPs are well-known entities. Instruction generation is useful to find unknown, irregular computational patterns that are not immediately apparent from looking at the code of the applications in a context.

Soft reconfigurable macros give many benefits. They allow the designer to work at a higher level of abstraction. Instead of dealing with basic arithmetic operations like addition, multiplication, etc, the designer can implement the application using function or block level structures. Matlab works at this level and is extremely popular in the signal processing community. Additionally, the soft reconfigurable macros can be highly optimized. A person familiar with the underlying architecture can design each macro. Therefore, the macros will be more efficient than if someone who doesn't understand the underlying architecture implemented them or if they were designed from basic operations using the synthesis flow. Finally, the compilation time of the applications using the macros is reduced. The macros can be pre-placed and routed. Therefore, the tool or designer must only determine the location of macro on the reconfigurable fabric. The placement and routing of the macro – an extremely time consuming task – is unnecessary.

In summary, instruction generation is an extremely important concept for context-specific architectures. Whether the underlying architecture is derived from a general purpose processor or a reconfigurable architecture, instruction generation is essential to the flexibility and performance of the system. Furthermore, instruction generation in the form of soft reconfigurable macros can reduce the time for synthesizing an application to reconfigurable architectures.

In the next section, we look at the role of instruction generation in a reconfigurable system compiler. The following section formalizes the problem of instruction generation. We discuss the specifics of the formulation of the instruction generation problem in Section 5.3. Section 5.4 proposes an iterative, constructive algorithm for simultaneous template generation and matching using graph profiling and edge contraction. In Section 5.5, we present experimental results. Then, in Section 5.6, we discuss related work. We conclude in Section 5.7.

5.2 Integration of Instruction Generation with Reconfigurable System Synthesis

Reconfigurable system synthesis has two different uses. One flow is architecture generation and another flow for application mapping. FIGURE 10 gives a high level overview for both of these flows. We use the SPS co-compiler as a representative reconfigurable system compiler.

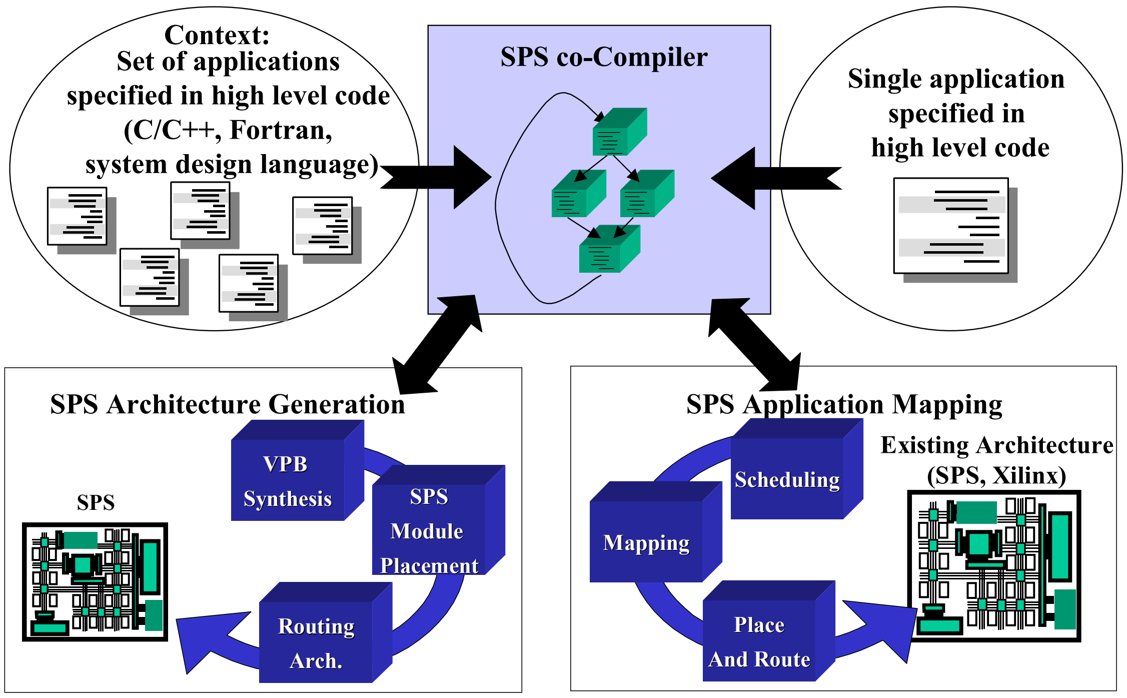


FIGURE 10 Two flows for the SPS system. The flow on the left is the SPS architecture generation flow. The application mapping flow is on the right. The co-compiler interfaces with both flows.

In both flows, the first task is to translate each of the applications of the context or the single application into a form that is suitable to interface with architecture generation and application mapping, respectively. The applications are given in a high level language. For example, the applications could be written in C/C++, Fortran or some other system design language, like SystemC [86], SpecC [87] or Esterel [88]. We must choose an intermediate representation (IR) for the co-compiler. We use the *control data flow graph (CDFG)* as the intermediate representation. A CDFG is a directed, labeled graph with data nodes corresponding to operations (addition, multiplication, shift) and control nodes that dictate the flow of the program. Control nodes allow branches and loops. The edges between the nodes represent control and data dependencies. The CDFG offers several advantages over other models of computation. The techniques of data flow analysis (e.g. reaching definitions, live variables, constant propagation, etc.) can be applied directly to CDFGs. Also, many high-level programming languages (e.g. Fortran, C/C++) can be compiled into CDFGs with slight modifications to pre-existing compilers; a pass converting a typical high-level IR into control flow graphs and subsequently CDFGs is possible with minimal modification. Therefore, we can leverage the front-end of many existing compilers for our reconfigurable system compiler.

On the other hand, CDFGs only have the ability to describe instruction level parallelism. In order to specify a higher level of parallelism, another model of computation (MOC) must be used. But, we could embed a CDFG into another MOC – one that can describe a higher level of parallelism. For example, we could embed CDFGs into finite state machines (FSM). Lee's *charts [89] do something similar; they embed synchronous data flow graphs into a FSM.

Instruction generation has a role in both the architecture generation flow and the application mapping flow. In the architecture generation flow, instruction generation determines the functionality of the hard macros – the VPBs in SPS. During application mapping, instruction generation is needed to determine soft reconfigurable macros.
There are many other tasks in the architecture generation and application mapping flows. For example, we must determine the exact placement of the VPBs, their interface with the reconfigurable fabric, the routing architecture, the exact number of each type of VPB

and so on. We focus on generating the VPB functionality and refer the interested reader to our other papers on these subjects [84, 90-94].

5.3 Problem Formulation

The instruction generation problem is an example of *regularity extraction*. Regularity extraction attempts to find common sub-structures (templates) in one or a collection of circuits (graphs). There are many applications for regularity extraction, including, but not limited to, scheduling during logic synthesis, system-level partitioning and FPGA mapping and placement. In our case, the templates we extract will be our instructions and the collection of graphs is the set of applications – the context – that we wish to target our system towards.

We aim to build a general profiling technique for simultaneous template generation and matching, which is applicable to any task that uses a directed labeled graph. We target the generation and matching algorithm towards instruction generation and selection, though the methods we present are general enough to be applied to any regularity extraction problem represented by a directed labeled graph.

5.3.1 Template Matching

Regularity refers to the repeated occurrence of computational patterns e.g. multiply-add patterns in an FIR filter and bi-quads in a cascade-form IIR filter. A *template* refers to an instance of a regular computational pattern.

We model an algorithm, circuit or system using a digraph *G(V,E)*. The nodes of the graph correspond to an instance of basic computational units. Examples of node types are add, multiply, subtract, etc. Each node has a label consistent with the type of operation that it performs. The edges of a graph model the dependencies between two operations. For instruction generation, the graph under consideration is a data flow graph.

It should be noted that systems/circuits must often be modeled by hypergraphs. A *hypergraph* is like an ordinary graph, but each *hyperedge*, connects multiple vertices instead of two as in a normal digraph. Extending our algorithms to consider hypergraphs is fairly straightforward.

We consider labeled digraphs in this work as we mainly target instruction generation, which use compiler data flow graphs; data flow graphs can be sufficiently modeled using labeled digraphs.

There are two general problems associated with template matching:

Problem 1: *Given a directed, labeled graph* G(V, E), *a library of templates, each of which is a directed labeled graph* T_i(V,E), *find every subgraph of* G *that is isomorphic to* T_i.

This problem is essentially equivalent to the subgraph isomorphism problem simplified due to the directed edges. Even with these simplification the general directed subgraph isomorphism problem is NP-complete [28].

Problem 2: *Given an infinite number of each set of templates* $\Omega = T_1, \dots , T_k$ *and an overlapping set of subgraphs of the given graph* G(V,E) *which are isomorphic to some member of* Ω*; minimize* k *as well as* $\Sigma\ x_i$ *where* x_i *is the number of templates of type* T_i *used such that the number of nodes left uncovered is the minimum.*

An example of these two problems is given in FIGURE 11. First, we must determine the exact location of all of the templates as stated in Problem 1. Once we have found every occurrence of the template, Problem 2 selects a set of templates that maximizes the covering of the graph using the templates.

We want to minimize both the number of distinct templates that are used in the covering while minimizing the number of instances of each template. Additionally, we want to cover as many nodes as possible. This problem is a fusion of the graph covering and the coin changing problems. It differs from the graph covering as it allows multiple instances of template in it's covering. The coin changing problem tries to find the minimum number of coins to produce exact change; this is similar to minimizing the number and types of templates to cover the graph.

The classical compiler problem of instruction selection falls into the realm of template matching. We are given the templates or instruction mappings corresponding to a directed, labeled graph of the program more commonly known as the IR. Instruction selection is directly related to Problem 2, with possibly additional objectives e.g. minimize runtime of the code, size of the code, etc.

5.3.2 Template Generation

Up until this point, it was assumed that the templates were given as an input. However, this may not always be the case; an automatic regularity extraction algorithm must develop its own templates.

Consider instruction generation for hybrid reconfigurable architectures. The instructions (templates) for traditional processors are fixed according to the target architecture. Since we are dealing with hybrid reconfigurable architectures, the instructions are not fixed. It is possible to arrange the reconfigurable fabric to perform virtually any combination of basic operations. Therefore, the instruction templates are not fixed in reconfigurable architectures and template generation is a necessary step for the hybrid reconfigurable architecture generation and compilation.

The configurable fabric allows the designer to implement custom instructions as well as perform any fixed instructions on more traditional embedded processing units. The custom instructions should be generated to maximally cover the data flow graph. In essence, the compiler must perform instruction generation and selection, equivalently template generation and matching.

Additionally, template generation is useful for creating macro libraries for both ASIC and FPGA architectures [95]. Also, template generation is needed for effective system-level partitioning [96].

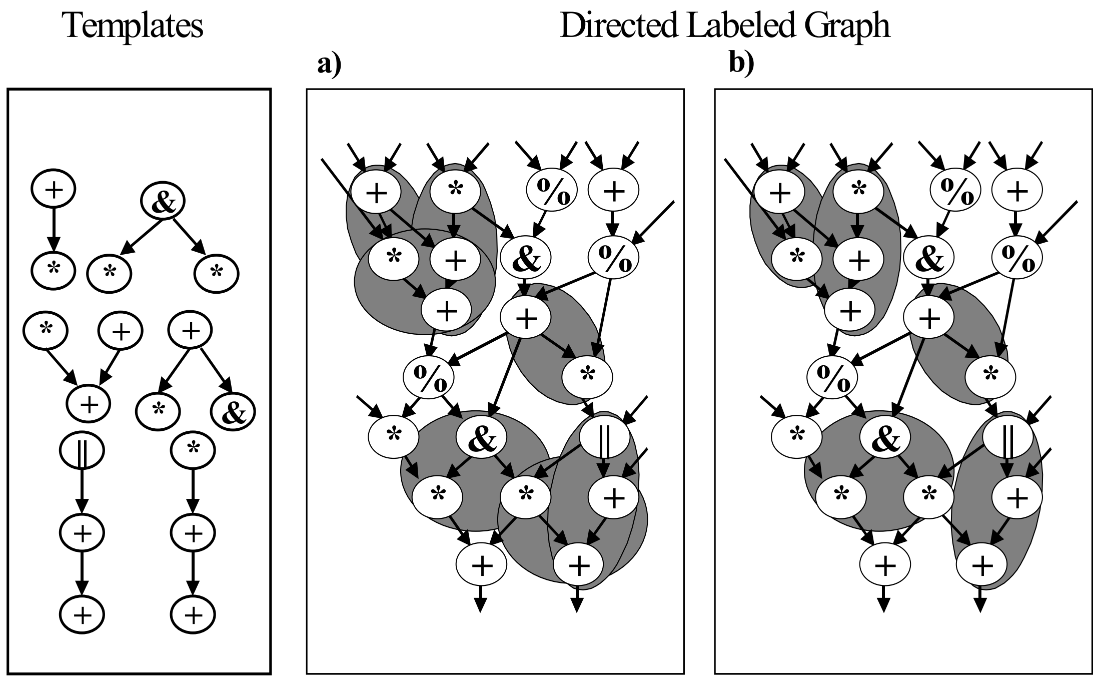


FIGURE 11 Part a) shows the subgraphs isomorphic to the given templates as described in Problem 1. Part b) selects a non-overlapping set of these subgraphs such that the covering of vertices is maximized

5.4 An Algorithm for Simultaneous Template Generation and Matching

In this section, we present an algorithm for template generation and matching which iteratively clusters nodes based on profiling. During the clustering, we generate templates and find a cover using the templates. The algorithm starts by profiling the graph for frequency of node and edge types. Based on the most frequently occurring edges, clustering is performed. An overview of the algorithm is given in FIGURE 12.

5.4.1 Algorithm Description

The algorithm starts by calling the function *profile_graph*, which traverses the graph to record the number of occurrences of every edge type. It returns the edge types ordered corresponding to the frequency of their appearance in the graph. Since the edges are not labeled, the two node types (head and tail) that an edge connects identify its type. If there are N distinct node types, the number of distinct edge types is $O(N^2)$.

```
1  **Given** a labeled digraph $G(V, E)$
2  # C is a set of edge types
3  $C \leftarrow \emptyset$
4  **while** stop_conditions_met($G$)
5       $C \leftarrow$ profile_graph($G$)
6            cluster_common_edges($G$, $C$)
```

FIGURE 12 Overview of clustering-based algorithm for template generation and matching

The function *cluster_commom_edges* takes the labeled digraph and the edge type frequencies and performs node clustering based on edge contraction. Given two vertices v_1 and v_2, *contraction* removes v_1 and v_2, replacing them with a new vertex v. The set of edges incident on v are the union of the set of edges incident to v_1 and v_2. Edges from v_1 and v_2 with mutual endpoints, e.g. $e_1 = (v_1, x)$ and $e_2 = (v_2, x)$, may or may not be merged into one edge; we merge them but we must remember that these two edges exist to preserve the sanity of the logic. We further discuss this and other issues concerning clustering later. Edges between v_1 and v_2 are removed to eliminate self-loops; self-loops must be eliminated when considering acyclic graphs as they introduce a cycle.

Finally, the function *stop_conditions_met* is called to possibly halt the algorithm. The function returns "true" if the algorithm has generated a sufficient amount of templates and/or the graph sufficiently covered. Without a stopping condition, the clustering process would continue until there is only one node. But, it is unlikely that we want this to happen. Often, we wish to stop once a certain number of templates are generated. Another possible stopping condition is when the generated templates cover every vertex of the graph. Most likely, the stopping condition function should be tailored to the particular application for template generation and matching. We discuss the stopping conditions we choose for instruction selection in the next section.

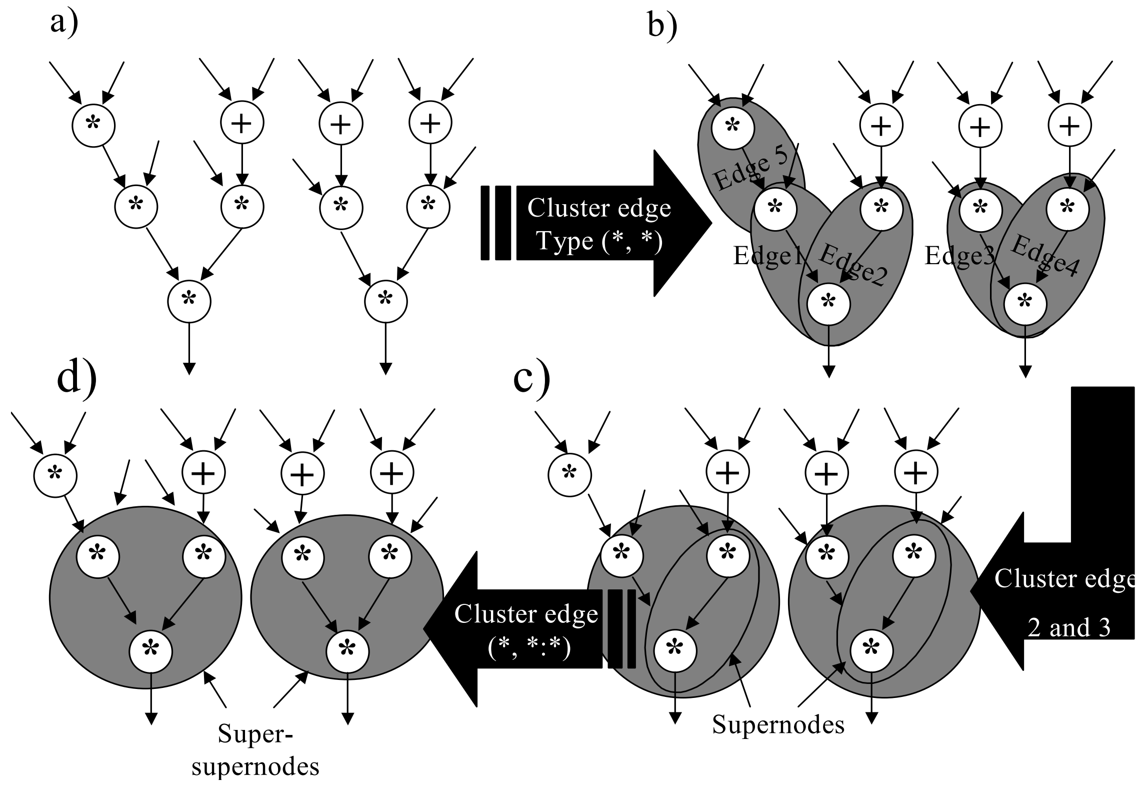


FIGURE 13 Contraction of edges to create supernodes. The supernodes correspond to templates.

Theorem 5.1 *One iteration of the clustering-based algorithm takes time* O(|E|) *on the graph* G(V,E).

Proof: One iteration spans steps 4 – 6 in FIGURE 12. The function *profile_graph* looks at every edge of the graph to determine the frequency of the edge types which takes *O(|E|)* time. Using a table of size *O(|L|^2)* where *L* is the number of distinct node labels (one entry for each possible edge type), we can determine the edge type in constant time for each edge by looking at the nodes adjacent to the edge. Furthermore, we know that

$O(|L|) \le O(|E|)$. Since we are only concerned with the most common edge type, we must only know of the edge type with largest number of occurrences. We can keep track of this when we are incrementing the edge type of the current edge. If the number of occurrences of the most common edge type is less than the number of occurrences of the edge type that we care currently incrementing, then we set the most common edge type as the current edge type; there is no need to sort the edges. The act of clustering (edge contraction) takes constant time for each edge we wish to contract. We must also determine the edges that we wish to contract. A simple method to choose the set of edges to contract is to consider each edge and contract the edge as if it is possible i.e. the previous edge contractions did not include a vertex that is incident to this edge. This takes $O(|E|)$ time but is not optimal. We consider other methods in Section 4.3 that are provable optimal, but may increase the runtime of an iteration. We assume that the halting condition of the algorithm takes constant time, which is true if our halting condition depends on the number of templates (constant time check) generated or if it halts once the graph is sufficiently covered (constant time check). Therefore, the total runtime is $O(|E|)$. □

FIGURE 13 demonstrates two passes of the algorithm. The initial graph (FIGURE 13 a) is profiled and the edge type (*,*) is chosen for clustering. You can see that there are many conflicting choices for edge contraction (FIGURE 13 b). We discuss how to resolve these conflicts later. Edges 2 and 4 are clustered to form a supernode (FIGURE 11 c). The next round of profiling chooses to contract the edge (*, {*:*}) where {*:*} is the supernode created in the previous pass. Edge contraction occurs to create a super-supernode. The algorithm stops having generated a template and a covering using these templates (FIGURE 13 d).

Edge contraction essentially creates a supernode from two nodes. The supernode must hold the DAG of the operations that it implements in order to realize the templates that it is generating; we call this the (super)node's *internal DAG.*

Every time we create a new supernode, we generate a new template corresponding to that new node. That template is the internal DAG of the supernode. It is possible that identical templates are generated through separate sequences of clustering. Therefore, we cannot identify the template based on its sequence of edge contractions; we must consider the supernode's internal DAG. A graph isomorphism algorithm is needed to identify whether two templates (generated through a different sequence of edge contractions) are identical.

5.4.2 Quick Rejection Graph Isomorphism

We developed a graph isomorphism algorithm for DAGs that quickly rejects dissimilar graphs while determining their isomorphism. The algorithm is sketched in FIGURE 14. Our algorithm is an extension of Gemini [97]; the Gemini algorithm iteratively colors and recolors vertices according to vertex invariants until every vertex contains a unique color equivalently each vertex is in a unique partition. An *invariant* is a property of a graph that does not depend on its presentation. More formally, an invariant is a function F such that $F(G_1) = F(G_2)$ if G_1 is isomorphic to G_2. The algorithm approaches the problem by finding a canonical label (coloring or partitioning) for each graph and then compares the labels of the graphs. If the labels are equivalent, then the graphs are isomorphic. The Gemini algorithm is used to solve the general graph isomorphism problem. We use additional invariant properties and checks related to labeled digraphs e.g. level information, to create a better initial coloring, which should decrease the number of iterations.

First, the algorithm does a simple check to verify if the two graphs G_1 and G_2 have the same edge and vertex cardinality. Then, the graphs are sorted in reverse topological order while adding level information to each vertex. The *level* of a vertex v is the minimum distance from v to $v' \in PO$ where PO is the set of primary output vertices (vertices with outdegree = 0). An example of a graph with level information is shown in FIGURE 15.

We use the level information as an initial color for the vertices (lines 6 – 10). At this point, we can compare the number of edges between levels as an addition check. Next, the vertices are iteratively recolored according to their colors and the colors of the adjacent edges (lines 12 – 16). This continues until each vertex of the graphs has a unique color. If the color for each and every vertex in G_1 matches a color for a distinct vertex in G_2, then the graphs are isomorphic (line 17). The coloring procedure is described in further detail in the Gemini graph isomorphism algorithm.

If we didn't care about runtime, we could solely use Gemini. But often this is overkill as it is rare that we create isomorphic templates. Therefore, the initial checks will most often quickly determine that the internal DAGs are non-isomorphic. Since we need to perform isomorphism checks after every iteration, it is essential to the overall runtime of the algorithm that we have a fast graph isomorphism algorithm.

```
1  **Given** labeled digraphs $G_1(V_1, E_1)$ and
   $G_2(V_2, E_2)$
2  **if** $|V_1|$ != $|V_2|$ **or** $|E_1|$ != $|E_2|$
3      **return** false
4  $R_1$ = reverse_topological_order($G_1$)
5  $R_2$ = reverse_topological_order($G_2$)
6  **for each** level $l$ of $R_1$
7      **for each** vertex $v$ in level($R_1$, $l$)
8          color($v$)
9      **for each** vertex $v$' in level($R_2$, $l$)
10         color($v'$)
11 //Now run the Gemini algorithm
12 **while** fully_partitioned($V_1$, $V_2$) = false
13     **for each** vertex $v$ in $V_1$
14         color($v$)
15     **for each** vertex $v$ in $V_2$
16         color($v$)
17 **return** equivalent_vertex_labels($G_1$, $G_2$)
```

FIGURE 14 An overview of the labeled DAG isomorphism algorithm

Theorem 5.2 *The labeled DAG isomorphism algorithm is correct.*

Proof: The basis of the algorithm is using the invariant properties of a graph to create a unique color or partition for each vertex. A vertex invariant is a property of the vertex that is preserved under isomorphism. It is trivial to show that the level of a vertex is invariant. Therefore, the initial coloring is invariant. The remainder of the algorithm iteratively colors (partitions) the vertices as stated in the Gemini algorithm, which is known to be correct. □

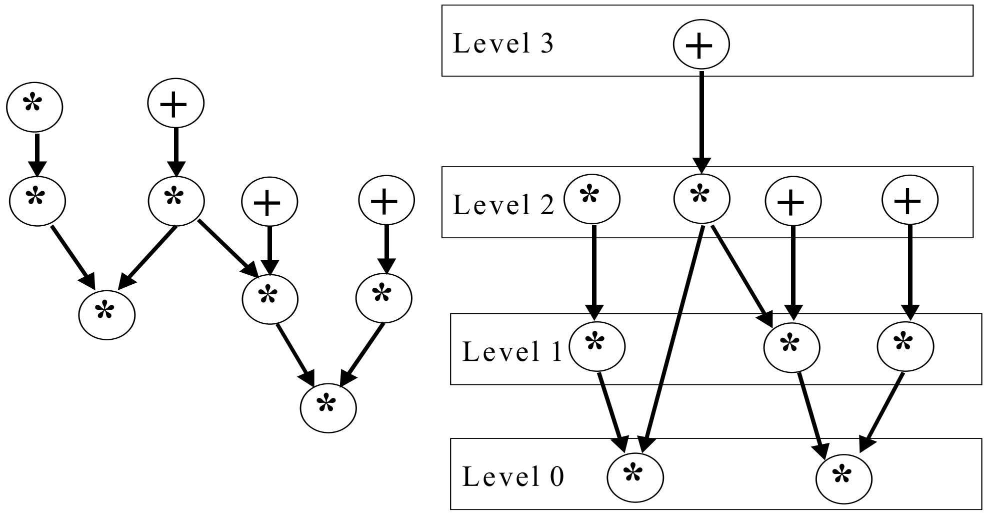


FIGURE 15 Levels of a digraph

5.4.3 Finding an Optimal Covering

Choosing the set of edges to contract can greatly affect the quality of the solution. Consider the graph in FIGURE 13. The best cover consists of one template consisting of two multiply operations feeding into another multiply (as demonstrated in FIGURE 13 d). But, if we contracted Edge 5 instead of Edge 2 and 3, we would not have been able to achieve this cover. This dilemma has haunted graph covering algorithm makers for a long time; there is no known exact method to avoid such ill-fated decisions. We employ a locally optimal heuristic based on maximum matching.

A *matching* of a graph is any set of pairwise disjoint edges. A maximum matching of a graph is the matching with maximum cardinality. The maximum matching problem for general undirected graphs can be solved in $O(n^{2.5})$ time [98].

To find the optimal set of edges to contract, we use the maximum matching of the edge induced undirected subgraph G_E of G. The edges inducing the subgraph have the most common edge type for the current iteration. Since we are trying to maximize the number of nodes that are covered using the minimum number of templates, we want to cover as many nodes possible at each step. This is accomplished by taking the maximum matching of the edge induced subgraph G_E. The edges in the maximum matching are chosen as the edges to contract.

Theorem 5.3 *Given a graph $G(V,E)$ to cover with a template T, the template instance assignment corresponding to the edges from the maximum matching of the edge induced subgraph G_E of G gives an optimal covering where optimality is defined as a covering of the maximum number of vertices in V.*

Proof: We prove the theorem by contradiction. Assume that the covering using maximum matching of G_E is not optimal. Since the number of vertices in each template instance is the same, a larger number of template instances correspond to a better matching (covering of more vertices). If the current matching M is not optimal, then there must be another covering M' such that the number of templates used in M' is greater than the number of templates used in M. By definition of a matching, the template instances in the covering of M' correspond to a set of edges in G_E that have no edges between them. Therefore, we can construct a matching with a larger cardinality than the maximum matching, giving the contradiction. □

Therefore, we can use the maximum matching algorithm to find a locally optimal covering of templates for the current iteration. Using the best know algorithm for maximum matching on a general undirected graph, we can optimally solve the template covering problem of any iteration of our template generation algorithm in *$O(n^{2.5})$* time.

5.5 Experimental Results

5.5.1 Experimental Setup

To test our template generation and matching algorithm, we implemented a hybrid reconfigurable system compiler front-end on top of the SUIF2 compiler system [52]. SUIF is a well-known intermediate format (IF) that is used heavily in industry and academia. It compiles C/C++/Fortran source code into a high-level IF. We used the Machine-SUIF [59] back end to create a low-level IF representation, i.e. a Control Flow Graph (CFG). From there, we implemented a pass to convert the CFG to a CDFG. Our template generation and matching algorithm was performed over all the data flow graphs of a CDFG.

5.5.2 Target Applications

Digital signal processing (DSP) applications tend to have a large amount of parallelism between instructions. They are ideal for mapping onto reconfigurable hardware, as the instruction parallelism can be exploited for increased runtime speed compared to executing the instructions sequentially on a traditional processor. In this work, we focus on DSP functions.

We look at the applications from the MediaBench test suite [99]. From these applications, we selected a set of files that implement DSP functions. Table 1 presents the characteristics of the selected DSP functions. In order to synthesize the results of our algorithm, we compiled and generated instructions for four programs: an image convolution algorithm, DeCSS (the decryption of DVD encoding), the DES encryption algorithm, and the Rijndael AES encryption algorithm. These algorithms are typical candidates for industrial hardware implementation (as cameras, DVD players, and embedded encryption devices must perform these operations). Additionally they are computationally intensive, leading to generally large DFGs which are benign to regularity extraction. From each compiled CDFG of the programs, four representative DFGs were selected for scheduling.

5.5.3 Results

We ran the template generation and matching algorithms on the MediaBench test files. For each test file, a set of templates was generated and a covering was produced using the generated templates. Templates were generated on the nodes that performed arithmetic operations. The stopping condition of the algorithm depended on the frequency of the most often occurring edge. If the edge type occurred less than *x*%, the algorithm completes. We call this the *cut-off percentage.*

TABLE 1: MediaBench Test Files

Benchmark	**C File**	**Description**
mpeg2	motion.c	Motion vector decoding
mpeg2	getblk.c	DCT block decoding
adpcm	adpcm.c	ADPCM to/from 16-bit PCM
epic	convolve.c	2D general image convolution
jpeg	jctrans.c	Transcoding compression
jpeg	jdmerge.c	Color conversion
rasta	fft.c	Fast Fourier Transform
rasta	noise_est.c	Noise estimation functions
gsm	gsm_decode.c	GSM decoding
gsm	gsm_encode.c	GSM encoding

We varied the cut-off percentage to measure the tradeoff of number of generated templates vs. percentage of the graph covered by those templates. As the cut-off percentage increases, the number of generated templates decreases and fewer nodes are covered. As it decreases to 0, the algorithm generates a larger number of templates and covers more nodes, but the additional templates that are generated may only cover a few additional nodes.

Remember that a template refers to a sequence of operations that will reside in the same vicinity and execute in sequence. The template can be placed as a macro in the configurable fabric or be used to specify the functionality of a VPB block. Either way, we lose system flexibility in hopes to increase performance, reduce power, etc. The gain of using a template in the system is a hard parameter to quantify. It depends on a number of factors, including scheduling, placement, routing, etc. Regardless of those other factors, a template must occur frequently in order to yield a favorable performance/power to flexibility ratio. Therefore, we aim to minimize the number of different templates while maximizing the covering of the graph(s).

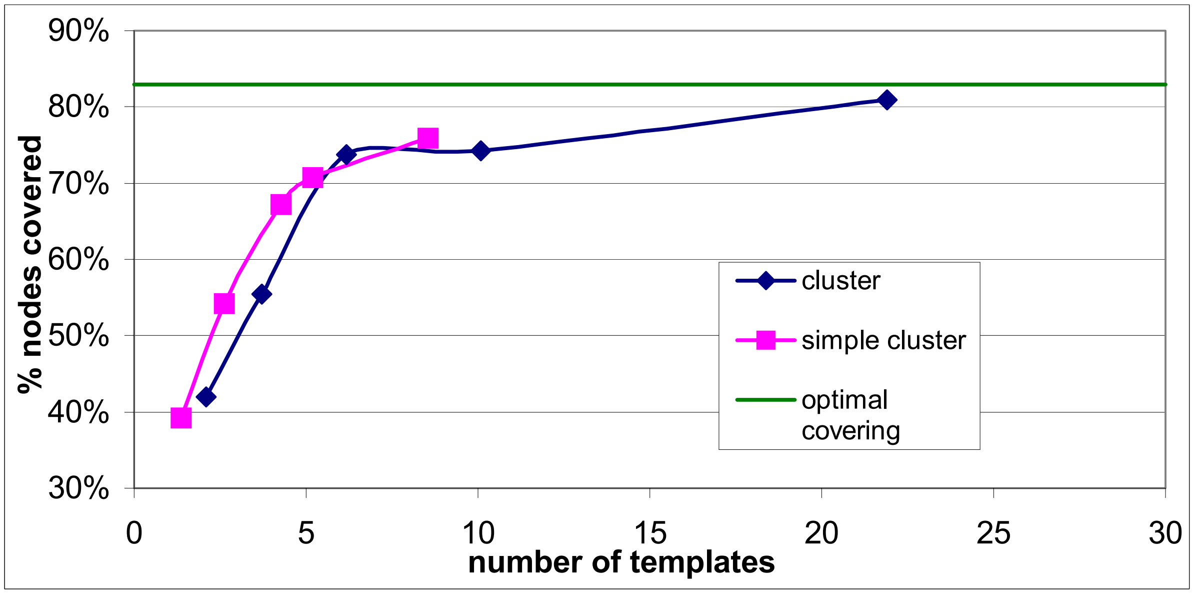


FIGURE 16 Comparison of clustering techniques

The “cluster” line in FIGURE 16 plots the average number of templates generated and the percentage of the graph covered using the generated templates. By varying the cut-off frequency we produced the points of the graph. A cut-off frequency of zero will cover every node by creating templates that occur a small number of times (including singleton templates). Sometimes a graph cannot be completely covered by templates, as an arithmetic node is isolated in a data flow graph (CFG node) by itself. In the benchmarks that we consider, the “optimal” covering is a covering of 83% of the nodes i.e., on average, 17% of the nodes are isolated. You can see that in order to generate an optimal covering the algorithm generates an average of 21.9 templates.

The slope of the line (Δ(% coverage)/ Δtemplates) gives much intuition into the amount of coverage you get by generating additional templates. When the number of templates is small (less than 5) the slope is large, meaning that adding another template gives you a large amount of additional graph coverage. As the number of templates increases, the slope decreases. It is interesting to note that the slope dramatically reduces around 5 templates. It seems to suggest that using five templates is a good number for covering the benchmarks.

During our experiments, we noticed that the number of operations (node) per template is small. We tried restricting the edge contraction so that only templates with 2 nodes would be generated. We called this “simple” clustering. The results using this clustering scheme were plotted in FIGURE 16. As with the previous experiment, the cut-off percentage is varied to generate the different points. As you can see, the results mimic those of the “complex” clustering technique. The main difference is that the simple technique cannot achieve an optimal covering like the complex technique. But, in order to achieve an optimal covering, the complex technique generated a large amount of templates. Many of these generated templates covered a limited number of nodes – a poor solution. Therefore, a template generation and matching which limits the templates to 2 nodes gives a solution with similar quality as the complex algorithm.

Additionally, we noticed that the types of complex templates varied widely across all the applications. Therefore, if we wanted to generate one “generic” system for all the benchmarks, e.g. a system of DSP applications, that we examined, there would be a large number of templates and each application would use only a small subset of those templates. On the other hand, we found that there was much less variation of template types when we used the simple templates.

To further explore this phenomenon, we looked at simple template combinations using add and multiply combinations – the two most frequently occurring arithmetic node types across all the benchmarks. Table 2 shows the results of the coverage using the 4 add/multiply sequences as individual templates. In the table, the notation OP1-OP2 denotes that the template consists of the two operations with an edge {OP1, OP2}.

TABLE 2:Coverage using simple add and multiple template combinations

Operation	MediaBench file name				
	motion	jdmerge	getblk	gsm_dec	jctrans
ADD	50.3%	84.6%	44.5%	29.6%	84.6%
MUL	36.3%	13.8%	24.0%	22.4%	13. 8%
Template Coverage					
MUL-MUL	0.0%	0.0%	1.3%	0.0%	0.0%
ADD-ADD	14.5%	9.1%	3.2%	3.6%	9.1%
ADD-MUL	0.0%	0.4%	0.6%	0.0%	0.4%
MUL-ADD	36.3%	13.0%	21.5%	22.4%	13.0%

We can gather a lot of information using these simple templates. For example, the sequence of operations deviates from the expected probability as the sequence MUL-ADD is found with much greater frequency than ADD-MUL. Probabilistically, these sequences should occur in the same proportion. Additionally, it shows that the MUL-

ADD[3] and ADD-ADD sequences could be implemented as a VPB or macro for DSP applications as it is widely used across all the applications. In summary, we presented evidence that templates can be limited to simple, two operation sequences while achieving good coverings using a small number of templates. We believe that this is due to the structure of the CDFGs.

In general, the data flow graphs of the CDFGs have a small number of levels and the actual number of arithmetic operations per data flow graph is not large enough to encourage templates with a large cardinality. These are two well-known phenomena and are the source of problems in exploiting parallelism for VLIW processors. Therefore, we believe that hybrid reconfigurable systems must leverage techniques from the VLIW domain such as predicated execution [100] and hyperblock construction [101] in order to realize larger template cardinality.

5.6 Related Work

While there has been a lot of work in regularity extraction, most of it focuses on template matching (similar to the graph covering problem) and not template generation. Regularity extraction was shown beneficial in reducing area and increasing performance for the PipeRench architecture [80]; PipeRench is a fully reconfigurable, pipelined FPGA. The benefits stem from the fact that the templates may be hand optimized. Additionally, the template operations are placed in the same vicinity on the chip. This reduces the interconnect delay as well as compacts the application into a smaller portion of the chip. Cadambi and Goldstein go on to show that templates lead to a decrease in area and delay for the PipeRench architecture; they suggest that profiling is beneficial for small granularity FPGAs e.g. LUT or PLA-based, though no empirical evidence is given to support this claim.

[3] The MUL-ADD should come as little surprise as we are profiling DSP applications and the multiply-add (MAC) instruction is a staple of the DSP instruction set.

Cadambi and Goldstein restrict their template generation to single output templates and limit the number of inputs. If the templates are going to be used as soft reconfigurable macros and placed in a configurable fabric, the number of inputs/outputs must be limited to maintain good routability. But, templates can also be used to generate the VPB functionality. Since the VPBs are ASIC blocks integrated into the reconfigurable fabric, the system architecture can place additional routing resources around VPBs to handle the additional routing needed by VPBs with a large number of inputs/outputs. Therefore, generated templates need not always have input/output restrictions.

Regularity extraction is used in a variety of other CAD applications. Templates are used during scheduling to address timing constraints and hierarchical scheduling [102]. Data path circuits exhibit a high amount of regularity; hence regularity extraction reduces the complexity of the program as well as increasing the quality of the result [103, 104]. System level partitioning is yet another use of regularity extraction [96]. Furthermore, proper use of templates can lead to low-power designs [105].

One of the earliest template matching works in the CAD community was by Kahrs [106], wherein a greedy, bottom-up procedure for a silicon compiler is described. Keutzer [107] modeled a system as a DAG and heuristically partitioned it to yield rooted trees and applied compiler techniques to test for pattern matches. Trees and single output templates are used by Chowdhary et al. [103] to cover data path circuits.
Rao and Kurdahi [96] addressed template generation for system-level clustering using the well-known first fit approach to bin filling. More recently, Cadambi and Goldstein [95] propose single output template generation via a constructive, bottom up approach. Both methods restrict the area and the number of pins for their templates. Our method attempts to find the best possible set of templates, regardless of area and size, though we can easily add pin and area restrictions to our algorithms. Additionally, we perform template generation and matching simultaneously.

IMEC's Cathedral Project [108] used a different model of computation in their high-level synthesis stage: instead of a CDFG they performed reductions on the signal flow graph of a DSP application. Their data path was composed of Abstract Building Blocks (ABBs), or instructions available from a given hardware library. The customized data path generated from many ABBs was referred to as an application specific unit (ASU). Cathedral's synthesis targeted ASUs, which could be executed in very few clock cycles. This goal was achieved via manual clustering of necessary operations into more compact operations, essentially a form of template construction. Whereas our template generation and matching algorithms are automated, the definition of clusters in Cathedral was a manual operation, mainly clustering loop and function bodies. Their results demonstrated an expected reduction of critical path length as well as interconnect as a result of clustering.

One of the more encouraging cases of performance gain via template matching was investigated by Corazao et al [109]. Their work assumed a given library of highly regular templates. These templates could be utilized during the high-level synthesis stage in order to minimize the number of clock cycles in a circuit's critical path. In circumstances where some parts of a template were not needed, partial matching was also allowed. With partial matching, some portions of a selected template go unused. Their experimental results demonstrated large performance gains without an unreasonable increase in area. Although many optimization techniques were utilized as part of the synthesis strategy, template selection had the largest impact on overall improvement in throughput.

The Totem Project [110, 111] endeavors to automate the generation of custom reconfigurable architectures based on a given set of applications. Built upon the RaPiD architecture [81], their optimizations are made at the placement and routing stages of synthesis, mapping coarse-grained components to a one-dimensional data path axis. Unlike our design, their input is a set of architecture netlists, which are transformed directly to a physical design while targeting the simultaneous goals of increased routing flexibility and decreased area.

5.7 Summary

In this work, we addressed the problem of instruction generation. We proposed an algorithm to solve the problem that performs simultaneous template generation and matching. Our algorithm generates instructions by profiling the graph and clustering common edges. Furthermore, we present some theory behind instruction generation.

Instruction generation is a relatively new and essential problem for compilation to reconfigurable systems. Instruction generation can be used to create soft reconfigurable macros, which are tightly coupled sequential operations that are placed in the same vicinity in a configurable fabric. Furthermore, the macros are ideal candidates for hand optimization. Additionally, template generation can be used to specify the functionality for pre-placed ASIC blocks (VPBs) in hybrid reconfigurable systems.

We developed a co-compiler for a hybrid reconfigurable system. Using DSP benchmarks, we showed that full-blown template generation is unnecessary as simple templates – templates with a sequence of two operations – create a graph covering with similar quality to more complex templates. This suggests that advanced compiler techniques such as predicated execution and hyperblock construction are needed in order to efficiently utilize large templates.

In the future, we plan to study the effect of predicated execution and hyperblock on template generation. Also, we intend to develop a complete back-end of a retargetable compiler for hybrid reconfigurable systems.

CHAPTER 6 Data Communication

6.1 Introduction

In order to facilitate the future design of embedded systems and system-on-chips, we must develop techniques to explore the design space of the system i.e. *system synthesis*. These tools will take a high-level specification of the application and produce a customized hardware system. The system may be comprised of many components – different types of processors (e.g. ARM, VLIW, superscalar) as well as reconfigurable logic devices (e.g. FPGA) and/or ASIC components.

The compiler straddles the boundary between application and hardware, making it a natural area to perform system exploration. The compiler can already map portions of the application to different processors by simply emitting code. This only allows exploration on a system composed of various numbers and types of processors. In order to complete the system exploration space – one with processors, ASIC and reconfigurable components, we need a path from the compiler to a hardware description language (HDL); this allows us to map portions of the application to ASICs, FPGAs and any other devices that accept HDL as an input.

An area of extreme importance is the translation of the compiler's intermediate representation (IR) to a form that is suitable for synthesis to hardware. During this translation, we should attempt to exploit the existing concurrency of the application and discover additional parallelism [84]. Also, we should determine the types of hardware specialization that will increase the efficiency of the application [80, 92]. Finally, we must take into account the hardware properties of the circuit, e.g. power dissipation, critical path and interconnect area.

Static single assignment [112, 113] transforms the IR such that each variable is defined exactly once. It is an ideal transformation for hardware because lone side effects of the transformation, Φ-nodes, are easily implemented in hardware as multiplexers. It has been used in many projects where the final output is an HDL [18, 20, 83]. Yet, SSA was originally developed to enable optimizations for fixed architectures; it was not originally meant for hardware synthesis.

In this work, we study SSA and its effect on the optimization of hardware properties of the circuit. We show how SSA can be used to minimize data communication; this has a direct effect on the area, amount of interconnect and delay of the final circuit. Furthermore, we show that SSA in its original form is not optimal in terms of data communication and give an optimal algorithm for the placement of Φ-nodes to minimize the amount of data communication.

In the next section, we give background material related to our research. We show how SSA is useful to minimize interconnect in the hardware in Section 6.2. Furthermore, we point out a fundamental shortcoming of traditional SSA and develop a new SSA algorithm to overcome this limitation. Section 6.3 presents experiments to illustrate the effect of these algorithms to minimize data communication. We discuss related work in Section 6.4 and provide concluding remarks in Section 6.5.

6.2 Minimizing Inter-Node Communication

In order to determine the data exchange between nodes in a CDFG, we establish the relationship between where data is generated and where data is used for calculation. The specific place where data is generated is called its *definition point*. A specific place where data is used in computation is called a *use point*. The data generated at a particular definition point may be used in multiple places. Likewise, particular use point may correspond to a number of different definition points; the control flow dictates the actual definition point at any particular moment.

In hardware, the path of execution between nodes can be controlled either by a central controller or by the nodes themselves. The former is called centralized control, whereas the latter is decentralized control. The hardware nodes are referred to as control nodes, and roughly correspond to nodes on a compiler's control flow graph.

If data generated in one control node is used in a computation in a second control node, these two control nodes must have a mechanism to transfer the data between them.

A distributed data communication scheme has a direct connection between the two control nodes (i.e. one node controls the other's execution through a signal). If a centralized scheme were used, the first control node would transfer the data to memory and the second control node would access the memory for that data. Therefore, in a centralized scheme minimizing the inter-node communication would have a direct impact on the number of memory accesses, and in a distributed scheme the interconnect between the control nodes would be reduced. However, in both control schemes real performance boosts can be realized through communication optimization. Thus, regardless of the scheme that we use, we should try to minimize the amount of inter-node communication.

6.2.1 Static Single Assignment

We can determine the relationship between the use and definition points through static single assignment [112, 113]. Static Single Assignment (SSA) renames variables with multiple definitions into distinct variables – one for each definition point.
We define a *name* to represent the contents of a storage location (e.g. register, memory). A name is unspecific to SSA. In non-SSA code, a name represents a storage location but we may not know the exact location; the precise location of the name depends on the control flow of the program. Therefore, we call a name in non-SSA code a *location.* SSA eliminates this confusion as each name represents a value that is generated at exactly one definition point. The SSA definition of a name is called a *value*.

In order to maintain proper program functionality, we must add Φ-nodes into the CDFG. Φ-nodes are needed when a particular use of a name is defined at multiple points. A Φ-node takes a set of possible names and outputs the correct one depending on the path of execution. Φ-nodes can be viewed as an operation of the control node. They can be implemented using a multiplexer. FIGURE 17 illustrates the conversion to SSA.

SSA is accomplished in two steps, first we add Φ-nodes and then we rename the variables at their definition and use points. There are several methods for determining the location of the Φ-nodes. The naïve algorithm would insert a Φ-node at each merging point for each original name used in the CDFG. A more intelligent algorithm – called the minimal algorithm – inserts a Φ-node at the iterated dominance frontier of each original name [112]. The semi-pruned algorithm builds smaller SSA form than the minimal algorithm. It calculates determines if a variable is local to a basic block and only inserts Φ-nodes for non-local variables [113]. The pruned algorithm further reduces the number of Φ-nodes by only inserting Φ-nodes at the iterated dominance frontier of variables that are live at that time [114]. After the position of the Φ-nodes is determined, there is a pass where the variables are renamed.

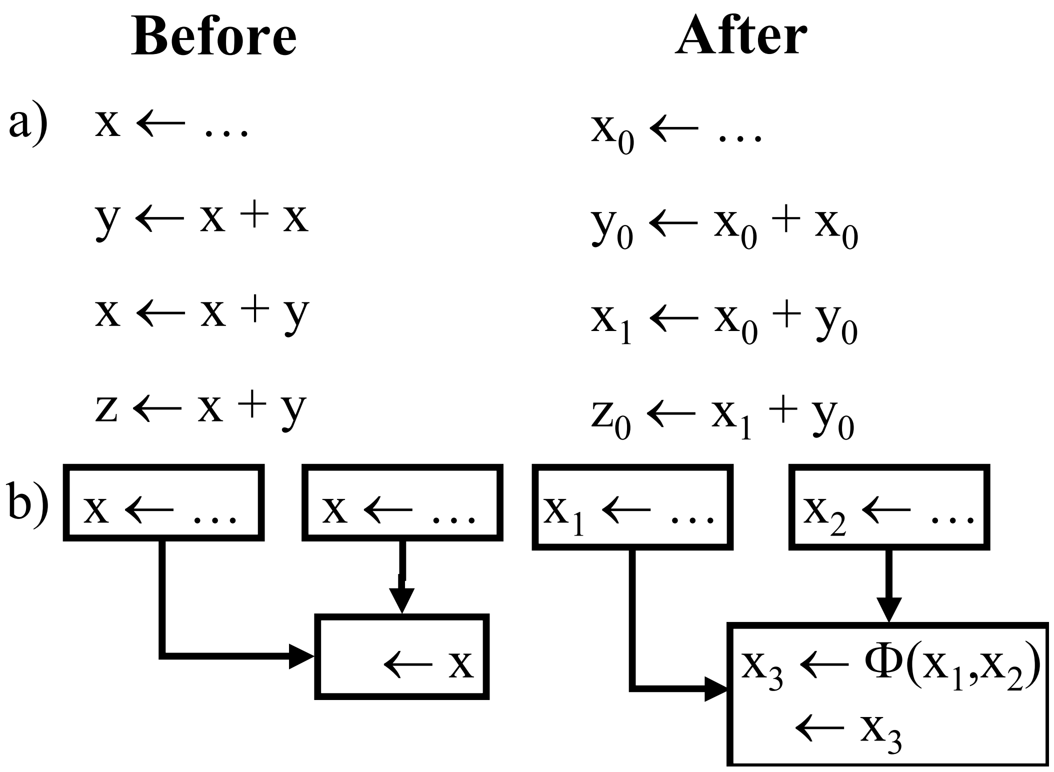


FIGURE 17 a) Conversion of Straight-line Code to SSA b) SSA Conversion with Control Flow

The minimal method requires $O(|E_{cfg}| + |N_{cfg}|^2)$ time for the calculation of the iterated dominance frontier. The iterated dominance frontier and liveness analysis must be computed during the pruned algorithm. There are linear or near linear time liveness analysis algorithms [115-117]. Therefore, the pruned method has the same asymptotic runtime as the minimal method.

We should suppress any unnecessary data communication between control nodes. Now we explain how to minimize the inter-node communication.

6.2.2 Minimizing Data Communication with SSA

SSA allows us to minimize the inter-node communication. The various algorithms used to create SSA all attempt to accurately model the actual need for data communication between the control nodes. For example, if we use the pruned algorithm for SSA, we eliminate false data communication by using liveliness analysis, which eliminates passing data that will never be used again.

SSA allows us to minimize the data communication, but it introduces Φ-nodes to the graph. We must add a mechanism that handles the Φ-nodes. This can be accomplished by adding an operation that implements the functionality of a Φ-node. A multiplexer provides the needed functionality. The input names are the inputs to the multiplexer. An additional control line must be added for each multiplexer to determine that the correct input name is selected.

A fundamental limitation of using SSA in a hardware compiler is the use of the iterated dominance frontier for determining the positioning of the Φ-nodes. Typically, compilers use SSA for its property of a single definition point. We are using it in another way – as a representation to minimize the data communication between hardware components (CFG nodes). In this case, the positioning of Φ-nodes at the iterated dominance frontier does not always optimize the data communication. We must consider spatial properties in addition to the temporal properties of the CDFG when determining the position of the Φ-nodes.

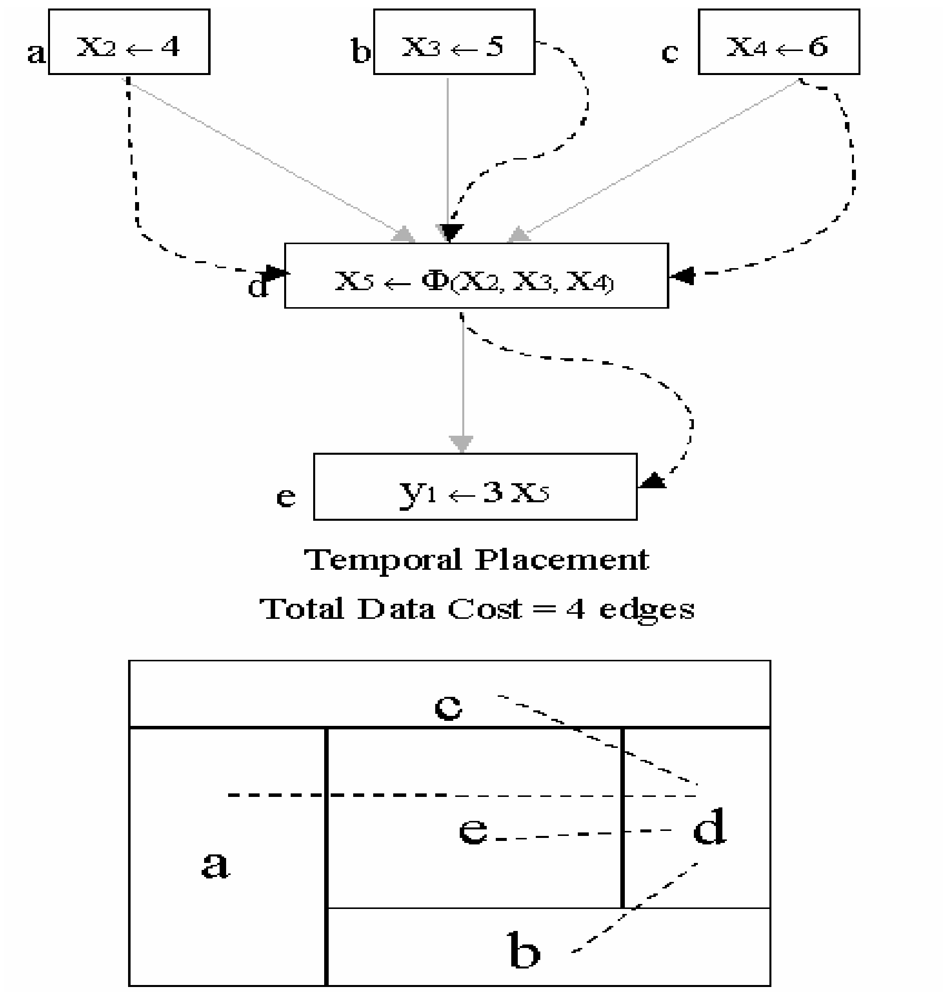


FIGURE 18 SSA form and the corresponding floorplan (dotted edges represent data communication, and grey edges represent control). Data communication = 4 units.

We illustrate our point with a simple example. FIGURE 18 a exhibits traditional SSA[4] form as well as the corresponding floorplan, containing control nodes a through e. The Φ-node is placed in control node d. In the traditional SSA scheme, the data values x_2, x_3, and x_4 (from nodes a, b, and c) are used in node d, but only in the Φ-node. Then,

[4] We use the terms "traditional SSA" and "temporal SSA" interchangeably to mean the SSA introduced by Cytron et al. {4}.

the data x_5 is used in node e. Therefore, there must be a communication connection from node a to node d, node b to node d and node c to node d, as well as a connection from node d to node e – a total of 4 communication links. In FIGURE 18 b, the Φ-node is distributed to node e. Then, we only need a communication connection from nodes a,b, and c to node e, a total of 3 communication links.

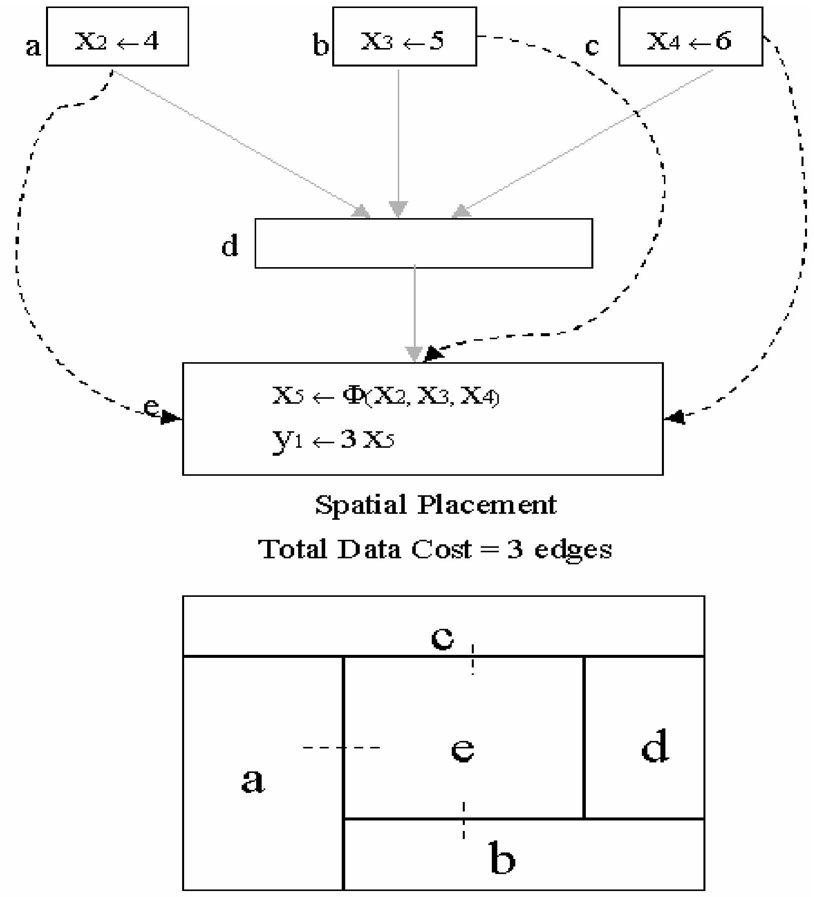


FIGURE 19 SSA form with the Φ-node spatially distributed, as well as the corresponding floorplan. Data communication = 3 units.

From this example, we can see that traditional Φ-node placement is not always optimal in terms of data communication. This arises because Φ-nodes are traditionally placed in a temporal manner. The iterated dominance frontier is the first place in the timeline of the program where the two (or more) locations of a variable merge. But, as you can see, this is not necessarily the only place where they can be placed. When considering hardware compilation, we must think spatially as well as temporally. By moving the position of the Φ-nodes, it is possible to achieve a better layout of our hardware design. In order to reduce the data communication, we must consider the number of uses of the value that a Φ-node defines as well as the number of values that the Φ-node takes as an input.

6.2.3 An Algorithm for Spatially Distributing Φ-nodes

The first step of spatially distributing Φ-nodes is determining which Φ-nodes should be moved. We assume that we are given the correct temporal positioning of the Φ-nodes according to some SSA algorithm (e.g. minimal, semi-pruned, pruned). The movement of a Φ-node depends on two factors. The first factor is the number of values that the Φ-node must choose between. We call this the number of Φ-node *source values* s. The second factor is the number of uses that the value of the Φ-node defines. We call this the Φ-node *destination value d*. Taking FIGURE 18 a as an example, the Φ-node source values are x_2, x_3, and x_4 whereas the Φ-node destination value is x_5. Determining s is simple; we just need to count the number of source values in the Φ-node. Finding the number of uses of the destination value is a more difficult. We can use def-use chains [51], which can be calculated during SSA.

The relationship between the amount of communication links C_T needed for a Φ-node in temporal SSA and the number of communication links C_S in spatial SSA is:

$$C_T = s + d \qquad C_S = s \cdot d$$

Using these relationships, we can easily determine if spatially moving a Φ-node will decrease the total amount of inter-node data communication. If C_S is less than C_T, then moving the Φ-node is beneficial. Otherwise, we should keep the Φ-node in its current location.

After we have decided on which Φ-nodes we should move, we must determine the control node(s) where we should move the Φ-node. This step is rather easy, as we move the Φ-node from its original location to control nodes that have a use of the definition value of that Φ-node. It is possible that by moving the Φ-node, we increase the total number of Φ-nodes in the design. But, we are decreasing the total amount of inter-node data communication. Therefore, the amount of data communication is not directly dependent on number of Φ-nodes.

```
1.  Given a CDFG $G(N_{cfg}, E_{cfg})$
2.  perform_SSA($G$)
3.  calculate_def_use_chains($G$)
4.  remove_back_edges($G$)
5.  topological_sort($G$)
6.  for each node $n \in N_{cfg}$
7.    for each Φ-node $\Phi \in n$
8.      $s \leftarrow |\Phi.sources|$
9.      $d \leftarrow$ |def_use_chain($\Phi.dest$)|
10.       if $s \cdot d < s + d$
11.         move_to_spatial_locations($\Phi$)
12. restore_back_edges($G$)
```

FIGURE 20 Spatial SSA Algorithm

It is possible that a use point of the definition value of Φ-node Φ_1 is another Φ-node Φ_2. If we wish to move Φ_1, we add the source values of Φ_1 into the source values of Φ_2; obviously, this action changes the number of source values of Φ_2. In order to account for such changes in source values, we must consider moving the Φ-nodes in a topologically sorted manner based on the CDFG control edges. Of course, any back control edges must be removed in order to have valid topologically sorting. We can not move Φ-nodes across back edges as this can induce dependencies between the source value and the destination value of previous iterations i.e. we can get a situation where $b_1 \leftarrow \Phi(b_1, \ldots)$. The source value b_1 was produced in a previous iteration by that same Φ-node. The complete algorithm for spatially distributing Φ-node to minimize data communication is outlined in FIGURE 19.

Theorem 6.1: Given an initially correct placement of a Φ-node, the functionality of the program remains valid after moving the Φ-node to the basic block(s) of all the use point(s) of the Φ-node's destination value.

Proof: There are two cases to consider. The first case is when the use point is a normal computation. The second case is when a use point is Φ-node itself.

We consider the former case first. When we move the Φ-node from it's initial basic block, we move it to the basic blocks of every use point of the Φ-node's destination value *d*. Therefore, every use of the *d* can still choose from the same source values. Hence, if the Φ-node source values where initially correct, the use points of *d* remain the same after the movement. We must also insure that moving the Φ-node does not cause some other use point that uses the same name but has a different value. The Φ-node will not move past another Φ-node that has the same name because by construction of correct initial SSA, that Φ-node must have *d* as one of its source values.

The proof of the second case follows similar lines to that of the first one. The only difference is that instead of moving the initial Φ-node Φ_i to that basic block, we add the source values to the Φ-node Φ_u that uses *d*. If we move Φ_i before Φ_u, then the functionality of the program is correct by the same reasoning of the first part of proof. Assuming that the temporal SSA algorithm has only one Φ-node per basic block per name, we can add the source values of Φ_i to Φ_u while maintain the correct program functionality. □

Theorem 6.2: Given a correct initial placement of Φ-nodes, the spatial SSA algorithm maintains the correct functionality of the program.

Proof: The algorithm considers the Φ-nodes in a topologically sorted manner. As a consequence of Theorem 6.1, the movement of a single Φ-node will not disturb the functionality of the program hence the Φ-node will not move past another value definition point with the same name. Since we are considering the Φ-nodes in forward topologically sorted order, the movement of any Φ-node will never move past a Φ-node which has yet to be considered for movement. Also, Φ-node can never move backwards across and edge (remember we remove back edges). Therefore, the algorithm will never move a value definition point past another value definition point with the same name. Hence every use preserves the same definition after the algorithm completes. This maintains the functionality of the program. □

Theorem 6.3: Given a floorplan where all wire lengths are unit length, the Spatial SSA Algorithm provides minimal data communication.

Proof: The source values of any given phi function are individual control nodes, and the cardinality of these nodes shall be referred to as *s*. Likewise, the destination points of any phi function are individual control nodes, and their cardinality will be referred to as *d*. The number of control nodes which define a given phi function will be referred to as *n*. The amount of data communication that this algorithm can reduce is restricted to the number of data edges coming into each phi node and the number of data edges coming out of each phi node. (The other data communication is already minimized, since SSA variables are actual data values. Therefore, SSA variables passed between control blocks are actual pieces of data that must be moved.) If a phi node is coalesced with its use point, then the number of out degree edges specifically leaving the phi node can be considered equal to zero. (The phi node's out degree data edges are now equal to the out degree of the use point, which cannot be reduced any further by the placement or removal of the phi node. Therefore the phi node's out degree of data will be considered equal to zero in this case.)

The total number of data communication points entering and exiting the phi nodes of a given phi function can be represented by a cost equation:

$$C = \sum_{\text{n } \Phi \text{ nodes}} in + out$$

where *in* is the number of inbound edges to each phi node and *out* is the number of outbound edges from each phi node.

In a floorplan where each edge has unit cost, this equation represents the total cost of this phi function in the graph.

In order to maintain correctness in a CDFG, every source value of a phi function must be coming into all phi nodes defining this function. (This is the only data that needs to enter a phi node.) Therefore, for all minimal cost cases, we can say that $in = s$ for every phi node and the data communication cost of the phi function can be restated as

$$C = ns \ + \sum_{\text{n } \Phi \text{ nodes}} out$$

since *s* is constant.

This leaves us with two values we can minimize: *n* (the number of total nodes defining a given phi function) and *out* (the out degree of a phi node), since *s* cannot be reduced (for correctness's sake). The most minimal cost we can have is when $n = 1$ or $out = 0$.

($n >= 1$, because at least one node must define the phi function. $out = 0$ is possible, as stated earlier.)

In the case that $out = 0$, the phi function will be coalesced with every use point of that function. That means that the total number *n* of nodes defining this function will equal *d* (the number of use points of the phi function). Therefore,

$$C = ns \ + \sum_{\text{n } \Phi \text{ nodes}} out = ns = d \cdot s = \mathbf{s} \cdot \mathbf{d}$$

(corresponding to spatial placement)

In the case that $n = 1$, that means that there is only one node defining a given phi function. This means that either a) there is a directed edge from this node to every use point or b) there is only one use point and this node has been coalesced with it.

In the case of part a, the total number of directed edges leaving the one phi node is equal to *d* (the number of use points) therefore

$$C = 1 * s \ + \sum_{\text{n } \Phi \text{ nodes}} out = s + out = \mathbf{s} + \mathbf{d}$$

(corresponding to temporal placement)

Part b is a special case of $C = s * d$ ($n = 1$, $out = 0$).

Therefore, we can minimize the total in/out degree of the phi node(s) by minimizing the equations ($C = s + d$, $C = s * d$). This corresponds to either choosing temporal placement (in the case of $s + d < s * d$) or choosing spatial placement (if $s + d > s * d$). This minimization of the degree of the phi node(s) leads to minimal data communication in the CDFG. □

6.3 Experimental Results

To measure the effectiveness of using SSA to minimize data communication between control nodes, we examined a set of DSP functions. DSP functions typically exhibit a large amount of parallelism making them ideal for hardware implementation. The DSP functions were taken from the MediaBench test suite [99] (See TABLE 3:). The files were compiled into CDFGs using the SUIF compiler infrastructure [52] and the Machine-SUIF [59] backend.

We performed SSA analysis with the SSA library built into Machine-SUIF. The library was initially developed at Rice [118] and recently adapted into the Machine-SUIF compiler.

First, we compare the amount of data communicated between the control nodes using the different SSA algorithms. Given two control nodes *i* and *j*, the *edge weight w(i,j)* is the amount of data communicated (in bits) from control node *i* to control node *j*. The *total edge weight (TEW)* is:

$$TEW = \sum_i \sum_j w(i, j)$$

TABLE 3:MediaBench functions

Application	C File	Description
mpeg2	getblk.c	DCT block decoding
adpcm	adpcm.c	ADPCM to/from 16-bit PCM
epic	convolve.c	2D general image convolution
jpeg	jctrans.c	Transcoding compression
rasta	fft.c	Fast Fourier Transform
rasta	noise_est.c	Noise estimation functions

Function	Name	# Control Nodes
adpcm_coder	adpcm1	33
adpcm_decoder	adpcm2	26
internal_expland	convolve1	101
internal_filter	convolve2	101
compress_output	jctrans	33
Decode_MPEG2_Intra_Block	getblk1	75
Decode_MPEG2_Non_Intra_Block	getblk2	60
decode_motion_vector	motion	15
FAST	fft1	14
FR4TR	fft2	76
comp_noisepower	noise_est1	153
Det	noise_est2	12

FIGURE 21 is a comparison of edge weights using three different algorithms for positioning the Φ-nodes. We compare the minimal, semi-pruned and pruned algorithms. Recall that the pruned algorithm is the best algorithm in terms of reducing the number of Φ-nodes, but worst in runtime. The minimal algorithm produces many Φ-nodes, but has small runtime. The semi-pruned algorithm provides a middle ground in terms of runtime and quality of result.

We divide the TEW of the minimal and semi-pruned algorithm (respectively) by the TEW of the pruned algorithm. We call this the *TEW ratio*. We use the pruned algorithm as a baseline because it consistently produces the smallest TEW. Referring to FIGURE 21, the TEW of the minimal algorithm is much worse than that of the pruned algorithm. For example, in the benchmark fft2, the TEW of the minimal algorithm is over 70 times that of the TEW of the pruned algorithm. The semi-pruned algorithm yields a TEW that is smaller than that of the minimal algorithm, but still slightly larger than the TEW of the pruned algorithm. All algorithms have the same asymptotic runtime and the actual runtimes for all the algorithms over all the benchmarks were very small (under 1 second). Therefore, we feel that one should use the pruned algorithm as it minimizes data communication much better than the other two algorithms. Furthermore, the actual additional runtime needed to run the pruned algorithm is miniscule.

Each of the algorithms we compared attempt to minimize the number of Φ-nodes, and not the data communication. There is obviously a relationship between the number of Φ-nodes and the amount of data communication. Every Φ-node defines additional data communication, but there can be inter-node data transfer without Φ-nodes. Furthermore, as we pointed out in Section 6.2, minimizing the number of Φ-nodes does not directly correspond to minimizing the data communication.

In FIGURE 21, we compare the ratio of Φ-nodes and the ratio of TEW using the minimal and pruned algorithms. As you can see, the number of Φ-nodes is highly related to the amount data communication. As the Φ-node ratio increases, the TEW ratio increases. Correspondingly, a large Φ-node ratio corresponds to a large TEW ratio. This lends validation to the using SSA algorithms to first minimize inter-node communication and then use the spatial Φ-node repositioning to further reduce the data communication. In other words, minimizing the number of Φ-nodes is a good objective function to initially minimize data communication.

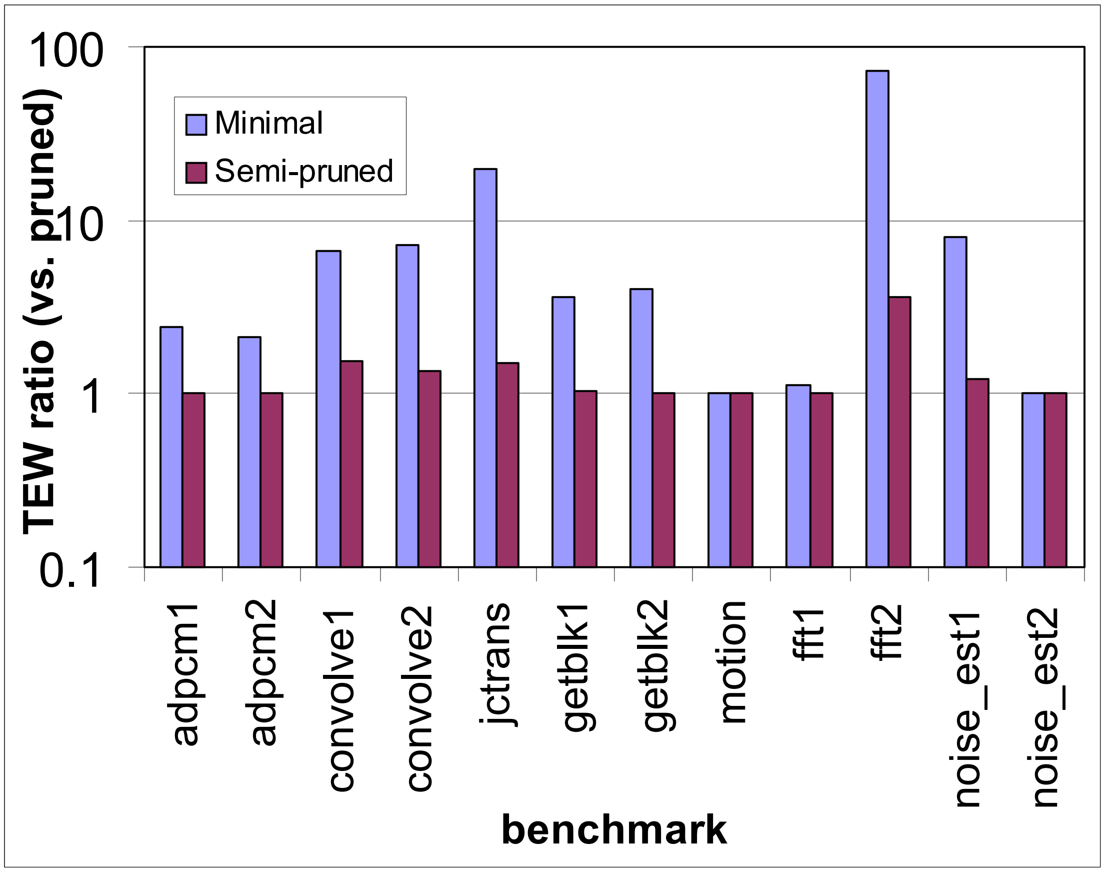


FIGURE 21 Comparison of total edge weight (TEW) between the minimal and semi-pruned TEW and the pruned TEW

We have focused on the total edge weight as a model of "goodness" for realizing the circuit in hardware. This model makes intuitive sense; by minimizing the amount of data that we pass within the circuit we should produce a "good" implementation of that circuit. Essentially, we are arguing that the TEW is a good metric in which to judge the quality of a circuit implementation.

In order to judge TEW as a metric of the quality of a circuit implementation, we used our system synthesis framework (described in CHAPTER 4) to realize each of the benchmarks as a hardware implementation. We synthesized each of the benchmarks using Synopsys Behavioral Compiler for architectural synthesis followed by Synopsys Design Compiler for logic synthesis. Then, we gathered the area statistics for each of the designs using the three different SSA algorithms. We present these statistics in TABLE 4: and 0.

TABLE 4: A circuit area comparison of the semi-pruned vs. pruned algorithm for the different benchmarks after logic synthesis.

benchmark	Combinatorial Area	Non-combinatorial Area	Net Interconnect Area	Total Area
Adpcm1	1.066027635	1.100004445	1.073872612	1.080671
Adpcm2	1.037206486	1.072816942	1.049062082	1.053845
convolve1	1.123069718	1.22310294	1.167967886	1.163818
convolve2	1.099277111	1.168578398	1.133835937	1.129477
jctrans	1.22556928	1.307342492	1.270941382	1.266603
getblk1	1.006448108	1.013098313	1.00529323	1.007845
getblk2	1.016337872	1.017142405	1.015978929	1.016409
Motion	1	1	1	1
fft1	0.960122699	1	0.99766437	0.987354
fft2	1.740987897	1.964909634	1.870670518	1.851336
Noise_est2	1	1	1	1
Average	1.115913346	1.16972687	1.144116995	1.141578

TABLE 5:A circuit area comparison of the semi-pruned vs. pruned algorithm for the different benchmarks after logic synthesis.

benchmark	Combinatorial Area	Non-combinatorial Area	Net Interconnect Area	Total Area
adpcm1	1.83250905	1.969198631	1.94100117	1.917612
adpcm2	1.879903115	2.067246881	1.984804053	1.981969
getblk1	2.93336955	4.635891202	3.095529065	3.461292
motion	1	1	1	1
fft1	2.375648891	1.218323587	2.105318539	1.866718
Noise_est2	1	1	1	1
average	1.836905101	1.981776717	1.854442138	1.871265

The two tables show three different elements for the area of the circuit. The combinatorial area is the area used by the gates that implement data path components (e.g. adders, multipliers). The non-combinatorial area is that of the memory and steering logic components (e.g. flip-flops, multiplexers). The area of the wires is given by the net interconnect area. The total area is simply the sum of the three previous elements. We give the ratio of the area of the semi-pruned (minimal) algorithm versus the pruned algorithm in a similar manner as the TEW ratios. Some of the benchmarks were omitted from the tables. These benchmarks ran out of memory on our servers (even a 4 processor Sun server with 2 GB memory!)

FIGURE 22 charts the total area ratios of the benchmarks. The figure demonstrates that our assumptions about minimizing the TEW have a good correlation with minimizing the area of the circuit. Comparing FIGURE 21 with FIGURE 22, you can see that the amount of reduction in TEW correlates with the amount of reduction in total area. For example, the TEW for the benchmark fft2 using the semi-pruned algorithm is approximately 5 times that of the pruned algorithm. A similar result is seen in the total area ratio; the area of the semi-pruned algorithm is about 1.8 times that of the area of the pruned algorithm. Furthermore, the area of the pruned algorithm is almost always the best algorithm in terms of total area. Fft1 is the lone exception, most likely due to its small size and limited number of phi-nodes. And even with this lone outlier, the overall average area improvements are 14% and 87% better using the pruned algorithm over the semi-pruned and minimal algorithm. It is safe to say that the type of SSA algorithm has a huge effect on the area of the circuit implementation. Furthermore, the results indicate that it is worth the small increase in runtime to use the pruned algorithm.

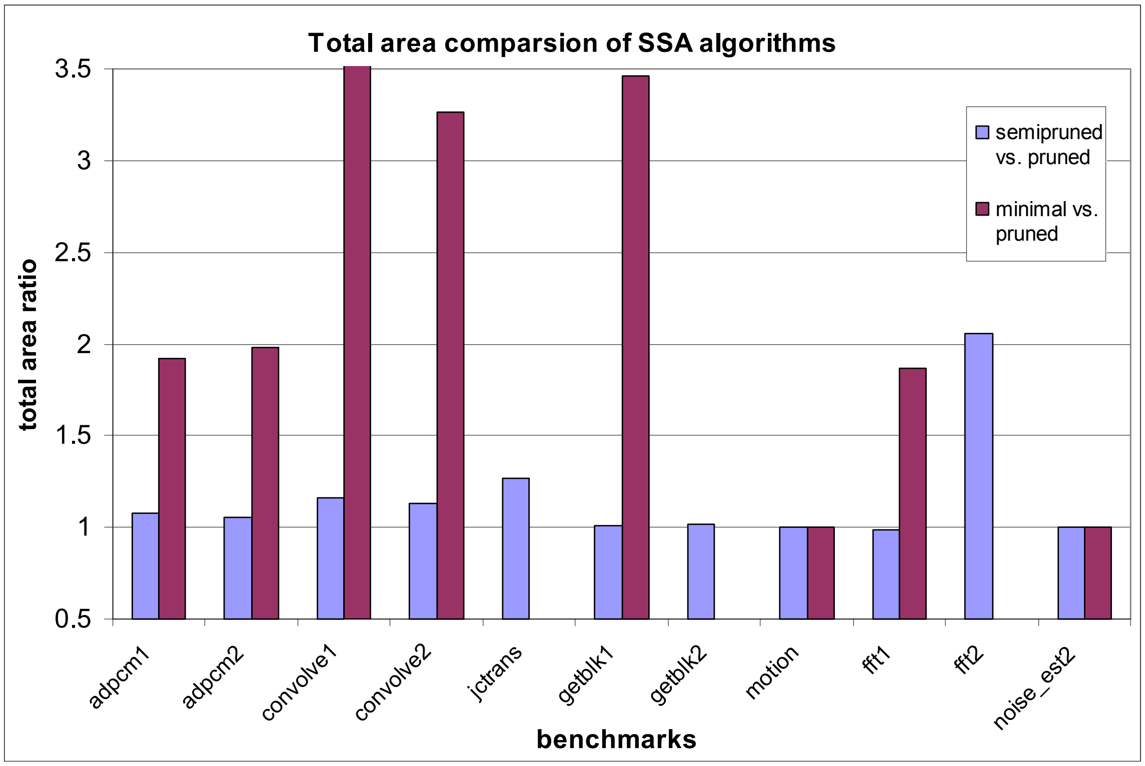


FIGURE 22 A total area comparison of the benchmarks after logic synthesis. The ratio is the minimal (semipruned) total area divided by the pruned total area.

Our next set of experiments focus on using spatial SSA Φ-node distribution to further minimize the amount of data communication. FIGURE 23 shows the number of Φ-nodes that are spatially distributed by the spatial SSA algorithm. We can see that these Φ-nodes are fairly common; in some of the benchmarks, over 35% of the Φ-nodes are spatially moved. The average number of distributed Φ-nodes over all the benchmarks is 11.65%, 18.21% and 13.56%[5] for the pruned, semi-pruned and minimal algorithms, respectively.

[5] Not all of the benchmarks are included in Figure 22; the omitted benchmarks have 0 Φ-nodes that should be distributed, but these benchmarks are included in the averages.

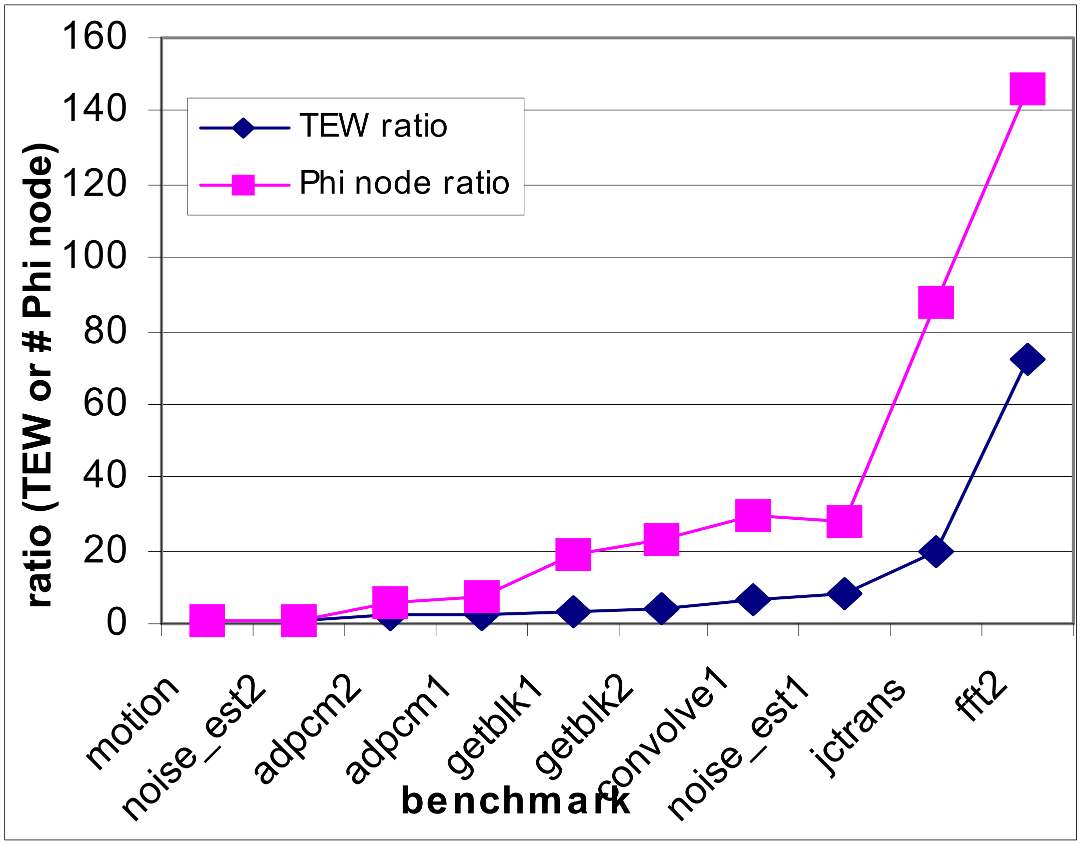


FIGURE 23 A comparison of total edge weight (TEW) and the number of Φ-nodes using the minimal and pruned algorithms.

FIGURE 25 gives the percentage of TEW improvement we achieve by spatially distributing the nodes. By spatially distributing the Φ-nodes, we reduce the TEW by 1.80%, 4.77% and 8.16% in the pruned, semi-pruned and minimal algorithms, respectively. We believe the small amount of improvement in TEW can be attributed to two things. First of all, the TEW contributed by the Φ-nodes is only a small portion of the total TEW. Also, when the number of Φ-nodes is small, the number of Φ-nodes to distribute is also small. This is apparent in the increasing trend seen by the pruned, semi-pruned and minimal algorithms. There are many Φ-nodes when we use the minimal algorithm and correspondingly, there TEW improvement of the minimal algorithm is the

8.16%. Conversely, the number of Φ-nodes in the pruned algorithm is small and the TEW improvement is also small.

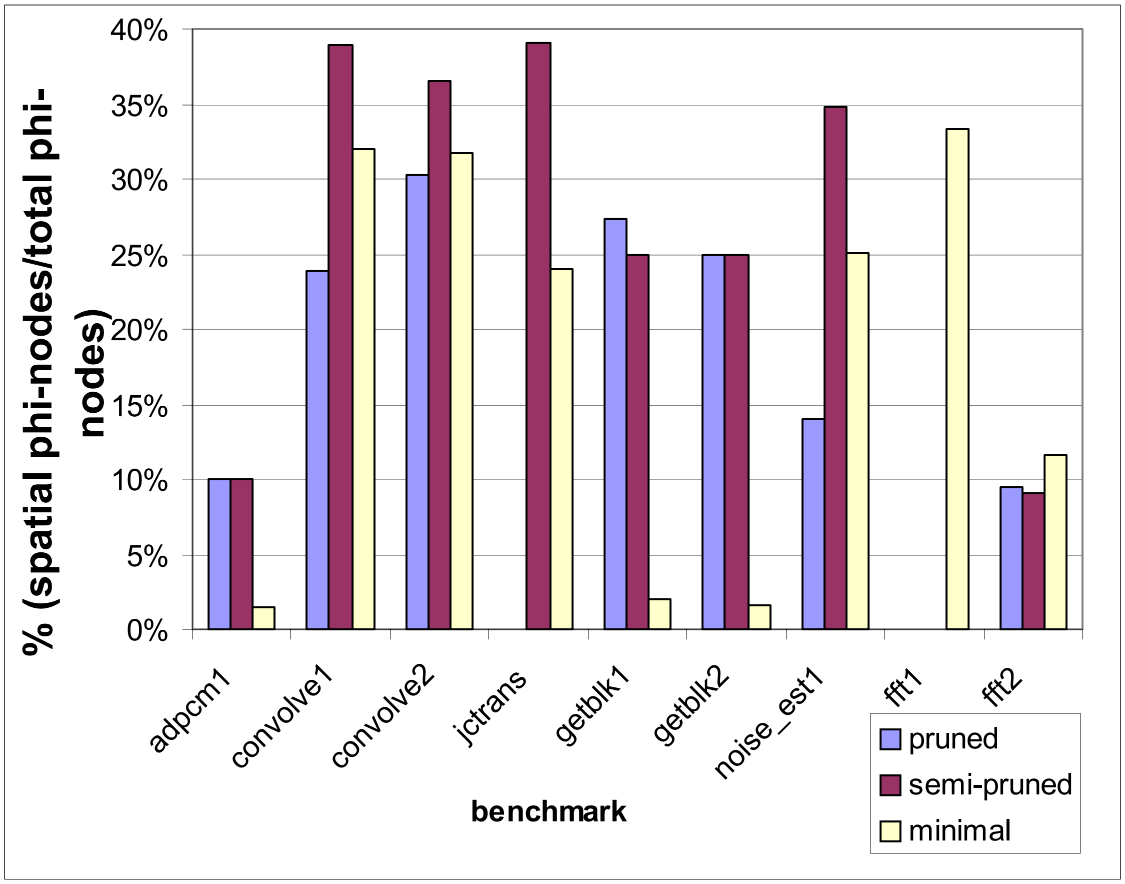


FIGURE 24 Comparison of the number of spatially distributed Φ-nodes and the total number Φ-nodes using the three SSA algorithms.

We ran the spatial algorithm through our system framework to determine the actual area improvements achieved by spatially distributing the phi-nodes. The results are shown in FIGURE 26. The results are very mixed and mostly negative. The chart plots the total area of the temporal (original) phi node placement divided by the total area of the spatial phi node placement as described our spatial phi node placement algorithm. A result above 1 denotes that the temporal area is larger than the spatial area, meaning that our spatial phi node placement algorithm is beneficial. The benchmarks getblk1 and getblk2 benefit immensely from the spatial phi node placement. The other benchmarks either have worse total area or the total area is approximately the same i.e. the total area ratio is equal to 1.

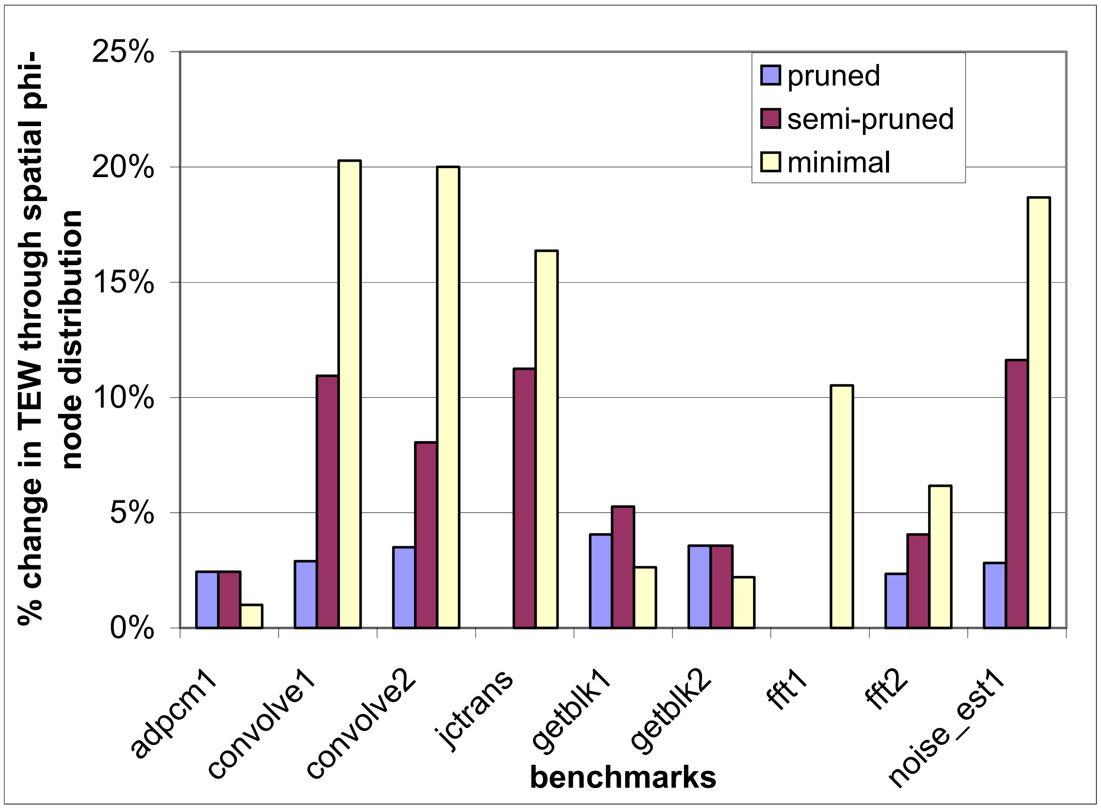


FIGURE 25 The percentage change in of total edge weight when we distribute the Φ-nodes using the three SSA algorithms.

We believe that the results are somewhat negative for several reasons. First of all, as stated previously, the TEW reduction when using the spatial algorithm is not that large. The TEW reduction was 1.80%, 4.77% and 8.16% using the pruned, semi-pruned and minimal algorithms. Furthermore, by spatially distributing the phi nodes we are duplicating the multiplexers in order to reduce the amount of global data communication. We are not taking into account the effect of duplicating the multiplexers, i.e. the additional area that they will contribute to the circuit. Essentially, we are assuming that the interconnect area of the global wires (those counted in the TEW) contribute more area to the circuit than the multiplexers that we are replicating. Depending on the distance that the global interconnect spans, this may not always be the case. We must take into account the additional area added by the multiplexers with respect to the interconnect area. This is one of the flaws of using the TEW as a model, since it ignores the area of the multiplexers and does not take into account the actual length of the edges.

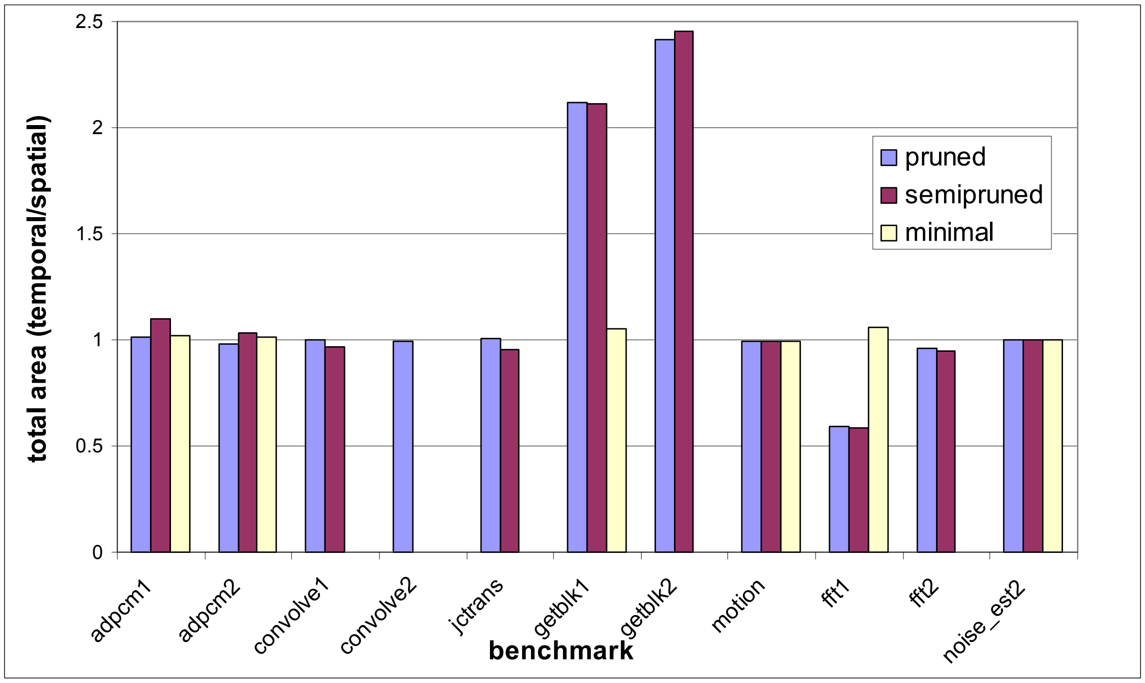


FIGURE 26 Comparison of the total area of the temporal versus spatial phi node placement for the three SSA algorithms.

6.4 Related Work

The idea of hardware compilation has been discussed since the 80's. At that time, it was under the guise of silicon compilation and related closely to what is referred to as behavioral synthesis nowadays.

The past 15 years have brought about a number of platforms that take high-level code and generate a hardware configuration for that platform. The PRISM project [119] took functions implemented in a subset of C and compiled them to their FPGA-like architecture. The Garp compiler [83] takes automatically maps C code to their MIPS +

FPGA architecture. The DeepC compiler [120] is the most similar to our work, as it synthesizable Verilog from C or Fortran. These are some of the more prevalent academic works in hardware compilation. The SystemC [121] and SpecC [87] languages have created much industrial interest in hardware compilation. Many companies including Synopsis and Cadence are exploring hardware compilation from these two languages.

Many compiler techniques use SSA for analysis or transformation [122-124]. Also, there have been modifications of SSA form [125, 126]. To the best of our knowledge, this is the first work that considers SSA form for hardware compilation.

6.5 Summary

In this work, we presented methods needed for hardware compilation. First, we described a framework for compiling a high-level application to an HDL. The framework includes methods for transforming a traditional compiler IR to an RTL-level HDL. We illustrated how to transform the IR into a CDFG form. Using the CDFG form, we explained methods to control the path of execution. Furthermore, we gave methods for communicating data between the control nodes of the CDFG.

We examined the use of SSA to minimize the amount of data communication between control nodes. We showed a shortcoming of SSA when it is applied to minimizing data communication. The temporal positioning of the Φ-node is not optimal in terms of data communication. We formulated an algorithm to spatially distribute the Φ-node to minimize the amount of data communication. We showed that this spatial distribution can decrease the data communication by 20% for some DSP functions. Additionally, we proved that if all data communication wire-lengths are of unit cost, the Spatial SSA Algorithm provides minimal data communication.

CHAPTER 7 Increasing Hardware Parallelism

7.1 Introduction

Hardware has a distinct advantage over software as operations can be spatially computed; this allows parallelism among the operations. Unfortunately, an application programmer does not always exploit all of the parallelism available in the application. We must employ techniques that discover additional parallelism so that the hardware performance is maximized.

The number of operations in a basic block – hence an initial control node[6] – can be quite small. Studies show that the average instruction level parallelism (ILP) per basic block is between 2 – 3.5 [127]. By combining basic blocks, one can increase the amount of ILP. There are many compiler optimizations that combine basic blocks such as trace scheduling [48], superblock [128] and hyperblock formation [101].

7.2 Trace Scheduling and Superblocks

Compilers for superscalar and VLIW processors increase parallelism by combining frequently executing sequences of basic blocks. Combining basic blocks allows advanced scheduling and optimization by ignoring the control constraints associated with the alternate paths of execution. *Trace scheduling* finds a *complete trace* – a path from the start to the end of the CDFG – and combines the basic blocks on that path.

Trace scheduling incurs a large amount of complexity (particularly during scheduling) in order to maintain correct program execution. In particular, side entrances to the trace create an immense amount of bookkeeping. The superblock eliminates side entrances; this makes scheduling and other compiler optimizations much easier. A

[6] Initially, control nodes are basic blocks. In this section, we show how to incorporate many basic blocks into a single control node. Hence, a control node may no longer be a basic block.

superblock is a set of basic blocks from one path of control where execution may only begin at the top but may leave at one or more exit points.

Every method that we discussed to increase the parallelism combines control nodes so that the operations in these nodes can be executed in parallel. There are many proposed methods for combining control node sequences. A runtime method profiles branching characteristics of the application. Based on this information, commonly occurring control node sequences are combined. This requires input sequences, which may not always be available. Also, different input sequences may cause drastically different profile information. But, when applications that have commonly occurring, well-represented input sequences, the profile information is immensely helpful for combining control nodes.

Hank et al. [129] develop a static method for superblock formation. They use heuristics to predict the commonly taken branches through the program structure. Using this branch frequency prediction, they combine basic blocks without hazards. A hazard is an instruction or group of instructions whose side effects may not be completely determined at compile time e.g. I/O instructions, subroutine calls, jumps with indirect target addresses. They show that static methods can rival the dynamic techniques.

There are many factors to consider during superblock formation. First and foremost, we wish to increase the parallelism. By increasing the number of operations in the superblock, we increase the opportunity for parallelism. But, there are several factors that may reduce the amount of parallelism we can achieve. For instance, a large amount of dependencies between operations in different control nodes of the superblock will limit the number of instructions that we can schedule at a particular time. Therefore, we want to minimize the length of the critical path of the superblock.

When we consider a hardware implementation of the application, the selection of control nodes to include in the superblock can have a direct effect on minimizing the amount data communication. By combining basic blocks into superblocks or traces, we are effectively choosing a hardware partitioning. The control nodes in the superblock must remain in the same vicinity or partition. Additionally, the data communication between the nodes of a superblock becomes local. *Local communication* – defined as data communication between control nodes in the same partition – has smaller delay than *global communication* – the data communication between partitions.

7.3 Hardware Partitioning using Trace Scheduling and Superblocks

In this section, we describe two problems associated with increasing the parallelism of the hardware. We wish to create a trace schedule or set of superblocks that produce a good partitioning of the application. In particular, we attempt to maximize the local interconnect of the hardware. This is accomplished by partitioning the operations such that the data communication within the partitions is maximized. In essence, we attempt to maximize the amount of local communication by placing control nodes that have a lot of communication between them in the same superblocks or a trace.

Given a CDFG $C(N,E_c,E_d)$, each node $n \in N$ has a weight w_n. The vertex weight w_n represents the amount of area occupied by an implementation of the computations in the node. We can use actual synthesized node as the area or some type of area estimation. The CDFG has two distinct sets of directed edges, E_c corresponds to the control flow constraints and E_d corresponds to the data communication between the nodes. There is a weight w_e for each $e_d \in E_d$. The edge weight w_e of edge $e(n_i, n_j)$ (i and j are two different control nodes) corresponds the amount of data that is being transferred along that edge (data being transferred from control node i to control node j). We wish to partition the CDFG such that the sum of the node weights within a partition is less than some constant. Additionally, the nodes in a partition must be superblock or complete trace. Furthermore, we attempt to maximize the sum of the edge weights between any two nodes in the same partition.

By partitioning the control nodes, we accomplish two things. First, the edges that are local to a partition have local data communication. Local communication has smaller delay than global communication. By maximizing the edge weights within the partitions, we minimize the amount of global delay. Second, by restricting the partition to a set of control nodes that form a superblock or complete trace, we increase the parallelism in the

CDFG. The area restriction on the partition insures that the CDFG for each partition has approximately the same size. We can use this to limit the amount of operations that can be optimized by data flow synthesis techniques. Also, this allows us to keep the silicon area of the each control node under a specified constraint.

In addition to area restrictions on the partitions, we could have parallelism limits. Ideally, control nodes would have a large amount of parallelism so that the data flow scheduler would have the freedom use take advantage of that parallelism, if it so desires. In order to realize this, we must have a method to determine the amount of parallelism present in a control node. A quick metric could use an ASAP scheduling. Then, we could determine the amount of parallelism at each time step. From that, an "average" parallelism can be determined. This method gives an upper bound on the average amount of parallelism. It also gives a lower bound on the delay. A more complex, yet more accurate metric could use one of the many data flow scheduling algorithms. Since trace scheduling must form a complete trace, it may not always allow an area constraint.
The trace scheduling partitioning and superblock partitioning problems can be formulated as follows:

Trace Scheduling Partitioning Problem Definition: Given a CDFG and a minimum parallelism constraint *P*, determine a complete trace that maximizes the data communication between the nodes of the trace such that the parallelism of the trace is greater than *P*.

Superblock Partitioning Problem Definition: Given a CDFG, a minimum parallelism constraint *P* and a maximum area constraint *A*, determine a set of superblocks that maximizes the local communication such that the parallelism of each superblock is greater than *P* and the combined node area of superblock is less than *A*.

In the next section, we show the effectiveness of trace scheduling and superblock formation for hardware partitioning.

7.4 Experimental Results

We developed two greedy algorithms for trace scheduling and superblock formation. The algorithms greedily select control nodes based on data communication. The trace scheduling algorithm starts at the entry control node and iteratively chooses the next control node of the trace based on the successor control node that has the greatest amount communication data with the nodes that are already in the trace. The superblock formation algorithm iteratively creates superblocks. An initial node for the superblock is selected by finding the node (which is not already in another superblock) with the greatest amount of data communication to other nodes that are not currently a part of another superblock. Then, the algorithm proceeds like the trace scheduling algorithm until it's successors are all part of another superblock or we reach the area limit. We use a function of additional area added by a node to the superblock and the amount of additional data communication added to the superblock in order to choose the next node to add to the superblock.

We ran the two algorithms on the benchmarks described in Table 3.1 in Section 3.6. We look at the number of operations per partition and the amount of local data communication after partitioning for the two algorithms. As in the previous section, we use the data communication edge weight as a measure of the data communication. First, we look at the number operations in each partition. The number of operations relates to the area of the partition. Since a trace schedule must form a complete trace, it is much harder to control the number of operations. When we use superblocks for partitioning, we can easily add an area constraint to the problem formulation since the superblock does not have the complete trace restriction.

FIGURE 27 displays the number of operations per partition for trace scheduling and superblock formation. For trace scheduling, the percentage is simply the number of operations in the trace compared to the total number of operations. For superblocks, the percentage is the average number of operations per superblock compared to the total number of operations. The number of operations in the trace can be quite large; the trace consumes approximately 80% in the adpcm1 benchmark. This mainly stems from the fact that there is no effective way of limiting the operations because we must have a complete trace. In our experiments, we choose limit the area of the operations for superblock scheduling. Most of the benchmarks had an average area under 20%. The two exceptions are fft1 and motion. Both of these benchmarks had a few nodes with a large amount of operations in them. The greedy algorithm chose to include these nodes as superblocks because they had a large amount of data communication going to and from them.

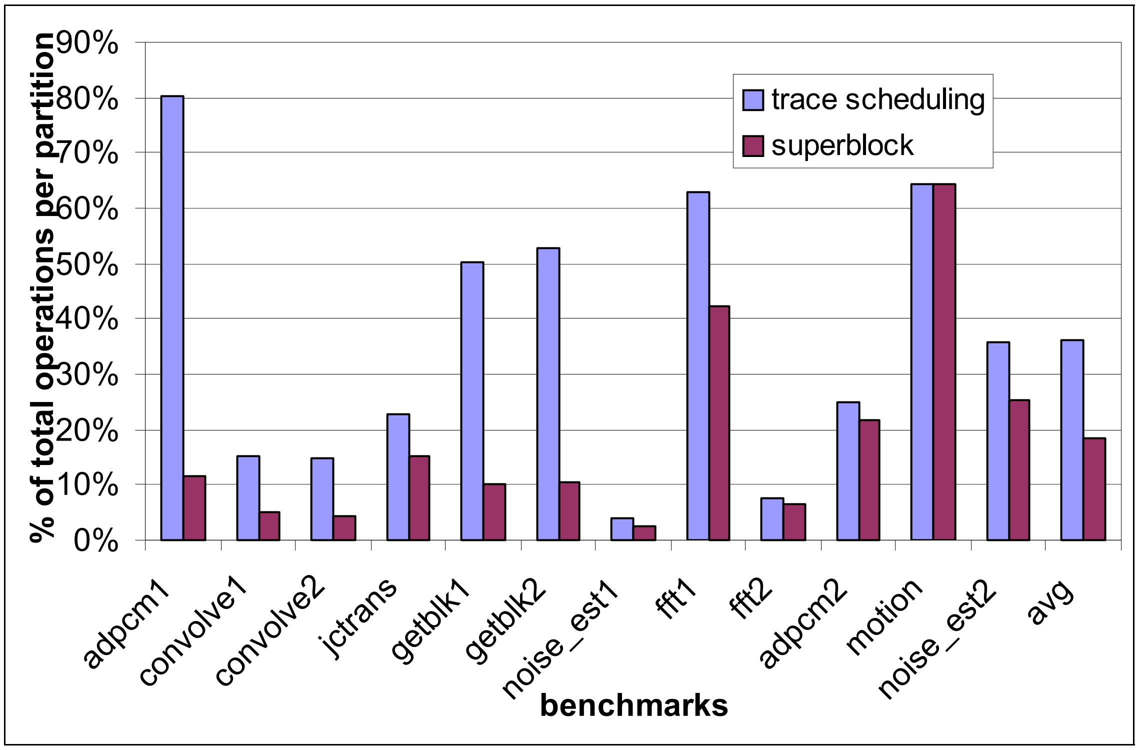


FIGURE 27 The percentage of operations per partition for trace scheduling and superblock formation.

The main objective of partitioning is to maximize the amount of local communication. FIGURE 28 shows the amount of local communication for both trace scheduling and superblock formation. Over all the benchmarks, trace scheduling converted an average of 12.7% of the total communication to local communication. On average, superblock formation localized 17.5% of the overall data communication. For the benchmarks where trace scheduling has more local communication than superblocks, the number of operations in the trace is inordinately large. For example, the trace scheduling partitioning of adpcm1, getblk1, getblk2, and fft1 outperform the superblock partitioning. This is mainly because the traces have a large amount of the total operations in them. Referring to FIGURE 27 each of these traces has over 50% of the total operations in it. This is unacceptable as a partition. Even though we restrict the number of operations in the superblocks, we can still get a good partitioning in terms of maximizing local data communication.

Our results show that superblock formation is much better technique for increasing parallelism during hardware partitioning. Additionally, the side entrances of trace scheduling constrain the scheduler. Superblocks eliminate side entrances; hence they do not have such a problem. Therefore, we believe that superblocks are a far more effective structure for increasing parallelism during hardware partitioning.

7.5 Related Work

The idea of hardware compilation has been discussed since the 80's. At that time, it was under the guise of silicon compilation and related closely to what is referred to as behavioral synthesis nowadays.

The past 15 years have brought about a number of platforms that take high-level code and generate a hardware configuration for that platform. The PRISM project [119] took functions implemented in a subset of C and compiled them to their FPGA-like architecture. The Garp compiler [83] takes automatically maps C code to their MIPS + FPGA architecture. The DeepC compiler [120] is the most similar to our work, as it creates synthesizable Verilog from C or Fortran. These are some of the more prevalent academic works in hardware compilation. The SystemC [121] and SpecC [87] languages have created much industrial interest in hardware compilation. Many companies including Synopsys and Cadence are exploring hardware compilation from these two languages.

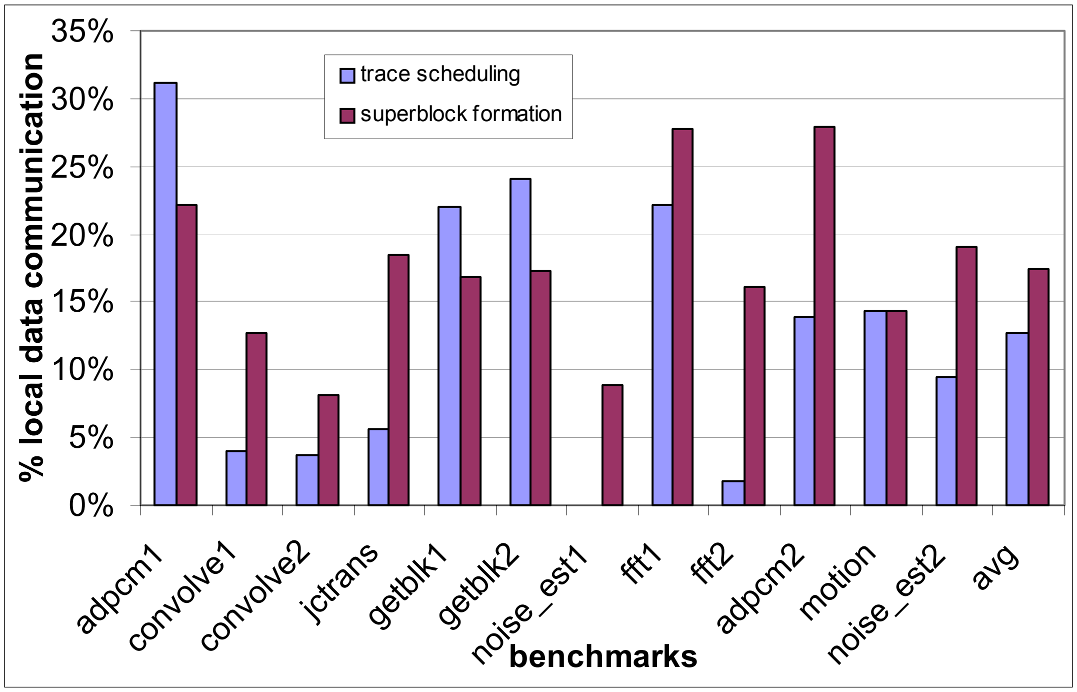


FIGURE 28 The percentage of local data communication in the partitions created by trace scheduling and superblock formation.

7.6 Summary

This chapter looked at several optimizations problems that occur during system synthesis. Specifically, we looked at techniques for mapping an application onto a microarchitecture, i.e. converting CDFGs to a synthesizable hardware description language. We described how several compiler techniques could be adapted for system synthesis.

We looked into the problem of increasing the hardware parallelism of an application. Two compiler methods for increasing parallelism – trace scheduling and superblock formation – were studied. We proposed additional constraints onto these two problems that transformed them into the problem of increasing parallelism during hardware partitioning. Our results show that the superblock is much more effective than trace scheduling for this purpose.

In the future, we plan to complete the flow from CDFG to synthesizable HDL. This allows us determine the effects that our optimizations have on an actual hardware implementation of the applications. Furthermore, we intend to develop better algorithms for the hardware partitioning using superblocks. Also, we will look into the benefits and drawbacks of using hyperblocks for hardware partitioning.

CHAPTER 8 Future Research Directions

There are many possible directions for future research. I will touch on a few directions that could be explored using my framework.

.

Expanding Template Generation and Matching using Predicated Execution: The first area involves techniques to expand the functionality of the versatile parameterizable blocks. We are restricted by control flow as the VPBs must execute atomically. But, if we could use predicated execution, i.e. speculatively perform part of the operations of the VPB, then we could be able to increase the complexity and occurrences of the VPBs. This would lead to gains in computation speed and power consumption, as we would perform fewer operations on the slow, power hungry fine-grain reconfigurable fabric.

Region Formation for Hybrid Reconfigurable Architectures: Another area of future research is the formation of regions – portions of the control data flow graph. The formation of regions has been studied in Very Long Instruction Word (VLIW) architectures. Yet, it has never been looked at in the context of hardware compilation for reconfigurable architectures. There are many fundamental differences between VLIW architectures and reconfigurable architectures. The main difference is that reconfigurable architectures allow different functional units various times during the running of the application and a possibly different set of functional units for different applications. The VLIW architecture is permanently fixed. I believe that this changes the problem in a basic manner and the VLIW region formation algorithms must be revisited for reconfigurable architectures.

Speculative Scheduling in Control Data Flow Graphs: Speculative execution executes an operation before control flow dictates that it executes. It is helpful to increase the parallelism, which in turn increases the hardware's efficiency. There are many factors to consider, including the critical path, the average number of operations to each of the exits, and increasing the number of operations in the application code. Once again, reconfigurable architectures present different challenges than other types of architectures making this a novel research area.

Predicated Execution and Hyperblock Formation for Hardware Synthesis: Speculatively executing operations and creating hyperblocks has been studied for for VLIW computers. But, it has not been looked at for hardware synthesis. I plan to identify the differences and between these two areas and develop algorithms for hyperblock formation. Additionally, I plan to look at different methods of predicated execution. Should there be a predicated bit for each register as in done in VLIW processors or is a different scheme better in hardware synthesis?

Application Partitioning/Estimation for System-on-chip: The majority of the research on hardware/software partitioning assumes a single processor (software) and an ASIC (hardware) on the same chip. Future computing systems will have many different types of computing elements – ASIC, many different processors, FPGA and other programmable hardware. We need partitioning methods to determine the computing element where each portion of the application should run. In order to quickly and accurately perform this partitioning, we need estimation engine for each of the computing elements. The estimation engine takes a portion of the application and determines different metrics (e.g. power consumption, speed, die area, utilization) for each computing element.

Memory Synthesis and Management for System-on-chip: The performance and location of the system's memory components have an enormous role in the throughput, latency, power, area and other performance characteristics of a system. Often, the components of the system have embedded local memory elements, e.g. embedded RAM of an FPGA. Efficient use of these elements is paramount to the performance of the application on the device. Additionally, different memory hierarchies can affect the area and timing characteristics of the system. A memory hierarchy for one application may not be suitable for another. Synthesis techniques that take into account the distinct features of the application are an interesting and powerful technique for optimizing system performance.

Hybrid Local/Global Controllers: A global control for determining the execution of the hardware resources is beneficial as the circuit is directed from only one location and one controller. Yet, the centralized nature of a global controller has the drawback of having to connect to every resource of the circuit. This may cause routing and signal delay problems depending on the size of the circuit. A set of local controllers alleviates this problem, yet introduces the problem of synchronization among the controllers. A meet in the middle approach would have a small number of "global" controllers that direct local controllers. This could reduce the routing complexity of routing without the need for a large amount of synchronization. This hierarchy could be extended even further, where there are multiple levels of global and local controllers, where synchronization is done within each level and the control is between two adjacent levels of the hierarchy. An automatic push-button synthesis for hybrid local/global controllers would be interesting and beneficial to the design of digital systems.

CHAPTER 9 REFERENCES

[1] M. J. Wirthlin and B. L. Hutchings, "A dynamic instruction set computer," Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, 1995.

[2] S. Hauck, "The roles of FPGAs in reprogrammable systems," *Proceedings of the IEEE*, vol. 86, pp. 615-38, 1998.

[3] D. A. Buell and K. L. Pocek, "Custom computing machines: an introduction," *Journal of Supercomputing*, vol. 9, pp. 219-29, 1995.

[4] A. DeHon, "Comparing computing machines," Proceedings of the Configurable Computing: Technology and Applications, 1998.

[5] J. E. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. H. Touati, and P. Boucard, "Programmable active memories: reconfigurable systems come of age," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 4, pp. 56-69, 1996.

[6] A. DeHon and J. Wawrzynek, "Reconfigurable computing: what, why, and implications for design automation," Proceedings of the Design Automation Conference, 1999.

[7] E. Mirsky and A. DeHon, "MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources," Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, 1996.

[8] J. R. Hauser and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor," Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 1997.

[9] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, "The Chimaera reconfigurable functional unit," Proceedings of the Symposium on Field-Programmable Custom Computing Machines, 1997.

[10] D. Gajski and R. Kuhn, "Guest Editors' Introduction: New VLSI Tools," *IEEE Computer*, vol. 16, pp. 11-14, 1983.

[11] K. Keutzer, A. R. Newton, J. M. Rabaey, and A. Sangiovanni-Vincentelli, "System-level design: orthogonalization of concerns and platform-based design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, pp. 1523-43, 2000.

[12] H. Trickey, "Flamel: A high level hardware compiler," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-6, pp. 259-69, 1987.

[13] S. A. Hayati, A. C. Parker, and J. J. Granacki, "Representation of control and timing behavior with applications to interface synthesis," Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors, 1988.

[14] G. De Micheli, *Synthesis and optimization of digital circuits*. New York: McGraw-Hill, 1994.

[15] Atmel, FPSLIC, http://www.atmel.com/atmel/products/prod39.htm

[16] "ARM-Based Embedded Processor PLDs," Altera Corporation 2001.

[17] Xilinx, "Virtex-II Pro Platorm FPGA Handbook," 2002.

[18] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to software: Raw machines," *Computer*, vol. 30, pp. 86-93, 1997.

[19] P. M. Athanas and H. F. Silverman, "Processor reconfiguration through instruction-set metamorphosis," *Computer*, vol. 26, pp. 11-18, 1993.

[20] K. Bondalapati, P. Diniz, P. Duncan, J. Granack, M. Hall, R. Jain, and H. Ziegler, "DEFACTO: a design environment for adaptive computing technology," Proceedings of the 11th IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing, 1999.

[21] M. B. Gokhale and J. M. Stone, "NAPA C: compiling for a hybrid RISC/FPGA architecture," Proceedings of, 1998.

[22] I. Page, "Constructing hardware-software systems from a single description," *Journal of VLSI Signal Processing*, vol. 12, pp. 87-107, 1996.

[23] T. J. Callahan and J. Wawrzynek, "Instruction-level parallelism for reconfigurable computing," Proceedings of, 1998.

[24] G. Vanmeerbeeck, P. Schaumont, S. Vernalde, M. Engels, and I. Bolsens, "Hardware/software partitioning of embedded system in OCAPI-xl," Proceedings of, 2001.

[25] C. J. Alpert and A. B. Kahng, "Recent directions in netlist partitioning: a survey," *Integration, The VLSI Journal*, vol. 19, pp. 1-81, 1995.

[26] F. Vahid, G. Jie, and D. D. Gajski, "A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning," Proceedings of, 1994.

[27] J. Henkel, "A low power hardware/software partitioning approach for core-based embedded systems," Proceedings of, 1999.

[28] M. R. Garey and D. S. Johnson, "Computers and intractability. A guide to the theory of NP-completeness," 1979.

[29] A. Kalavade, "System Level Codesign of Mixed Hardware-Software Systems," UCB, PhD Dissertation ERL 95/98, September 1995.

[30] D. E. Thomas, J. K. Adams, and H. Schmit, "A model and methodology for hardware-software codesign," *IEEE Design & Test of Computers*, vol. 10, pp. 6-15, 1993.

[31] A. Kalavade and E. A. Lee, "A hardware-software codesign methodology for DSP applications," *IEEE Design & Test of Computers*, vol. 10, pp. 16-28, 1993.

[32] A. Kalavade and E. A. Lee, "The extended partitioning problem: hardware/software mapping, scheduling, and implementation-bin selection," *Design Automation for Embedded Systems*, vol. 2, pp. 125-63, 1997.

[33] R. K. Gupta and G. De Micheli, "System-level synthesis using re-programmable components," Proceedings of the European Conference on Design Automation, 1992.

[34] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," *IEEE Design & Test of Computers*, vol. 10, pp. 64-75, 1993.

[35] P. Zebo and K. Kuchcinski, "An algorithm for partitioning of application specific systems," Proceedings of the European Conference on Design Automation, 1993.

[36] R. Niemann and P. Marwedel, "Hardware/software partitioning using integer programming," Proceedings of the European Design and Test Conference, 1996.

[37] J. Henkel and R. Ernst, "A hardware/software partitioner using a dynamically determined granularity," Proceedings of the Design Automation Conference, 1997.

[38] F. Vahid and L. Thuy Din, "Extending the Kernighan/Lin heuristic for hardware and software functional partitioning," *Design Automation for Embedded Systems*, vol. 2, pp. 237-61, 1997.

[39] Celoxica, Frequently Asked Questions: DK1 Design Suite, http://www.celoxica.com/products/technical_papers/faqs/FAQDK1002_2.pdf

[40] A. Balboni, W. Fornaciari, and D. Sciuto, "Co-synthesis and co-simulation of control-dominated embedded systems," *Design Automation for Embedded Systems*, vol. 1, pp. 257-89, 1996.

[41] R. Camposano and J. Wilberg, "Embedded system design," *Design Automation for Embedded Systems*, vol. 1, pp. 5-50, 1996.

[42] M. Chiodo, D. Engels, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, K. Suzuki, and A. Sangiovanni-Vincentelli, "A case study in computer-aided co-design of

embedded controllers," *Design Automation for Embedded Systems*, vol. 1, pp. 51-67, 1996.

[43] L. Yanbing, T. Callahan, R. Harr, U. Kurkure, and J. Stockwood, "Hardware-software co-design of embedded reconfigurable architectures," Proceedings of the Design Automation Conference, 2000.

[44] T. Callahan and J. Wawrzynek, "Adapting Software Pipelining for Reconfigurable Computing," Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), 2000.

[45] M. Weinhardt and W. Luk, "Pipeline vectorization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, pp. 234-48, 2001.

[46] T. C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, pp. 841-848, 1961.

[47] P. G. Paulin and J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, pp. 661-79, 1989.

[48] J. A. Fisher, "Trace scheduling: a technique for global microcode compaction," *IEEE Transactions on Computers*, vol. C30, pp. 478-90, 1981.

[49] R. Potasman, J. Lis, A. Nicolau, and D. Gajski, "Percolation based synthesis," Proceedings of the ACM/IEEE Design Automation Conference, 1990.

[50] F. J. Kurdahi and A. C. Parker, "REAL: a program for REgister ALlocation," Proceedings of the ACM/IEEE Design Automation Conference, 1987.

[51] S. S. Muchnick, *Advanced compiler design and implementation*. San Francisco, Calif.: Morgan Kaufmann Publishers, 1997.

[52] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, L. Shih-Wei, E. Bugnion, and M. S. Lam, "Maximizing multiprocessor performance with the SUIF compiler," *Computer*, vol. 29, pp. 84-9, 1996.

[53] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys*, vol. 26, pp. 345-420, 1994.

[54] J. J. Dongarra and A. R. Hinds, "Unrolling loops in FORTRAN," *Software - Practice and Experience*, vol. 9, pp. 219-26, 1979.

[55] S. Weiss and J. E. Smith, "A study of scalar compilation techniques for pipelined supercomputers," Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems, 1987.

[56] M. Lam, "Software pipelining: an effective scheduling technique for VLIW machines," Proceedings of the Conference on Programming Language Design and Implementation, 1988.

[57] B. R. Rau and C. D. Glaeser, "Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing," Proceedings of the Microprogramming Workshop, 1981.

[58] J. Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein, "Software pipelining showdown: optimal vs. heuristic methods in a production compiler," Proceedings of the Programming Language Design and Implementation, 1996.

[59] M. D. Smith and G. Holloway, "An introduction to machine SUIF and its portable libraries for analysis and optimization," Division of Engineering and Applied Sciences, Harvard University.

[60] A. M. Dewey, *Analysis and design of digital systems with VHDL*. Boston: PWS Pub. Co., 1997.

[61] 1076.3-1997 IEEE Standard VHDL Synthesis Packages,

[62] G. Holloway and M. D. Smith, "Machine SUIF Control Flow Graph Library," Division of Engineering and Applied Sciences, Harvard University 2002.

[63] G. Holloway and M. D. Smith, "Machine-SUIF SUIFvm Library," Division of Engineering and Applied Sciences, Harvard University 2002.

[64] G. Holloway and M. D. Smith, "Machine-SUIF Machine Library," Division of Engineering and Applied Sciences, Harvard University 2002.

[65] M. C. Rinard and P. C. Diniz, "Commutativity analysis: a new analysis framework for parallelizing compilers," Proceedings of the Programming Language Design and Implementation, 1996.

[66] M. C. Rinard and P. C. Diniz, "Commutativity analysis: a new analysis technique for parallelizing compilers," *ACM Transactions on Programming Languages and Systems*, vol. 19, pp. 942-91, 1997.

[67] R. Rugina and M. Rinard, "Pointer analysis for multithreaded programs," Proceedings of the Programming Language Design and Implementation (PLDI), 1999.

[68] R. Rugina and M. Rinard, "Automatic parallelization of divide and conquer algorithms," Proceedings of the Symposium on Principles and Practice of Parallel Programming, 1999.

[69] K. Zee and M. Rinard, "Write barrier removal by static analysis," *SIGPLAN Notices*, vol. 37, pp. 32-41, 2002.

[70] A. Salcianu and M. Rinard, "Pointer and escape analysis for multithreaded programs," Proceedings of the Symposium on Principles and Practice of Parallel Programming, 2001.

[71] F. Vivien and M. Rinard, "Incrementalized pointer and escape analysis," *SIGPLAN Notices*, vol. 36, pp. 35-46, 2001.

[72] R. Rugina and M. Rinard, "Symbolic bounds analysis of pointers, array indices, and accessed memory regions," Proceedings of the Programming Language Design and Implementation (PDLI), 2000.

[73] J. Whaley and M. Rinard, "Compositional pointer and escape analysis for Java programs," Proceedings of the Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99), 1999.

[74] R. Schreiber, S. Aditya, B. R. Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider, "High-level synthesis of nonprogrammable hardware accelerators," Proceedings of the International Conference on Application-Specific Systems, Architectures, and Processors, 2000.

[75] R. E. Gonzalez, "Xtensa: a configurable and extensible processor," *IEEE Micro*, vol. 20, pp. 60-70, 2000.

[76] M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti, "Building and using a highly parallel programmable logic array," *Computer*, vol. 24, pp. 81-9, 1991.

[77] P. M. Athanas and A. L. Abbott, "Real-time image processing on a custom computing platform," *Computer*, vol. 28, pp. 16-25, 1995.

[78] Z. Peixin, M. Martonosi, P. Ashar, and S. Malik, "Using configurable computing to accelerate Boolean satisfiability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, pp. 861-8, 1999.

[79] R. P. S. Sidhu, A. Mei, and V. K. Prasanna, "String matching on multicontext FPGAs using self-reconfiguration," Proceedings of the International Symposium on Field Programmable Gate Arrays, 1999.

[80] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "PipeRench: a reconfigurable architecture and compiler," *Computer*, vol. 33, pp. 70-77, 2000.

[81] C. Ebeling, D. C. Cronquist, and P. Franklin, "RaPiD-reconfigurable pipelined datapath," Proceedings of the Workshop on Field-Programmable Logic and Applications, 1996.

[82] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The raw microprocessor: a computational fabric for software circuits and general-purpose programs," *IEEE Micro*, vol. 22, pp. 25-35, 2002.

[83] T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "The Garp architecture and C compiler," *Computer*, vol. 33, pp. 62-69, 2000.

[84] S. Ogrenci Memik, E. Bozorgzadeh, R. Kastner, and M. Sarrafzadeh, "Strategically programmable systems," Proceedings of the Reconfigurable Architecture Workshop, 2001.

[85] A. DeHon, "DPGA utilization and application," Proceedings of the International Symposium on Field Programmable Gate Arrays, 1996.

[86] T. Grotker, *System Design with SystemC*. Boston: Kluwer Academic Publishers, 2002.

[87] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauser, and S. Zhoa, *SpecC: Specification Language and Methodology*. Boston: Kluwer Academic Publishers, 2000.

[88] S. A. Edwards, "An Esterel compiler for large control-dominated systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, pp. 169-83, 2002.

[89] A. Girault, L. Bilung, and E. A. Lee, "Hierarchical finite state machines with multiple concurrency models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, pp. 742-60, 1999.

[90] E. Bozorgzadeh, S. Ogrenci Memik, R. Kastner, and M. Sarrafzadeh, "Pattern selection: customized block allocation for domain-specific programmable systems," Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, 2002.

[91] E. Bozorgzadeh, S. Ogrenci Memik, R. Kastner, and M. Sarrafzadeh, "SPS: Strategically programmable system - fully automated architecture generation and application compilation," UCLA, Los Angeles, Technical Report 020004, 2002.

[92] R. Kastner, S. Ogrenci Memik, E. Bozorgzadeh, and M. Sarrafzadeh, "Instruction generation for hybrid reconfigurable systems," Proceedings of the International Conference on Computer Aided Design, 2001.

[93] R. Kastner, E. Bozorgzadeh, S. Ogrenci Memik, and M. Sarrafzadeh, "Compiler techniques for system synthesis optimization," UCLA, Los Angeles, Technical Report 020002, 2002.

[94] S. Ogrenci Memik, E. Bozorgzadeh, R. Kastner, and M. Sarrafzadeh, "A super-scheduler for embedded reconfigurable systems," Proceedings of the International Conference on Computer Aided Design, 2001.

[95] S. Cadambi and S. C. Goldstein, "CPR: a configuration profiling tool," Proceedings of the Symposium on Field-Programmable Custom Computing Machines, 1999.

[96] D. S. Rao and F. J. Kurdahi, "On clustering for maximal regularity extraction," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, pp. 1198-208, 1993.

[97] C. Ebeling and O. Zajicek, "Validating VLSI circuit layout by wirelist comparison," Proceedings of the International Conference on Computer-Aided Design, 1983.

[98] S. Micali and V. V. Vazirani, "An O(√|V| |E|) algorithm for finding maximum matching in general graphs," Proceedings of the Symposium on Foundations of Computer Science, 1980.

[99] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," Proceedings of the International Symposium on Microarchitecture, 1997.

[100] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W. W. Hwu, "A comparison of full and partial predicated execution support for ILP processors," Proceedings of the International Symposium on Computer Architecture, 1995.

[101] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," Proceedings of the International Symposium on Microarchitecture, 1992.

[102] L. Tai, D. Knapp, R. Miller, and D. MacMillen, "Scheduling using behavioral templates," Proceedings of the Design Automation Conference, 1995.

[103] A. Chowdhary, S. Kale, P. Saripella, N. Sehgal, and R. Gupta, "A general approach for regularity extraction in datapath circuits," Proceedings of the International Conference on Computer-Aided Design, 1998.

[104] T. J. Callahan, P. Chong, A. DeHon, and J. Wawrzynek, "Fast module mapping and placement for datapaths in FPGAs," Proceedings of the International Symposium on Field Programmable Gate Arrays, 1998.

[105] R. Mehra and J. Rabaey, "Exploiting regularity for low-power design," Proceedings of the International Conference on Computer-Aided Design, 1996.

[106] M. Kahrs, "Matching a parts library in a silicon compiler," Proceedings of the International Conference on Computer-Aided Design, 1986.

[107] K. Keutzer, "DAGON: technology binding and local optimization by DAG matching," Proceedings of the Design Automation Conference, 1987.

[108] S. Note, W. Geurts, F. Catthoor, and H. De Man, "Cathedral-III: architecture-driven high-level synthesis for high throughput DSP applications," Proceedings of the Design Automation Conference, 1991.

[109] M. R. Corazao, M. A. Khalaf, L. M. Guerra, M. Potkonjak, and J. M. Rabaey, "Performance optimization using template mapping for datapath-intensive high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, pp. 877-88, 1996.

[110] K. Compton, A. Sharma, S. Phillips, and S. Hauck, "Flexible Routing Architecture Generation for Domain-Specific Reconfigurable Subsystems," Proceedings of the International Symposium on Field Programmable Logic and Applicaions, 2002.

[111] K. Compton and S. Hauck, "Totem: Custom Reconfigurable Array Generation," Proceedings of the Symposium on FPGAs for Custom Computing Machines Conference, 2001.

[112] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeick, "An efficient method of computing static single assignment form," Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, 1989.

[113] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson, "Practical improvements to the construction and destruction of static single assignment form," *Software - Practice and Experience*, vol. 28, pp. 859-81, 1998.

[114] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, pp. 451-90, 1991.

[115] S. L. Graham and M. Wegman, "A fast and usually linear algorithm for global flow analysis," *Journal of the Association for Computing Machinery*, vol. 23, pp. 172-202, 1976.

[116] K. Kennedy, "A survey of data flow analysis techniques," in *Program flow analysis. Theory and applications*, N. D. Jones, Ed. Englewood Cliffs, NJ, USA: Prentice-Hall, 1981, pp. 5-54.

[117] J. B. Kam and J. D. Ullman, "Global data flow analysis and iterative algorithms," *Journal of the Association for Computing Machinery*, vol. 23, pp. 158-71, 1976.

[118] P. Briggs, T. Harvey, and L. Simpson, "Static Single Assignment Construction," 196.

[119] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. Athanas, H. Silverman, and S. Ghosh, "PRISM-II compiler and architecture," Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines, 1993.

[120] J. Babb, M. Rinard, C. A. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasinghe, "Parallelizing applications into silicon," Proceedings of the Seventh

Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 1999.

[121] Open SystemC Initiative, http://www.systemc.org

[122] P. Briggs and K. D. Cooper, "Effective partial redundancy elimination," Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, 1994.

[123] P. Briggs, K. D. Cooper, and L. T. Simpson, "Value numbering," *Software - Practice and Experience*, vol. 27, pp. 701-24, 1997.

[124] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting equality of variables in programs," Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, 1988.

[125] L. Carter, B. Simon, B. Calder, and J. Ferrante, "Predicated static single assignment," Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques, 1999.

[126] W. Amme, N. Dalton, J. von Ronne, and M. Franz, "SafeTSA: a type safe and referentially secure mobile-code representation based on static single assignment form," *SIGPLAN Notices*, vol. 36, pp. 137-47, 2001.

[127] H. C. Torng and S. Vassiliadis, *Instruction-level parallel processors*. Los Alamitos, Calif.: IEEE Computer Society Press, 1995.

[128] W. M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm,

and D. M. Lavery, "The superblock: an effective technique for VLIW and superscalar compilation," *Journal of Supercomputing*, vol. 7, pp. 229-48, 1993.

[129] R. E. Hank, S. A. Mahlke, R. A. Bringmann, J. C. Gyllenhaal, and W. W. Hwu, "Superblock formation using static program analysis," Proceedings of the 26th Annual International Symposium on Microarchitecture, 1993.