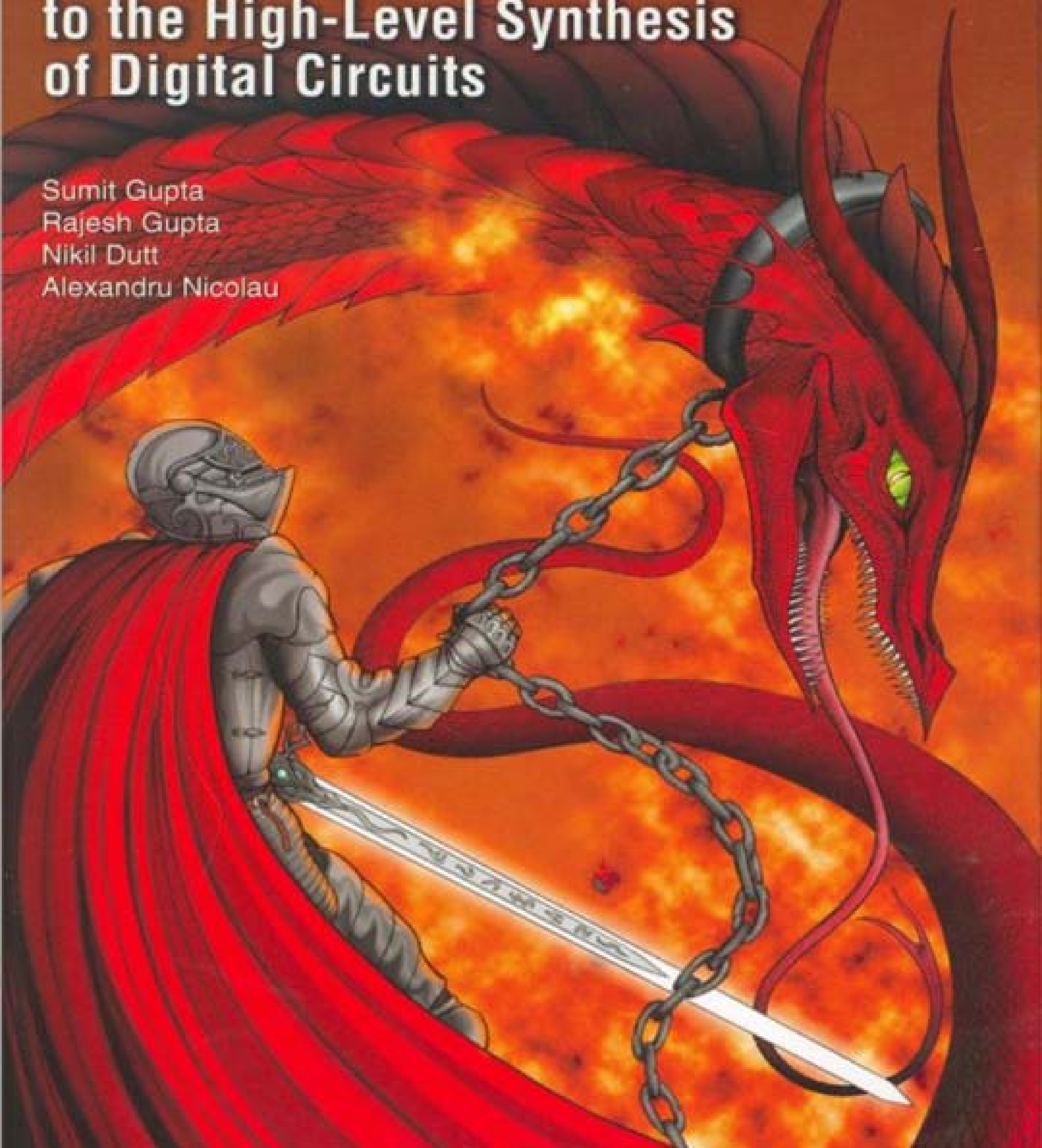
SPARK

A Parallelizing Approach to the High-Level Synthesis of Digital Circuits

Sumit Gupta
Rajesh Gupta
Nikil Dutt
Alexandru Nicolau



SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits

SPARK: A PARALLELIZING APPROACH TO THE HIGH-LEVEL SYNTHESIS OF DIGITAL CIRCUITS

SUMIT GUPTA
Center for Embedded Computer Systems
University of California
San Diego and Irvine, USA

RAJESH K. GUPTA
Department of Computer Science and Engineering
University of California
San Diego, USA

NIKIL D. DUTT
Center for Embedded Computer Systems
University of California
Irvine, USA

ALEXANDRU NICOLAU
Center for Embedded Computer Systems
University of California
Irvine, USA

KLUWER ACADEMIC PUBLISHERS
NEW YORK, BOSTON, DORDRECHT, LONDON, MOSCOW

CD-ROM available only in print edition
eBook ISBN: 1-4020-7838-2
Print ISBN: 1-4020-7837-4

Created in the United States of America

Visit Springer's eBookstore at: http://www.ebooks.kluweronline.com
and the Springer Global Website Online at: http://www.springeronline.com

This book is dedicated to our families.

Contents

IV Future Directions 185

V Appendix 191

List of Figures

List of Tables

Preface

Perhaps nothing better states the capability, reach and potential of microelectronic chips than Moore's Law. Driven by the integration advantages, semiconductor chips have become the driver for continued advance in systems capabilities for a broad range of domains from computing, communications to automotive and consumer electronics. However, as manufacturing costs of these parts drops, our ability to build the future system-chips has not kept pace. Today, our ability to manufacture complex system-chips strictly exceeds the ability to correctly design and verify these. As a consequence, a significant part of the inherent capabilities of the silicon is left on the table: ASIC parts barely run at 500 MHz speeds on processes that can easily support 2-3 GHz operation. In addition to this, the barrier to chip design is becoming higher: it takes $20M to design a reasonably complex ASIC part. Consequently, designers must find efficient ways to exploit semiconductor manufacturing advantages in building systems. The movement to structured ASICs and FPGA is a part of that process. But that is not enough: the increasing complexity of design demands moving to higher levels of abstraction and an automated approach to the synthesis of circuits from these abstractions.

An important step in this move to higher levels of abstraction is high-level synthesis. *High-level synthesis* (HLS) is the process of building microelectronic circuits from their behavioral descriptions. These descriptions are often written in a programming language. Despite its great need, HLS has not found wide acceptance in the engineering practice. The primary barriers to its adoption have been reluctance by the circuit designers to let go of the cycle boundary (in other words, describe circuit behavior without reference to specific cycle-by-cycle operations) and loss in quality and controllability of the circuits when using automated synthesis tools compared to manually crafted designs. Both of these barriers are coming down – driven by the ever expanding scope of what is put on a single chip coupled with shrinking design times and resources.

This book describes a novel approach to HLS – that of *parallelizing high-level synthesis* (PHLS). This approach employs aggressive code parallelizing and code motion techniques to discover circuit optimization opportunities beyond what is possible with traditional high-level synthesis. In this way, our PHLS approach creates a path to synthesis that competes with the quality and controllability of manually designed circuits.

The code parallelization techniques in our PHLS approach consist of a range of coarse-grain and fine-grain transformations. We devise ways to first apply these techniques prior to actual synthesis (in a "pre-synthesis" phase) with the goal of exposing opportunities for code parallelization that can be used in later scheduling and synthesis tasks. During scheduling, we employ code transformations that consist of dynamic (compiler) transformations that re-order, speculate and sometimes even replicate code to improve design performance. Selecting the right subset of these and employing them in conjunction with other synthesis related transformations such as operation chaining, resource sharing, et cetera is a difficult problem. Our approach is to build a toolbox of transformations that are guided by a set of heuristics specific to design styles that guide application of these transformations so that cycle time and circuit delay is optimized while keeping circuit area, interconnect costs, and critical path lengths in check.

Parallelizing compilers, i.e., compilers that produce code for execution on parallel machines, have been a fertile ground for design of automatic parallelization techniques, often with a strong focus on the completeness of the depth and scope of parallelization achieved. However, a straightforward application of known parallelizing compiler techniques to hardware synthesis does not work. For hardware synthesis, control costs – circuits that control the flow of data and control in the design and hence, enable parallelization of operations and tasks – are not fixed and can vary significantly by changes in the behavioral description. Rather than brute force parallelization, one needs to find a judicious balance between parallelization and sequencing that would ultimately result in a faster, better circuit. Doing so in a systematic manner for all possible approaches to code parallelization is a tall order. This book focuses on parallelization across control boundaries – for these often define the bottleneck in achieving the highest speeds and in exploitation of the hardware available. That means, examining and transforming control structures – such as complicated nested loops and conditionals – to achieve competitive synthesis results.

This book presents a systematic approach to the use of parallelizing transformations in HLS through speculation and dynamic code transformation techniques. We begin by giving some background and an overview of HLS and describe how parallelizing transformations fit in with HLS. In Chapter 2, we review relevant previous work in the domains of HLS and parallelizing compilers. We then first introduce the models and representations we use in the rest of the book. We describe the intermediate representation we use to capture the design, models for hardware resources and the speculative code motions. We also formulate the scheduling problem for high-level synthesis of designs with control flow. This sets up the stage for describing our solution for this problem.

In Chapter 4, we present our parallelizing high-level synthesis methodology; we give an overview of the design flow through our methodology and list the passes and code transformations that we found to be useful for high-level synthesis. We then describe these code transformations in detail over the next two chapters: the pre-synthesis compiler transformations in Chapters 5 and the parallelizing compiler, synthesis, and dynamic transformations we employ during scheduling in Chapter 6. Throughout the book, and particularly in these two chapters, we use examples to illustrate how the compiler transformations optimize code for hardware synthesis. However, as mentioned earlier, straightforward application of compiler transformations does not always lead to optimized circuits. So, in Chapter 7, we present algorithms that we developed to guide the compiler transformations employed during the scheduling phase. We also illustrate how we employ these compiler transformations in conjunction with more traditional HLS transformations such as operation chaining and resource sharing.

The aggressive code parallelization transformations that we apply on the design significantly increases resource sharing and hence, the size and complexity of the steering (multiplexers) and control logic. In order to manage this complexity, we developed a resource binding methodology that reduces the size of the interconnect or multiplexers. We present this methodology and the network flow solution we developed in Chapter 8. In this chapter, we also discuss our control synthesis technique that generates a finite-state machine (FSM) controller for the synthesized design.

We implemented our PHLS methodology and compiler techniques in a software framework called SPARK. We present implementation and usage details of the SPARK software package in Chapter 9. Detailed instructions on the command-line options and how to use the synthesis scripts in the SPARK software are listed later in the Appendix (along with the results of some sample runs). In Chapter 9, we also demonstrate how a designer can control and guide the synthesis process in SPARK using the synthesis scripts and command-line directives. In the next chapter, we look at a few design examples derived from multimedia and image processing applications to illustrate the efficacy and results of the parallelizing optimizations. We then (in Chapter 11) use a case study of an instruction length decoder derived from the Intel Pentium-class of microprocessors to walk the reader through the steps involved in the synthesis of a typical microprocessor functional block with complex control flow. We conclude the book with a discussion of future directions in Chapter 12.

We included a CD with this book that contains the SPARK prototype software (please see CD for disclaimer and copyright notice). This software release consists of the SPARK executable (for Linux, Solaris, and Windows), a User Manual, and a tutorial that walks the user through the synthesis of a design example extracted from a MPEG-1 player using SPARK. This software release and updates are also available on the Internet at:

http://www.cecs.uci.edu/~spark

We look forward to your comments and suggestions about this book and the accompanying software. Please send them at spark@cecs.uci.edu

S. GUPTA, N. D. DUTT, R. K. GUPTA, A. NICOLAU

Acknowledgments

We thank the many people, past and present, at the Center for Embedded Computer Systems (CECS) who have contributed to the formulation and refinement of the ideas presented in this book and in reviewing early manuscripts and providing valuable feedback. We specially acknowledge the contribution of Nick Savoiu to the underlying SPARK framework. Others who contributed to the SPARK code-base include Mehrdad Reshadi and Sunwoo Kim. We are grateful to Melanie Sanders for going out of her way to help everyone at CECS over the years. We thank the following people for reviewing early drafts of this book (in the form of Sumit Gupta's Ph.D. thesis): Nikhil Bansal, Radu Cornea, Frederic Doucet, Andreas Gerstlauer, Ravindra Jejurikar, Arun Kejariwal, Manev Luthra, Prabhat Mishra, Dan Nicolaescu, Cristiano Pereira, Mehrdad Reshadi, and Srikanth Srinivasan. We would also like to thank Arun Kejariwal for helping with the aesthetics and layout of the book.

We thank the Semiconductor Research Corporation (SRC) and Intel Incorporated for funding the SPARK project. We particularly acknowledge the mentorship of Justin Harlow from SRC and the advice and collaboration of Shai Rotem, Timothy Kam, and Michael Kishinevsky from the Intel Strategic CAD Labs.

Part I

Introduction to High-Level Synthesis

1

INTRODUCTION

1.1 System-Level Design of Hardware Systems

Microelectronic systems have become the medium of choice for the state of the arts in end capabilities in human endeavors from science (computing) to space exploration. These systems can be found *embedded* in all aspects of our daily lives ranging from equipment, places and even the human body. Advances in microelectronic technologies enable integration of ever increasing computation and communication capabilities on a single chip. However, the rate of increase of technology advances has far outpaced the ability of designers to use the real-estate and speed available on chips. Furthermore, rapid changes in the protocols, applications and consumer demands of embedded systems have created a need to design, synthesize and deploy these systems within a short turnaround time. As a result, it is now possible to build microelectronic chips that execute large and complex applications, but it is not possible to do so without spending a small fortune and while meeting the short times-to-market that are the hallmark of modern consumer products.

To manage the growing size and complexity of these microelectronic systems and increase designer productivity, we need to perform design modeling, synthesis and validation at higher levels of abstraction. As designers we continually seek the ability to partially or completely synthesize the hardware and software of an embedded system from a system level description. This book focuses on one aspect of this goal, namely, high-level synthesis. *High-level synthesis is the automated generation of the hardware circuit of a digital system from a behavioral description.*

High-level synthesis is an important component of a system level design methodology. Figure 1.1 shows an overview of the design flow through a typical system level design methodology. The input to this system level design methodology is the specification of an application in a high-level language and the output is an implementation of the application on a system-on-a-chip (SOC) platform. Several system level design languages that are variants of C or C++ have been proposed to enable this methodology. These languages attempt to provide a unified means of specifying hardware and software. SystemC [GL97, SyC] and SystemVerilog currently find attention, but are a part of a long line of research in languages for system design [GL97, VSB99, GZD+00].

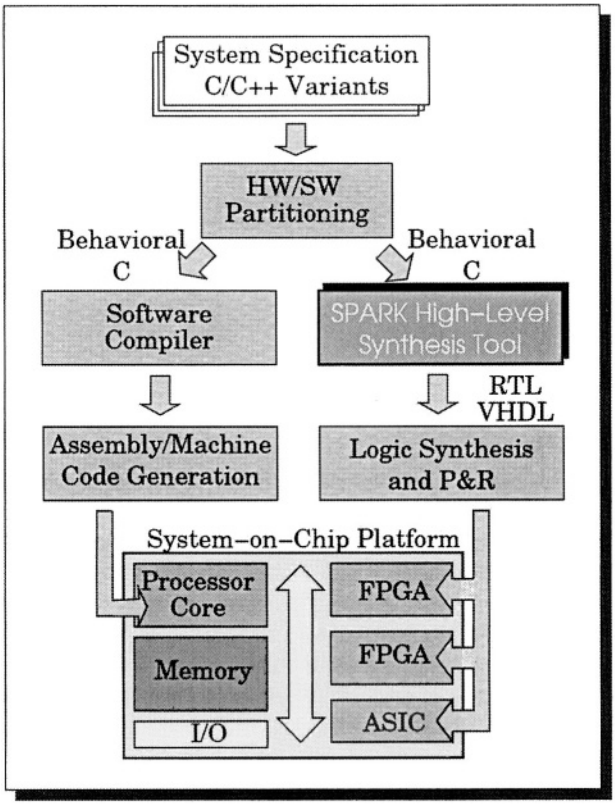


Figure 1.1: *A system-level design methodology that incorporates high-level synthesis. This book describes the* Spark *high-level synthesis framework and parallelizing high-level synthesis methodology.*

Ideally, we would like to be able to perform an analysis of the system specification to determine a hardware-software partitioning that meets the timing, hardware, performance, and power constraints placed on the application. Then, as shown in Figure 1.1, the software portion can be compiled using a software compiler for the processor core and the hardware portion is synthesized with a high-level synthesis tool, followed by logic synthesis and place and route tools.

However, in practice, existing high-level synthesis techniques do not produce circuits whose quality can compete with manual designs. As a result, there is a gap between the models represented in the system level languages and the implementation of the hardware component. The focus of this book is in bridging this gap by proposing a novel approach to high-level synthesis – that of parallelizing high-level synthesis – that generates results that are competitive with manual designs and thus, finally fulfills the promise of high-level synthesis.

1.2 Why High-Level Synthesis and Why Now

High-level synthesis was first proposed two decades ago. Several commercial tools have been released over the past decade [BC, InaCs, Cel, DS], but there has been limited adoption of these tools.

However, recently several factors have led to a renewed interest in high level synthesis from behavioral descriptions, both in the industry and in academia. Firstly, recent years have seen the widespread acceptance and use of language-level modeling of digital designs. Increasingly, the typical design process starts with design entry in a hardware description language at the register-transfer level, followed by logic synthesis. Secondly, the increasing size and complexity of digital systems and developments such as the advent of systems-on-a-chip, have fueled the use of system level behavioral modeling in high level languages for initial system specification and analysis. Also, whereas traditionally system architects could leverage experience to examine design alternatives, the large size and complexity of present day systems requires automated architectural-level design space exploration and synthesis tools.

Although high-level synthesis has been the subject of research for two decades, there are *two chief factors that have limited the utility and wider acceptance of high-level synthesis*: (a) The quality of synthesis results is compromised because of the significant control flow in typical descriptions. This control flow limits the application of most data flow driven behavioral optimizations. (b) And designers are usually given minimal controllability of the synthesis process and its result. In this work, we address these two limitations.

Poor synthesis results can be attributed to several factors. Most of the optimizing transformations that have been proposed over the years are operation level transformations. In contrast, language level optimizations refer to transformations that change the circuit description at the source level, for example, loop unrolling, loop-invariant code motion, et cetera. Few language level optimizations have been explored in the context of high-level synthesis and their effects on final circuit area and speed are not well understood. Most approaches demonstrate the effectiveness of an optimization in isolation from other transformations and on small, synthetic designs and with little or no analysis of control and hardware costs. It is thus difficult to judge if an optimization has a positive impact beyond scheduling results.

Our work in high-level synthesis (HLS) has been motivated by the advances in *parallelizing* compiler technology that enable exploitation of extreme amounts of parallelization through a range of code motion and code transformation techniques. While we found no single code transformation technique (including the ones we developed specifically for HLS) to be universally useful, we found that a judicious balance of a number of these techniques driven by well considered heuristics is likely to yield HLS results that compare in quality to the manually designed functional blocks. The challenge then, for us, has been to identify and isolate a useful set of transformations and couple these with the rest of a high-level synthesis framework. This, in essence, is the contribution of the *Spark* parallelizing high-level synthesis methodology to the domain of high-level synthesis.

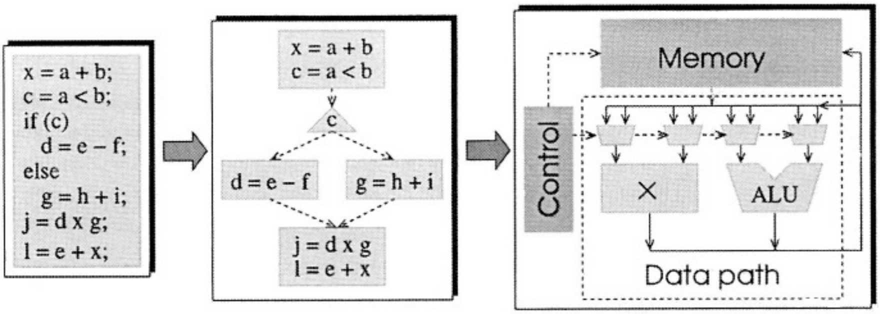


Figure 1.2: *An overview of high-level synthesis: going from a behavioral description to a digital circuit that consists of a data path, a controller and memory.*

1.3 Overview of High-Level Synthesis

An overview of high-level synthesis is shown in Figure 1.2. High-level synthesis is the process of converting a behavioral description into a digital circuit that consists of a data path, a controller and memory elements. The first task in high-level synthesis is to capture the behavioral description in an intermediate representation that captures both control flow and data flow. Thereafter, the high-level synthesis problem has usually been solved by dividing the problem into several sub-tasks. Typically the subtasks are:

- **Allocation:** This tasks consists of determining the number of resources that have been allocated or alloted to synthesize the hardware circuit. Typically, designers can specify an allocation in terms of the number of resources of each resource type. Resources consist not only of functional units (like adders and multipliers), but may also include registers and interconnection components (such as multiplexers and buses).

- **Scheduling:** The scheduling problem is to determine the time step or clock cycle in which each operation in the design executes. The ordering between the "scheduled" operations is constrained by the data (and possibly control) dependencies between the operations. Scheduling is often done under constraints on the number of resources (as specified in the resource allocation).

- **Module Selection:** Module selection is the task of determining the resource type from the resource library that an operation executes on. The need for this task arises because of the presence of several resources of different types (and different area and timing) that an operation may execute on. For example, an addition may execute on an adder, an ALU, or a multiply-accumulate unit. There are area, performance, and power trade-offs in choosing between these components. Module selection must make a judicious choice between different resources such that a metric like area or timing is minimized.

- **Binding:** Binding determines the mapping between the operations, variables and data (and control) transfers in the design and the specific resources in the resource allocation. Hence, operations are mapped to specific functional units, variables to registers, and data/control transfers to interconnection components.

- **Control Generation and Optimization:** Control synthesis generates a control unit that implements the schedule. This control unit generates control signals that control the flow of data through the data path (i.e., through the multiplexers). Control optimization tries to minimize the size of the control unit and hence, improve metrics such as area and power.

Each of these tasks are interlinked and often dependent on each other. To delve deeper into each of these tasks, we refer the reader to [DM94, GDWL92, CW91, Kna96, JDKR97].

1.4 Role of Parallelizing Compiler Transformations in High-Level Synthesis

We focus on synthesizing the computationally expensive portions of multimedia and image processing applications and of low latency functional blocks from microprocessors. These applications typically consist of arithmetic operations embedded in deeply nested loops with a complex mix of conditional (if-then-else) constructs. The choice of these control constructs (if-then-else, loops) has a dramatic effect on the quality of hardware synthesis results. Language level modeling of digital designs (especially in system-level design methodologies) allows additional freedom in the way a behavior is described compared to register-transfer level (RTL) descriptions. The ordering and placement of operations in high-level behavioral descriptions is usually governed by programming ease and varies from designer to designer. Very often this ordering is not conducive to or optimal for downstream high-level synthesis and optimization tasks [CGR93a].

A range of parallelizing transformations has been explored in the compiler community to alleviate the problems associated with code with complex control flow. Previous work has found that the average number of operations within a basic block is typically 4 to 5 [TF70] and thus, to increase resource utilization and performance, code transformations have to look beyond basic block boundaries to exploit parallelism. We found that these parallelizing transformations are equally essential for the high-level synthesis of behavioral descriptions with nested conditionals and loops.

Thus, we developed a methodology that applies a **coordinated set of coarse-grain and fine-grain parallelizing compiler, compiler, and synthesis transformations** so as to improve the synthesis results of the behavioral codes by exposing and increasing the parallelism available in the algorithmic description. Our methodology aims to improve the quality of the high-level synthesis results by many-fold and at the same time, give the designer control over the transformations applied. We employ transformations and techniques that exploit instruction-level parallelism, such as speculative code motions, percolation scheduling and Trailblazing, and coarse grain transformations such as loop transformations. Loop transformations either attempt to exploit inter-iteration parallelism (loop unrolling and loop pipelining) or inter-loop parallelism (loop fusion, loop interchange). Another class of loop transformations attempts to either reduce the number of operations executed (loop invariant code motion) or the strength of operations executed; i.e., replace computationally expensive operations such as multiplies by additions using strength reduction techniques and induction variable analysis.

Most of these parallelizing transformations coupled with basic compiler transformations, such as common sub-expression elimination, copy propagation, dead code elimination, et cetera, are essential to improve the quality of high-level synthesis results – particularly, when synthesizing from high-level languages such as behavioral VHDL, C and C++ (and variants) due to freedom in programming style afforded by these languages.

An important aspect of our approach to high-level synthesis is the application of speculative code motions that move operations across conditionals and loops based on the time criticality of an operation and in the process expose the parallelism available in the algorithm. These transformations are guided by heuristics that optimize the circuit quality in terms of latency, cycle time, circuit size, and interconnect costs.

However, straightforward application of compiler transformations does not work for high-level synthesis. This is because the increases in control, interconnect (multiplexing) and area costs are unacceptably large – particularly for designs with complex nested control flow – and are often accompanied by a corresponding increase in the critical path lengths in the synthesized circuits. Indeed, we have shown that exposing maximum parallelism does not necessarily yield the best results, and parallelizing transformations must be applied judiciously – sometimes even moving operations a non-parallelizing way (e.g., reverse and conditional speculation) in order to gain improvement in overall synthesis results. Also, the cost functions and optimization criteria for compilers are different from those of high-level synthesis. Whereas, compilers often pursue maximum parallelization, we found that in high-level synthesis, parallelizing transformations have to be tempered by their effects on the control and area (in terms of interconnect) costs. Indeed, the very nature of transformations that are useful for high-level synthesis is different from those that are useful for compilers. Transformations such as predication and loop unrolling that are very useful for compilers are not as effective and at times not even useful for high-level synthesis. Conversely, transformations that we proposed for high-level synthesis such as dynamic common-subexpression elimination and conditional branch balancing are not particularly useful in compilers.

1.5 Our Parallelizing High-Level Synthesis Methodology

We present a high-level synthesis methodology that incorporates parallelizing compiler and compiler transformations into the traditional high-level synthesis framework, both during a "pre-synthesis" phase and the scheduling phase. An overview of this methodology is shown in Figure 1.3. This methodology takes an input of the design description in a high-level language such as C and captures it using an intermediate

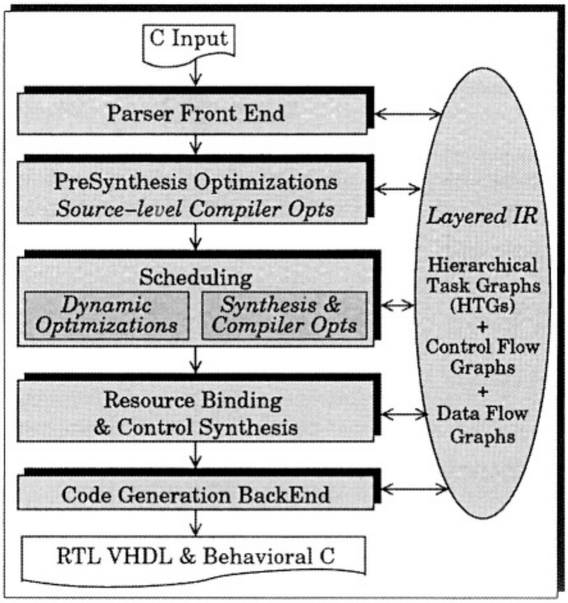


Figure 1.3: ***An overview of the our high-level synthesis flow incorporating compiler transformations during the pre-synthesis source-to-source transformation phase and the scheduling phase.***

representation that maintains the code structure and other code information such as loop index variables, et cetera.

We first apply a set of source level transformations to the input design description in a *pre-synthesis phase*. These transformations, such as common sub-expression elimination (CSE), copy propagation, dead code elimination and loop-invariant code motion [ASU86] aim to reduce the number of operations executed and remove redundant and unnecessary operations. Also, we use coarse-level loop transformation techniques such as loop unrolling to restructure the code. This increases the scope for applying parallelizing optimizations in the scheduling phase that follows.

The scheduling phase employs an innovative set of speculative, beyond-basic-block code motions that reduce the impact of syntactic variance or programming style on the quality of synthesis results. These code motions enable movement of operations through, beyond, and into conditionals with the objective of maximizing performance by extracting the inherent parallelism in designs and increasing resource utilization. Since these speculative code motions often re-order, speculate and duplicate operations, they create new opportunities to apply additional transformations "dynamically" during scheduling such as dynamic common sub-expression elimination. Dynamic branch balancing is another transformation we apply during scheduling to enable code motions, particularly those involving code duplication. These compiler transformations are integrated with the standard high-level synthesis techniques such as resource sharing, scheduling on multi-cycle operations and operation chaining. Also, since our methodology targets mixed control-data flow designs, often operations have to be chained across conditional boundaries. After scheduling the design once, we employ a loop pipelining technique called loop shifting that incrementally exposes and exploits parallelism across loop-iterations.

The pre-synthesis and scheduling transformations result in significantly shorter schedule lengths and higher resource utilization. However, these techniques also lead to an increase in the complexity of the steering logic, i.e., the multiplexers and associate control logic, in the design. Note that, we also refer to this steering logic as *interconnect.*

We minimize the complexity of the interconnect by means of a resource binding methodology that first does operation binding and then variable binding. Operations are bound to functional units such that operations with the same inputs or outputs are bound to the same functional units. Similarly, variables are bound to registers such that variables that are inputs or outputs to the same functional units are mapped to the same registers [GDWL92].

A control synthesis and optimization pass then generates a finite state machine (FSM) controller for the scheduled and bound design. This controller executes the operations as per the timing specified by the schedule and generates control signals to guide the data through the interconnect as specified by the resource binding. To complete the path from behavioral specification down to the design net-list, our methodology employs a back-end code generator that can interface with standard logic synthesis tools. This enables the evaluation of the effects of several coarse and fine-grain optimizations on logic synthesis results.

1.6 Contributions of this Work

Given the maturity of high-level synthesis techniques and equally well-researched compiler techniques, it is natural for the reader to be skeptical about the novelty of the contributions in this work. The primary contribution of this work is the careful analysis, experimentation and integration of diverse high-level synthesis, compiler and parallelizing compiler techniques for hardware synthesis. We have introduced novel speculative code motions, dynamic transformations, and a technique to chain operations across conditional boundaries. Also, we have identified a set of useful transformations from the myriad transformations that have been proposed over the years in high-level synthesis. We have devised a novel synthesis methodology that applies source-level compiler transformations in pre-synthesis phase, before scheduling the design. We apply the speculative code motions during scheduling and these are aided by a set of dynamic transformations that increase the scope of the code motions and exploit new opportunities for eliminating operations that the code motions create.

We implemented our parallelizing high-level synthesis methodology in a C-to-VHDL high-level synthesis framework called *Spark*. *Spark* has a toolbox of transformations and a set of algorithms that employ these transformations. Such a toolbox approach enables the designer to employ heuristics to drive selection and control of individual transformations under realistic cost models for high-level synthesis. The *Spark* synthesis framework is a *complete* high-level synthesis system that provides a path from an unrestricted input behavioral description down to register-transfer level (RTL) code. This RTL code can then be synthesized using commercial logic synthesis tools.

Spark enables experimentation with large “real-life” designs and more importantly it enables an analysis of the impact of the transformations on control and area costs of the gate level circuit. We give the designer the ability to control the transformations applied on the design description using scripts. Graphical outputs aid in visualizing the intermediate results. Results for experiments performed on complex industrial applications from the multimedia and image processing domain validate the utility of our approach. Furthermore, we present a case study for a design from the microprocessor domain, namely, the *instruction length decoder* from the Intel Pentium class of processors. This case study demonstrates how a designer can develop scripts to employ the code transformations in the *Spark* toolbox to synthesize designs from different domains.

1.7 Book Organization

The organization of this book is shown in Figure 1.4. We begin by presenting a survey of previous work in Chapter 2. Chapter 3 presents the models and representations used in this book. We present a 3-layered intermediate representation that captures the design description with a combination of control flow, data flow and hierarchical control graphs. We then present models for the additional input information required for scheduling such as the resource library, the clock information, data types et cetera. Based on these definitions, we formulate the resource-constrained scheduling problem in high-level synthesis. We then introduce the notion of control flow in design descriptions and present a model of speculative code motions that can be used to move operations past the control flow. Finally, we formulate the minimum resource scheduling problem in the presence of control flow.

In Chapter 4, we present our parallelizing high-level synthesis (PHLS) methodology and the design flow through a high-level synthesis framework in which this methodology is implemented. We also present some of the passes required in the transformations toolbox of a PHLS framework.

Chapter 5 presents the pre-synthesis optimizations in our PHLS framework. These include common sub-expression elimination (CSE), loop-invariant code motion, loop unrolling and loop index variable elimination.

In Chapter 6, we present the compiler and synthesis transformations employed during scheduling. We present the speculative code motions, dynamic CSE, chaining of operations across conditional boundaries, and loop shifting. We also demonstrate through examples how these transformations can be employed to improve scheduling results.

In Chapter 7, we present algorithms that integrate the various compiler and synthesis transformations into a high-level synthesis list scheduler. We first describe the software architecture of the scheduler and then present each algorithm implemented in this architecture. This includes the main list scheduling algorithm, the algorithm to gather operations to schedule, the algorithm to traverse the design graph and the algo-

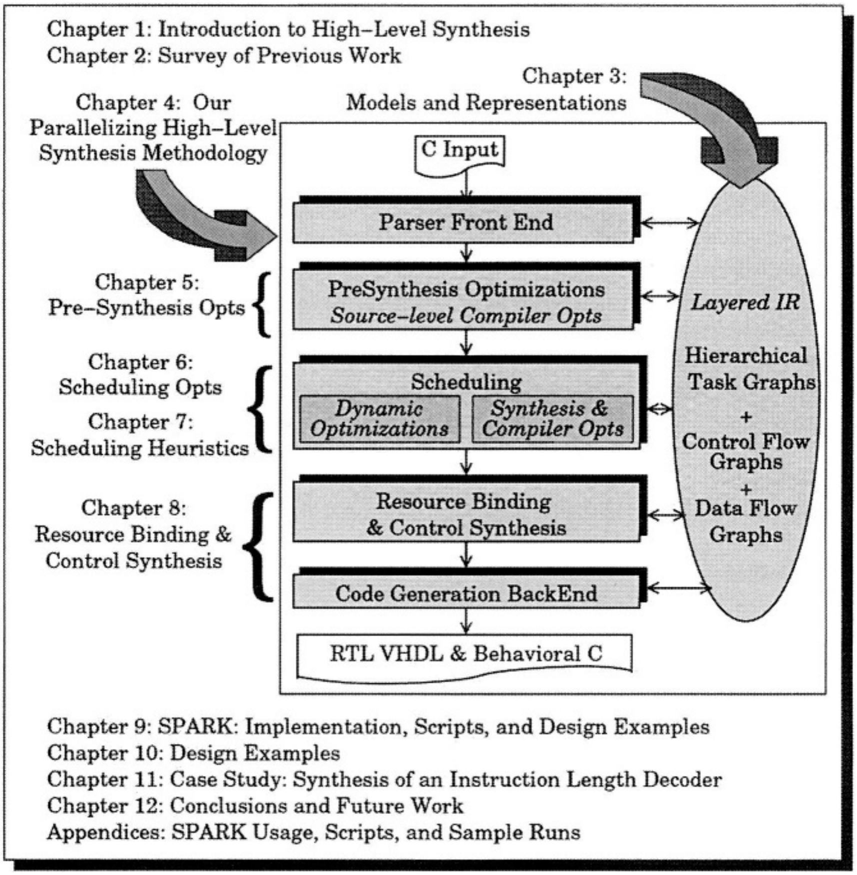

Chapter 1: Introduction to High-Level Synthesis
Chapter 2: Survey of Previous Work
Chapter 3: Models and Representations
Chapter 4: Our Parallelizing High-Level Synthesis Methodology
C Input
Parser Front End
Chapter 5: Pre-Synthesis Opts
PreSynthesis Optimizations
Source-level Compiler Opts
Chapter 6: Scheduling Opts
Chapter 7: Scheduling Heuristics
Scheduling
Dynamic Optimizations
Synthesis & Compiler Opts
Chapter 8: Resource Binding & Control Synthesis
Resource Binding & Control Synthesis
Code Generation BackEnd
RTL VHDL & Behavioral C
Layered IR
Hierarchical Task Graphs
+
Control Flow Graphs
+
Data Flow Graphs
Chapter 9: SPARK: Implementation, Scripts, and Design Examples
Chapter 10: Design Examples
Chapter 11: Case Study: Synthesis of an Instruction Length Decoder
Chapter 12: Conclusions and Future Work
Appendices: SPARK Usage, Scripts, and Sample Runs

Figure 1.4: *An overview of the organization of this book.*

rithm to move operations in the design graph. We also present algorithms for dynamic CSE, dynamic branch balancing and operation chaining.

Our approach to resource binding and control synthesis is described in Chapter 8. We first describe the interconnection minimizing resource binding technique that we have developed. We then describe our finite state machine based control synthesis methodology. We then describe how synthesizable VHDL code can be generated after synthesis.

In Chapter 9, we present the implementation and usage of the *Spark* high-level synthesis framework. We describe how synthesis scripts can be used to control the application of transformations and heuristics in *Spark* by comparing the quality of results when individual transformations and heuristics are employed.

In Chapter 10, we compare the quality of results for a set of design examples derived from the multimedia and image processing domains. We first list the characteristics of the designs used in the experiments and follow this up with detailed scheduling and logic synthesis results on each optimization described in Chapters 5 and 6.

In Chapter 11, we present a methodology to synthesize low latency microprocessor functional blocks using the *Spark* PHLS framework. We first present the transformations that are particularly useful to this methodology and then present a case study for the synthesis of the instruction length decoder from the Intel Pentium processor. We describe the transformations applied by *Spark* to synthesize this functional block.

Chapter 12 wraps-up the book with conclusions of our work and a discussion of future work.

2

SURVEY OF PREVIOUS WORK

In this chapter, we survey some of the related work done in the high-level synthesis and compiler communities and discuss how this relates to our work. This survey represents a small sampling of the work done in these communities over the past three decades. At the end of the chapter, we give an overview of the reasons that we believe have hindered the widespread acceptance of high-level synthesis in the design community.

2.1 Early Work in High-Level Synthesis

High-level synthesis techniques have been investigated for two decades now. Over the years, high-level synthesis techniques have been chronicled in several books [GDWL92, CW91, KM92, DM94, Kna96, JDKR97].

A lot of the early work focused on scheduling heuristics for data-flow designs. The simplest scheduling approach is to schedule all the operations *as soon as possible* (ASAP) [KT85, TS86, Mar86, Tri87]. The converse approach is to schedule the operations *as late as possible* (ALAP). Scheduling heuristics such as urgency scheduling [Gir84] and force-directed scheduling (FDS) [PK89a] schedule operations based on their urgency and mobility respectively. *Urgency* of an operation is the minimum number of control steps from the last step in the schedule that the operation can be scheduled in before a timing constraint is violated. *Mobility* of an operation is the difference between the ASAP and ALAP start times of the operation.

Another category of heuristics uses a list scheduling approach whereby operations are scheduled after they are ordered based on control and data dependencies [HT83, MRSC86, PG86]. There are several other types of scheduling approaches that either iteratively reschedule the design [PK91] or first schedule operations along the critical path through the behavioral description [PPM86]. In the MAHA system [PPM86], operations on the critical path are scheduled first and then the critical path is divided into several pipeline stages. Finally, the operations on the off-critical paths are scheduled based on a notion of *freedom*. This notion of freedom of an operation is similar to the mobility of an operation as defined above.

Resource allocation and binding techniques ranging from clique partitioning [TS86] to knowledge-based systems [KT85] have been explored in the past. The optimization goals for resource binding vary from reducing registers and functional units to reducing wire delays and interconnect costs [GDWL92, CW91, DM94]. Tseng et al. [TS86] use clique partitioning heuristics to find a clique cover for a module allocation graph. Paulin et al. [PK89b] perform exhaustive weight-directed clique partitioning of a register compatibility graph to find the solution with the lowest combined register and interconnect costs. Stok et al. [SP91] use a network flow formulation for minimum module allocation while minimizing interconnect. Mujumdar et al. [MJS96] consider operations and registers in each time-step one at a time and use a network flow formulation to bind them.

Since the sub-tasks of high-level synthesis are highly interrelated, there have been several works that attempted to perform these tasks simultaneously. Approaches using ILP formulations frequently fall into this category [HLH91, GE92, LMD94, WMGB95]. Gebotys et al. [GE92] present an integer-programming model for simultaneous scheduling and allocation that minimizes interconnect. A 0/1 integer programming model is proposed for simultaneous scheduling, allocation and binding in the OSCAR system [LMD94]. Wilson et al. [WMGB95] presented a generalized ILP approach that gives an integrated solution to the various high-level synthesis sub-tasks. The authors in [SZ90] and [KAH97] use simulated annealing to search the design space while performing simultaneous scheduling, allocation and binding.

Pipelining has been the primary technique to improve performance for data-flow design [PK89a, PG86, PP88, Gir87, CLS93]. The *Sehwa* system [PP88] automatically pipelines designs into a fixed number of stages to achieve the maximum performance. In the HAL system [PK89a], designs are pipelined with the help of user-specified pipeline boundaries, so that the resources are distributed uniformly across concurrent control steps. In the Maha system [PPM86], the critical path is repeatedly divided into pipeline stages till the user-specified timing bound is satisfied.

Several optimization techniques such as algebraic transformations, re-timing and code motions across multiplexers for improved synthesis results have been proposed [PR94, WT89, IPDP93]. Walker and Thomas [WT89] proposed a set of transformations that were implemented within the System Architect's Workbench [TDW$^+$88]. In this system, the input description is captured in a variant of data flow graphs known as a value trace [McF78]. The system first applied coarse-level transformations such as function inlining, inverse function inlining and removal of uncalled functions (or dead code elimination of whole functions). Then the system applied transformations that moved operations into and out of select operations (or multiplexers). These code motion transformations are similar to the speculative code motions used in designs with control flow.

Another commonly used flow graph transformation is *tree height reduction* (THR) [HC89]. In this technique, the height of an expression tree or number of operations that execute in a chain is reduced by employing the commutative and distributive properties of the operators. Nicolau and Potasman [NP91] proposed an incremental THR technique that opportunistically applies tree height reduction during scheduling if leads to better scheduling results.

2.2 High-Level Synthesis for Behaviors with Complex Control Flow

The high-level synthesis of designs with control flow (conditional branches and loops) has been studied by several groups over the last decade. Path-based Scheduling [Cam91] is an exact approach that creates a As-Fast-As-Possible (AFAP) schedule by scheduling each control path independently. However, the order of operations on each control path is fixed and scheduling is formulated as a clique covering problem on an interval graph; this means that speculative code motions are not supported. An extension of this technique was presented in [BRT97] that allowed operation reordering within basic blocks. On the other hand, *tree-based scheduling* [Hea93a] supports the speculation code motion and removes all the join nodes from the design, thus leaving the control-data flow graph looking like a tree. Both these approaches deal with loops by removing the loop's feedback path and scheduling just one iteration of the loop. The PUBSS approach [WTH+92] captures scheduling information in the behavioral finite state machine (BFSM) model. The input description (in behavioral VHDL) is captured as a network of communicating BFSMs and the schedule is generated using constraint-solving algorithms.

In the CVLS approach, [WY89, WT92, Wak99] a condition vector is generated for each operation based on the conditions under which the operation executes. Operations whose condition vectors are complementary are considered mutually-exclusive and are free to be scheduled on the same resource. Speculation and operation duplication across a merge node are supported.

Radivojevic et al. [RB96] present an exact symbolic formulation that schedules each control path (or trace) independently and then creates an ensemble schedule of valid, scheduled traces. The traces are validated by using a backtracking procedure that can be computationally expensive. Haynal [Hay00] uses an automata-based approach for symbolic scheduling of cyclic behaviors under sequential timing and protocol constraints. This is an exact approach, but can grow exponentially in terms of internal representation size. The "Waveschedule" approach [LRJ98] incorporates speculative execution into high level synthesis to achieve its objective of minimizing the expected number of cycles. Kim et al. [KYLL94] transform a control-data flow graph into a data flow graph by exploring various conditional resource sharing possibilities. The data flow graph is scheduled and then transformed back into a control-data flow graph.

Santos et al. [dSJ99, dS97, dS98] and Rim et al. [RFJ95] support generalized code motions during scheduling in synthesis systems, whereby operations can be moved globally irrespective of their position in the input. Santos creates and evaluates the cost of a set of solutions for the given design and then picks the solution with the lowest cost. This approach supports all types of code motions and employs code motion pruning techniques to reduce the search space and hence, the run time of the approach.

Paulin et al. [PFH+95] present a hardware-software co-design methodology that uses the Amical tool [KDJ94] for synthesizing the control-intensive parts of the design and the Cathedral-2/3 tool [MRSC86] for DSP-oriented synthesis. The Amical tool enables the use of large function components in the hardware resource library.

The Cathedral-2/3 tool is specifically targeted toward the synthesis of high-throughput application specific units.

The Olympus system [MKMT90] is a complete set of high-level synthesis, logic-level synthesis and simulation tools. The high-level synthesis tool, *Hercules,* accepts an input description in a variant of C called *HardwareC* [KM88], that introduces the notion of concurrency and timing into C. Relative scheduling [KM90] is used to schedule a design with operations that have unbounded delay.

Prior work on pre-synthesis transformations has focused on altering the control flow or extracting the maximal set of mutually exclusive operations [LG96, PMH02]. Li and Gupta [LG97] restructure the control flow and attempt to extract common sets of operations within conditionals to improve synthesis results. Kountouris and Wolinski [KW99] perform operation movement and duplication *before* scheduling and also attempt to detect and eliminate false control paths in the design. Function inlining, where a function call is replaced by the contents of the function, was presented in the context of high-level synthesis by Walker and Thomas [WT89].

2.3 Intermediate Representations in High-Level Synthesis

Traditionally, control-data flow graphs (CDFGs) have been the most popular intermediate representation for high-level synthesis [OG86]. However, in our work, in addition to maintaining control flow graphs and data flow graphs, we also adopted a hierarchical intermediate representation (IR) that maintains the hierarchical structuring of the design such as if-then-else blocks and for and while loops. This representation consists of basic blocks encapsulated in *Hierarchical Task Graphs* (HTGs) [GP92]. HTGs enable the efficient application of coarse-grain code restructuring transformations, besides supporting operation level transformations [NN93].

Of course, several other representation models have been proposed earlier for high-level synthesis [GDWL92]: such as Value Trace (VT) [McF78], Yorktown Intermediate format (YIF) [BCM$^+$88], Assignment Decision Diagrams (ADDs) [CGR92], Behavioral Finite State Machines (BFSMs) [WTH$^+$92], Sequencing Graphs [KM90], Hierarchical Conditional Dependency Graphs (HCDGs) [KW00], et cetera . Also, Rim et al. [RFJ95] and Bergamaschi [Ber99] have proposed new design representation models that attempt to bridge the gap between high-level and logic-level synthesis and aid in estimating the effects of one on the other. Chaiyakul et al. [CGR92, CGR93b] proposed using assignment decision diagrams (ADDs) as the intermediate representation to minimize syntactic variance in the input description caused by complex control flow.

However, we found HTGs to be the most natural choice for our parallelizing transformations. This is because, HTGs retain important code structure information such as control flow, hierarchy of control and loop constructs, loop characteristics, et cetera. This information makes it possible to make design optimizations at multiple levels from pre-synthesis transformations (such as loop unrolling and loop fusion) to very low level compiler transformations (such as dead code elimination). Our choice of HTGs also helped in implementation and analysis of parallelizing transformations such as *Trail-*

blazing [NN93] and *Resource-Directed Loop Pipelining* [NN96] that were originally developed using HTGs as the underlying intermediate representation.

2.4 Related Work in Compiler and Parallelizing Compiler Approaches

Compiler transformations such as CSE and copy propagation predate high-level synthesis and are standard in most software compilers [ASU86, Muc97]. These transformations are applied as passes on the input program code and as cleanup at the end of scheduling before code generation. Compiler transformations were developed for improving code efficiency. Their use in digital circuit synthesis has been limited. For instance, CSE has been used for throughput improvement [IPDP93], for optimizing multiple constant multiplications [PSC96, PSD+99] and as an algebraic transformation for operation cost minimization [JCM94, MCJM98].

A converse of CSE, namely, *common sub-expression replication* has been proposed to aid scheduling by adding redundant operations [LP91, PR92]. *Partial redundancy elimination* (PRE) [KCL+99] inserts copies of operations present in only one conditional branch into the other conditional branch, so as to eliminate common subexpressions in subsequent operations. The authors in [JCM94, GMCG00] propose doing CSE at the source-level to reduce the effects of the factorization of expressions and control flow on the results of CSE.

In the context of parallelizing compilers, *Mutation Scheduling* [NN94] also performs local optimizations such as CSE during scheduling in an opportunistic, context-sensitive manner. A range of parallelizing code transformation techniques has also been previously developed for high-level language software compilers (especially parallelizing compilers) [Kuc78, KKP+81, Pol88, GPN90, KA01, Wol96, Fis81, EN89, AN88a, NN93].

Although the basic compiler transformations (e.g. dead code elimination, copy propagation) can be used in synthesis as well, other transformations need to be re-instrumented for synthesis by incorporating ideas of mutual exclusivity of operations, resource sharing and hardware cost models. Cost models of operations and resources in compilers and synthesis tools are particularly very different. For example, in compilers there is generally a uniform push towards executing operations as soon as possible by speculative code motions. Indeed, the optimality claims in percolation and trace scheduling are based entirely upon maximum movement of operations out of conditional branches. In the context of high-level synthesis, such notions of optimality have little relevance. In circuit synthesis, code transformations that lead to increased resource utilization, also lead to higher hardware costs in terms of steering logic and associated control circuits. Some of these costs can be mitigated by interconnect minimizing resource binding techniques [GSD+01].

The use of code motions as a transformation technique has been explored at length in the context of parallelizing compilers [Fis81, Nic85a, Hea93b, GS90, NN93, Wal91, LW92]. Trace scheduling [Fis81] uses profiling information to aggregate sequences of basic blocks (or control paths) that execute frequently into *traces.* These traces in the design are scheduled one at a time starting with the most time critical one. Hence, code motions are allowed only within traces and not across traces; in effect, code motions that duplicate or eliminate operations across multiple traces are not supported. Compensation code must be added in a bookkeeping pass applied after scheduling.

Percolation scheduling [AN88a, Nic85a, Nic85b] applies a set of atomic semantic-preserving transformations to move or "percolate" operations in the design and create a resource constrained schedule. Given an infinite number of resources, percolation scheduling has been shown to produce optimal schedules. Percolation scheduling suffers from two main limitations: (a) operations are moved *linearly* by visiting each node on the control path being traversed, and (b) there is a lot of unnecessary duplication of operations into multiple control paths.

Trailblazing [NN93] overcomes the limitations of percolation scheduling by moving operations across large pieces of code with the help of a hierarchically structured intermediate representation. The intermediate representation, called *Hierarchical Task Graphs* (HTGs) [GP92], captures the code structure along with the control flow between basic blocks.

2.5 Use of Loop Transformations in Compilers and High-Level Synthesis

Loop transformations that unroll and pipeline loops have long been recognized in the software community as key to exploiting much larger amounts of parallelism in program codes than is possible by looking at just a single iteration of the loop body ([RG81, AN88b, Lam88, NN96, ANN95, JA90] to name a few). Modulo scheduling and its variants [RG81, Lam88] create a schedule for one iteration of the loop body that can be repeated at a regular initiation interval without violating any constraints. The resource-constrained software pipelining [ANN95] and perfect pipelining [AN88b] techniques iteratively unroll and schedule the loop body until a repeating pattern of operations emerges. Loop shifting was originally proposed as a loop pipelining technique by Ebcioglu [Ebc87] and used later as a part of the *resource-directed loop pipelining* (RDLP) technique [ANN95, NN96]. RDLP first unrolls the loop several times and then attempts loop shifting and compaction.

Early work on loop pipelining in high-level synthesis focused on the innermost loops of DSP applications with no conditional constructs. These works include loop winding [Gir87], rotation scheduling [CLS93], percolation based synthesis (uses perfect pipelining) [PLNG90], and loop folding (in Cathedral-II) [GVM89].

Holtmann and Ernst [HE95] apply loop pipelining to designs with conditional branches by scheduling operations on the most probable path through the loop body and deferring operations on other paths. Yu et al. [YSPJ97] extend rotation scheduling for control-data flow graphs (CDFGs) by creating a *branch anticipation* controller to store and propagate branch control signals across loop iterations. Lakshminarayana et al. [LRJ98] speculatively execute operations from future loop iterations. Radivojevic and Brewer [RB95] extend the loop folding technique for CDFGs.

We found that for designs with a large amount of conditionals embedded inside loops, loop unrolling can lead to worse circuit delays and area due to a large control and multiplexer overhead. Instead, we proposed using an incremental loop transformation technique called loop shifting that incrementally exploits parallelism when the number of resources is not enough to justify loop unrolling or initiating another iteration of the loop body (see Chapter 6).

2.6 What is Hindering Adoption of High-Level Synthesis Tools

Despite this rich body of research work and several commercial efforts, high-level synthesis still enjoys limited adoption among designers. The two chief reasons for this are:

- The quality of synthesis results is poor in behavioral descriptions with complex and nested conditionals and loops.
- Designers are often given minimal controllability over the transformations that affect these results.

Recent work in academia has looked at speculative code motions for improving synthesis results in designs with complex control flow. However, most of these works present code transformations in isolation and on small, synthetic designs. There is no clear analysis and understanding of how the range of coarse-grain and fine-grain code transformations interact. In particular, there is no clear understanding of the affect of language-level coarse-grain transformations on the quality of synthesis results.

Furthermore, previous works compare the effectiveness of their transformations and scheduling algorithms primarily in terms of schedule lengths. They give little insight into the control and area costs of the transformations. Thus, it is not clear if a transformation has a positive impact beyond scheduling. Industry experience shows that, often critical paths in control-intensive designs pass through the control unit and steering logic. To this end, Rim et al. [RFJ95] use an analytical model to estimate the cost of additional interconnect and control caused by code duplication during code motions. Bergamaschi [Ber99] proposes the behavioral network graph to bridge the gap between high-level and logic-level synthesis and aid in estimating the effects of one on the other.

Several approaches use intermediate representations such as control-data flow graphs for design entry. The lack of robust *language* front-ends restricts these tools to the synthesis of small, synthetic designs. For the few approaches that do use high-level languages as input, synthesizability is guaranteed on a small, constrained sub-set of the input language. As a result, language level optimizations are few and their effects on final circuit area and speed are not well understood.

All these factors continues to limit the acceptance of high-level synthesis tools among designers. High-level synthesis tools do not produce competitive results as compared to manual implementations. We address these factors and demonstrate through our design examples that a parallelizing high-level synthesis approach can produce designs that are competitive with manual design. This can make high-level synthesis part of the microelectronic design flow, thus, bringing about the much needed productivity gain required to meet competitive market forces.

2.7 Summary

In this chapter, we presented a survey of related previous work. We first examined early work in high-level synthesis (HLS) followed by work on synthesizing designs with control flow in Section 2.2. In Section 2.3, we briefly discussed intermediate representations that have been used in the past for HLS. We then examined related work in the compiler community with particular focus on parallelizing compiler techniques. In Section 2.5, discussed loop transformations that have been proposed for compilers and HLS. Finally, in Section 2.6, we presented the chief reasons that have limited the adoption of high-level synthesis tools among designers. The contributions of this chapter are a detailed survey of techniques that have been proposed both in the high-level synthesis and the compiler communities. This survey forms the basis of exploring techniques that are relevant to the synthesis of designs with complex control flow.

3

MODELS AND REPRESENTATIONS

3.1 Modeling the Problem

This chapter describes the models we use to represent the various pieces of information required and generated during the application of the code transformations and in particular during the scheduling task of high-level synthesis. We first present the intermediate representation used to represent designs in our work and how we model the hardware resources and timing information. We then present the traditional formulation of the minimum-latency resource-constrained scheduling problem. Next, we introduce the notion of control flow and re-formulate the scheduling problem in the presence of control flow. We also model speculative and hierarchical code motions and show how these integrate into the scheduling problem.

Scheduling uses and generates several pieces of information. The most important among these are:

- ❒ *Intermediate Representation* (IR): required to capture the design description.
- ❒ *Hardware Description*: The specification and representation of functional units and resources such as registers, buses et cetera allocated to schedule the design,
- ❒ *Timing Information*: Clock period, execution times of the various functional units, and information generated by the scheduling task such as start times of the operations.
- ❒ *Resource Mapping*: The mapping of each operation in the design to a resource that it will execute on. Resource binding determines the mapping of the operation to a particular instance of the resource type that the operation has been scheduled on.

3.2 Design Description Modeling

We have chosen the high-level programming language "C" as the input language for our synthesis methodology. There are several reasons for our choice; the most important of these is that system architects often verify the functionality of a proposed architecture by modeling it in the "C" programming language. Furthermore, nearly all the system level design approaches that have been proposed in recent years use a high level programming language that is a variant of "C" or "C++" [GL97, SyC, GZD+00, VSB99]. Designer's prefer high level languages such as "C" over behavioral hardware description languages (HDLs) such as behavioral VHDL since they give the designer more freedom for describing a behavior and also, enable specifying a behavior that is free of hardware and software implementation details. This is an important prerequisite of system level specifications since the tasks in the specification have not yet been partitioned into hardware and software components.

The input "C" description to our methodology is a sequential list of statements. Statements may be operation expressions, conditional constructs (if-then-else, switch-case) and loop constructs (for, while, do-while loops). Besides the operations supported by "C", we also support Boolean conditional checks. These Boolean checks are *produced* by comparison or relational tests such as <, ==, $\geq$, $\neq$, et cetera. We decompose complex expressions into three-address expressions (of type a=b+c) [ASU86]. Each three-address expression is then called an *operation* and represented by an abstract syntax tree.

We capture the control and data flow in the input description using a control flow graph (CFG) and a data flow graph (DFG) respectively. Additionally, we also capture the program structure in the input description using a hierarchical intermediate representation called a hierarchical task graph (HTG). The nodes of these three graphs form a 3-layered graph such that there is a relation between the nodes of each successive layer. Layered graphs are defined as follows:

Definition 3.1. *A* ***k-layered graph*** *is a connected graph in which the vertices are partitioned into k sets $L = l_1, ..., l_k$ and edges run between the vertices of successive layers, l_i and l_{i-1}.*

In our case, the top-level layer consists of the nodes from the HTG, next layer has nodes from the CFG and lowest level layer has nodes from the DFG. We first define these three types of graphs and then will return to the layered graph representation in Section 3.2.6.

3.2.1 Modeling Data Dependencies

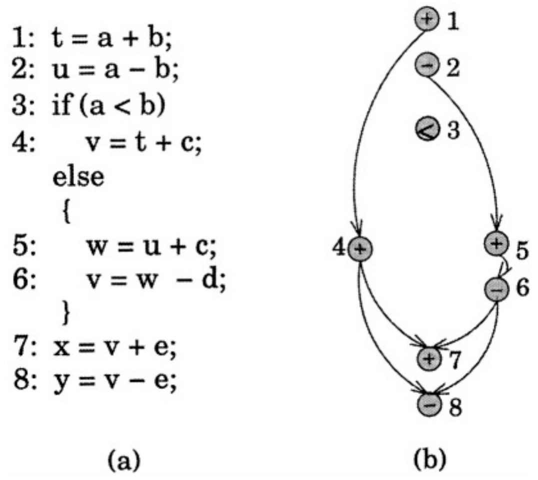
There are four types of data dependencies that can exist between operations op_i and op_j [KKP+81, Muc97]:

(i) *flow* or read-after-write dependency: op_j reads the result written by op_i.

(ii) *anti* or write-after-read dependency: op_j writes to a variable after it is read by op_i.

(iii) *output* or write-after-write dependency: op_j writes to the same variable that is written by op_i.

(iv) *input* or read-after-read dependency: op_j reads a variable after it is read by op_i.



```
1: t = a + b;
2: u = a - b;
3: if (a < b)
4:     v = t + c;
   else
   {
5:     w = u + c;
6:     v = w  - d;
   }
7: x = v + e;
8: y = v - e;
```

Figure 3.1: *(a) Example C description (b) Corresponding Data-Flow Graph (DFG).*

Of these, *input* dependencies do not affect scheduling of the design.

To formulate the scheduling problem, we use only flow data dependencies (in practice, we maintain all the data dependencies; see next section). A flow data dependency determines when an operation can start execution – if op_j has a flow dependency with op_i, then op_2 can start execution only after op_1 has finished execution. We can define a data flow graph that captures flow dependencies as follows:

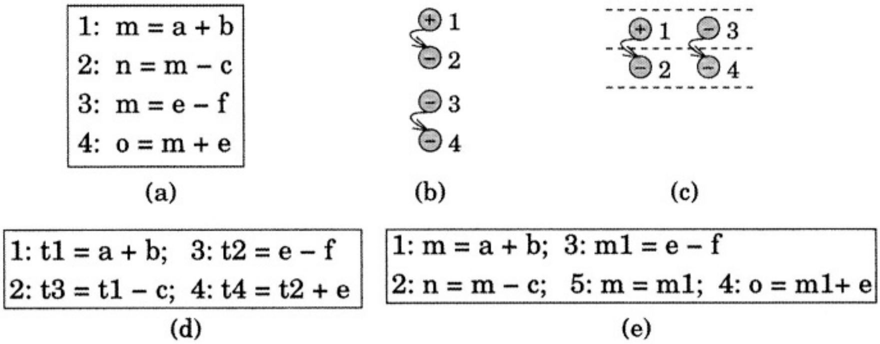
Definition 3.2. *The* **data flow graph** *is a directed acyclic graph $G_{DFG}(Ops, E_{data})$, where the vertex set $Ops = \{op_i;\ i = 1, ..., n_{ops}\}$ is the set of n_{ops}* ***operations*** *in the design and the edge set $E_{data} = \{(op_i, op_j);\ i, j = 1, ..., n_{ops}\}$ represents the flow data dependencies. A directed edge $e_{ij} = (op_i, op_j)$ exists in E_{data} if data produced by operation op_i is read by operation op_j.*

An example of a data flow graph is given in Figure 3.1(b) for the sample "C" description in Figure 3.1 (a). Operations in the data flow graph (DFG) are denoted by circular nodes with the operator sign within. The operation numbers in the DFG correspond to the line numbers in the "C" code in Figure 3.1 (a). The expression in each operation node is stored as an abstract syntax tree (AST) [ASU86].

3.2.2 Better Design Visualization by Maintaining Variable Names

Maintaining only flow data dependencies means that the information about the variable names from the original description is discarded. This impairs the ability to correlate the input description with the intermediate representation and the final output code generated after synthesis. This, in turn, hinders the *visualization* of the results of applying transformations to the design vis-a-vis the original input description.

To understand this, consider the "C" description of an example in Figure 3.2(a) and its corresponding data flow graph (DFG) in Figure 3.2(b). One possible schedule is



```
1: m = a + b
2: n = m - c
3: m = e - f
4: o = m + e
```

(a)

```
1: t1 = a + b;   3: t2 = e - f
2: t3 = t1 - c;  4: t4 = t2 + e
```

(d)

```
1: m = a + b;  3: m1 = e - f
2: n = m - c;  5: m = m1;  4: o = m1+ e
```

(e)

Figure 3.2: *(a) Example C description. (b) Corresponding DFG. (c) DFG after scheduling: operations* 1 *and* 3 *and operations* 2 *and* 4 *are scheduled concurrently. (d) Scheduled output code generated without maintaining non-flow data dependencies. (e) Output code generated if non-flow data dependencies are maintained. Operations in the same line denote concurrent execution.*

shown in Figure 3.2(c), where operations 1 and 3 and operations 2 and 4 are scheduled concurrently. The output code corresponding to this scheduled design, when only flow data dependencies are maintained, is given in Figure 3.2(d). In this output code, we have to create new variables that store the result of each operation in the scheduled design (Figure 3.2(c)). Clearly, it is difficult to correlate the operation statements in this output code with the operation statements in the input code.

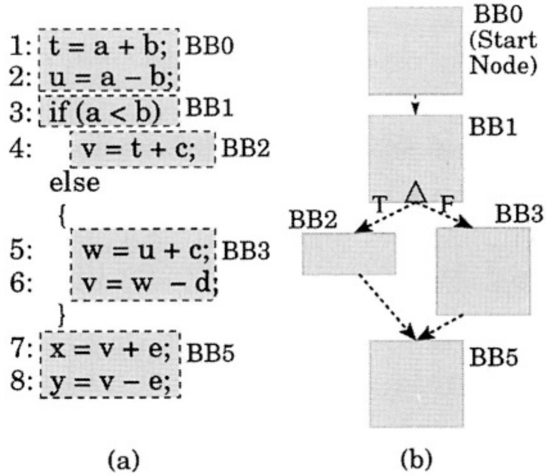
If we maintain the non-flow data dependencies, we can generate the output code given in Figure 3.2(e). In this code, the variables that each operation writes to are maintained as per the original code. By inspecting this code, we see that concurrent execution of operations 1 and 3 requires renaming the result variable of operation 3 to $m1$. A copy operation, operation 5, from the new variable $m1$ to the variable m from the original code is inserted in the code as shown in Figure 3.2(e). Thus, operation 4 can be executed concurrently with operations 2 and 5 using the new variable $m1$ after employing dynamic variable renaming (see Chapter 4).

We, therefore, employ a data dependency analysis pass to capture the full set of data dependencies given in the input description and employ techniques such as dynamic variable renaming to aid in reducing the restrictions imposed by these dependencies.

3.2.3 Modeling Control Flow

The presence of conditional and loop constructs lead to the notion of *control flow* through a design. During program execution, when a condition operation is encountered, the control flow branches into two control flows, based on whether the condition evaluates to true or false. At the end of the conditional or loop construct, the two control flows converge or merge back into a single thread of control flow.

Conditional constructs, hence, add two types of control nodes to the design graph; a *fork* node and a *join* node. Fork nodes signify the point at which a condition causes



```
t = a + b;
u = a - b;
if (a < b)
    v = t + c;
else
    {
    w = u + c;
    v = w  - d;
    }
x = v + e;
y = v - e;
```

Figure 3.3: ***(a) Example C description (b) Corresponding Control-Flow Graph (CFG).***

the control flow to branch into multiple control paths. Conversely, join nodes are the merging points of multiple control flow paths. The presence of control flow introduces the notion of basic blocks. A *basic block* is a sequence of statements from the input description with no conditionals or loops between them.

In a general program description in C, the presence of jumps can lead to an arbitrary control flow between basic blocks. However, in our synthesis methodology, we ensure synthesizability only for code that leads to reducible graphs [ASU86]. This means that jumps are allowed in the code as long as the control flow graph is a reducible graph. In our control flow model, each basic block can have two *input* control paths corresponding to a join node and two *output* control paths corresponding to a fork node. A segment of code with more than two paths originating from a fork node can be converted into this model by recursively partitioning the multiple fork nodes into two output fork nodes. For example, a switch-case statement can be converted into multiple if-then-else statements. Join nodes can similarly be converted into two input join nodes. A loop with multiple exit points can be converted into a single exit loop by inserting an empty basic block that acts as the exit point for all the loop exits [GP92].

The output control paths of a basic block have a condition of "true" or "false" associated with them; consequently, these control paths are known as the *true path* and the *false path.* Basic blocks that do not have a conditional check in them (no control flow fork), have only a default output true path. Basic blocks that are the last basic block in the design do not have any output control paths. Similarly, the first basic block in the design (defined below) corresponding to the top-level entry point into the design does not have input control paths.

We define a control flow graph that captures the control flow information between basic blocks as follows:

Definition 3.3. *A* ***control flow graph*** *is a directed graph $G_{CFG}(BB, E_{control})$, where the vertex set $BB = \{bb_i;\ i = 1, 2, ..., n_{bbs}\}$ is the set of n_{bbs} basic blocks in the design and the edge set $E_{control} = \{(bb_i, bb_j);\ i, j = 1, 2, ..., n_{bbs}\}$ corresponds to the control flow between the basic blocks. Also, there exists an unique initial basic block $bb_0 \in BB$ from which all paths in the flow graph originate; bb_0 can be obtained by $FirstBB(G_{CFG})$. Each edge has a labeling $L_{Cond} = \{T_{edge}, F_{edge}\}$ that signifies if the edge is a true path or a false path. A directed edge $e_{ij} = (bb_i, bb_j)$ exists in $E_{control}$ if basic block bb_i is a* **predecessor** *of basic block bb_j. Basic block bb_j is denoted as a successor of bb_i.*

We also define some additional functions that are called by the algorithms presented later in Chapter 7 as follows:

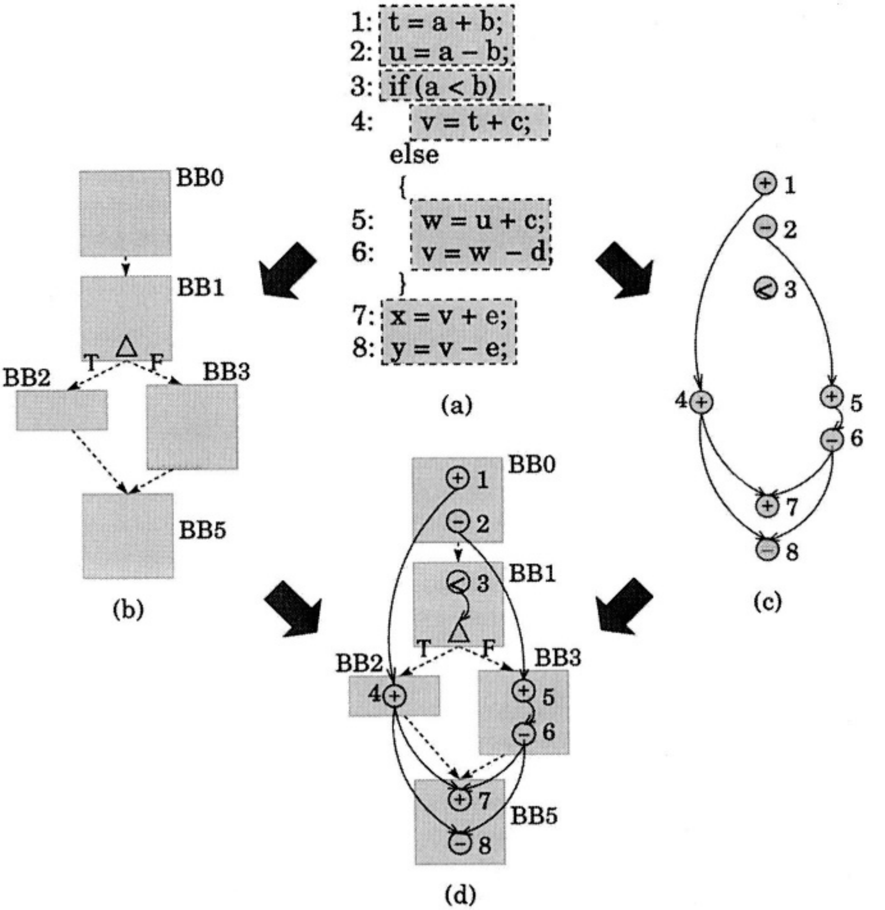
Definition 3.4. *The set of successors of a basic block bb_i can be obtained by the function $SUCCS(bb_i)$ and the predecessors by $PREDS(bb_i)$. Of the successor basic blocks of bb_i, the basic block on the "true" path is obtained by $NextTrue(bb_i)$, i.e., $L_{Cond}(e_{ij}) = T_{edge}$ for $bb_j = NextTrue(bb_i)$, such that $e_{ij} \in E_{control}$. Similarly, the basic block on the "false" path is obtained by the function $NextFalse(bb_i)$. Also, the function $IsJoin(bb_i)$ returns true if multiple (two) control flow edges merge at basic block bb_i. Similarly, the function $IsFork(bb_i)$ returns true if multiple (two) control flow edges branch at basic block bb_i.*

The earlier example "C" description from Figure 3.1(a) is reproduced in Figure 3.3(a). Each sequence of operations in the source code with no control flow between them is aggregated into a basic block (shown by shaded boxes in Figure 3.3(a)). The control flow between these basic blocks is shown in the corresponding control flow graph in Figure 3.3(b). The basic blocks are labeled from bb_0 to bb_5. A triangle denotes a Boolean conditional check or a fork in control flow with a true path and a false path. All other control edges are true by default and their labeling L_{Cond} is omitted.

3.2.4 Mapping between Data Flow and Control Flow Graphs

Definition 3.5. *Given a data flow graph $G_{DFG}(Ops, E_{data})$ and a control flow graph $G_{CFG}(BB, E_{control})$ as per Definitions 3.2 and 3.3, there exists a many-to-one mapping of the operations to the basic blocks BB_{Ops}: $Vop \mapsto BB$. The basic block that an operation op_i belongs to can be obtained by $BB_{Ops}(op_i) = bb_j\ \forall\ op_i \in Ops$, such that $bb_j \in BB$.*

The combined control and data flow graphs for the running example is shown in Figure 3.4(d) along with the original description and the control and data flow graphs. Although these control and data flow graphs capture all the operation level information in a design description, information about the code structure from the input description is lost. In the next section, we describe a hierarchical intermediate representation that extends control flow graphs to retain this structural information.



```
1: t = a + b;
2: u = a − b;
3: if (a < b)
4:    v = t + c;
   else
   {
5:    w = u + c;
6:    v = w − d;
   }
7: x = v + e;
8: y = v − e;
```

Figure 3.4: *(a) Example C description, Corresponding (b) Control-Flow Graph (CFG), (c) Data-Flow Graph (DFG), and (d) Mapping between the Control and Data Flow Graphs.*

3.2.5 HTGs: A Model for Designs with Complex Control Flow

Traditionally, control-data flow graphs (CDFGs) have been the most popular intermediate representation for high-level synthesis. CDFGs consist of operation and control nodes with edges for both data flow and control flow. CDFGs work very well for the traditional high-level synthesis tasks of scheduling and binding. However, we found the abstraction level offered by CDFGs is too thin for the range of coarse-grain and fine-grain parallelizing compiler transformations that we proposed.

In order to enable the range of optimizations explored by our work, we use an intermediate representation that maintains the hierarchical structuring of the design such as

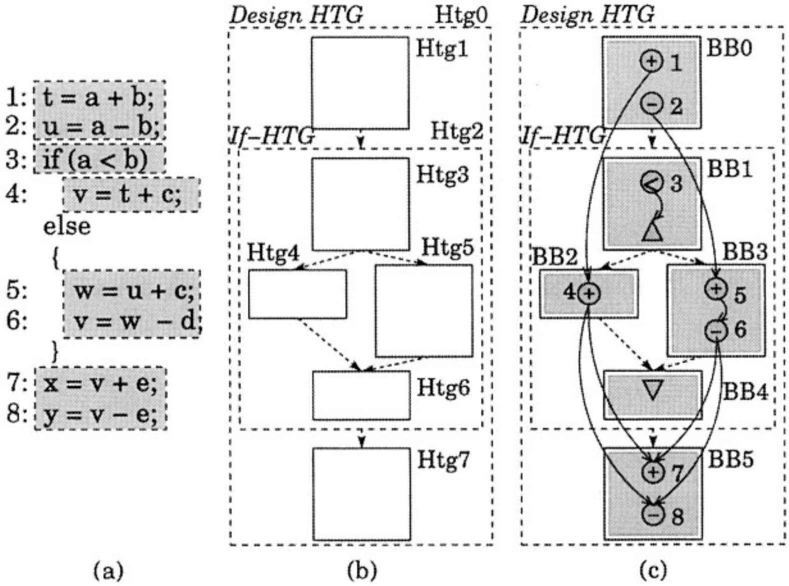


Figure 3.5: ***(a) Example C description (b) Corresponding Hierarchical Task Graph representation (HTG) (c) HTG representation with the control and data flow graphs overlaid on top. Each basic block is encapsulated in a Single HTG Node. Basic block bb_4 is added to the control flow graph as a join node for the If HTG node.***

if-then-else blocks and for- and while-loops. This intermediate representation consists of basic blocks encapsulated in *Hierarchical Task Graphs* (HTGs) [GP92, NN93].

Definition 3.6. *A* ***hierarchical task graph*** *(HTG) is a directed acyclic graph HTG = (HV, HE) with unique* **Start** ***and*** **Stop** *nodes belonging to HV such that there exists a pathfrom the* **Start** *node to every node in HV and a pathfrom every node in HV to the* **Stop** *node. Edges, HE, in a HTG represent control flow between HTG nodes. Each node $HV \in HTG$ can be one of three types $T_{HTG} = \{ \mathcal{SN}, \mathcal{CN}, \mathcal{LN} \}$, corresponding to single nodes, compound nodes and loop nodes:*

1) **Single nodes** *represent nodes that have no sub-nodes and are used to encapsulate basic blocks.*

2) **Compound nodes** *are hierarchical in nature, i.e., they contain other HTG nodes. They are used to represent structures like if-then-else blocks, switch-case blocks or a series ofHTGs.*

3) **Loop nodes** *are used to represent the various types of loops (for, while-do, do-while). Loop nodes consist ofa loop head and a loop tail that are single nodes and a loop body that is a compound node.*

The **Start** *and* **Stop** *nodes ofa HTG can be obtained by Start(HTG) and Stop(HTG). Also, the* **Start** *and* **Stop** *nodes for all compound and loop HTG nodes are always sin-*

gle nodes. Note that the **Start** *and* **Stop** *nodes of a single node are the node itself. The type of a node $hv_i \in HV$ can be obtained by $T_{HTG}(hv_i)$.*

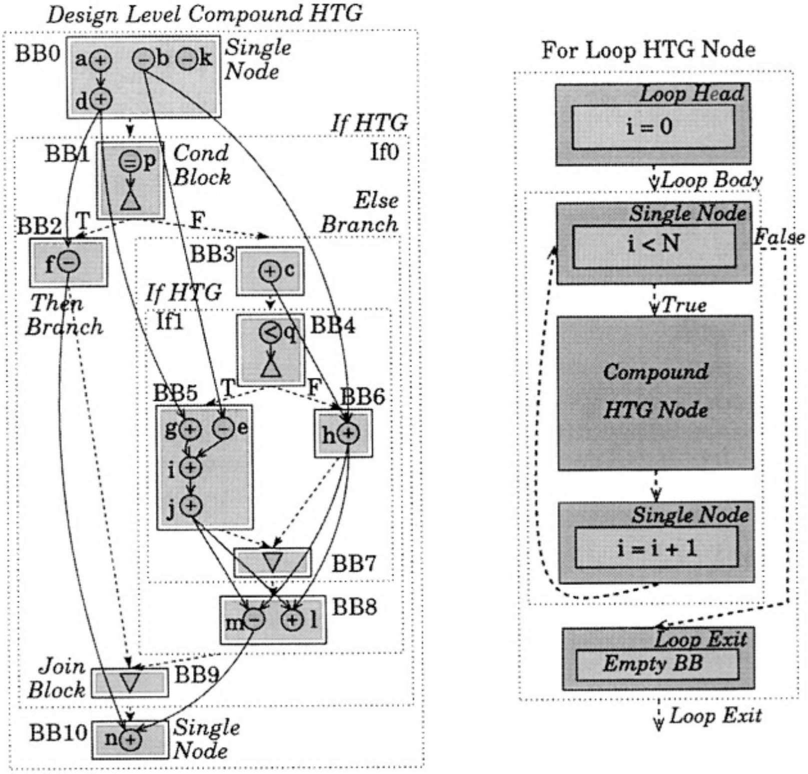
Since HTGs maintain a hierarchy of nodes, they are able to retain coarse, high level information about program structure in addition to operation level and basic block level information. This aids in coarse-grain code restructuring (such as that done by loop transformations) and also, in operation movement by reducing the amount of compensation code required. Furthermore, non-incremental moves of operations across large blocks of code are possible without visiting each intermediate node [NN93].

The HTG representation of the earlier example (reproduced in Figure 3.5(a)) is given in Figure 3.5(b). In this figure, we show the HTG nodes, $Htg0$ to $Htg6$, with the control flow between them. The HTG node $Htg1$ is a If-HTG (compound) node, whereas all other nodes are single nodes. In Figure 3.5(c), we show how the control and data flow graphs can be overlaid onto the HTG graph. Each basic block is encapsulated in a single node. Also, during the construction of HTGs, **we add empty basic blocks as "Join" basic blocks** at points where multiple control flow path merge into a basic block.

In Figure 3.5(c), basic block bb_4 corresponds to a join basic block. The control flow merge is denoted as an *inverted triangle.* Join basic blocks serve as the *Stop* nodes of compound HTG nodes and enable an easier and more structured approach to the hierarchical composition of nodes. Empty basic blocks are also added during HTG construction to ensure that every HTG node has a unique *Start* node and a unique *Stop* node (both of which are single nodes with a basic block in them). Hence, unstructured designs with jumps within loops are converted into structured designs using these extra basic blocks. These issues, along with detailed notes on HTG construction are covered in detail in [GP92].

In the HTG representation in Figure 3.5(b), each basic block from the original control flow graph is encapsulated by a single node. Control flow edges between the HTG nodes are shown by dashed arrows. The if-then-else control construct from the source code is encapsulated in a if-HTG node. As shown in the figure, an if-then-else HTG node consists of a single node for the conditional check, a compound node for the true/then branch, a compound node for the false/else branch and a single node with an empty basic block for the join node. Note that, the basic blocks that comprise the true and false branches in this figure are shown to be encapsulated only in a single node for clarity. In practice, the single node is then encapsulated in the compound HTG node that forms the true or false branch. The *Start* node for an If-HTG is the single node with the conditional check and the *Stop* node is the join node. We also show the data flow graph super-imposed on the HTG representation in Figure 3.5(c); in practice, data flow graphs are maintained separate to HTGs.

Figure 3.6(a) illustrates the HTG for the synthetic benchmark "waka" [WT92] along with the data flow dependencies. In this figure, the dashed arrows indicate control flow between HTG nodes and the solid lines indicate data flow between operations. This design contains an If-HTG node, whose false/else branch contains another If-HTG node. bb_0 to bb_{10} denote basic blocks. Again, each basic block is encapsulated in a single HTG node.



(a) (b)

Figure 3.6: ***(a) The hierarchical task graph (HTG) representation of the "waka" benchmark. Flow data dependencies are also shown. (b) The HTG representation of a For-Loop.***

A For-loop HTG, as shown in Figure 3.6(b), consists of 3 sub-nodes: (a) Loop head and *Start* node: consists of a single node with an optional initialization basic block; (b) Loop iteration body: a compound HTG node contains a single node for the conditional check basic block and a compound HTG node for the main body of the loop and an optional single node for the loop index increment basic block; and (c) Loop tail/exit and *Stop* node: a single node with an empty basic block. There is a backward control flow edge from the end of the loop body to the conditional check single node. Maintaining the loop hierarchy allows us to treat the back edges as implicit self-loops on composite nodes [GP92]. Therefore, at any hierarchy level, the HTG is a directed acyclic graph.

HTGs are constructed from the input description by first creating a compound HTG node for the design level HTG. Each sequential piece of code in the input description forms a sub-node of this HTG. The *Start* and *Stop* nodes of the design level HTG correspond to the *Start* node of the first sub-node HTG and *Stop* node of the last sub-node in the design respectively. Hence, for the *waka* design shown in Figure 3.6(a), the design level HTG node has three sub-nodes. The first sub-node is a single node with basic block bb_0, the second sub-node is the If-HTG node, IF_0 and the third sub-node is the single node with basic block bb_{10}. The *Start* and *Stop* nodes for this design are the single nodes that encapsulate basic blocks bb_0 and bb_{10}.

The hierarchical structuring in HTGs is useful in implementing global code motion techniques such as *Trailblazing* [NN93]. For example, when the *Stop* node of a HTG node is encountered while moving an operation, Trailblazing can move the operation directly to the *Start* node of the HTG node without visiting each node in the HTG – provided the operation does not have any data dependencies with the operations in the HTG node [NN93, GDGN03d].

For clarity, we make several simplifications in the figures used for the examples in the rest of this book. We omit the single HTG node that encapsulates basic blocks. Control flow edges in HTG representations are shown to originate from basic blocks and terminate at basic blocks. Control edges emanating from a fork node are shown to originate from the triangle in the figures that denotes the control flow fork. These simplifications aid the understanding of the flow of control between the basic blocks in the designs.

Using the definitions of HTGs and the control flow graph, we can now define a design HTG as:

Definition 3.7. *A* ***design HTG*** *is a directed acyclic graph $G_{HTG} = (V_{HTG}, E_{HTG})$ with unique* **Start** *and* **Stop** *nodes belonging to V_{HTG}. Each hierarchical or compound node in G_{HTG} corresponds to a control construct in the design. There is a one-to-one mapping of the single nodes in the design HTG G_{HTG} and the basic blocks in the control flow graph $G_{CFG}(BB, E_{control})$ defined in Definition 3.3 given by: BB_{SN} : $V_{HTG} \mapsto BB \; \forall \; htg_i \in V_{HTG}$ such that $T_{HTG}(htg_i) = \mathcal{SN}$. New join basic blocks may be inserted into the control flow graph to aid in the construction of HTGs.*

Since, the **Start** and **Stop** nodes of compound and loop HTG nodes are always single nodes, the first and last basic blocks of a HTG node $htg_i \in V_{HTG}$ can be obtained by $BB_{SN}(Start(htg_i))$ and $BB_{SN}(Stop(htg_i))$ respectively. For short, in the rest of this book, we will use $Start(htg_i)$ and $Stop(htg_i)$ to denote the first and last basic block in a HTG node htg_i.

3.2.6 Capturing the Complete Design Description

We capture the input description using a design graph that is defined as follows:

Definition 3.8. *A* ***design graph,*** *$\mathcal{DG}$, is a layered graph with three layers of nodes corresponding to the nodes of the graphs G_{HTG}, G_{CFG}, and G_{DFG}. The edges of $\mathcal{DG}$ are the mapping BB_{Ops} between the operations in G_{DFG} and the basic blocks in G_{CFG} and the mapping BB_{SN} between the single nodes in G_{HTG} and the basic blocks in G_{CFG}.*

The design graph for the running example is shown in Figure 3.7. Note that, HTGs capture information about the control flow between hierarchical nodes and CFGs capture information about the control flow between basic blocks. Hence, we maintain both graphs since HTGs are an efficient way to traverse the hierarchy of the design, whereas CFGs are efficient for traversing the basic blocks in the design.

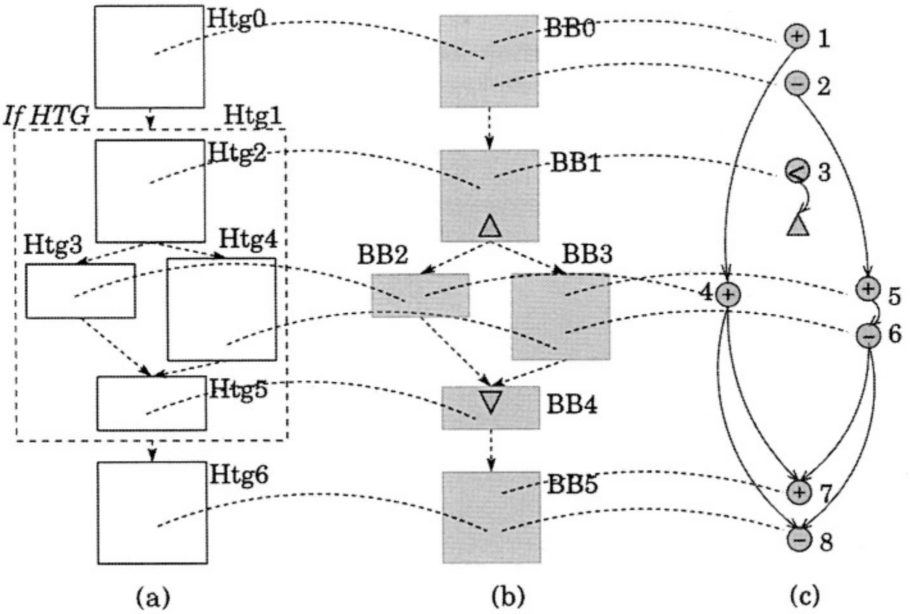


Figure 3.7: *(a) HTG representation of the example C description, (b) Control flow graph for the example, and (c) Data flow graph for the example. Together these form a 3-layered* Design Graph, $\mathcal{DG}$. *The dashed lines denote the edges between the different layers of the design graph.*

3.3 Modeling Hardware Resources, Timing and Data Types

In addition to the design description, high-level synthesis tools require information about the hardware resources allocated to schedule the design and the timing of these resources. We also require information about the data types in order to generate output code in a hardware description language (HDL), as explained in the next section.

3.3.1 Modeling the Data Type Information

The programming language “C” supports several data types including integer, float, character and variants of these such as short, long, double et cetera. Furthermore, each of these data types can be signed or unsigned. We enable a designer to specify the range of the various data types as a table in a hardware description file. This table has three columns: the data type, lower data range and upper data range.

An example of a data type table is shown in Table 3.1. In this table, we have specified that variables of type “integer” range from -32767 to 32768; this corresponds to a 16 bit Boolean in hardware. Also, this same table can be used to make entries for specific variables from the input description. This enables the designer to use his

Table 3.1: *An example of a table of data types in the hardware description file.*

Data type	Lower Data Range	upper data range
integer	-32767	32768
myVariable	0	15

or her knowledge of the design to provide constraints on the range of the variables. For example, in Table 3.1, we have specified that variable "myVariable" in the design description has a range of only 0 to 15. The synthesis tool can use this information to generate a 4 bit Boolean in hardware for this variable instead of the 16 bit Boolean generated for all the other integers.

The data type information is essential for generating synthesizable HDL code, since logic synthesis tools require the exact range of data types in the HDL code. Hence, the back-end code generator for a high-level synthesis tool needs this table of data types for correct VHDL generation.

3.3.2 Modeling the Hardware Resources

We model functional resources or *functional units* by means of a *hardware resource library.* The hardware resource library contains the following information:

- *Operations that can be mapped to the resource.* For example, we can specify that both adds and subtracts can be mapped to an *ALU* unit. In a technology library, there may be several functional units on which an operation can be executed. The task of choosing a particular functional unit from a library of components for each operation in the design is known as *module selection.* Module selection has been the subject of much research in the past [LT81, MPC90, RCHP91, IM91]; however, for the purpose of the work presented in this book, we assume that the designer does the resource allocation and module selection a priori.

- *Number of inputs and outputs of the resource.* Currently, the resource model in our system is limited to a 2-input (or 1-input) and 1-output configuration. The exception is function calls that we also model as hardware resources. The number of inputs and outputs of a function call is determined from the function declaration in the input description. Hence, one way to model multi-input, complex resources in our high-level synthesis framework is to declare them as function calls.

- *Cycles and execution time (in nanoseconds) of the resource.* This information is used for determining if the resource may be chained or is a multi-cycle resource. The information in these two parameters is redundant; only execution time would suffice.

- *Number of units of each resource type.* This corresponds to the resource allocation for the design.

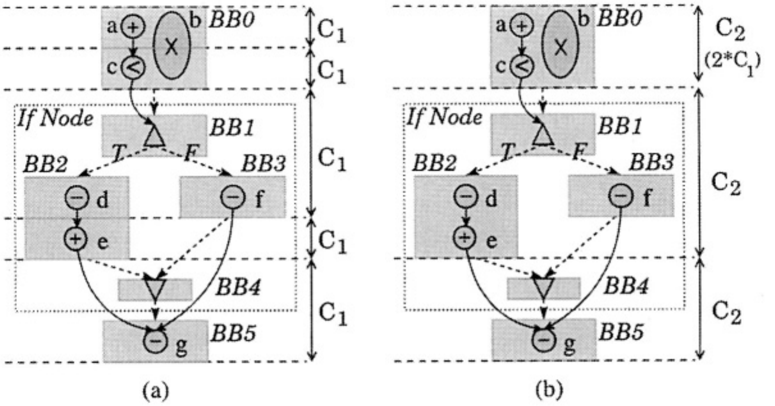


Figure 3.8: ***(a) HTG representation of an example scheduled with a 2-cycle multiplier; all other resources are single cycle. C_1 is the clock cycle period. (b) Clock period is doubled to C_2. Now, multiplier executes in one cycle and other resources in half a cycle. This enables chaining of operations a and c in basic block bb_0 and operations d and e in bb_2.***

➢ *Area cost of the resource.* This can be incorporated into cost functions used by module selection heuristics when choosing between several resources that an operation can be mapped to.

The hardware resource model presented above can be extended to include information about structural pipelining and register, communication, interconnect and bus allocations. Memory resources such as registers and read-only and random-access memories (ROMs and RAMs) can also be explicitly modeled in the same way.

3.3.3 Modeling Clock Cycle Timing

The hardware description file also contains information about the *clock period* of the design, denoted by C_{CLK}. The clock period signifies the time available in each control step or cycle for operations to execute. The scheduler uses this time and the resource execution time (defined in the hardware resource library, as explained in the previous section) to schedule operations into each clock cycle.

Consider the example in Figure 3.8(a). The clock cycle period assigned to this design is C_1. The design has been scheduled with a 2-cycle multiplier and one each of a single cycle adder, subtracter and comparator. So, lets say C_1 is 10 nanoseconds (ns), then the multiplier executes in $20ns$ and the other resources in $10ns$. The dashed lines denote the cycle boundaries. Hence, as shown in this figure, the multi-cycle multiply operation (b) is scheduled to execute over two cycles. Note that, all the C_1 time spans are of equal value even though they appear unequal in Figure 3.8(a).

3.3.4 Modeling Operation Chaining

Operation chaining is an important high-level synthesis technique. Two operations that are chained together execute back-to-back in the same cycle without any memory element in between to store the intermediate result. That is, two operations are chained within a basic block by scheduling them into the same control step (or clock cycle) and tying the outputs of one operation to the inputs of the other operation.

If we double the clock period in the example in Figure 3.8(a) to $C_2 = 2 * C_1$, then one possible scheduled design is as shown in Figure 3.8(b). The multiply operation now takes one cycle of the new clock period C_2 to execute (both are $20ns$). In this example, operations a and c, and the operations d and e, are chained together; that is, they execute back-to-back in the same cycle with no memory elements in between. This is because the total execution time of the chained operations is $10 + 10 = 20ns$. In this example and in the rest of our work, we assume that multiplexing and control overheads are included in the execution times given for each resource in the hardware resource library. Also, with the chained operations, basic blocks bb_0 and bb_2 have only one control step each versus the example in Figure 3.8(a), where they had two control steps each.

3.4 Formulation of the Scheduling Problem without Control Flow

Scheduling of operations during high-level synthesis is constrained by data dependencies and hardware resource allocations, as discussed in the next two sections.

3.4.1 Constraints due to Hardware Resource Allocation

The scheduling task in high-level synthesis is typically area constrained. This constraint on area is translated into a constraint on the number of functional units that are allocated to schedule the design, as given by the following definition:

Definition 3.9. *The set of resource types Res_{type} is defined as $Res_{type} = \{res_k; k = 1, 2, ..., n_{res}\}$ of n_{res} resource types. The number of resources of each resource type is given by the* **resource allocation** $\alpha = \{a_k; k = 1, 2, ..., n_{res}\}$ *that is a mapping $\alpha : Res_{type} \rightarrow Z^+$from the set of resource types to positive integers. $\alpha(res_k)$ denotes the number of resources allocated for resource type $res_k \in Res_{type}$. We define the execution time (in nanoseconds) ofthe resource types in Res_{type} as: $T_{res} = \{tres_k; k = 1, 2, ..., n_{res}\}$. We also define a* **resource list** $\mathcal{R} = \{res_l; l = 1, 2, ..., \sum a_k{}_{k=1}^{n_{res}}\}$ *that is a list of all resources of all resource types allocated to schedule the design.*

Using this definition of resource allocation, we can define a function that maps operations to resources as follows:

Definition 3.10. *We can denote the unique resource type that implements an operation by the function $\mathcal{T} : Ops \rightarrow Res_{type}$.*

This function assumes that there is only one resource type that can implement the given operation. If there is more than one resource type that an operation can execute on, then we also have to solve the module selection problem [LT81] and the function $\mathcal{T}$ becomes a one-to-many mapping. For the purposes of this work, we have assumed a one-to-one mapping of operations to resources.

3.4.2 Constraints due to Data Dependencies

Given a data dependency graph $G_{DFG}(Ops, E_{data})$ as defined in Definition 3.2, we can define a set of operation *execution delays* as $D_{ops} = \{dop_i; i = 1, ..., n_{ops}\}$. These operation execution delays are equivalent to the execution time of the resource (given by T_{res}) that the operation is scheduled on, as explained in the previous section. Let $T_{ops} = \{t_i; i = 0, 1, ..., n_{ops} - 1\}$ be the execution *start times* of the operations. Then, as per the flow data dependency graph, if an operation op_i reads the result of another operation op_j, then op_i can start execution only after op_j has finished execution. This can be expressed as:

$$t_i \geq t_j + dop_j \;\; \forall\, i, j \;\; : \; (op_j, op_i) \in E \tag{3.1}$$

3.4.3 Resource-Constrained Scheduling

In a *resource-constrained* scheduling approach, the number of resources of each resource type are upper bound by α as defined above. Hence, the number of operations mapped to a resource type in any clock cycle cannot exceed the upper bound of that resource type. De Micheli [DM94] has defined the *minimum-latency resource-constrained* scheduling problem as follows:

Definition 3.11. *Given a set of operations Ops with integer delays Dops, a partial order on the operations due to data dependencies E_{data}, n_{res} resource types whose numbers are upper bound by $\alpha = \{a_k; k = 1, 2, ..., n_{res}\}$, the* ***resource-constraint scheduling problem*** *is to find an integer labeling of the operations $\varphi : Ops \rightarrow Z^+$ such that $t_i = \varphi(op_i)$, $t_i \geq t_j + dop_j \; \forall\, i, j$ s.t. $(op_j, op_i) \in E_{data}$, $|\{op_i : \mathcal{T}(op_i) = res_k$ and $t_i \leq l < t_i + dop_i\}| \leq a_k$ for each resource type res_k, $k = 1, 2, ..., n_{res}$ and control step $l = 1, 2, ..., t_n$. The* ***minimum-latency*** *resource-constraint scheduling problem is an integer labeling φ such that t_n is minimum.*

The above definition essentially says that we have to find the start time of all the operations, while making sure that each operation starts only after all the operations it depends on have finished execution ($t_i \geq t_j + d_j$). Also, the number of operations mapped to each resource type (res_k) in any control step (l) has to be less than or equal to available resources of that type (a_k). De Micheli goes on to point out that if all the resources are of a given type (e.g., ALUs), then the problem reduces to the classical multiprocessor scheduling problem. The minimum-latency multiprocessor scheduling problem is intractable [GJ79]. The minimum-latency resource constrained scheduling problem is NP-complete [GDWL92, DM94, NR00].

3.4.4 Incorporating Operation Chaining in Scheduling Formulation

Operation chaining can be incorporated into the scheduling formulation given above without any modification, as long as the chained operations obey the following constraint:

Definition 3.12. *Two operations, op_i and op_j that have a flow data dependency between them can be chained together if their execution times dop_i and dop_j are such that, $dop_i + dop_j \leq C_{CLK}$, where C_{CLK} is the clock cycle period allocated to schedule the design. Also, operation op_j can start execution only after op_i has finished execution. This definition assumes the time taken by multiplexing and control is incorporated in the execution times of the operations.*

3.5 Modeling Parallelizing Code Motions

The presence of control flow means that operations *cannot* be arbitrarily scheduled in any basic block. A condition, based on the conditional construct the operation is in, is associated with each operation. The condition for operations that are not in conditional constructs is "true". Operations can be moved along their control path by employing speculative and non-speculative code motions as explained in the next section.

3.5.1 Modeling Speculative Code Motions

Operations can be moved freely along the control flow path they lie on as long as a fork or join node is not encountered. Operations may be moved across these control nodes as per the rules given below.

- *Moving past a predecessorjoin node:* If an operation op is in basic block bb_{op}, where the predecessor basic block bb_{pred} of bb_{op} is a *join* node, i.e., multiple basic blocks $bb_i : i = 1, 2, ..j$ merge at bb_{pred}, then op is moved past bb_{pred} by placing *identical* duplicates of op in all the basic blocks $bb_i : i = 1, 2, ..j$.

- *Moving past a predecessor fork node:* If an operation op is in basic block bb_{op}, where the predecessor basic block bb_{pred} of bb_{op} is a *fork* node, i.e., multiple basic blocks $bb_i : i = 1, 2, ..j$ branch out at bb_{pred} ($bb_{op} \in \{bb_i : i = 1, 2, ..j\}$), then op is moved past bb_{pred} by duplicating op into two non-identical operations op_1 and op_2. Two common approaches to create these operations are:

  - A new operation op_1 is created with the computation from the original operation op that stores the result in a new variable *newResultV ar*. op_1 is placed in the basic block with the fork node, i.e. bb_{pred} (before the fork). Another new operation op_2 is created that is a *copy* operation that copies the new variable *newResultV ar* to the result variable of the original operation op. Operation op_2 is placed in basic block bb_{op} in place of op.

- Alternatively, the original operation op is placed in bb_{pred} and compensation code is inserted in the basic blocks $\{bb_i : i = 1, 2, ..j\}$ that are in the control paths that branch out of the fork basic block. This compensation code undoes any effects that executing the original operation op in bb_{pred} will have on the system state.

In our approach, we have chosen the former approach.

- *Moving past a successor fork node:* If an operation op is in basic block bb_{op}, where the successor basic block bb_{succ} of bb_{op} is a *fork* node, i.e., multiple basic blocks $bb_i : i = 1, 2, ..j$ branch out of bb_{succ}, then op is moved past bb_{succ} by placing *identical* duplicates of op into all the basic blocks $bb_i : i = 1, 2, ..j$.

- *Moving past a successor join node:* If an operation op is in basic block bb_{op}, where the successor basic block bb_{succ} of bb_{op} is a *join* node, i.e., multiple basic blocks $bb_i : i = 1, 2, ..j$ merge at bb_{succ} $(bb_{op} \in \{bb_i : i = 1, 2, ..j\})$, then in our approach, op cannot be moved past bb_{succ}. This code motion can, however, be enabled by finding and merging identical operations in all the basic blocks $\{bb_i : i = 1, 2, ..j\}$ in the control paths that merge at the bb_{succ}.

3.5.2 Modeling Hierarchical Code Motions

In addition to the speculative code motions described above, the hierarchical representation employed by our methodology – hierarchical task graphs (HTGs) – enables operations to be moved hierarchically across large pieces of code. When an operation encounters a fork (or join) node during code motion, the operation can be directly moved to the corresponding join (or fork) node if the operation does not have any data dependencies with the operations in the hierarchical HTG node that the fork-join nodes belong to. These code motions are enabled by a code motion technique called *Trailblazing* [NN93] and are governed by the following rules:

- If an operation op is in basic block bb_{op}, where the predecessor basic block bb_{pred} of bb_{op} is a *join* node, i.e., multiple basic blocks $bb_i : i = 1, 2, ..j$ merge at bb_{pred}, then op can be moved to the basic block in the *Start* node of HTG node htg_i, provided: (a) bb_{pred} is in the *Stop* node of a HTG node htg_i in the design HTG, G_{HTG} (i.e. $Stop(htg_i) = bb_{pred}$), and (b) there are no data dependencies between op and the operations in htg_i.

- If an operation op is in basic block bb_{op}, where the successor basic block bb_{succ} of bb_{op} is a *fork* node, i.e., multiple basic blocks $bb_i : i = 1, 2, ..j$ branch out of bb_{succ}, then op can be moved to the basic block in the *Stop* node of HTG node htg_i, provided: (a) bb_{succ} is the *Start* node of the HTG node htg_i in the design HTG, G_{HTG} (i.e. $Start(htg_i) = bb_{succ}$), and (b) there are no data dependencies between op and the operations in htg_i.

The hierarchical code motions described above can be achieved by the basic speculative code transformations described in Section 3.5.1 by duplicating the operations at the fork/join node and then merging the identical copies at the corresponding join/fork node. However, the hierarchical structuring of HTGs provides an efficient and elegant way to enable these hierarchical code motions.

3.6 Scheduling Designs with Control Flow

The presence of control flow introduces another dimension to the high-level synthesis scheduling problem. Besides data dependencies and hardware resource constraints, the scheduler is also constrained by the control flow semantics of the design description, as explained in the next two sections.

3.6.1 Notion of Scheduling Steps within Basic Blocks

Scheduling assigns time steps in which operations execute. Frequently, several operations are scheduled to execute concurrently. Furthermore, operations on mutually exclusive control paths may execute concurrently as long as the number of resources used in each control path does not exceed the resource allocation. To capture the control or time steps that belong to different control paths, we introduce the notion of scheduling steps. **We define a *scheduling step* as an aggregation of operations that execute concurrently within a basic block.** A basic block is, hence, a sequence of scheduling steps with no control flow between them. A more precise definition of scheduling steps can be given as follows:

Definition 3.13. *There exists a set of* n_{steps} *scheduling steps* $Steps = \{step_k; k = 1, 2, ..., n_{steps}\}$ *in a design graph* $\mathcal{DG}$ *such that each step* $step_k$ *belongs to a basic block* $bb_i \in BB$.

There is a partial ordering on Steps since within each basic block the scheduling steps are ordered as per the sequence in which they execute. Hence, there exists an ordered set of $bbsteps_i$ *scheduling steps* $StepsInBB_i = (stepi_l; l = 1, 2, ..., bbsteps_i)$ *within a basic block* bb_i*, such that* $StepsInBB_i \subseteq Steps$*. The first scheduling step in* bb_i *is given by* $FirstStep(bb_i) = stepi_1 \in StepsInBB_i$*.* $NextStep(bb_i,\ stepi_l)$ *returns the next step in basic block* bb_i *after* $stepi_l$*. The last step in* bb_i *does not have a next step.*

The mapping of steps to basic blocks is given by $BB_{Steps} : Steps \rightarrow BB$*. Hence,* $BB_{Steps}\,(step_k) = bb_i$ *is the basic block that* $step_k$ *is in. There also exists a many-to-one mapping of operations to scheduling steps within a basic block* bb_i *given by* SS : $Ops \rightarrow Steps$ *such that if the scheduling step of an operation* op_j *is* $SS(op_j) = step_k$*, then* $BB_{Ops}(op_j) = bb_i = BB_{Steps}(step_k)\ \forall\ op_j \in Ops$*.*

When the input description is captured by the control and data flow graphs, one scheduling step is created corresponding to each operation in the design. That is, initially there is no operations executing concurrently. This is because the input language to our methodology is “C”, which is a sequential language with no notion of concurrency between operations. So, initially the mapping SS is a one-to-one mapping. After scheduling, operations may be scheduled to execute concurrently and hence, SS

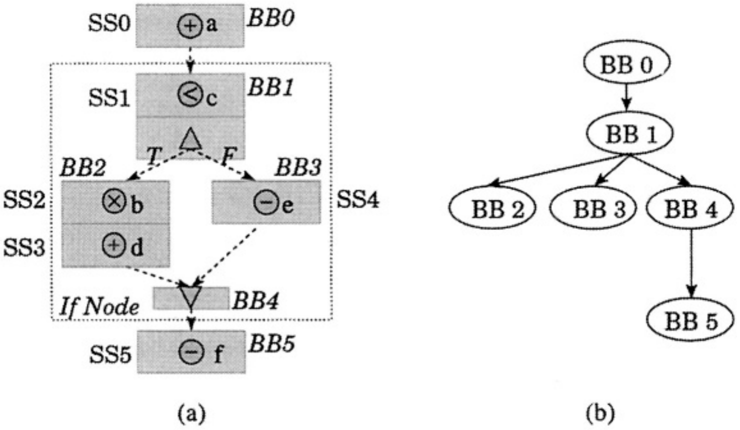


Figure 3.9: ***(a) Example to explain how control paths determine which operations are eligible to be scheduled in the various scheduling steps. (b) Dominator tree for this example.***

becomes a many-to-one mapping. Thus, the scheduling function also leads to a new mapping of the operations to scheduling steps in basic blocks.

3.6.2 Formulation of Scheduling Problem with Conditional Constructs

The basic scheduling problem formulation remains the same as given in Definition 3.11. When there is no control flow in a design, the only constraint to select the operations that may be scheduled on the scheduling step under consideration is that operations have to obey their data dependencies. However, the presence of control flow in a design means that operations can only be scheduled in basic blocks that have a control path from the basic block the operation is currently in.

The relationship between basic blocks in a control flow graph can be captured using *dominator trees* [ASU86]. These trees can be constructed using the following definition:

Definition 3.14. *A node n in a control flow graph (CFG) is said to* ***dominate*** *another node m, if every path from the Initial node of the flow graph to m goes through n. Note that, every basic block dominates itself.*

Let us understand dominator trees using the example in Figure 3.9(a) and the corresponding dominator tree in Figure 3.9(b). In this example, basic block bb_1 dominates basic blocks bb_2, bb_3 and bb_4 and is itself dominated by bb_0. bb_4 in turn dominates bb_5.

We can now define the set of operations that is eligible for scheduling into the scheduling step under consideration as:

Definition 3.15. *An operation op in basic block bb_{op} is eligible for scheduling into a scheduling step $step_k$ in basic block bb_{stepk}, if either bb_{stepk} dominates bb_{op} or bb_{op} dominates bb_{stepk}. The operation is still subject to data dependency constraints.*

This definition means that there has to be a control path from bb_{op} to bb_{step} or vice versa for the operation to be eligible for scheduling in *step*. However, the constraint given by this definition is too restrictive. Consider the example in Figure 3.9(a) again. By Definition 3.15, operation f in basic block bb_5 cannot be scheduled into basic block bb_2, since bb_5 does not dominate bb_2 (see dominator tree in Figure 3.9(b)). However, operation f can be scheduled into bb_2, as long as we *duplicate* it into basic block bb_3 as well; this was explained earlier in Section 3.5.1. This introduces the notion of dominance by a set of basic blocks.

Definition 3.16. *A set of nodes $\mathcal{N}$ in a control flow graph (CFG) is said to* ***dominate*** *another node m, if every path from the initial node of the flow graph to m goes through at least one node in $\mathcal{N}$.*

Now, we can update Definition 3.15 as:

Definition 3.17. *An operation op in basic block bb_{op} is eligible for scheduling into a scheduling step $step_k$ in basic block bb_{stepk}, if there exists a set of basic blocks $\mathcal{B}$, $bb_{stepk} \in \mathcal{B}$, such that either $\mathcal{B}$ dominates bb_{op} or bb_{op} dominates $\mathcal{B}$. Operation op has to be duplicated into each basic block in $\mathcal{B}$.*

As per this definition, operations can be duplicated at fork and join nodes and scheduled in the basic blocks of conditional branches.

Using these definitions of scheduling steps and restrictions on operation scheduling, we can now define resource-constrained scheduling of designs with control-flow as:

Definition 3.18. *We are given:*

- ❒ *A data flow graph, $G_d(Ops, E_{data})$ with a vertex set of operations $Ops = \{op_i;\ i = 1, 2, ..., n_{ops}\}$ and a directed edge set $E_{data} = \{(op_i, op_j);\ i, j = 1, 2, ..., n_{ops}\}$ that corresponds to a partial order due to data dependencies.*
- ❒ *A control flow graph, $G_{CFG}(BB, E_{control})$ with a vertex set of basic blocks $BB = \{bb_i; i = 1, 2, ..., n_{bbs}\}$ and a directed edge set $E_{control} = \{(bb_i, bb_j); i, j = 1, 2, ..., n_{bbs}\}$ that corresponds to the control flow between the basic blocks.*
- ❒ *A many-to-one mapping of the operations to the basic blocks $BB_{Ops} : Ops \rightarrow BB$. The operations have integer delays D and have to be mapped onto n_{res} resource types whose numbers are upper bound by $\{a_k;\ k = 1, 2, ..., n_{res}\}$.*

The resource-constrained scheduling problem is a function $\psi : BB_{Ops} \rightarrow BB_{Ops_{sched}}$ that finds a new mapping of the operations to basic blocks, $BB_{Ops_{sched}} : Ops \rightarrow BB$, such that:

- ◆ $\forall\ i$, *if $bb_n = BB_{Ops}(op_i)$ and $bb_m = BB_{Ops_{sched}}(op_i)$, then either bb_m dominates bb_n or bb_n dominates bb_m or $\exists$ a set of basic blocks $\mathcal{B} \subset BB$, $bb_m \in \mathcal{B}$, such that either $\mathcal{B}$ dominates bb_n or bb_n dominates $\mathcal{B}$ and operation op_i has been duplicated in each basic block in $\mathcal{B}$.*

- *Start times of the operations,* $\varphi : Ops \rightarrow Z^+$*, are such that:* $t_i = \varphi(op_i)$, $t_i \geq t_j + d_j \; \forall\, i, j$ *s.t.* $(op_j, op_i) \in E_{data}$,

- *Each operation is mapped to a resource type* $\mathcal{T} : Ops \rightarrow \{1, 2, ..., n_{res}\}$ *such that:* $|\{op_i : \mathcal{T}(op_i) = k$ *and* $t_i \leq l < t_i + d_i\}| \leq a_k$ *for each resource type* $k = 1, 2, ..., n_{res}$ *and* $\forall$ *control steps* $l = 1, 2, ..., t_n$ *within each basic block* $bb_j \in BB$*. This can also be stated as* $|\{op_i : \mathcal{T}(op_i) = k\}| \leq a_k$ *for each scheduling step* $stepj_k \in Steps(bb_j) = \{stepj_k; k = 1, 2, ..., nj_{steps}\}$ *in each basic block* $bb_j \in BB$.

The *minimum-latency* resource-constrained scheduling optimization problem is then to minimize t_n, i.e., to minimize the schedule length of the design. The ψ scheduling function also finds a new many-to-one mapping of operations to scheduling steps within a basic block bb_i given by $SS_{sched} : Ops \rightarrow Steps$ such that if the new scheduling step of an operation op_j is $SS_{sched}(op_j) = step_k$, then $BB_{Ops_{sched}}(op_j) = bb_i = BB_{Steps}(step_k) \; \forall \; op_j \in Ops$.

3.6.3 Modeling Resource Utilization

In order to generate a schedule as per the definition given above, a scheduler has to maintain information about the resource utilization in each scheduling step. A resource is said to be *utilized* in a clock cycle on a control path if an operation is scheduled on the resource in that clock cycle on that control path. This can be defined more formally as:

Definition 3.19. *When an operation* op_i *is scheduled on a resource* res_l *in a scheduling step* $step_k$*, we denote this by* $step_k(res_l) = op_i$*. Resource* res_l *is then said to be* ***utilized*** *in* $step_k$*. Conversely, if there is no operation scheduled on resource* res_l *in* $step_k$*, then* $step_k(res_l) = \emptyset$*. Resource* res_l *is then said to be an* ***idle resource*** *in* $step_k$*.*

To understand the information required and generated by the scheduler, consider the HTG representation of an example shown in Figure 3.10(a). The flow data dependencies among the operations are also shown in this HTG representation. The comparison operation for the if-HTG in basic block bb_1 has been split into the comparison operation itself that produces result "c" and the check of this Boolean condition (denoted by a triangle).

Now consider that an allocation of one adder, one subtracter, one comparator and one multiplier is available to schedule this design. The multiplier executes in two cycles whereas all other units execute in one cycle. A schedule where all the operations are scheduled *as soon as possible* (ASAP) is shown in Figure 3.10(b). In this schedule, operations a and c are scheduled concurrently in basic block bb_0. Also, since operation b is scheduled on the multiplier, its dependent operation d can only start execution two cycles after b. The resource utilization in each scheduling step in the (non-empty) basic blocks of this scheduled HTG is shown in Figure 3.10(c). The shaded boxes indicate that an operation is scheduled on the resource and the faded white boxes indicate

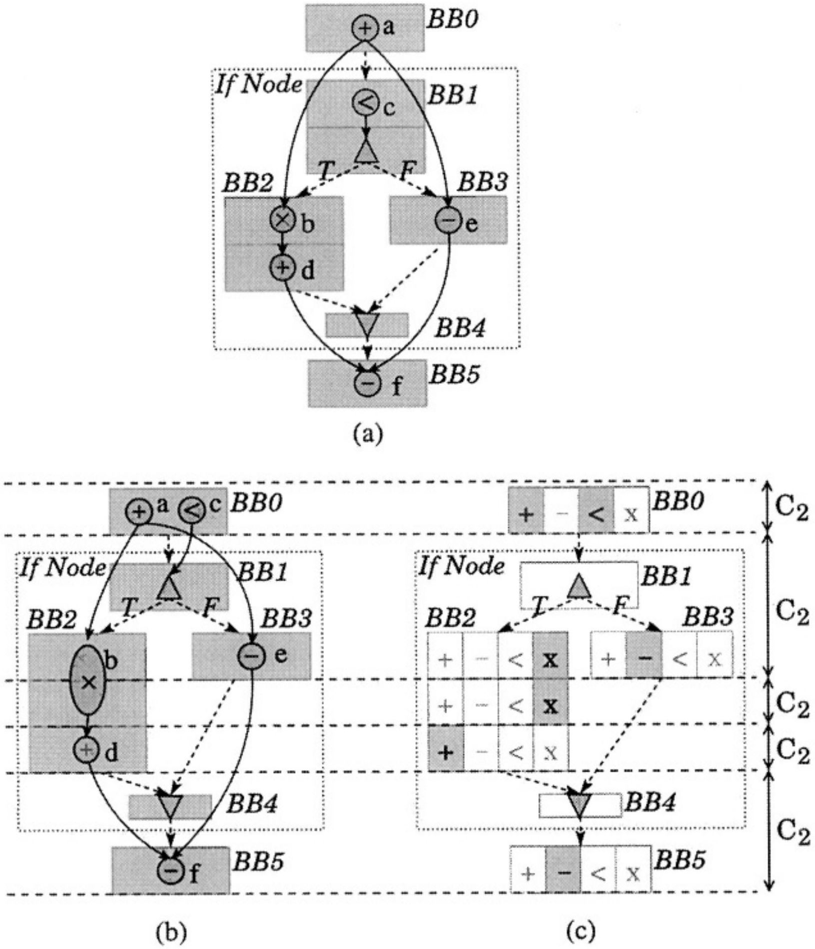


Figure 3.10: ***(a) An example HTG representation along with*** **flow** ***data dependencies. (b) HTG of the example scheduled with a resource allocation of one adder, one subtracter, one comparator and one (two cycle) multiplier. (b) Resource utilization in each scheduling step. A shaded box indicate that an operation is scheduled on the resource.***

idle resources. A resource is considered to be **idle** in a scheduling step if there is no operation scheduled to execute on that resource in that step.

Hence, the scheduler maintains and generates the following information:

- Operations that are scheduled.
- Scheduling step in which each operation is scheduled.
- Resource type of the resource that the operations are scheduled on.

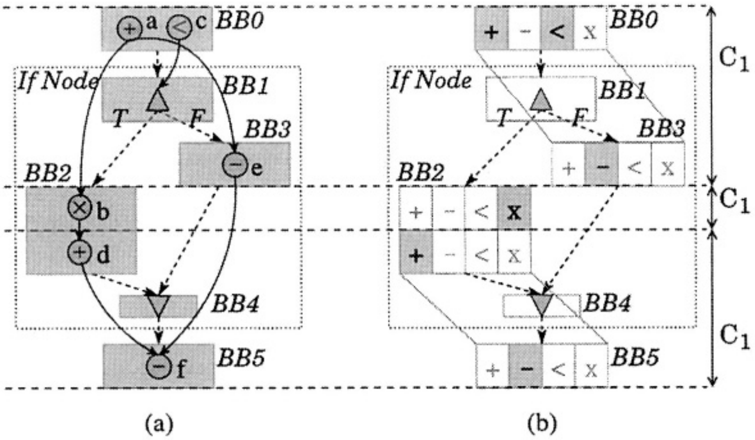


Figure 3.11: *(a) Example from Figure 3.10(a) scheduled after doubling cycle time; the multiplier now executes in one cycle and all other resources in half a cycle. Hence, operation e is chained with operation a across the conditional boundary. (b) The resource utilization of the resources in each scheduling step.*

- Resource utilization of each scheduling step.

Note that, the particular instance of a resource type that an operation is scheduled on is determined by the resource binding task.

Operations in the mutually exclusive branches of a conditional block such as an if-HTG node can share the same resources in the same cycle. Hence, the operations in basic blocks bb_2 and bb_3 in Figure 3.10(b) can share the resources in the same cycle.

Within this resource utilization model, operations can be chained to execute together within a scheduling step by tying the resources that they are scheduled on, together. Two consecutive scheduling steps may also be chained to execute in the same cycle, albeit only if they are chained across a conditional boundary. This is explained in the next section.

3.6.4 Modeling Operation Chaining across Conditional Boundaries

In this book, we introduce a novel technique of chaining operations across conditional boundaries. This technique chains operations that execute under different conditions (hence, are in different scheduling steps) to execute back-to-back in the same cycle. This chaining of operations *across conditional boundaries* requires the scheduler to look at the resource utilization of multiple basic blocks during scheduling.

Consider the example in Figure 3.11(a). Also, consider that in this example the multiplier executes in one cycle and all the other resources execute in half a cycle. We observe that it is possible to chain operation e in basic block bb_3 with operation a in basic block bb_0, as shown in Figure 3.11(a). The clock boundaries are demarcated by dashed lines. In this figure, we placed bb_3 a little higher than basic block bb_2 to indicate that the scheduling step in basic block bb_3 is chained across the conditional boundary with the scheduling step in basic block bb_0.

Figure 3.11(b) illustrates the resource utilization for this scheduled HTG. To indicate the chaining across conditional boundaries, we have shown a connection between the scheduling steps in bb_0 and bb_3 in Figure 3.11 (b). Similarly, in this example, we also chain operation f with operation d since they both execute in half a cycle. Note that, in Figure 3.11(b), the clock period C_1 is the same in all three clock cycles although it appears different in the figure.

3.7 Summary

In this chapter, we presented the models and representations used in this book. We first presented the layered graph intermediate representation used to capture the input description in Section 3.2. In Section 3.3, we discussed the additional information that is given as input with a design, such as the hardware resource library and the clock cycle period. We then presented the resource-constrained scheduling problem – without considering control flow – in Section 3.4. In Section 3.5, we introduced the modeling of the speculative and hierarchical code motions employed for scheduling control flow designs. We then extended the model for the scheduling problem to include control flow in Section 3.6. In this section, we also presented the modeling of resource utilization across mutually exclusive control paths in a design. The contributions of this chapter are the layered intermediate representation, the modeling for code motions across control flow and the formulation of the scheduling problem for designs with control flow.

Part II

Parallelizing High-Level Synthesis (PHLS)

4

OUR PARALLELIZING HIGH-LEVEL SYNTHESIS METHODOLOGY

In this chapter, we present a methodology for parallelizing high-level synthesis (PHLS) and discuss the design flow through a framework that implements this methodology. We present an overview of a range of synthesis and compiler transformations and code refinement passes that form part of this methodology. We propose a modular, tool-box construction of the PHLS framework to enable research into heuristics that guide the code transformations. We begin this chapter by presenting the design flow through our proposed PHLS framework, followed by a detailed look at some of the passes in the methodology.

4.1 Design Flow through a PHLS Framework

An overview of our proposed PHLS framework is shown in Figure 4.1. The design flow through this framework is as follows: the framework accepts a behavioral description of a design in high-level language such as C, C++ (or their variants) et cetera, captures the description using a multi-layer hierarchical intermediate representation (see Chapter 3), runs a data dependency analysis pass, schedules the design, binds the resources, performs control synthesis, and finally generates an output in a register-transfer level (RTL) hardware description language such as VHDL or Verilog. As shown in Figure 4.1, the framework also requires additional input consisting of the hardware resource library, resource and timing constraints and user directives for the various heuristics and transformations.

In this framework, we first apply a set of coarse-grain and fine-grain code transformations to the input description during a *pre-synthesis* phase before performing the traditional high-level synthesis tasks of scheduling, allocation and binding. The transformations in the *pre-synthesis phase* include (a) coarse-level code restructuring by function inlining and loop transformations (loop unrolling, loop fusion et cetera), (b) transformations that remove unnecessary and redundant operations such as common sub-expression elimination (CSE), copy propagation, and dead code elimination, (c)

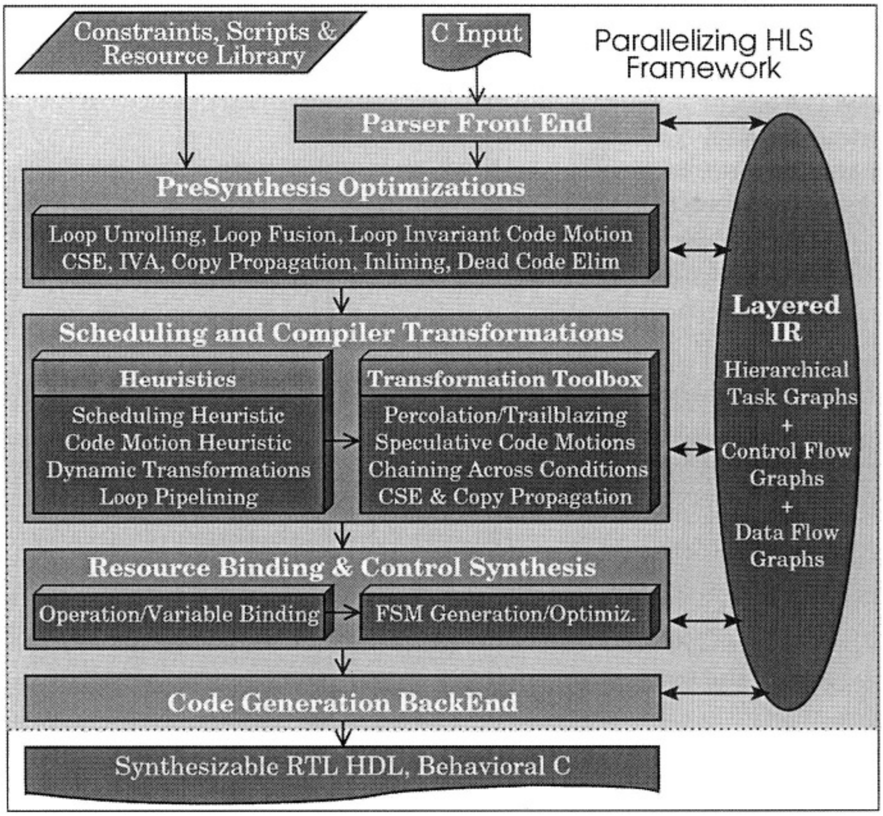

Constraints, Scripts & Resource Library
C Input
Parallelizing HLS Framework
Parser Front End
PreSynthesis Optimizations
Loop Unrolling, Loop Fusion, Loop Invariant Code Motion
CSE, IVA, Copy Propagation, Inlining, Dead Code Elim
Scheduling and Compiler Transformations
Heuristics
Scheduling Heuristic
Code Motion Heuristic
Dynamic Transformations
Loop Pipelining
Transformation Toolbox
Percolation/Trailblazing
Speculative Code Motions
Chaining Across Conditions
CSE & Copy Propagation
Resource Binding & Control Synthesis
Operation/Variable Binding
FSM Generation/Optimiz.
Code Generation BackEnd
Synthesizable RTL HDL, Behavioral C
Layered IR
Hierarchical Task Graphs
+
Control Flow Graphs
+
Data Flow Graphs

Figure 4.1: *An Overview of our proposed parallelizing high-level synthesis framework.*

transformations such as loop-invariant code motion, induction variable analysis (IVA) and operation strength reduction that reduce the number of operations within loops and replace expensive operations (multiplications and divisions) with simpler operations (shifts, additions and subtractions).

The pre-synthesis phase is followed by the scheduling and allocation phase (see Figure 4.1). We will only discuss the scheduling framework here. There has been a body of work in resource allocation and module selection that is applicable and complementary to our work (see Chapter 2). The scheduler is organized into two parts: the heuristics that perform scheduling and a transformations toolbox. The *transformations toolbox* contains speculative code motion transformations, code motion techniques (such as *Percolation* and *Trailblazing* [NN93, Nic85a]), *dynamic renaming* of variables et cetera. The synthesis transformations include chaining operations across conditional blocks, scheduling on multi-cycle operations, resource sharing et cetera [DM94].

Besides the traditional high-level synthesis transformations, the scheduling phase also employs several compiler transformations applied “dynamically” during scheduling. These dynamic transformations, such as dynamic CSE and dynamic copy propagation, exploit the new opportunities created by code motions. A dynamic branch balancing technique also adds scheduling steps in conditional branches dynamically during scheduling to enable code motions; this is particularly useful for enabling code motions such as conditional speculation that duplicate operations in conditional branches.

Passes from the toolbox are called by a set of heuristics that guide how the code refinement takes place. The heuristics and the underlying transformations that they use are kept completely independent. This allows the heuristics to employ the various transformations as and when required, thus enabling a modular approach that allows the easy development of new heuristics. Also, the use of the passes and transformations can be controlled by the designer using synthesis scripts.

In addition to the transformations applied during scheduling, we apply an incremental loop pipelining technique called *loop shifting* after scheduling the design once [GDGN04]. As described later in Chapter 6, loop shifting operates on loops after scheduling and moves a set of operations from the beginning of the loop body to the end of the loop body and inserts a copy in the loop header. The SPARK scheduler then *reschedules* the loop and applies parallelizing compiler transformations to once again compact the loop body. We found that loop shifting is extremely useful for extracting inter-loop iteration parallelism in designs with complex control flow. Whereas traditional aggressive loop pipelining techniques such as modulo scheduling can result in large increases in the control and interconnect logic, incremental loop pipelining techniques like loop shifting expose just as much parallelism as can be utilized by code compaction techniques without adversely affecting circuit area and delay.

The scheduling phase is followed by a resource binding and control synthesis phase. This phase binds operations to functional units, ties the functional units together (interconnect binding), allocates and binds storage (registers), generates the steering logic, and generates the control circuits to implement the schedule. As we will discuss in Chapter 8, the focus of our resource binding approach is to minimize the interconnect between functional units and registers. After binding, we generate a finite state machine controller for the scheduled and bound design.

Finally, a back-end code generation pass generates register-transfer level (RTL) code in a hardware description language (HDL) such as VHDL or Verilog. This RTL HDL code belongs to the subset of the HDL that is synthesizable by commercial logic synthesis tools [DC, Xil]. This enables our synthesis framework to complete the design flow path from architectural design to final design netlist. Note that, the output RTL HDL code is structural with all operations bound to resources (i.e., components are instantiated in the HDL code) and variables bound to registers.

We believe that such a parallelizing high-level synthesis framework should also have a back-end code generation pass that generates ANSI-C (or C++). This behavioral output code represents the scheduled and optimized design. The output “C” can be used in conjunction with the input “C” to perform functional verification and also, to improve visualization for the designer on the affects of the transformations applied by the PHLS framework on the design.

We implemented our parallelizing high-level synthesis methodology in the *Spark* PHLS framework. Implementation details about this framework are discussed in Chapter 9.

4.2 Passes and Techniques in the PHLS Framework

We present the various transformations in this PHLS framework over the next four chapters. In the rest of this chapter, we discuss three techniques that aid the transformations in the PHLS framework, namely, the data dependency analysis pass, dynamic variable renaming, and the Trailblazing code motion technique.

4.2.1 Data Dependency Analysis Pass

A data dependency analysis (DDA) pass is required to create the data flow graph from the design description. This DDA pass traverses the control flow in the design and determines the dependencies between operations based on this control flow. A list of live variables is maintained as the control flow in the design is traversed. The live variable list is duplicated whenever a fork is encountered in the control flow and the variable list of branches merging at a join are also merged to form a new live variable list.

Recall from the earlier discussion in Section 3.2.1 (Chapter 3), there are four types of data dependencies that can exist between two operations [KKP+81, Muc97]: *a flow* dependency is said to exist when an operation that writes to a variable is followed by an operation that reads the same variable, an *anti* dependency is when one operation that reads a variable is followed by an operation that writes to the same variable, an *output* dependency exists when two operations write to the same variable one after the other and an *input* dependency when two operations read from the same variable. Of these, input dependencies do not affect scheduling.

In our PHLS methodology, we believe it is important to construct data dependency graphs from the input description that maintain all the types of data dependencies between the variables in the source code. This is important since it enables the PHLS framework to maintain the variable names used in the input description. Hence, users can correlate the variables and operations from the input description to the intermediate representation used by the synthesis tool (see Chapter 3 for an example). This improves the ability to visualize the intermediate and final results of the transformations applied by the PHLS framework.

4.2.2 Eliminating data dependencies by Dynamic Renaming

Non-flow data dependencies, however, place constraints on operation movement. For example, if an operation op_1 that reads variable x is followed by an operation op_2 that writes to x, then it would seem that op_2 cannot be executed until op_1 has finished reading variable x.

However, these constraints can often be resolved by dynamic renaming and combining [CF87, ME92]. Figures 4.2(a) to (c) demonstrate how one operation can be

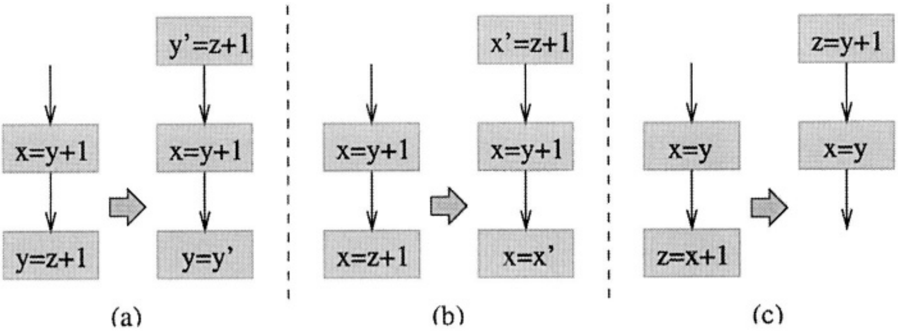


Figure 4.2: ***Moving one operation across another operation while eliminating (a) an anti dependency (b) an output dependency and (c) a flow dependency.***

moved past another one while dynamically eliminating data dependencies. In Figure 4.2(a), the two operations that write to variables x and y have a anti-dependency between them (since the first operation reads y and the second one writes to y). The operation that writes to variable y can, however, be scheduled at an earlier time step by moving only the right hand side of the operation. The result is written to a new destination variable y' and the original operation is replaced by a copy operation of the new destination variable y' to the original variable y. Similarly, in Figure 4.2(b), an output dependency between two operations that write to the same variable x can be resolved by creating a new destination variable x' while moving the operation and replacing the original operation with a copy operation.

Copy operations introduced by dynamic renaming can also be circumvented by a technique known as *combining* [ME92]. Combining replaces the copy in the operation being moved by the variable being copied; this is nothing but copy propagation that is performed while an operation is being moved. This is demonstrated in Figure 4.2(c), where the operation $z = x + 1$ is moved past the copy operation $x = y$. The variable x is replaced with the variable y on the right hand side of the moving operation.

In this way, dynamic renaming and combining, when performed in conjunction with code motion techniques such as Trailblazing and Percolation, can lead to considerable easing of the constraints imposed by data dependencies.

4.2.3 The *Trailblazing* Code Motion Technique

As mentioned earlier, the speculative code motions employed in our PHLS framework are enabled by the *Trailblazing* code motion technique [NN93]. Trailblazing is a hierarchical code motion technique that builds on earlier work done on *Percolation scheduling* [Nic85b, Nic85a]. Whereas Percolation suffers from the problems of incremental operation moves and code explosion [NN93], Trailblazing can efficiently move operations across large pieces of code.

Trailblazing exploits the information about the hierarchical structure of the design maintained by hierarchical task graphs (HTGs) [GP92] (see Section 3.2.5). The global

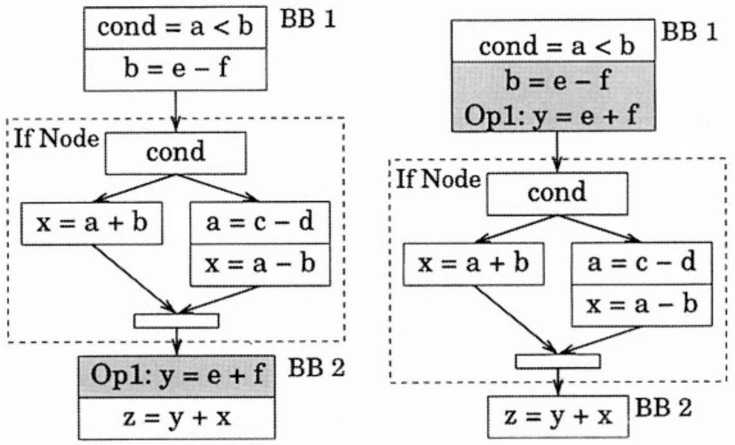


Figure 4.3: *Trailblazing: Operation* $op1$ *is moved from basic block* bb_2 *to basic block* bb_1 *across the if-then-else HTG node without visiting each basic block inside the node.*

structural information maintained by HTGs about the input description means that non-incremental moves can be made without visiting every basic block that is bypassed. At the lowest level, Trailblazing is able to perform the same fine-grained transformations as Percolation. However, at a higher level, Trailblazing is able to move operations across hierarchical blocks of code.

To understand the hierarchical moves performed by Trailblazing, consider the example in Figure 4.3. In this example, we want to move the operation $Op1 : y = e + f$ from basic block bb_2 to basic block bb_1. While moving this operation, Trailblazing encounters the join node of an if-HTG node. It determines if the moving operation has any dependencies with the if-HTG node. Since, in this example, there are no dependencies, then the operation is moved across the if-HTG node to bb_1 without visiting each sub-node of the if-HTG, as shown in Figure 4.3(b) (in practice $Op1$ is moved to the fork or Start node of the if-HTG). To perform the same code motion, Percolation would have duplicated $Op1$ into both the branches of the if-HTG, then moved it up each branch, and finally unified the copies back into $Op1$ at the conditional check, hence, in the process visiting each node in the if-then-else block.

We use the Trailblazing technique in our code motion algorithm albeit with modifications for high-level synthesis. Our modified algorithm, the *TrailSynth* algorithm, is presented in Chapter 7. *TrailSynth* also supports a dynamic branch balancing technique and high-level synthesis techniques such as operation chaining.

4.3 Summary

In this chapter, we presented a methodology for parallelizing high-level synthesis and described the design flow through a PHLS framework. We discussed the various passes and transformations that form part of this methodology. In Section 4.2, we presented three passes/techniques that aid the transformations applied in the PHLS framework. We first discussed a data dependency analysis pass that captures all types of data dependencies to enable better design visualization. In Section 4.2.2, we presented the dynamic variable renaming technique that is employed by the scheduler to eliminate non-flow data dependencies between operations. Next, we discussed the *Trailblazing* code motion technique that efficiently moves operations across large pieces of code. The contributions of this chapter are the presentation of our PHLS methodology and the design flow and transformations through a PHLS framework.

5

PRE-SYNTHESIS COMPILER OPTIMIZATIONS

In this chapter, we discuss the compiler transformations applied in the pre-synthesis phase of our parallelizing high-level synthesis methodology. Specifically, we discuss common sub-expression elimination (CSE), loop-invariant code motion, loop unrolling, and loop index variable elimination. The goal of applying these transformations before scheduling is three-fold: (i) to remove redundant and unnecessary operations, (ii) to reduce the number of operations that execute in the design (particularly in loops), and (ii) to increase the scope of the code motions and other transformations applied during scheduling.

Note that, besides the transformations discussed in this chapter, the PHLS framework also applies several standard compiler passes such as copy and constant propagation and dead code elimination. These passes are applied both to the input description and after scheduling to remove any unnecessary or redundant code.

5.1 Common Sub-Expression Elimination

Common sub-expression elimination (CSE) is a well-known transformation that attempts to detect repeating sub-expressions in a piece of code, stores them in a variable and reuses the variable wherever the sub-expression occurs subsequently [ASU86]. This is demonstrated by the example in Figure 5.1(a). The common sub-expression $b + c$ in operations 2 and 3 can be replaced with the result of operation 1, resulting in the code in Figure 5.1(b).

Whether a common sub-expression between two operations can be eliminated depends on the control flow between the locations or basic blocks of the two operations. One common approach to capture the relationship between basic blocks in a control flow graph is using *dominator trees* [ASU86]. A definition for dominator trees has been given earlier in Definition 3.14. We repeat this definition here:

Definition 5.1. *A node n in a control flow graph (CFG) is said to* **dominate** *another node m, if every path from the initial node of the flow graph to m goes through n. The*

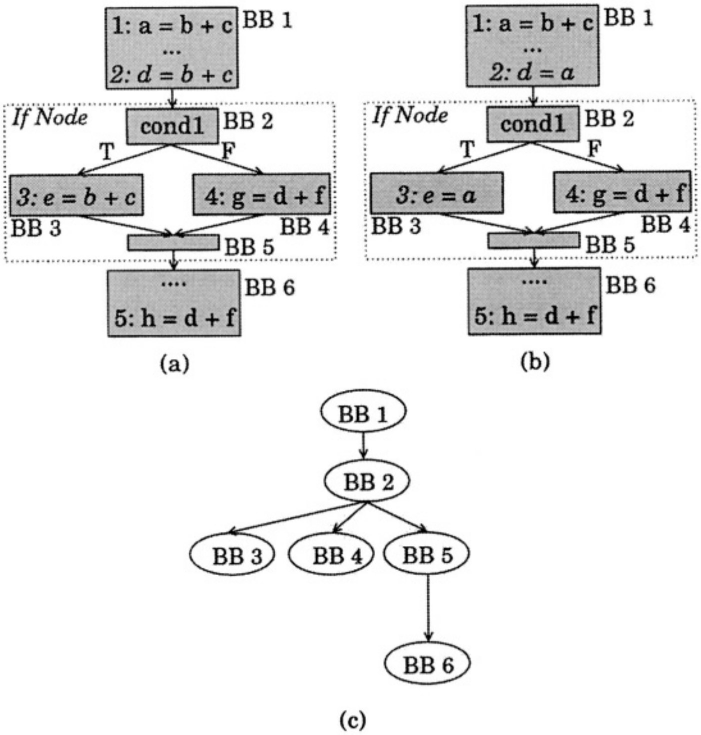


Figure 5.1: *CSE: (a) a sample HTG (b) the common sub-expression $b + c$ in operations 2 and 3 has been replaced with the variable a from operation 1. (c) Basic block dominator tree for this example. Operation 5 cannot be eliminated by the expression in operation 4, since bb_4 does not dominate bb_6*

dominator set *of node m,* **dom**(m)*, is formed by all the nodes that dominate m. By definition, every node in a CFG dominates itself, i.e., $m \in dom(m)$.*

The dominator tree for the example in Figure 5.1 (a) is given in Figure 5.1(c). In this example, basic block bb_2 dominates basic blocks bb_3, bb_4 and bb_5 and is itself dominated by bb_1. bb_5 in turn dominates bb_6. We can now define CSE in terms of dominator trees as follows:

Definition 5.2. *In order to preserve the control-flow semantics of a CFG, the common sub-expression in an operation op_2 can only be replaced with the result of another operation op_1, if op_1 resides in a basic block bb_1 that dominates the basic block bb_2 in which op_2 resides; i.e., $bb_1 \in dom(bb_2)$.*

So, in the example in Figure 5.1(a), operations 2 and 3 can be eliminated using the result of operation 1 as per the dominator tree shown in Figure 5.1(c). Conversely, bb_4 does *not* dominate bb_6 and hence, the common sub-expression in operation 5 cannot be replaced with the result of operation 4.

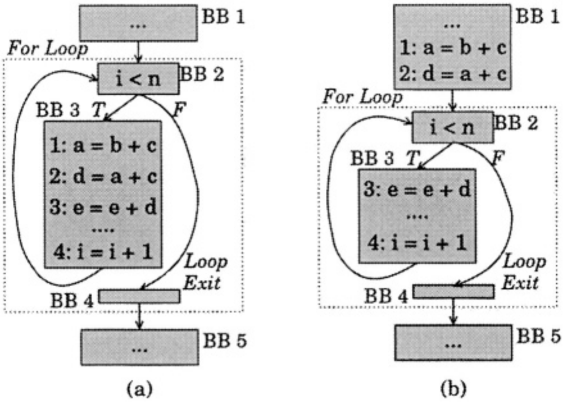


Figure 5.2: ***Loop-Invariant Code Motion: (a) The HTG of an example with a loop. The loop body (basic block $BB3$) has operations 1 through 4. (b) The loop-invariant operations op_1 and op_2 are moved outside the loop body.***

In this way, we use dominator trees while applying CSE to the design description. Note that, dominator trees have been extensively used previously for data flow analysis and transformations such as loop-invariant code motion and CSE [ASU86, SGL97].

Although transformations such as CSE were originally proposed as operation level or fine-grain transformations, recent work has shown that these optimizations are more effective when applied at the source level with a global view of the code structure [GMCG00].

5.2 Loop-Invariant Code Motion

Frequently, there exist computations within a loop body that produce the same results each time the loop is executed. These computations are known as *loop-invariant code* and can be moved outside the loop body, without changing the results of the code. In this way, these computations will execute only once *before* the loop, instead of for *each* iteration of the loop body. Consequently, this leads to better design performance, albeit only if the loop is executed at least once. Loop-invariant code motion is defined as follows.

Definition 5.3. *An operation op is said to be* loop-invariant *if: (a) its operands are constant, or (b) all operations that write to the operands of operation op are outside the loop, or (c) all the operations that write to the operands of the operation op are themselves loop invariant [ASU86, Muc97].*

We demonstrate loop-invariant code motion with the example in Figure 5.2. In the following discussion we refer to the operations by the line number shown next to them in the figure. In this example, operands b and c of operation op_1 are not written to by any operation within the loop. Hence, op_1 is loop-invariant. Similarly, the operand

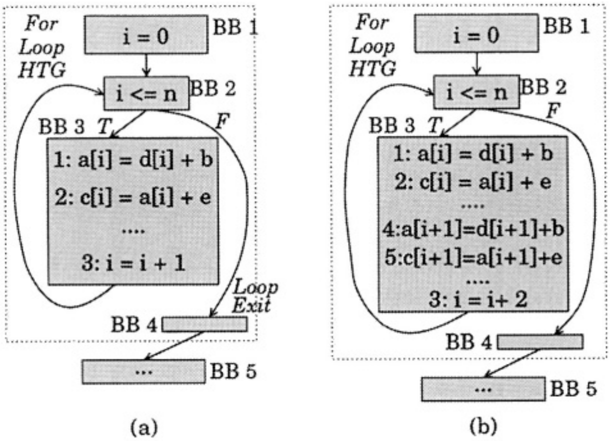


Figure 5.3: ***Loop Unrolling: (a) The HTG of an example with a loop and some operations. (b) The loop is unrolled once***

a of operation op_2 is written only by the loop-invariant operation op_1 and its other operand, *c*, is not written within the loop. Hence, op_2 is also loop-invariant. Thus, these operations can be moved out of the loop body into basic block bb_1 as shown in Figure 5.2(b). Operations op_3 and op_4 are not loop-invariant since one of their operands is written to, from within the loop.

When describing behaviors in high-level languages, designers frequently place several loop-invariant operations within loops for ease of understanding and readability of the code. Furthermore, loops themselves are used as a programming convenience and often do not expose all the available parallelism in the design. Thus, it is imperative to explore loop transformations such as loop unrolling that significantly alter the structure of the code and potentially expose parallelism across loop boundaries.

5.3 Loop Unrolling

Loop unrolling is the process of placing a duplicate of one or more iterations of the loop body at the end of the current loop body. The loop index variable increment (or decrement) is updated as necessary. Loop unrolling was developed to enable software compilers to perform optimizations across loop iterations and facilitate global code optimizations. However, loop unrolling can lead to code explosion. So loops are unrolled one iteration at a time, followed by code compaction by parallelizing transformations until no further improvements can be obtained. Loops are seldom unrolled fully.

On the other hand, in the context of hardware design descriptions, loops are only a programming convenience and latency constraints generally dictate the amount of unrolling a loop has to undergo. For instance, if a design is targeted to, say, three clock cycles, it implies that *all* the operations within *all* the iterations of the loop have to be executed in these three cycles. Hence, when this design is mapped to hardware, it will generate a design in which the loop is, in essence, unrolled within these three cycles. Some hardware architectures such as microprocessor functional blocks are low latency designs that must often be executed in just one cycle. Loops in these type of single cycle designs must be unrolled completely (see Chapter 11).

Loop unrolling is demonstrated in Figure 5.3. Figure 5.3(a) shows the HTG of a synthetic example, which has a loop and some operations within this loop. Operation op_1 uses the loop index variable i to read the array d and another operand, b, to generate the result $a[i]$. This result is used by operation op_2 to generate the result $c[i]$. When this loop is unrolled once, the resulting HTG is as shown in Figure 5.3(b). This unrolled loop exposes the inherent parallelism among the operations in the loop body – the operations op_1 and op_4 can be executed concurrently, followed by the concurrent execution of operations op_2 and op_5. Without loop unrolling, the two iterations of the loop body would have been executed sequentially.

In the *Spark* framework, the amount of loop unrolling for each loop is currently user-directed. The designer can experiment with different unrolls of the loop and determine the trade-offs. In our experiments with loop unrolling, we found that – as is the case with software – the code explosion caused by loop unrolling can be concern in hardware design as well. This is because the larger number of operations in the design after loop unrolling have to be mapped to the same number of resources as before. This leads to more complex interconnect (multiplexers) and associated control logic. The size of the FSM controller also increases since more states are required to execute the loop body (even though the number of iterations are fewer) [GDGN03c].

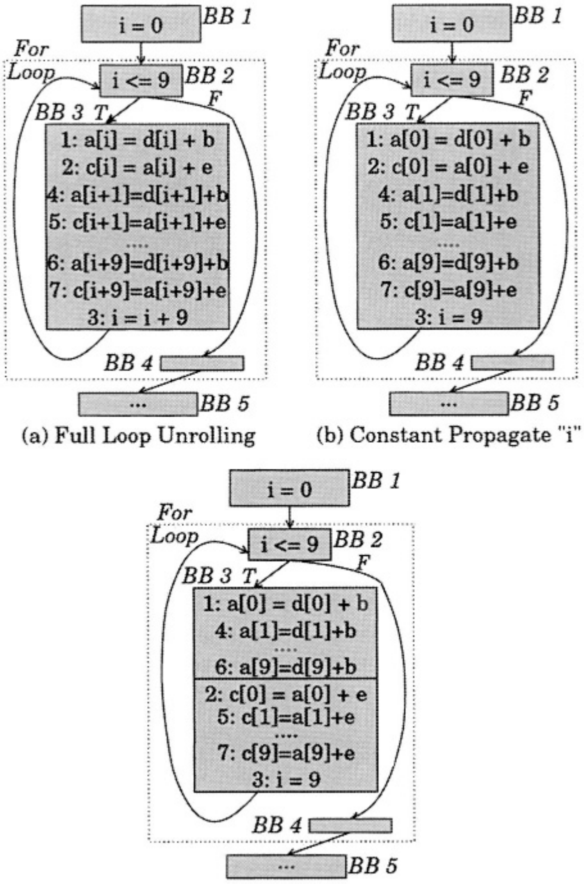
5.4 Loop Index Variable Elimination

We use the term *loop index variable elimination* to refer to copy propagation of the loop index variable after a loop is unrolled completely. We demonstrate this transformation using the previous example from Figure 5.3(a). Consider that the loop bound n is equal to 9 and that we unroll this loop completely; the resulting design is shown in Figure 5.4(a)[1]. The value of the loop index variable is now known statically in all the loop iterations. Hence, the initial value assigned to the loop index variable, $i = 0$, can now be propagated as a constant throughout all the iterations. The resultant design is shown in Figure 5.4(b).

In this way, the loop index variable is completely eliminated from the loop body. This removes the data dependencies that exist between the operations in the loop body and loop index variable, thus allowing the application of further code parallelizing transformations. In this example, the code motion transformations applied during the later scheduling phase can concurrently calculate all the values of the $a[]$ array followed by the concurrent calculation of all $c[]$ array values (assuming that the resources to do so are available), as shown in Figure 5.4(c).

This transformation is not new; indeed it is simply a combination of loop unrolling, constant propagation, and dead code elimination. However, we have shown in [GKK$^+$02] that this combination is an essential transformation for the synthesis of

[1] Note that, although not shown in this figure, the loop construct along with the loop check and the expression that updates the loop index variable can be removed after loop unrolling.



(a) Full Loop Unrolling

(b) Constant Propagate "i"

(c) Compact/Parallelize Loop Body

Figure 5.4: *(a) The loop in the example in Figure 5.3(a) is unrolled completely (loop bound $n = 9$). (b) Constant propagation of loop index variable, i. (c) Loop body compaction: array* a[] *values are calculated concurrently followed by concurrent calculation of the* c[] *array values. Note that, the loop control constructs can be removed after full loop unrolling.*

microprocessor functional blocks. In contrast, for designs from the multimedia and image processing domains, we found that even partial loop unrolling can lead to an explosion in the number of operations in the design. This in turn puts too much pressure on the control and steering logic in synthesized circuit (see experimental results presented in Chapter 9).

5.5 Summary

In this chapter, we presented several pre-synthesis transformations that are important to our parallelizing high-level synthesis methodology. We presented common subexpression elimination, loop invariant code motion, loop unrolling, and loop index variable elimination in Sections 5.1, 5.2, 5.3, and 5.4 respectively. We demonstrated these transformations using examples. The contribution of this chapter is a presentation of the pre-synthesis transformations that increase the scope for applying code transformations during the scheduling phase of high-level synthesis.

6

COMPILER AND SYNTHESIS TRANSFORMATIONS EMPLOYED DURING SCHEDULING

The ordering and placement of operations in high-level behavioral descriptions is usually governed by programming ease and varies from designer to designer. Very often this ordering is not conducive to or optimal for downstream high-level synthesis and optimization tasks [CGR93a]. This is particularly true of control-intensive designs due to the presence of nested conditionals and loops. An important aspect of our approach to high-level synthesis is the application of parallelizing transformations that move operations across conditionals and loops based on the time criticality of an operation and in the process expose the parallelism available in the algorithm.

To this end, we developed a set of speculative code motions to alleviate the effects of programming styles and constructs on the quality of synthesis results. These code motions enable the movement of operations through, across, and into conditionals with the objective of maximizing performance. However, this means that the heuristics that guide these code motions have to carefully manage the resource utilization across several basic blocks. This is especially true for hardware-expensive code motions such as conditional speculation that duplicate operations into the branches of a conditional block (see Section 6.3.4). Conditional speculation should only be employed when the resource utilization techniques are able to find idle or unused resources in multiple basic blocks in the conditional branches.

In this chapter, we discuss several of the parallelizing compiler transformations employed by our parallelizing high-level synthesis methodology during scheduling. We start off with a description of a set of speculative code motions and demonstrate how these code motions can enable new opportunities for applying compiler transformations such as CSE and copy propagation, *dynamically,* during scheduling. We then discuss a classical high-level synthesis transformation, namely, *operation chaining,* albeit modified to handle the control-intensive designs that our approach targets. Finally, we discuss an incremental loop pipelining technique called *loop shifting.* The algorithms that guide these transformations are presented in the next chapter.

Note that, in our work we distinguish between speculative code motions and techniques such as Trailblazing, Percolation scheduling, and Trace scheduling that employ these code motions during scheduling. Whereas we have already described the Trailblazing code motion technique in Chapter 4, in this chapter we focus on the speculative code motion transformations themselves.

6.1 Limits of Parallelism within Basic Blocks

Several classes of applications and particularly, multimedia and image processing applications are characterized by the presence of a considerable number of unpredictable branches. These control constructs limit the amount of instruction-level parallelism that can be exploited from the input description [Wal91, LW92]. Since the average number of operations within a basic block is typically 4 to 5 [TF70], there are usually not enough operations available for execution to utilize all the resources in each cycle (or control step). Hence, there are a number of "idle" resources in a basic block.

A resource is said to be *idle* in a control step if there is no operation scheduled to execute on that resource in that control step (the converse of an idle resource is a *busy* resource). These idle resources can be utilized by moving and scheduling operations from subsequent or preceding basic blocks. The candidate operations for these *"code motions"* are operations whose data dependencies are satisfied but the conditions under which they execute may not have been evaluated. There are two primary techniques that have been developed for compilers to circumvent such control dependencies: speculative execution and predicated execution.

6.2 Speculation and Predicated Execution in Compilers

Speculative execution, popularly known as speculation, refers to the execution of an operation before the branch condition that controls it has been evaluated. If the condition under which the operation was to execute evaluates to false, then compensation code may have to be executed. This is often referred to as *control speculation.* In contrast, in *data speculation,* an operation is executed before an operation that it is dependent on has been executed; thus, the speculated operation may potentially use incorrect operand values. For example, a load may be executed before a preceding store that may write to the same location. This type of speculation typically targets long latency operations (or instructions) whose execution can be overlapped with the execution of other instructions. In general, speculation is useful either when there are insufficient operations in a basic block to keep the functional units busy or to execute operations on the critical path on a priority basis.

In this work we focus on control speculation as employed by a compiler. There are also hardware-assisted approaches that speculatively execute operations dynamically at runtime using architectural support such as branch predictors in superscalar processors [SLH90, MBVS97].

Predicated execution [AKPW83, HD86, RYYT89, DT93, Man00] is a technique that eliminates branch instructions and converts control dependencies into data dependencies. Predicated execution refers to the conditional execution of an instruction based on a Boolean operand called a guarding predicate. The instruction either executes normally or is nullified based on whether the predicate evaluates to true or false. This technique requires architectural support in the form of an additional Boolean operand guarding each operation, comparator units to compute the predicates and the ability to nullify the result of an instruction if its predicate evaluates to false.

Predicated execution is often employed to eliminate several branches completely by merging them into one large block of straight line code with predicates. This is known as *if-conversion* [AKPW83, DHB89, PS91, MLC$^+$92] and is employed by compilers to boost the opportunities for instruction-level parallelism. Whereas speculation is generally employed to execute operations on control paths that have a higher probability of being taken, predicated execution of entire conditional blocks is usually done when the branch probabilities are unknown or are equal for both branches.

6.3 Role of Speculative Code Motions in High-Level Synthesis

Speculation is generally more useful than predicated execution for high-level synthesis. This is because an important notion in hardware synthesis is that of mutual exclusivity of operations. Two operations are said to be *mutually exclusive* if they execute under complementary conditions. High-level synthesis schedulers can take advantage of this property and schedule two mutually exclusive operations in the same cycle on the same resource. The operation that actually executes is based on the evaluation of the conditions. This is known as *resource sharing.* We demonstrate this with an example in the next section.

Thus, in the context of high-level synthesis, if we if-convert the branches of a conditional block and execute them with predicates, we are discarding the mutual exclusivity information about the operations. In effect, this leads to a longer schedule length, since both the branches of the conditional execute. Furthermore, predicated execution is employed by compilers for processors that have an abundance of resources (functional units) to execute the predicated instructions. On the other hand, the architectures targeted by high-level synthesis are usually tightly resource-constrained. Thus, there are generally only a few resources idle in each cycle. The high-level synthesis scheduler thus attempts to find and speculatively execute an operation from a successor (or predecessor) basic block.

Over the next four sections, we explore a set of speculative code motions that are useful for high-level synthesis. These code motion transformations re-order, speculate and sometimes even duplicate operations in the design to achieve maximum parallelism and shorter schedule lengths. This code restructuring also reduces the impact of programming style on the quality of synthesis results. Hence, after applying the code motion transformations, the operation placement in the code is optimized for improved synthesis results. In contrast, designers place operations based on programming ease.

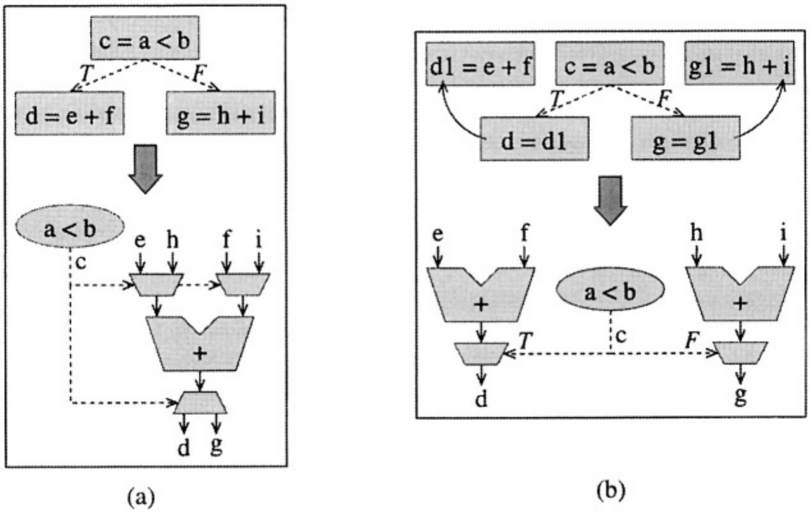


Figure 6.1: ***Extracting the inherent parallelism in a design by speculatively executing the addition operations. This requires an additional resource, but leads to a reduction in the longest path.***

We begin by studying speculation. Although the trade-offs of speculation have been explored at length in the compiler domain, its utility and impact on the overall high-level synthesis results – particularly, the hardware overheads – is not well-understood. We explore these issues in the next section.

6.3.1 Speculation in High-Level Synthesis

The notion of speculation that we adopt for high-level synthesis is similar to that of a compiler's. In high-level synthesis, when an operation is executed speculatively, we store its result in a new memory location. In place of the original operation, we leave a copy operation so that if the condition that the operation was to execute under evaluates to true, then the stored result is committed to the variable from the original operation; else the stored result is discarded. Hence, no compensation code has to be added or executed if the condition evaluates to false.

We demonstrate speculation by an example in Figure 6.1. In Figure 6.1 (a), variables d and g are calculated based on the result of the calculation of the conditional c. Since the operations that produce d and g are executed on different branches of a conditional block, these two operations are *mutually exclusive*. Hence, they can be scheduled on the same hardware resource with appropriate multiplexing of the inputs and outputs as shown in the circuit in Figure 6.1(a).

Now, consider that an additional adder is available. Then the operations within the conditional branches can be executed *speculatively* and concurrently with the calculation of the conditional c as shown in Figure 6.1(b). The corresponding hardware circuit is also shown in this figure. Based on the evaluation of the conditional, one of

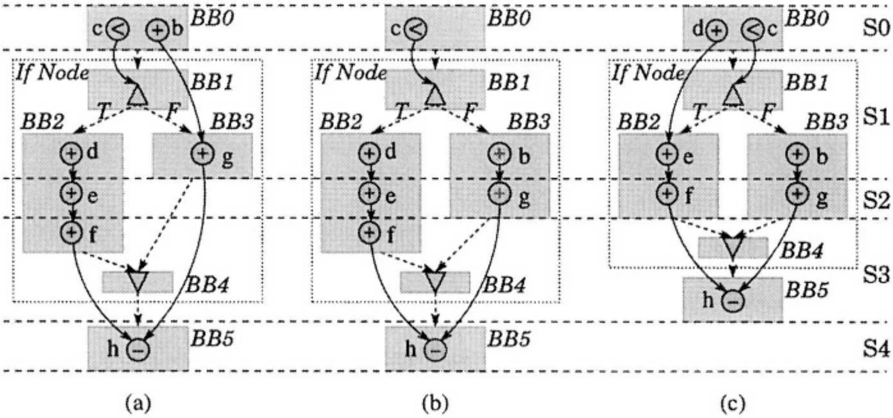


Figure 6.2: ***Reverse Speculation: (a) An example design (b) Operation b is reverse speculated into the branch of the conditional that uses its result, i.e., the* false *branch. (c) Operation d is now speculated into basic block bb_0; the schedule length thus is reduced by one cycle.***

the results will be discarded and the other committed. As shown by the corresponding hardware circuits in Figures 6.1 (a) and (b) that as a result of this speculation, the longest path gets shortened from being a sequential chain of a comparison followed by an addition to being a parallel computation of the comparison and the additions.

This example also demonstrates the additional costs of speculation. Speculation requires more functional units and potentially more storage for the intermediate results. **So, uncontrolled aggressive speculation can lead to worse results due to the extra multiplexing and control overheads.** On the other hand, idle resources can be better utilized by executing operations speculatively on them.

Besides moving operations out of conditionals and executing them speculatively, we find that in high-level synthesis it is sometimes useful to move operations into conditionals as explained next.

6.3.2 Reverse Speculation

Reverse speculation refers to downward motion and *duplication* of an operation at a conditional fork node into the subsequent conditional branches. This code motion is useful in instances where an operation inside the conditional branch is on the longest (critical) path through the design, whereas an operation before the conditional is not. The operation before the conditional branch can then be duplicated down or *reverse speculated* into both the conditional branches, so that the resource made idle by this move can be better utilized by the operation on the longest path. Reverse speculation has been variously referred to as *lazy* code motion or execution and *duplicating down* in past literature [RFJ95, Muc97].

Reverse speculation is demonstrated by an example in Figure 6.2(a). In this design, the two longest data dependency paths are: $< d, e, f, h >$ and $< b, g, h >$ (solid

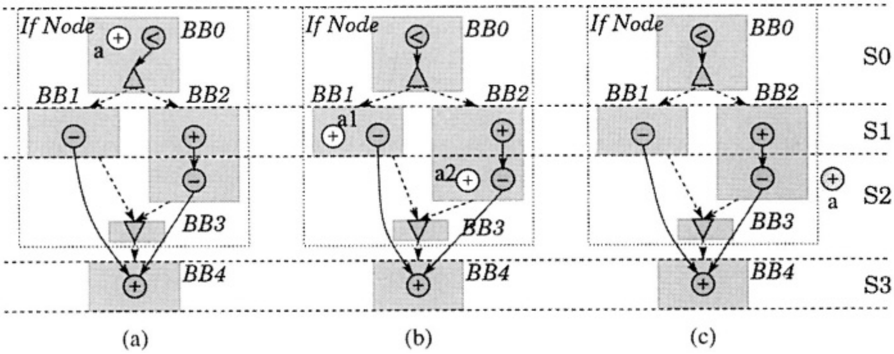


Figure 6.3: ***(a) Example HTG representation (b) Operation a is reverse speculated by duplicating as operations $a1$ and $a2$ (c) Downward code motion of operation a without duplication; this is an illegal move in our system.***

lines in this HTG representation denote data dependencies). Of these, the first one $< d, e, f, h >$ is the longest path through the design. However, the operation placement in this design is such that operation b that is on the shorter dependency path is placed outside the conditional whereas operation d that is on the longer dependency path is placed in the *true* branch of the conditional. Hence, operation b can be *reverse speculated* or moved into the conditional branches as shown in Figure 6.2(b). This means that the adder in basic block bb_0 is now idle. We can thus speculatively execute operation d in basic block bb_0 as shown in Figure 6.2(c).

The dashed lines in Figure 6.2(a), (b) and (c) demarcate the state assignments (50 through *S4)* for the three designs. Clearly, the final design in Figure 6.2(c) after reverse speculation of operation b and speculation of operation d requires one state or cycle less than the original design in Figure 6.2(a).

Note that, as shown in Figure 6.2(b), the reverse speculation algorithm detects that the result of operation b is used only in *the false* branch of the conditional and hence, moves it only into that branch. In this case, reverse speculation behaves in the exact opposite manner of speculation. However, in the general case, reverse speculation duplicates the operation into both the *true and false* conditional branches.

There is an important distinction between the reverse speculation code motion that duplicates an operation into the branches of a conditional versus just scheduling the operation in parallel to the branches of the conditional block. Duplicating operations at a fork node means that these operations can be scheduled in different control steps or states within the basic blocks in the conditional branches, independent of each other. Hence, the scheduler has greater flexibility in scheduling the duplicated operations in mutually exclusive basic blocks.

This distinction can be understood by the example in Figure 6.3(a). In this example, when operation a is reverse speculated, it is duplicated as operations $a1$ and $a2$ in basic blocks bb_1 and bb_2 respectively, as shown in Figure 6.3(b); operations $a1$ and $a2$ are scheduled in different states in the their respective basic blocks. This is unlike a downward code motion in which operation a is not duplicated, but instead is scheduled

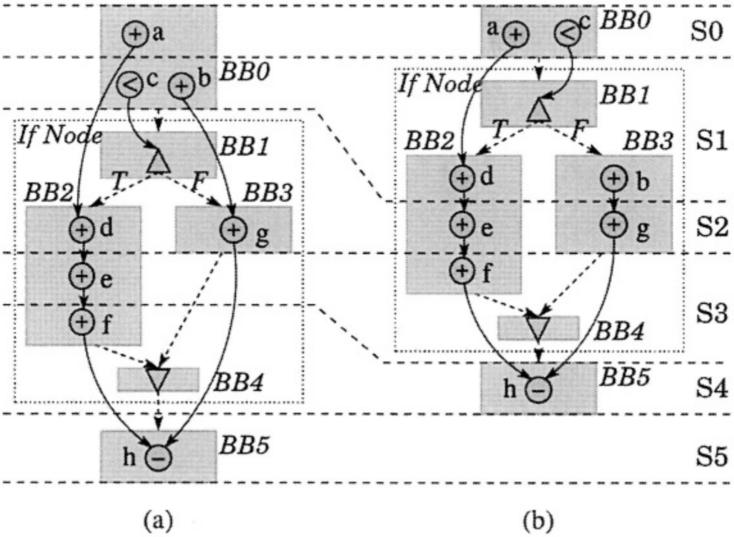


Figure 6.4: ***Early condition execution: (a) original design (b) comparison operation c is scheduled as soon as possible to enable early condition evaluation. The unscheduled operation b before the conditional check is reverse speculated into the conditional branches.***

to execute in, say, state *S2,* as shown in Figure 6.3(c). This type of downward code motion past a fork node without duplication is *not allowed* in our PHLS methodology.

6.3.3 Early Condition Execution

Early condition execution is a novel transformation that restructures the original code, so as to evaluate conditional checks as soon as possible. This in effect means that the conditional check is "moved up" in the design, and hence, all unscheduled operations before the conditional are reverse speculated into the conditional. This transformation is motivated by the fact that evaluating a conditional check early resolves the control dependency for operations within conditional branches. These operations are thus available for scheduling sooner.

Early condition execution is demonstrated by the example in Figure 6.4(a). In this example, comparison operation c computes a conditional that is checked in basic block bb_1 (the Boolean conditional check is denoted by a triangle). We can schedule this comparison operation concurrently with operation a in state *S*0 in basic block bb_0, as shown in Figure 6.4(b). Now the conditional check in basic block bb_1 can be executed "early" and the unscheduled operations before this check can be reverse speculated. For this example, operation b is reverse speculated into basic block bb_3 (and not into bb_2 since its result is used only in bb_3). These code motions lead to an overall shorter schedule length, as shown by the state assignments in Figures 6.4 (a) and (b).

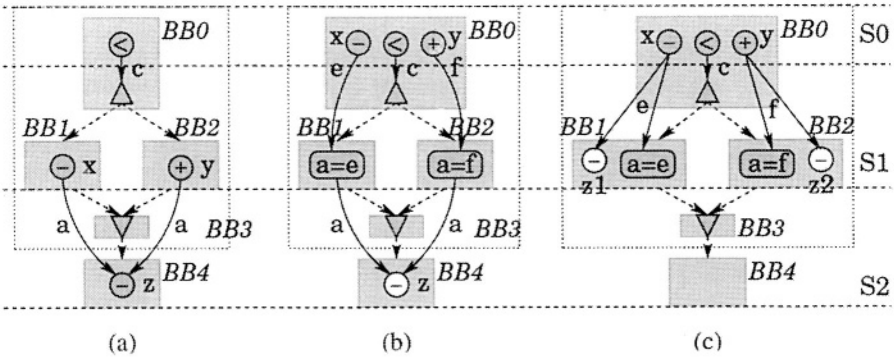


Figure 6.5: *Conditional Speculation: (a) HTG representation of an example. (b) Operations x and y are speculated leaving resources idle in the conditional branches. (c) Operation z is conditionally speculated into conditionals bb_1 and bb_2.*

6.3.4 Conditional Speculation

We have shown how speculation can be used to schedule operations on idle resources in basic blocks before conditionals. However, design descriptions often have instances where the control steps in the basic blocks of a conditional have idle resources. Speculation also leaves resources idle in the conditional branches. We utilize these idle resources by *duplicating* operations that lie in basic blocks after the conditional branches into both the branches of the conditional. We call this code motion *conditional speculation.* This is similar to the duplication-up code motion used in compilers and the node duplication transformation discussed by Wakabayashi et al. [WT92].

Let us understand conditional speculation with the example in Figure 6.5(a). In this example, operations x and y both write to the variable a in the conditional branches bb_1 and bb_2. Consider that we allocate one adder, one subtracter and one comparator to schedule this design. Then, operations x and y can be speculatively executed in basic block bb_0, as shown in Figure 6.5(b). The results of the speculated operations are written into two new destination variables: e and f. These variables are committed (or written back) to the variable a only within the conditional branches, after the conditional check has been evaluated.

Figure 6.5 (b) demonstrates that the speculation of these operations leaves all the resources (adder, subtracter and comparator) idle in basic blocks bb_1 and bb_2. This allows us to *conditionally speculate* operation z – that lies in the basic block after the conditional branches – and schedule its duplicates $z1$ and $z2$ on the idle subtracter in both the branches of the conditional, as illustrated in Figure 6.5(c). Also, operation z is dependent on either the result of operation x or operation y depending on how the condition evaluates (i.e., operation z is dependent on the variable a). Hence, the duplicated operations, $z1$ and $z2$, directly read the results of operations x and y respectively. We also show the state assignments ($S0$, $S1$ and $S2$) for the three designs in Figures 6.5 (a), (b) and (c) by dashed lines. Clearly, for this example, this set of code motions leads to a design that requires one less state (or cycle) to execute.

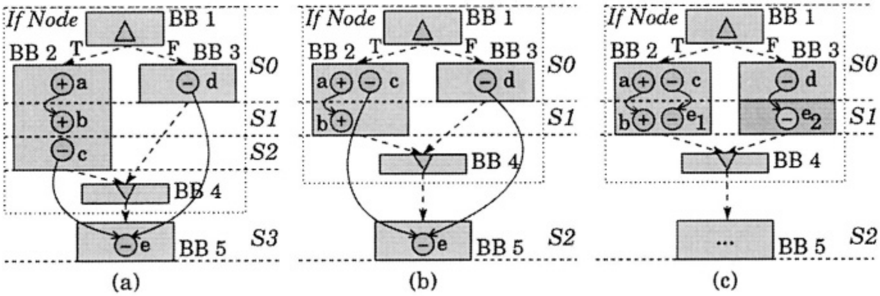


Figure 6.6: ***Dynamic branch balancing: (a) HTG representation of an example, (b) After scheduling basic block bb_2, (c) Insertion of a new scheduling step in basic block bb_3 enables conditional speculation of operation e.***

Note that, condition speculation does not necessarily need speculation to be performed first to activate it as shown in the example above. As stated earlier, there are often idle resources within conditional branches that go unused unless operations are conditionally speculated from after the conditional branches.

6.4 Enabling New Code Motions by Dynamic Branch Balancing

Often design descriptions are written so that one conditional branch in an if-then-else HTG node has fewer scheduling steps than the other. We call this an If-HTG with *unbalanced* conditional branches. Consider the input description shown in Figure 6.6(a). One possible scheduled design (with a resource allocation of an adder and a subtracter) is as shown in Figure 6.6(b): operation a and c execute concurrently in state *S*0 in basic block bb_2. The state assignments (*S*0, *S*1, and so on) are demarcated by dashed lines in these figures. We can see from Figure 6.6(b) that after scheduling this example, the false branch (bb_3) of the If-HTG node has fewer scheduling steps than the true branch (bb_2). Thus, "IfNode" in Figure 6.6(b) is an If-HTG with unbalanced conditional branches.

In such unbalanced If-HTGs, it is possible to insert a new scheduling step in the branch with fewer scheduling steps, without increasing the length of the longest path through the If-HTG. We call this *dynamic branch balancing* (dynamic because it is employed as and when needed during scheduling). Hence, in the scheduled design in Figure 6.6(b), we can insert a new scheduling step in basic block bb_3 since bb_2 has more scheduling steps than bb_3. This new step and the presence of a scheduling step in bb_2 with an idle subtracter enables the conditional speculation of operation "e", as operations "e_1" and "e_2" in basic blocks bb_2 and bb_3 respectively. The resulting design is shown in Figure 6.6(c).

The design in Figure 6.6(c) requires one state less to execute than the scheduled design in Figure 6.6(b). Thus, branch balancing can introduce new opportunities for applying conditional speculation and thus, further compaction of the design schedule.

Also, since the longest path through the If-HTG is unaltered, this technique does not lead to an increase in the longest path length through the design. Note that, if profiling information is available, we can instead insert scheduling steps into basic blocks in branches that are *less likely* to be taken.

Our PHLS methodology employs branch balancing during two stages of the scheduler [GDGN03b]:

1. *Branch Balancing during Design Traversal* (BBDDT): In this technique, scheduling steps are inserted to balance the branches of unbalanced conditional blocks as they are encountered while traversing the design during scheduling. This is detailed later in Section 7.6.1 in Chapter 7.

2. *Branch Balancing during Code Motions* (BBDCM): This technique inserts new scheduling steps in unbalanced conditional blocks if this enables a code motion (specifically conditional speculation) required to move the candidate operation during scheduling (see Section 7.4.2). Note that, this means that during the candidate validater task, we validate operations that can be moved if branch balancing is employed (see Section 7.4.1).

6.5 Dynamic Common Sub-Expression Elimination

The code restructuring and operation duplication done by the speculative code motions can create new opportunities for applying transformations such as common subexpression elimination (CSE) and copy propagation. We demonstrate this with the help of the example in Figure 6.7(a). In this example, classical CSE cannot eliminate the common sub-expression in operation 4 with operation 2, since operation 4's basic block bb_6 is not dominated by operation 2's basic block bb_3 (see Section 5.1 for a definition of dominator trees). Consider now that the scheduling heuristic decides to schedule operation 2 in bb_1 and execute it speculatively as operation 5, as shown in Figure 6.7(b). Now, basic block bb_1, containing this speculated operation 5, dominates operation 4's basic block bb_6. Hence, operation 4 in Figure 6.7(b) can be eliminated and replaced by the result of operation 5, as shown in Figure 6.7(c).

Since CSE is traditionally applied as a pass, usually before scheduling, it can miss these new opportunities created *during* scheduling. This motivated us to develop a technique by which CSE can be applied *dynamically* while the design is being scheduled. *Dynamic CSE* is a technique that operates after an operation has been moved and scheduled on a new basic block [GRS$^+$02]. It examines the list of remaining ready-to-be-scheduled operations and determines which of these have a common sub-expression with the currently scheduled operation. This common sub-expression can *now* be eliminated if the new basic block containing the newly scheduled operation dominates the basic block of the operation with the common sub-expression. We use the term "dynamic" to indicate that CSE is applied *during* scheduling versus the phase ordered application of CSE *before* scheduling.

We can also see from the example in Figure 6.7 that applying CSE as a pass *after* scheduling is ineffective compared to dynamic CSE. This is because the resource freed

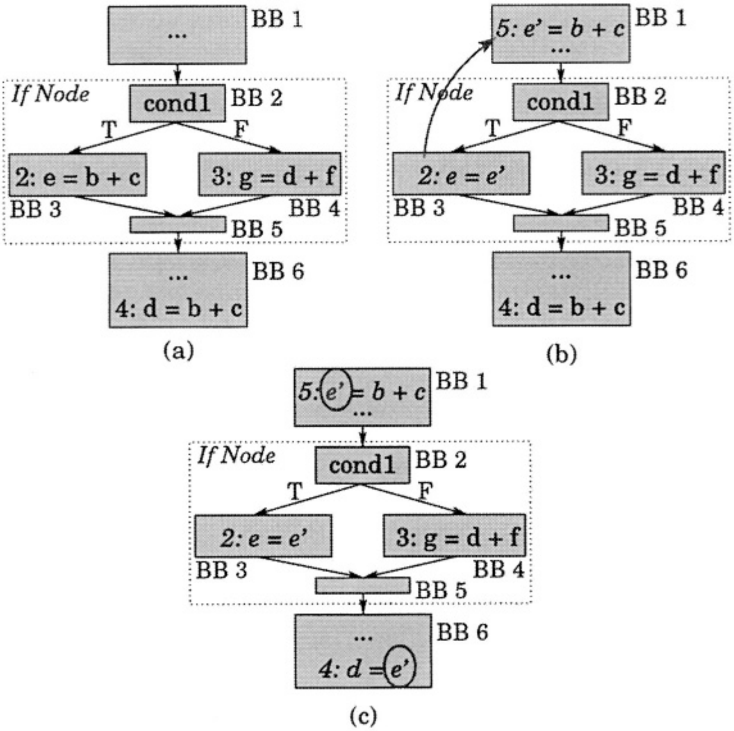


Figure 6.7: ***Dynamic CSE: (a) A hierarchical task graph of an example, (b) Speculative execution of operation** 2 **as operation** 5 **in** bb_1**, (c) This allows dynamic CSE to replace the common sub-expression in operation** 4.*

up by eliminating operation 4, can potentially be used by the scheduler to schedule another operation in basic block bb_6. Performing CSE after scheduling is too late to effect any decisions by the scheduler.

6.5.1 Conditional Speculation and Dynamic CSE

We find that conditional speculation frequently enables new opportunities to apply dynamic CSE. Consider the example in Figure 6.8(a). In this example, operation 2 in bb_9 has a common sub-expression, $b + c$, with operation 1 in bb_6. But since bb_9, is not dominated by bb_6, this common sub-expression cannot be eliminated by classical CSE. Consider that the scheduling heuristic decides to conditionally speculate operation 1 into the branches of the if-then-else conditional block, $IfNode_1$. In the resulting design, shown in Figure 6.8(b), the operation is duplicated up as operations 3 and 4 in basic blocks bb_2 and bb_3 respectively. Thus after conditional speculation, operations with the common sub-expression $b + c$ exist in all control paths leading up to bb_9.

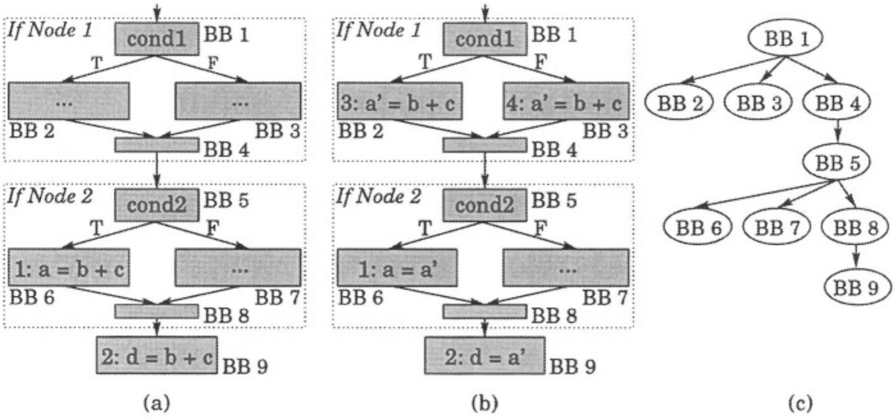


Figure 6.8: ***Dynamic CSE after conditional speculation: (a) A sample HTG (b) Operation** 1 **has been conditionally speculated into** bb_2 **and** bb_3**. This allows dynamic CSE to be performed for operation** 2 **in** bb_9**. (b) Dominator tree for this example***

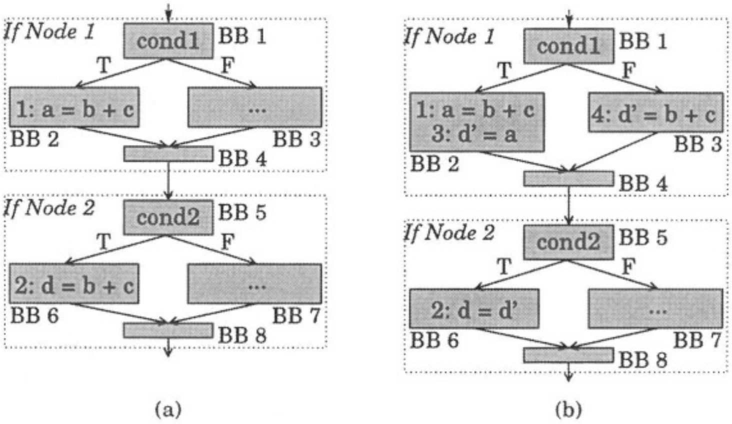


Figure 6.9: ***(a) Another example HTG (b) Operation** 2 **has been conditionally speculated into** bb_2 **and** bb_3**. The common sub-expression** $b + c$ **in the duplicated operation** 3 **is eliminated due to the presence of operation** 1 **in basic block** bb_2**.***

Hence, we can now apply dynamic CSE and operation 2 can use the result a' of operations 3 and 4, as shown in Figure 6.8(b).

This leads to the notion of dominance by sets or groups of basic blocks [SGL96], as defined below (also given earlier in Definition 3.16):

Definition 6.1. *A set of basic blocks $\mathcal{BB}$ in a control flow graph G_{CFG} is said to dominate another basic block bb if every path from the initial node bb_0 of the control flow graph G_{CFG} to bb goes through at least one of the basic blocks in the set $\mathcal{BB}$.*

By this definition, in Figure 6.8(b), basic blocks bb_2 and bb_3 together dominate basic block bb_9, hence enabling dynamic CSE of operation 2. In this manner, we use this property of domination by sets of basic blocks while performing dynamic CSE along with code motions such as reverse and conditional speculation that duplicate operations in the design graph.

Another case in which dynamic CSE is applied in conjunction with conditional speculation, arises when an operation is duplicated into a basic block in which another operation with the same expression already exists. In this case, the operation being duplicated is instead replaced with a copy operation using the result of the already present operation with the same expression. This is illustrated by the example in Figure 6.9(a). Consider that, in this example operation 2 is conditionally speculated as operations 3 and 4 in basic blocks bb_2 and bb_3. Then, the sub-expression $b + c$ that is common between operations 1 and the duplicated operation 3 can be eliminated in operation 3, as shown in Figure 6.9(b).

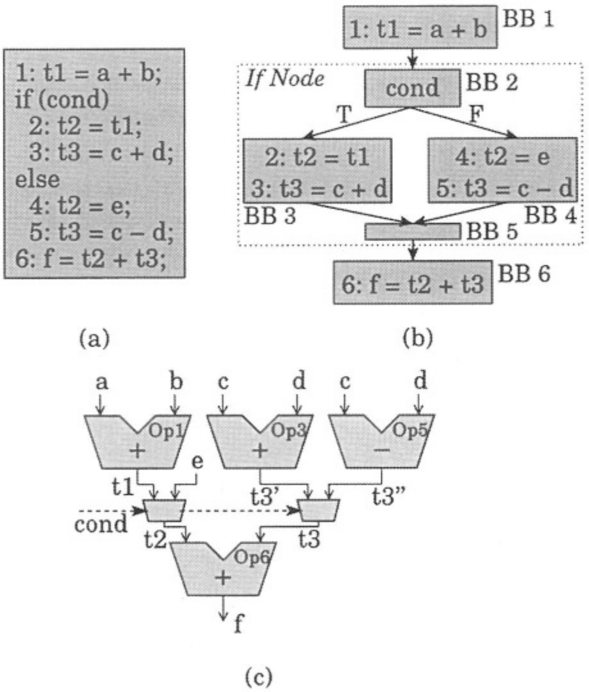
6.5.2 Dynamic Copy Propagation

The concept of dynamic CSE can also be applied to *copy propagation.* After applying code motions such as speculation and transformations such as CSE, there are usually several copy operations left behind. *Copy operations* are operations that read the result of one variable and write them to another variable. For example in Figure 6.9(b), the copy operation 2 in basic block bb_6 copies variable d' to variable d.

Copy propagation is a compiler pass that replaces the variable written by a copy operation op_{copy} by the variable read by op_{copy} in all the operations that have flow dependencies with op_{copy}. So, in the example in Figure 6.9(b), copy propagation would propagate the copy operation $d = d'$ forward by replacing d with the variable d' in the operations that have flow dependencies with this copy operation.

Again, traditionally copy propagation is done as a compiler pass before and after scheduling to eliminate unnecessary copies and use of variables. However, we found that it is essential to propagate the copies created by speculative code motions and dynamic CSE during scheduling itself, since this enables opportunities to apply dynamic CSE on subsequent operations that read these variable copies. After copy propagation, these dependent operations can directly use the result of the operation that creates the variable in the first place, rather than the result of the copy operation. A dead code elimination pass after scheduling can then remove unused copies.

In this way, the scheduling heuristic can dynamically employ compiler transformations such as speculative code motions, dynamic CSE and dynamic copy propagation to increase resource utilization and improve the quality of synthesis results. These transformations form an integral part of the scheduling strategy employed by the *Spark* framework. Besides these compiler transformations, high-level synthesis schedulers also employ several high-level synthesis transformations. Of these, one commonly used transformation is operation chaining. This is discussed in the next section.



```
1: t1 = a + b;
if (cond)
  2: t2 = t1;
  3: t3 = c + d;
else
  4: t2 = e;
  5: t3 = c – d;
6: f = t2 + t3;
```

Figure 6.10: ***An example of operation chaining across conditional boundaries: (a) sample "C" code, (b) its HTG representation, (c) corresponding hardware, with functional units connected via steering logic.***

6.6 Chaining Operations Across Conditional Boundaries

Operation chaining is an important technique that is supported by most high-level synthesis tools [GDWL92]. *Chaining* of operations means that the result of one operation is used immediately by another operation without storing it in an intermediary latch or register. In the corresponding hardware, the functional units, on to which the operations are mapped, have to be connected to each other without any memory elements in between.

We have extended this classical high-level synthesis technique to chain operations *across conditional boundaries.* Chaining operations across conditional boundaries (or basic blocks) is required for the type of designs targeted by our approach, i.e., designs that contain a mix of control and data operations. In hardware, this type of operation chaining leads to functional units that are connected via often complex, steering logic such as multiplexers.

Let us understand this with the aid of an example. Consider the sample fragment of "C" code in Figure 6.10(a) and the corresponding HTG representation in Figure 6.10(b). Consider also that this design description has to be scheduled in one cycle. To achieve this, all the operations in the description have to be chained together, across

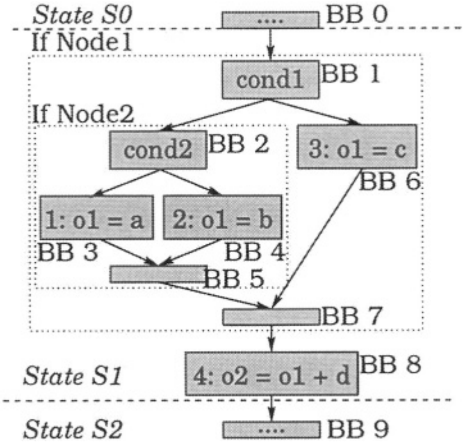


Figure 6.11: ***Operation* 4 *is scheduled in the same cycle as operations* 1, 2 *and* 3. *Hence, we have to check that chaining is possible on all* chaining trails *up from* bb_8.**

the if-then-else conditional block. One possible hardware implementation for this is shown in Figure 6.10(c). The operations Op_1 to Op_6 correspond to the line numbers in Figure 6.10(a). In the circuit in Figure 6.10(c), the inputs to the operation Op_6 are obtained by multiplexing the outputs of the Op_1, Op_3 and Op_5, based on the condition *cond*. Variables $t3'$ and $t3''$ are the temporary results of operations Op_3 and Op_5, that are immediately multiplexed to produce the result $t3$. Since all the operations in this fragment of code are chained together, none of the variables, $t1$, $t2$ and $t3$, have to be stored in registers. We refer to variables that are not stored in registers as *wire-variables*: this is discussed in detail in Section 6.6.2.

Hence, chaining operations across conditional boundaries has two effects on the scheduling strategy: firstly, the scheduling heuristic has to keep track of the resource utilization of multiple scheduling steps in several basic blocks that are chained into the same clock cycle. Thus, the scheduler has to use a modified resource utilization and operation scheduling model that looks across the conditional boundaries. This modified resource utilization model has already been presented in Section 3.6.4 in Chapter 3. Secondly, chaining an operation with operations that are in the branches of a conditional check requires a detailed analysis of the control flow paths in which the chained operations are, as discussed in the next section.

6.6.1 Chaining with Operations in the Branches of a Conditional Block

To be eligible for chaining across a conditional boundary, an operation has to satisfy two main criteria: (a) a resource on which the operation can execute should be idle in all the scheduling steps chained together, including the current scheduling step, and (b) if there are operations in the steps being chained together that the current operation being scheduled has dependencies with, then the total execution time of these chain of operations should be less than the clock period of the design. Recall that a scheduling step represents an aggregation of operations that execute concurrently in the same cycle within a basic block (see Section 3.2.5).

We explain these two criteria using the example in Figure 6.11. W want to schedule operation 4 in the same cycle as operation 1. First, the chaining heuristic determines all the basic blocks that have scheduling steps scheduled in the same cycle as the current step under consideration. The heuristic does this by traversing all the paths or *chaining trails* back wards from the basic block that operation 4 is in (bb_8), looking for scheduling steps scheduled in the same cycle. In this example, there are three trails comprising the basic blocks:
Trail 1: $<bb_8, bb_7, bb_5, bb_3, bb_2, bb_1>$
Trail 2: $<bb_8, bb_7, bb_5, bb_4, bb_2, bb_1>$
Trail 3: $<bb_8, bb_7, bb_6, bb_1>$.

Note that, for the rest of the discussion of this example, since each basic block has just one scheduling step, we will use the basic block name to refer to the corresponding scheduling step in it.

The first criteria is satisfied in this case since none of the operations in the steps being chained together uses an adder (we assume that at least one adder has been allocated to schedule this design). For the second criteria, we determine the dependency chain of operation 4 in each trail. The operations that will be chained with operation 4 in the trails are operations 1, 2 and 3 respectively, each of which writes to the variable $o1$. We determine that operation 4 can be executed in the same cycle as these operations by using the appropriate value of $o1$, depending on the evaluation of the condition. However, to actually chain these operations together, the chaining algorithm along with the code motion algorithm has to ensure that the correct hardware corresponding to the chained operations is generated to implement the schedule. This is discussed in the next section.

6.6.2 Creating Wire-Variables to enable Chaining on each Chaining Trail

To enable chaining, results from one operation must be read by another operation in the same cycle. In hardware, this corresponds to a connection between the two functional units on which the operations execute. In the intermediate graph representation, we represent operation chaining using *wire-variables.* These are variables that can be written to and read in the same cycle. Thus, wire-variables are explicitly marked as being wires and are not mapped to registers. Note that, we initially assume that each variable in the input behavioral description may potentially be mapped to a register. Thus, the variables that are not marked as wire-variables can only be read one cycle after they are written to.

Consider an operation Op_1 that writes a result, r_1 and another operation Op_2 that reads this result:

$$\boxed{\begin{array}{l} r_1 = Op_1(arguments) \\ r_2 = Op_2(r_1) \end{array}}$$

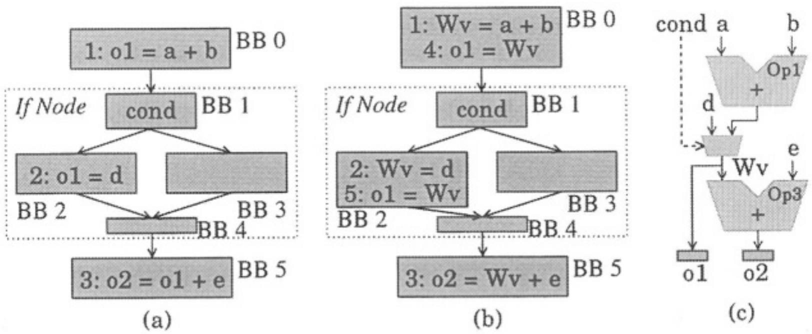


Figure 6.12: *(a) HTG of an example, (b) operation 3 is chained with operations. 1 and 2; so, wire-variable Wv and copy operations 4 and 5 are inserted (c) corresponding hardware; Wv becomes a wire and $o1$ a register.*

To chain operations Op_1 and Op_2, the code has to be modified to:

$$\boxed{\begin{array}{l} wireVar = Op_1(arguments) \\ r_2 = Op_2(wireVar) \\ r_1 = wireVar \end{array}}$$

where variable $wireVar$ is marked as being a wire and r_1 is (potentially) mapped to a register. In the RTL VHDL generated after synthesis, r_1 is mapped to a VHDL signal and $wireVar$ is mapped to a VHDL variable.

Often, as was the case in the example in Figure 6.11, several operations in different basic blocks may write to a variable. When operations are chained across conditional checks, we have to insert operations that write to "wire-variables" in all the trails that lead back from the chained operation, i.e., in all the branches of the preceding conditional blocks. We explain this using the example in Figure 6.12(a). In this example, operations 1 and 2 write to variable $o1$ in basic blocks bb_0 and bb_2 respectively. Operation 3 in basic block bb_5 reads the value of $o1$ and writes to variable $o2$. Consider that the scheduling algorithm schedules the entire fragment of code in this figure within one clock cycle. Then, to enable operation chaining, a wire-variable Wv is introduced and the copy operations 4 and 5 are inserted, as shown in Figure 6.12(b). In the resulting hardware, shown in Figure 6.12(c), variable Wv becomes a wire and the variables $o1$ and $o2$ are bound to registers. Operation 3 uses the multiplexed result of both the operations that write to wire-variable Wv. Note that, the copy operation 4 could also have been inserted into basic block bb_3, leading to the same hardware.

Similarly consider the fragment of code in the Figure 6.13(a). In this example, variable $o1$ is written to only in the true branch of a conditional block and is read by operation 2 in basic block bb_5. This code implies that if the condition evaluates to "false", then a value of $o1$ from a previous write (not shown here) will be used by operation 2. In order to chain the operations in this code, a variable copy to wirea-variable Wv has to be inserted in both branches of the conditional block, as shown in Figure 6.13(b). So, the operation 2 now reads the variable Wv instead. Again, in hardware, variable Wv will be mapped to a wire and variable $o1$ to a register.

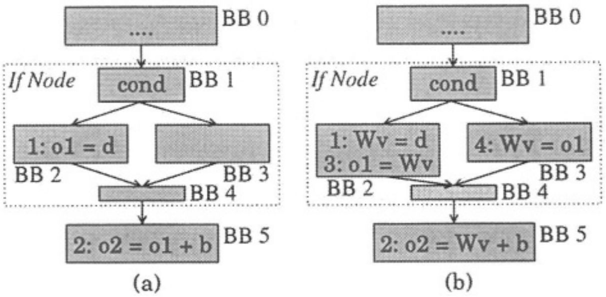


Figure 6.13: ***(a) HTG of another example (b) Wire-variable Wv and copy operations (3 and 4) are added in all chaining trails.***

In this way, wire-variables are introduced in all paths in the chaining trails for variables that are written and read in the same cycle. A dead code elimination pass later removes any unnecessary variables and variable copies. Hence, the chaining heuristic has to traverse all the chaining trails leading up to the current scheduling step and insert copy operations to wire-variables for all the variables/operands read by the operation being scheduled.

Chaining operations across conditional blocks is particularly useful for the design of low latency blocks such as microprocessor functional blocks [GKK+02] (see Chapter 11). These blocks are usually targeted to an implementation within a single or a few cycles and hence, all the operations in the design description have to be chained together. In general, there are often situations in control-intensive designs similar to those shown in Figures 6.12 and 6.13, wherein there exist copy operations within conditional branches that assign a value, computed before the conditional, to a variable that is subsequently read by an operation after the conditional block.

6.7 Loop Shifting

The computationally expensive portions of multimedia and image processing applications typically contain arithmetic operations embedded in deeply nested loops with a complex mix of conditional (if-then-else) constructs. The presence of these nested loops limits the scope of parallelizing code motion transformations to within one loop iteration.

To achieve the next level of performance improvement, we use an incremental loop pipelining transformation, called *loop shifting,* that moves operations from one iteration of the loop body to its previous iteration [GDGN04]. It does this by shifting a set of operations from the beginning of the loop body to the end of the loop body; a copy of these operations is also placed in the loop head or prologue. Parallelizing transformations can then operate on the shifted operations to further compact the loop body. Thus, loop shifting shifts a set of operations one at a time, thereby, exposing just as much parallelism as can be exploited by the available resources.

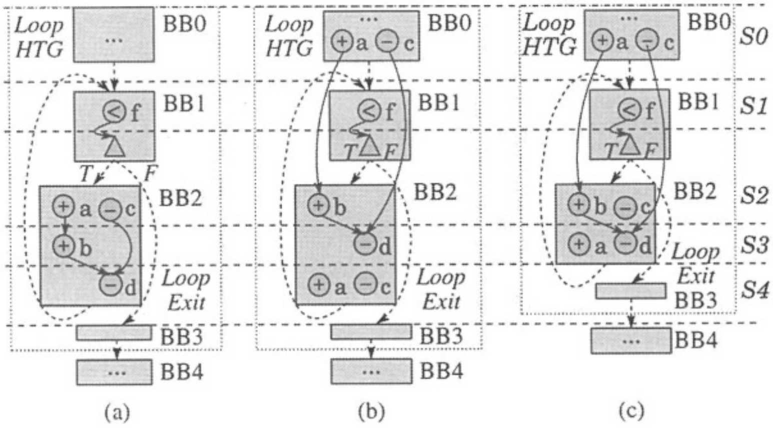


Figure 6.14: ***Loop Shifting: (a) An example with a loop. (b) Operations a and c are shifted to the end of loop body (basic block bb_2) and copies are inserted in the loop head (bb_0). (c) Shorter schedule length after code compaction.***

In contrast, loop pipelining techniques such as modulo scheduling conceptually overlap several iterations of the loop body at constant time (initiation) intervals [RG81]. We found that the overlapped iterations can lead to a sharp increase in the number of operations executed in one iteration. This leads to an increase in the number of operations mapped to each resource in the design, which in turn leads to a large increase in the size and complexity of the control and steering (multiplexing) logic associated with the resources. These increases in the control and multiplexing logic have a deleterious affect on the total input to output circuit delay and thus, the control costs of these pipelining techniques outweigh the gains achieved in schedule lengths (we demonstrate this for loop unrolling through experiments in Chapter 9).

Hence, although loop unrolling and traditional loop pipelining have been shown to be useful for the high-level synthesis of straight-line DSP algorithms, we found that for designs with complex control flow, they can lead to worse synthesis results. This is particularly true when the resource utilization is already high because of either high instruction level parallelism in the design or as a result of prior loop unrolling. In such a situation, we propose applying loop shifting to incrementally move code across iterations in a controlled way so that operations can be compacted further without excessive increase in control costs.

Loop shifting is a technique whereby an operation ***op*** is moved from the beginning of the loop body to the end of the loop body, along the back-edge of the loop. To preserve the correctness of the program, a copy op_c of operation ***op*** is placed in the loop head/prologue. Thus, op_c is executed before the first iteration of the loop body and the original operation ***op*** is then executed at the end of the loop body. This execution corresponds to the execution of ***op*** from the next loop iteration as per the original code.

We demonstrate loop shifting with an example in Figure 6.14. In this example, basic blocks bb_1 and bb_2 form the body of a loop and bb_0 is the loop head and bb_3 is the loop exit or tail. Solid arrows indicate data flow and dashed arrows indicate control

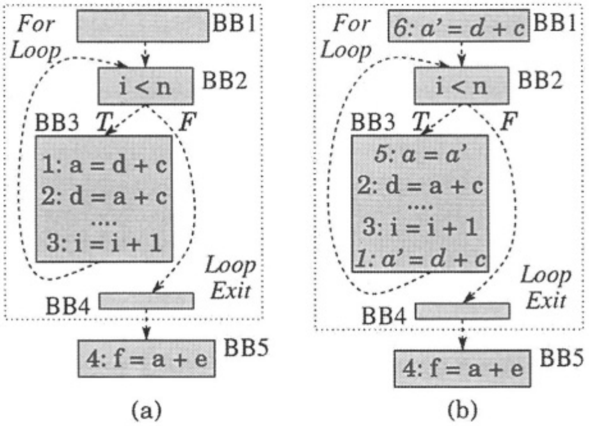


Figure 6.15: *(a) An example design. (b) Copy operation, $a = a'$ is left in place of shifted op* 1 *to ensure code correctness.*

flow. Consider that we shift operations a and c from the loop body in the original design in Figure 6.14(a) to the end of the loop body (bb_2) and copies of a and c are inserted in the loop head (bb_0). The resultant design is shown in Figure 6.14(b).

We can now compact the code inside the shifted loop body using parallelizing transformations. In the shifted design, it is possible to schedule operation a concurrently with operation d and c concurrently with operation b. The resultant, compacted design is shown in Figure 6.14(c). The state assignments ($S0$ to $S4$) for these three designs are demarcated by dashed lines. Clearly, the design in Figure 6.14(c), after shifting and compaction, has a shorter schedule length than the original design in Figure 6.14(a).

Thus, as a result of loop shifting and compaction, the loop body executes in fewer cycles. These fewer cycles multiplied by the loop iteration count give us the reduction in execution cycles of the design. However, loop shifting is useful only when the gains in performance of the loop body is larger than the overhead of the copies of the shifted operations that are placed in the loop head.

6.7.1 Ensuring the Correctness of Code

Shifting an operation leads to one extra execution of the operation over the number of times it is executed in the original code. This can be understood by the shifted design shown earlier in Figure 6.14(c). In this design, if the loop executes for 8 iterations, then the shifted operation a executes 8 times inside the loop body plus once in the loop head (basic block bb_0). In contrast, in the original design in Figure 6.14(a), operation a executes only 8 times inside the loop body.

To ensure that executing the shifted operation one extra time does not change the behavior of the program, we write the result of the shifted operation, *op*, to a new variable, $newVar$ and in place of *op*, we leave a copy operation from $newVar$ to the result variable of the original operation *op*.

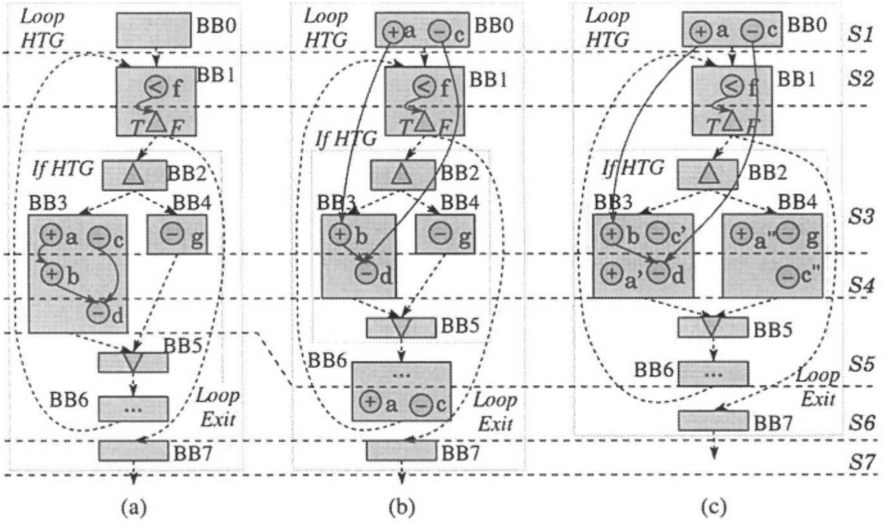


Figure 6.16: ***(a) A design with a if-then-else conditional block inside a loop. (b) Operations a and c are shifted from the longer conditional, bb_3. (c) Code compaction by duplicating operations a and c into both conditional branches.***

We demonstrate this through an example in Figure 6.15(a). Here, the result of operation 1 in the loop body (in basic block bb_3) is read by operation 4 after the loop. Consider that we shift operation 1 to the end of the loop body and place a copy as operation 6 in the loop head. Both these operations write to a new variable a' and a copy operation $a = a'$ is left in place of the original operation 1. This ensures that operation 4 gets the correct value of a after loop shifting. The resultant design is shown in Figure 6.15(b).

We also have to maintain the inter and intra-iteration data dependencies while applying loop shifting since a shifted operation may have data dependencies across loop iterations. In the example in Figure 6.15(a), operation 1 reads the variable d that is written by operation 2. Hence, after shifting operation 1, we have to add a data dependency arc from operation 2 to shifted operation 1.

6.7.2 Shifting Loops with Conditional Branches

In loops with conditional constructs, we shift operations from within a conditional branch. Since the goal of our approach is to minimize the length of the longest path through the design, we shift operations from the branch of the conditional with the *longer schedule length.*

Consider the example in Figure 6.16(a). This example has an if-then-else conditional block within the body of a loop. Since the true branch (basic block bb_3) of this if-block has a longer schedule length (of 3) than the false branch (bb_4), we choose to shift operations from basic block bb_3.

Hence, consider that we shift operations a and c from bb_3, as shown in Figure 6.16(b). The parallelizing code transformations can now compact the shifted code by conditionally speculating or duplicating operations a and c into both branches of the if-block, as operations a' and a'' and c' and c''. The resultant design is shown in Figure 6.16(c).

It is interesting to note here that the shifted and duplicated operations may be scheduled in different states or control steps in the two conditional branches, as is the case for this example. Also, it is clear from this example that loop shifting can lead to a significant increase in the number of operations in the design. Thus, loop shifting does have a control and multiplexing cost associated with it, as we will see when we present results in Chapter 9.

Also note that, when shifting operations out of conditional branches, we have to store the result of the shifted operation in a new variable. The result is committed to the original result variable of the operation only within the conditional branch. Hence, we insert a copy operation in the same manner as discussed earlier in Section 6.7.1.

We perform loop shifting after scheduling the loop body once. This means that the scheduler may schedule some operations to execute concurrently in the same cycle, i.e., in the same scheduling step. Thus, in our approach instead of shifting one operation at a time, we shift an entire scheduling step across loop iterations. This is because shifting only one of several concurrent operations will not eliminate the scheduling step and hence, the schedule length of the basic block (and loop body) will not decrease. In the design in Figure 6.16(a), we chose to shift the first scheduling step in bb_3, i.e., both operations a and c (instead of just one of them).

6.8 Summary

In this chapter, we first discussed speculation and predication and how these techniques apply to high-level synthesis. We then presented a set of speculative code motion transformations that form an important part of our parllelizing high-level synthesis methodology. In Section 6.5, we presented the dynamic common subexpression elimination (CSE) and the dynamic copy propagation techniques. We showed how speculation and conditional speculation enable several opportunities for dynamic CSE. In Section 6.6, we presented a technique that chains operations across conditional boundaries. We presented the issues that come up when operations on multiple control paths that have to be chained together and discussed the notion of wire-variables that are required by this technique. In Section 6.7, we presented loop shifting – an incremental loop pipelining technique – that operates at the end of scheduling phase. The contributions of this chapter are the presentation of various compiler, parallelizing compiler, and synthesis techniques that are an essential part of our parallelizing high-level synthesis methodology and how these techniques interact with each other.

7

CODE TRANSFORMATIONS AND SCHEDULING

Resource-constrained scheduling for high-level synthesis is the task of assigning operations to control steps or time intervals so that the allocated resources can compute the operations assigned to each step [GDWL92]. The minimum-latency resource constrained scheduling problem has been shown to be NP-complete [GDWL92, DM94, GJ79, NR00]. Several scheduling heuristics have been proposed in the literature that attempt to trade-off schedule computation time against the quality of the solution as measured for schedule length, controller size, energy/power et cetera. The use of parallelizing code transformations explored in this book provides another level of flexibility in scheduling tasks by providing either additional slots or additional operations for scheduling in a given control step.

To evaluate the effectiveness of the code transformations, we design a scheduler where the code transformations can be run fairly independently of the choice of the scheduling algorithm. For the purpose of demonstration, we choose a priority-based list scheduling heuristic since it is a popular heuristic that forms the basis of a large number of scheduling algorithms including force-directed scheduling, path-based scheduling etc. In this chapter, we describe the various algorithms that form a part of the scheduling framework.

7.1 Software Architecture of the Scheduler

Figure 7.1 shows the overall architecture of the scheduler in our PHLS framework. The components of this scheduler framework are:

(a) An *IR Walker* (IR stands for intermediate representation) that traverses the design and returns the next step (and basic block) to schedule.

(b) A *Candidate Fetcher* that itself consists of two components:

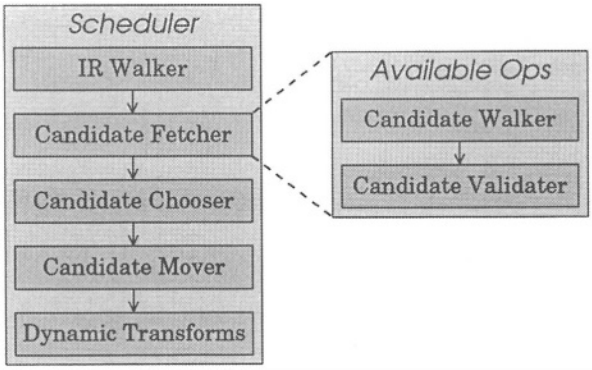


Figure 7.1: ***The architecture of the scheduler in our PHLS framework***

(i) A *Candidate Walker* that traverses the design and finds the unscheduled operations that are candidates for scheduling on the current step being scheduled. These candidate operations are called *Available Operations.*

(ii) A *Candidate Validater* that removes those unscheduled available operations whose data dependencies are not satisfied or that cannot be moved to the current step being scheduled.

(c) A *Cost Function* that calculates the cost of scheduling each candidate in the available operations list. The scheduler then picks the operation with the lowest cost.

(d) A *Candidate Mover* that moves the chosen operation from its current basic block to the current step being scheduled.

(e) A *Dynamic Transformations* pass that applies low level compiler optimizations such as CSE and copy propagation dynamically during scheduling based on the new position and possible duplication of the scheduled operation.

7.2 Priority-based Global List Scheduling Heuristic

The priority-based global list scheduling heuristic that is the basis of our scheduler is listed in Algorithm 1. The inputs to this heuristic are the unscheduled design graph $\mathcal{DG}$ and the list of resources $\mathcal{R}$. In each clock cycle and for each resource in the resource list, the scheduling heuristic collects a list of unscheduled operations and orders them according to a cost or *priority*. The operation with the lowest cost is picked and scheduled in the current clock cycle. Dynamic transformations are applied either in this process (e.g., dynamic branch balancing) or after scheduling the operation (e.g., dynamic CSE). The algorithm is global in the sense that all unscheduled operations in the design graph are considered for scheduling in each clock cycle – this is in contrast to a local algorithm that would only consider unscheduled operations within the current

Algorithm 1 : PriorityListScheduling ($\mathcal{DG}, \mathcal{R}, CMs$)
/* Schedules the Design Graph $\mathcal{DG}$ */

```
1:  $Pr \leftarrow$ CalculatePriority( $G_{DFG}$ )
2:  $step \leftarrow$ FirstStep ( FirstBB($G_{CFG}$) ) /* First step of first basic block in $G_{CFG}$ */
3:  while $step \neq \phi$ do
4:    for all resources $res \in \mathcal{R}$ do
5:      $\mathcal{A} \leftarrow$ GetAvailableOperations($\mathcal{DG}, step, res, CMs$)
6:      if ($\mathcal{A} \neq \emptyset$) then
7:        CalculateCostOfAvailOps($\mathcal{A}, Pr$)
8:        Pick Operation $op \in \mathcal{A}$ with lowest cost
9:        TrailblazeOp($op, res, step, G_{HTG}, G_{CFG}, CMs, \mathcal{R}$)
10:       Mark $op$ as scheduled
11:       ApplyDynamicCSE($\mathcal{A}, op, \mathcal{DG}$)
12:     end if
13:   end for
14:   $step \leftarrow$ GetNextSchedulingStep($G_{HTG}, G_{CFG}, step$)
15: end while
```

basic block. The various function calls made by the scheduling heuristic are presented in detail over the next few sections.

We can also specify a list of allowed code motions, *CMs,* (i.e., speculation, reverse speculation, conditional speculation et cetera) to the scheduling heuristic. This gives us control over the code motions employed while scheduling the design by selecting and de-selecting code motions from *CMs.* In this way, we can study the effectiveness and performance-area trade-offs of individual code motions.

The heuristic starts by assigning a priority to each operation in the input description by calling the function *CalculatePriority. Priority* of an operation is calculated as the length of the data dependency chain from the operation to an output of the design (see Section 7.2.2 for a detailed definition of priority). The *CalculatePriority* function returns the function Pr that gives the priority of an operation op_i as $Pr(op_i)$.

Scheduling is then done one scheduling step at a time while traversing the basic blocks in the design graph $\mathcal{DG}$. This design traversal is done by the function *GetNextSchedulingStep.* Scheduling starts with the first scheduling step of the first basic block of the design CFG G_{CFG}. For each scheduling step in the design, we iterate over each resource *res* in the resource list $\mathcal{R}$ and collect a list of *available* operations $\mathcal{A}$ that may be scheduled on res by calling the function *GetAvailableOperations* (lines 3 to 5 in the algorithm in Algorithm 1).

Available operations is a list of operations whose data dependencies are satisfied and that can be moved in the design graph and scheduled on the given resource at the current scheduling step. If the available operations list is not empty, the scheduler calculates a cost for each operation in $\mathcal{A}$. Currently, this cost is the negative of the operation's priority, that is:

$$cost(op_i) = -Pr(op_i) \ \forall \ op_i \in G_{DFG}$$

The scheduler then picks the operation op with the *lowest* cost from the available operations list (line 8 of Algorithm 1). Effectively, this chooses the operation with the highest priority in the available list and hence, favors operations that are on the longest path through the design. Thus, this cost function attempts to minimize the longest delay through the design. It is important to note that minimizing a different cost function, such as average delay, can be done by incorporating control flow information into the cost function. Also, if we have profiling information about which control paths are more likely to be taken, then we can give operations on those paths a higher priority than operations on less taken paths. Future work entails enhancing the cost function to include hardware (control and area) cost models of the code transformations.

The scheduler calls the *TrailSynth* algorithm (see Section 7.4) to move the chosen operation op from its current scheduling step and basic block in the design and schedule it on the resource *res* at the scheduling step *step*. This is done by a function call to *TrailblazeOp*. The *TrailblazeOp* may duplicate op into multiple basic blocks. The scheduler then marks op as scheduled in *step* on the resource *res*.

A dynamic transformations pass then operates using the scheduled operation. As shown in Algorithm 1, the scheduler calls the dynamic CSE (common sub-expression elimination) algorithm to eliminate any operations in $\mathcal{A}$ that have common sub-expressions with op and that may now be eliminated due to the code motion (and possible duplication) of op to *step*. Dynamic copy propagation and dynamic dead code elimination are also applied after dynamic CSE.

This scheduling procedure is repeated for all the resources in each scheduling step as the basic blocks in the design are traversed from the initial basic block to the last. Since operations with higher priority may be speculated into a scheduling step *step* in a basic block bb_{step}, the (lower priority) operations already in *step* in bb_{step} can be left unscheduled. Either new scheduling steps are added to the basic block bb_{step} to schedule these operations or if reverse speculation has been enabled, then these unscheduled operations are reverse speculated into the subsequent conditional branches (if possible). If the successor basic block of bb_{step} is a join basic block, then the only option is to add new scheduling steps in bb_{step}, since we do not support code motion of operations past a *successor* join node. Note that, we do allow code motions of an operation past a predecessor join node using the conditional speculation code motion.

7.2.1 Scheduling Loops

Scheduling of loops is done by the same procedure outlined above. However, user-specified loop transformations such as loop unrolling et cetera are applied first. Also, the scheduler cannot move operations into or out of the loop body. This can only be done by transformations such as loop-invariant code motion or loop pipelining. Hence, the available operations algorithm does not collect unscheduled operations from inside a loop body to schedule them outside the loop body. Also, while scheduling the loop body of a loop node, available operations are collected only from within the loop body.

The *Spark* framework can schedule all types of loops, including those with unknown loop iteration bounds. Thus, at the end of a loop body, the next state in the finite state machine (FSM) generated by the framework is either the first state in the loop body or the state after the loop body, depending on whether the loop condition is satis-

Algorithm 2 : CalculatePriority (G_{DFG})
Returns: Priority function Pr for each op_i in $Vops$
/* Calculates priority of each operation in design */

```
Initialize $Pr(op_i) = 0 \; \forall \; op_i \in Vops$
for all $op_i \in Vops$ do    /* $Vops$ is the set of operations in $G_{DFG}$ */
    $Pr(op_i) \leftarrow$ GetOpPriority($op_i$)
end for
```

Algorithm 3 : GetOpPriority (op_i)
Returns: Priority $Pr(op_i)$ of op_i
/* Recursive function that returns priority of an operation in the design */

```
if ($Pr(op_i) = 0$) then
    for all $e_{ij} \in E_{data}$ do    /* All outgoing data flow edges from $op_i$ */
        $Pr(op_i) \leftarrow$ max($Pr(op_i)$, GetOpPriority($op_j$)+1)
    end for
end if
return $Pr(op_i)$
```

fied or not. Hence, loop bounds are not required for generating correct, synthesizable VHDL. However, when the loop bounds are unknown, several loop transformations cannot be applied to the design and we cannot establish the number of cycles that the loop takes to execute.

We now present the various algorithms that the scheduler employs, starting with the algorithm that calculates the priority of the operations in the design graph.

7.2.2 Calculating Priority

Our primary objective is to minimize the *longest delay* through the design; hence, priorities are assigned to each operation based on their distance, in terms of the data dependency chain, from the primary outputs of the design. The priority of an operation is calculated as one more than the maximum of the priorities of all the operations that use its result. Hence, an operation that produces an output is not read by any operation and thus, these *output* operations have a priority of zero. Thereafter, operations whose results are read by output operations have a priority of one and so on.

An algorithm that calculates the priorities of all the operations, Pr, in a data flow graph, G_{DFG}, is given in Algorithms 2 and 3. The algorithm iterates over all the operations in G_{DFG} and starts by assigning each operation op_i a priority of zero. It then calls the function *GetOpPriority* that iterates over all the operations that read the result of op_i and assigns op_i a priority that is one more than the maximum of the priorities of all these dependent operations. The function *GetOpPriority* recursively calls itself to find the priority of the dependent operations, hence, performing a depth first calculation of the operation priorities.

The priority assignment of operations for the waka benchmark is indicated next to the operations in Figure 7.2. In this design, the priority assignment of the output operations, n, l and k is 0, and the operations that they depend on have priority 1 and

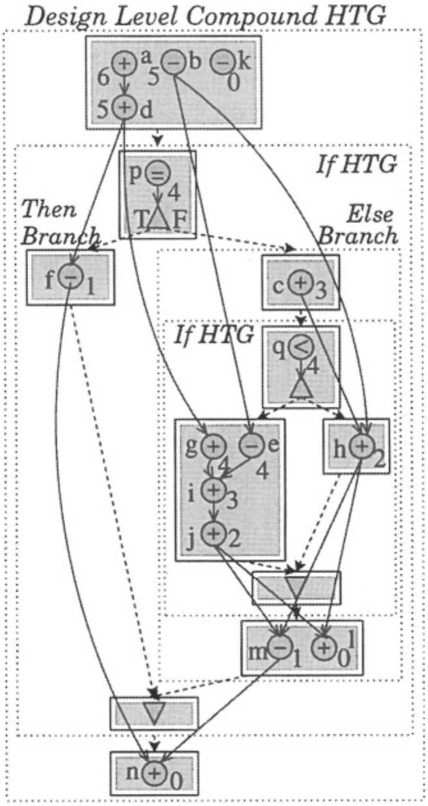


Figure 7.2: ***Priority assignment for the operations in the "waka" benchmark.***

so on. The priority of an operation that creates a conditional check (operations p and q in the figure), is assigned the maximum of the priorities of all the operations in the conditional branches of the If-HTG. Note that, this has not been shown in Algorithms 2 and 3.

7.3 Collecting the List of Available Operations

Available operations is a list of operations that can be scheduled on the given resource *res* at the current scheduling step *step.* Pseudo-code for collecting the list of available operations is given in Algorithm 4. Initially, all unscheduled operations in the design that can be executed on the resource type of *res* and whose data dependencies have been satisfied are added to the available operations list. These unscheduled operations are collected by the candidate walker function *CollectUnscheduledOps* presented in the next section.

```
Algorithm 4 : GetAvailableOperations ( $\mathcal{DG}$, $step$, $res$, $CMs$ )
Returns: Available Operations List $\mathcal{A}$
/* Gets operations available to be scheduled on res in step */

   /* Find all unscheduled operations in $G_{CFG}$ that can be scheduled on res */
1: CollectUnscheduledOps($res$, $step$, $BB_{Steps}(step)$, $G_{DFG}$, $G_{CFG}$, $\mathcal{A}$)
   /* Remove all operations that cannot be moved to step using CMs */
2: for all $op_i \in \mathcal{A}$ do
3:    if (IsTrailblazeOpPossible($op_i$, $step$, $G_{HTG}$, $G_{CFG}$, $CMs$, $\mathcal{R}$) = false) then
4:       $\mathcal{A} \leftarrow \mathcal{A}$ - $op_i$
5:    end if
6: end for
7: return $\mathcal{A}$
```

Once these unscheduled operations have been collected, the available operations algorithm calls the *I sTrailblazeOpPossible* function to determine which operations cannot be moved in design to the current scheduling step using the *allowed* code motions, CMs. These operations are removed from the available list (line 4 of the algorithm in Algorithm 4).

The *I sTrailblazeOpPossible* function corresponds to the candidate validater function shown earlier the scheduler architecture in Figure 7.1. The operations remaining in $\mathcal{A}$ after validation are possible candidates for scheduling on *step*. This available list $\mathcal{A}$ is returned to the scheduling algorithm.

7.3.1 Collecting the Unscheduled Operations from the Design Graph

The algorithm for the function *CollectUnscheduledOps* is presented in Algorithm 5. This algorithm takes as input the resource *res* we are scheduling on and the current scheduling step *startStep*. The algorithm starts by examining each scheduling step in the current basic block bb_i and looks for unscheduled operations that can be executed on *res*. These unscheduled operations are added to the available list if their data dependencies have been satisfied. This check is done by calling the function *AreDataDepsSatisfied* (described in the next section). This corresponds to lines 1 to 6 in Algorithm 5.

When the unscheduled operations from all the scheduling steps in the current basic block bb_i have been collected, the *CollectUnscheduledOps* function traverses the basic blocks on the control flow paths leading out of the basic block bb_i. Thus, the *CollectUnscheduledOps* function recursively calls itself to collect the unscheduled operations from each successor basic block bb_j of the current basic block bb_i (lines 12 to 14 of the algorithm).

Note that, in practice, there are several ways to improve the efficiency of this algorithm. Firstly, we can set a limit to the depth/number of basic blocks that the function should look at for unscheduled operations. Secondly, we can maintain a list of unscheduled operations ordered by the resource they execute on.

Although not shown in Algorithm 5, this basic block traversal algorithm skips over the loop body of any loop HTG nodes it encounters. This is because operations from

Algorithm 5 : CollectUnscheduledOps(res, $startStep$, bb_i, G_{DFG}, G_{CFG}, $\mathcal{A}$)
/* Adds all unscheduled ops in basic blocks after bb_i to $\mathcal{A}$ */

```
$step_k = startStep$
while $step_k \neq \phi$ do
  for all $op_l \in step_k$ do    /* Add each unscheduled op in $step_k$ whose data dependencies are satisfied and can be executed on the resource type of $res$ */
    if (($op_l$ is not scheduled) and ($\mathcal{T}(op_l) == res$)) then
      if (AreDataDepsSatisfied($op_i$, $G_{DFG}$, $step$, $res$)) then
        $\mathcal{A} \leftarrow \mathcal{A} \cup op_l$
      end if
      $step_k \leftarrow$ NextStep($bb_i$, $step_k$)
    end if
  end for
end while
/* Recursively traverse all outgoing control edges from $bb_i$ */
for all $e_{ij} \in E_{control}$ do
  CollectUnscheduledOps($res$, FirstStep($bb_j$), $bb_j$, $G_{DFG}$, $G_{CFG}$, $\mathcal{A}$)
end for
```

within loop nodes can only be moved outside the loop body by transformations such as loop-invariant code motion and loop pipelining. Similarly, when the scheduler is scheduling the loop body of a loop node, available operations are only collected from within the loop body.

7.3.2 Algorithm for the *AreDataDepsSatisfied* function

The algorithm for the function *AreDataDepsSatisfied* is listed in Algorithm 6. This function iterates over all the incoming data flow edges of operation op_i. For each operation op_j whose result op_i reads, the function checks if op_j is scheduled and if it has finished execution by the time *step* starts executing (lines 2 and 3). Else if op_j is scheduled in *step,* then the algorithm checks if op_j and op_i can be chained in *step.* This is possible if the sum of their execution times does not exceed the clock period (line 3 in algorithm). Recall that $T_{res}(res)$ gives the execution time of the resource *res* as per Definition 3.9.

If any of these conditions is not satisfied, i.e., if op_j is not scheduled or does not finish execution in time for op_i to execute, then the function returns a false result (line 7 in Algorithm 6). If all the operations that op_i depends on have finished execution, then the function returns a true result. Note that, the efficiency of this function can be improved by marking operations whose data dependencies have been satisfied. Then, the next time the *AreDataDepsSatisfied* function is called, it can first check if the operation has been previously marked as satisfied.

While validating the candidate operations, the available operations algorithm calls the function *IsTrailblazeOpPossible* to determine if these operations can be moved to the scheduling step under consideration. This function in turn uses the *Trailblazing* code motion technique as explained in the next section.

Algorithm 6 : **AreDataDepsSatisfied(** op_i, G_{DFG}, $step$, res)
Returns: True if data dependencies are satisfied, else false
/* Checks if the data dependencies of op_i are satisfied */

```
for all $e_{ji} \in E_{data}$ do     /* All incoming data flow edges to $op_i$ */
    if ($op_j$ is scheduled) then
      /* Check if either $op_j$ has finished execution by step */
      /* OR $op_j$ and $op_i$ can be chained in step */
      if (($t_j + dop_j <$ start time of operations in step) or
         ($op_j$ is scheduled in step and $T_{res}(res) + dop_j \leq C_{CLK}$ )) then
         Continue for all loop to next edge
      end if
    end if
    /* Either $op_j$ is not scheduled or has not finished execution */
    /* or cannot be chained with $op_i$ */
    return false
end for
return true     /* All data dependencies are satisfied */
```

7.4 *TrailSynth*: A code motion algorithm based on *Trailblazing*

Our code motion algorithm employs *Trailblazing* to find the trails required to move operations in the design. But besides this our algorithm also analyzes resource utilization and employs a novel dynamic branch balancing technique. We refer to our code motion algorithm as *TrailSynth* since it is based on *Trailblazing.*

Trailblazing is a hierarchal code motion technique that finds all the control paths that an operation will have to move along in order to be scheduled in a given basic block. These control paths are represented by *trails* as per the following definition:

Definition 7.1. *A* trail tr *is an ordered list of basic blocks with unique source and sink basic blocks, Source(tr) and Sink(tr). The list of basic blocks in* tr *denotes the basic blocks that will be visited in moving an operation opfrom its current basic block* $BB_{Ops}(op)$ *to the basic block target BB* $\in$ *Vbb. Hence,* $Source(tr)$ = $BB_{Ops}(op)$ *and Sink(tr) = targetBB.*

The *Trailblazing* algorithm returns a list of trails given by *TrailList.* There is a trail for each control path that leads from the current basic block that operation op is in, to the scheduling step, *step,* that the operation op is being scheduled on **or** to one of the basic blocks into which op will be duplicated. Consider the example in Figure 7.3. To move operation b from basic block $bb_1 2$ to basic block bb_0, the *Trailblazing* algorithm will return two trails:
Trail 1: $<bb_{12}, bb_{11}, bb_8, bb_7, bb_6, bb_3, bb_1, bb_0>$
Trail 2: $<bb_{12}, bb_{11}, bb_8, bb_7, bb_2>$.

Trail 1 is a trail for operation b to basic block bb_0. However, the second trail stops in basic block bb_2 since operation b has a data flow dependency with the operation a in

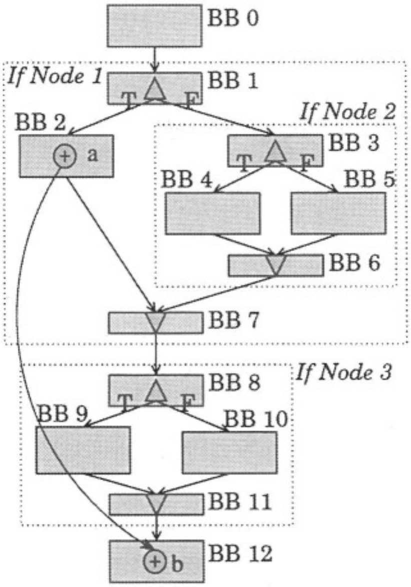


Figure 7.3: *Finding the* Trailblazing *trails in an example HTG.*

bb_2. Hence, to move operation b to bb_0, we would have to leave a duplicated copy in basic block bb_2. We do not present the *Trailblazing* algorithm here; our implementation is similar to the algorithm presented in [NN93].

The example in Figure 7.3 also demonstrates the hierarchical nature of *Trailblazing*. The trails returned for moving operation b jump over the If-HTG nodes *IfNode3* and *IfNode2* in this example. This is because operation b does not have a data dependency with any operations in these HTG nodes. It is this hierarchical code motion capability of *Trailblazing* that makes it efficient for moving operations across large pieces of code without duplicating the operation at every join node encountered and then unifying the copies at fork nodes.

In our implementation of *Trailblazing,* we also supply the *Trailblazing* algorithm with the list of allowed code motions, *CMs,* that is specified by the user. While generating the list of trails, the *Trailblazing* algorithm also determines the code motions that will be required to move the operation on these trails. If a required code motion is not in the list of allowed code motions (*CMs*), then the *Trailblazing* algorithm returns an empty list of trails, i.e., it fails to move the operation to the requested target basic block.

Hence, for the example in Figure 7.3, if we disable conditional speculation, then *Trailblazing* will return an empty list of trails when we query it for the trails to move operation b to basic block bb_0. This is because to move operation b to bb_0, it has to be conditionally speculated at the join basic block bb_7 and a duplicate copy has to be left in basic block bb_2. Similarly, the returned trails will be empty, if we disallow speculation in *CMs* and ask *Trailblazing* for trails to move operation a to bb_0.

```
Algorithm 7 : IsTrailblazeOpPossible(op, step, $G_{HTG}$, $G_{CFG}$, $CMs$, $\mathcal{R}$)
Returns: Returns true if Operation op can be moved to step, else returns false
/* Determines if op can be moved from its current basic block to target basic block */
1:  $targetBB \leftarrow BB_{Steps}(step)$
2:  $currBB \leftarrow BB_{Ops}(op)$
    /* Call Trailblazing to find trails */
3:  $TrailList \leftarrow$ FindTrails($currBB$, $targetBB$, $G_{HTG}$, $G_{CFG}$, $CMs$)
4:  if ($TrailList == \emptyset$) then
5:     return false
6:  end if
7:  for all trail $\in$ TrailList do
8:       lastBBInTrail $\leftarrow$ Sink(trail)
9:     if ( (lastBBInTrail $\neq$ targetBB) and (lastBBInTrail is not scheduled) )
       then
10:       return false
11:    end if
12:    $stepInBB \leftarrow$ FindIdleResInBB(op, $lastBBInTrail$, $\mathcal{R}$)
13:    if ($stepInBB = \emptyset$) then
14:       if (IsBranchBalancingPossible($G_{HTG}$, $lastBBInTrail$) = false) then
15:          return false
16:       end if
17:    end if
18: end for
19: return true   /* Found idle resources in all trails */
```

This means that there is no fixed order for the application of code motions. The code motions required to move an operation to the current scheduling step depends on the basic block that the operation is in relative to the current scheduling step. Also, since *CMs* are user-specified in a script file read by *Spark* at initialization, it enables us to experiment with the affects of the various code motions on synthesis results.

We employ two variants of the *TrailSynth* algorithm: (a) the available operations algorithm calls the *IsTrailblazeOpPossible* algorithm to determine if is possible to move each operation in the available list to the scheduling step currently being scheduled, and (b) the *TrailblazeOp* function called later by the scheduler to actually move and duplicate the operation chosen from the available list for scheduling. We present these two algorithms next.

7.4.1 Algorithm for the *IsTrailblazeOpPossible* Function

The algorithm for *IsTrailblazeOpPossible* function is listed in Algorithm 7. This algorithm determines if operation *op* can be moved in the design to the scheduling step *step* using the code motions *CMs*. The algorithm starts by determining the current basic block *currBB* of *op* and the target basic block *targetBB* in which *step* is (lines 1 and 2). It then calls the *Trailblazing* algorithm (*FindTrails*) to find the trails from the *currBB* to *targetBB* using the code motions *CMs*. The returned list of trails is null if ***op*** cannot be moved to *targetBB* (see previous section); in this case, the *IsTrailblazeOpPossible* function returns a false result. Note that, *TrailList* contains only one trail if ***op*** will not have to be duplicated in order to schedule it on *targetBB*.

If the list of trails is not null, the *IsTrailblazeOpPossible* function iterates over each trail in the list and returns a false result if the sink basic block (*lastBBInTrail*) of any trail is not scheduled. This is because the resource utilization in an unscheduled basic block is unknown and thus, we want to prevent the duplication of an operation into unscheduled basic blocks.

For each scheduled *lastBBInTrail,* the *IsTrailblazeOpPossible* function calls the *FindIdleResInBB* function to get a scheduling step in *lastBBInTrail* that has an idle resource on which ***op*** can be scheduled. This function is presented later in Section 7.4.3. This function is not called for trails that end in *targetBB,* since ***op*** will be scheduled in scheduling step *step* in *targetBB.*

If the *FindIdleResInBB* function is unable to find a scheduling step to accommodate a duplicated copy of ***op*** in **any** of the sink basic blocks of the trails, then the *IsTrailblazeOpPossible* function returns a false result (lines 12 to 15 in Algorithm 7). However, before returning a false result, the *IsTrailblazeOpPossible* function calls the function *IsBranchBalancingPossible* to check if the dynamic branch balancing technique can create a new scheduling step in the basic block(s) that do not have an idle resource. This function and dynamic branch balancing are explained in detail in Section 7.7.

The *IsTrailblazeOpPossible* function returns a true result either if it finds an idle resource for ***op*** in each basic block that ***op*** will be duplicated into or if it is possible to create a new scheduling step (and thus, an idle resource) using the branch balancing technique.

7.4.2 Algorithm for the *TrailblazeOp* Function

To actually move the operation chosen for scheduling, the scheduling heuristic calls the *TrailblazeOp* algorithm. The algorithm for this function is presented in Algorithm 8. The *TrailblazeOp* algorithm takes as input the operation being scheduled ***op***, the scheduling step to schedule it on *step* and the resource *res* in *step* to schedule ***op*** on.

The *TrailblazeOp* is similar to the *IsTrailblazeOpPossible* function described above. This function also calls the *Trailblazing* algorithm to find the trails that lead from the current basic block *currBB* of operation ***op*** to the target basic block *targetBB* of the scheduling step *step.* The *TrailblazeOp* function then finds a step in which to schedule ***op*** in the sink basic block of each trail by calling the *FindIdleResInBB* function. For the *targetBB,* ***op*** is inserted in *step* and for all other sink basic blocks, i.e., *lastBBInTrail,* ***op*** is duplicated and inserted in the step, *stepInBB,* found by *FindIdleResInBB.* If *FindIdleResInBB* does not find a step in a basic block, then the branch balancing algorithm is called (call to *BalanceBranches* in line 12 of Algorithm 8) to insert a new scheduling step in *lastBBInTrail.*

Since the *IsTrailblazeOpPossible* function has already been executed earlier, we can be sure that either the *FindIdleResInBB* function or the *BalanceBranches* function will find or create a scheduling step in *lastBBInTrail.* The resource *res*

Algorithm 8 : **TrailblazeOp**(op, $step$, G_{HTG}, G_{CFG}, CMs, $\mathcal{R}$)
/* Moves (and possibly duplicates) *op* from its current basic block to *step* */

```
1:  $targetBB \leftarrow BB_{Steps}(step)$
2:  $currBB \leftarrow BB_{Ops}(op)$
    /* Call Trailblazing to find trails */
3:  $TrailList \leftarrow$ FindTrails($currBB$, $targetBB$, $G_{HTG}$, $G_{CFG}$, $CMs$)
4:  for all $trail \in TrailList$ do
5:    $lastBBInTrail \leftarrow Sink(trail)$
6:    if ($lastBBInTrail = targetBB$) then
7:      $stepInBB \leftarrow step$
8:      $stepInBB(res) \leftarrow op$
9:    else /* $lastBBInTrail \neq targetBB$ */
10:     $stepInBB \leftarrow$ FindIdleResInBB($op$, $lastBBInTrail$, $\mathcal{R}$)
11:     if ($stepInBB = \emptyset$) then
12:       $stepInBB \leftarrow$ BalanceBranches($G_{HTG}$, $lastBBInTrail$)
13:     end if
14:     $stepInBB(res) \leftarrow$ Duplicate($op$)
15:     Update data dependencies in $G_{DFG}$ effected by $op$ duplication
16:   end if
17: end for
```

in *step* and *stepInBB* is marked as having *op* scheduled on it. The *TrailblazeOp* function also updates any changes in data dependencies due to the duplication of *op* (line 15 in Algorithm 8).

7.4.3 Finding an Idle Resource in a Basic Block

The algorithm to find an idle resource for an operation *op* in a basic block *bb* is outlined in Algorithm 9. This algorithm starts by calling the function *FindMatchingResForOp* (not given here) to determine the list of resources, *matchingResList,* on which the operation *op* can be executed. There may be multiple resources in *matchingResList* since, although there is only one resource type on which *op* may execute, there may be several instances of this resource type in the resource list $\mathcal{R}$. The *FindldleResInBB* function then finds the first scheduling step in *bb* that does not have an operation with a data dependency with *op* by calling the function *GetStepInBBAfterDataDeps.* This function (not given here) looks for operations whose result *op* reads and that are in basic block *bb.* It then finds the last scheduling step in *bb* with any of these operations that *op* depends on and returns the next scheduling step. This returned step, *currStep,* signifies the first scheduling step in *bb* that *op* can be potentially scheduled on. Recall that the scheduling steps in a basic block are ordered as per their execution sequence.

Using this scheduling step *(currStep)* as a starting point, the algorithm determines if there is an idle resource in *currStep* or any of its successor steps in basic block *bb* (shown by the while loop in Algorithm 9). Each resource *res* in *matchingResList* in *currStep* is checked to see if it is idle, i.e., there is no operation scheduled on

Algorithm 9: FindIdleResInBB(*op*, *bb*, $\mathcal{R}$, G_{CFG})
Returns: A scheduling step in *bb* with idle resource for *op*
/* Finds an idle resource in basic block *bb* for scheduling operation *op*

```
 1: $matchingResList \leftarrow$ FindMatchingResForOp($\mathcal{R}$, op)
 2: $currStep \leftarrow$ GetStepInBBAfterDataDeps(bb, op)
    /* Iterate over all remaining steps in bb */
 3: while currStep $\neq \phi$ do
 4:   for all res $\in$ matchingResList do
 5:     if ($currStep(res) == \emptyset$) then
 6:       $numSteps \leftarrow T_{res}(res)/C_{CLK}$ - 1
 7:       $prevStepList \leftarrow$ GetPrevSteps($G_{CFG}$, currStep, numSteps)
 8:       $succStepList \leftarrow$ GetSuccSteps($G_{CFG}$, currStep, numSteps)
          /* If an operation is scheduled in any step in prevStepList or */
          /* succStepList, then continue to next matching resource in step */
 9:       for all otherStep $\in$ prevStepList $\cup$ succStepList do
10:         if ($otherStep(res) \neq \emptyset$) then
11:           Continue for all (res) loop     /* Goes to line 4 */
12:         end if
13:       end for
14:       return currStep     /* Found currStep with idle resource */
15:     end if
16:   end for
17:   currStep $\leftarrow$ NextStep(bb, currStep)     /* Get next step in bb */
18: end while
19: return $\emptyset$     /* No step found. Return null step */
```

it and hence, it is potentially available for scheduling the operation *op* (line 5 in the algorithm).

However, the current resource *(res)* being checked may be a multi-cycle resource. Hence, the scheduling steps before and after the current step have to be checked to make sure that the resource is idle in them for the duration of its execution time starting in *currStep.* First, the number of steps that need to be checked is calculated (*numSteps*); this is one less than the execution cycles of the resource. The execution cycles of a resource are determined by dividing its execution time T_{res} with the clock period C_{CLK} allocated to the design.

The algorithm then calls the *GetPrevSteps* and *GetSuccSteps* functions to get *numSteps* predecessor steps and *numSteps* successor steps (lines 7 and 8 in Algorithm 9). Since the predecessor and successor steps can, and frequently are, in the predecessor and successor basic blocks of *bb,* these two functions (not described here) look for steps not only in the current basic block *bb* but may also traverse to the predecessor and successor basic blocks of *bb.* Hence, the resource utilization of the resource *res* has to be checked beyond the current basic block.

If the resource *res* is not used in any of these predecessor and successor steps, then an idle resource has been found in the current step *currStep* and the algorithm

Algorithm 10 : ApplyDynamicCSE($\mathcal{A}$, *op*, $\mathcal{DG}$)
/* Eliminates operations from $\mathcal{A}$ by employing dynamic CSE */

```
1: $cseOpsList \leftarrow$ GetOperationsWithCSE($\mathcal{A}$, $op$)
2: for all operations $cseOp \in cseOpsList$ do
3:    if ($BB_{Ops}(op)$ dominates $BB_{Ops}(cseOp)$) then
4:       ApplyCSE($cseOp$, $op$, $G_{DFG}$)
5:    end if
6: end for
```

terminates by returning *currStep*. However, if *res* is used in any of these steps, then the procedure is repeated for the next resource in the *matching Res List* and so on. This is done for all the steps following *currStep* in the given basic block *bb*, until either a step with an idle resource is found or all the steps in *bb* have been visited.

The *TrailSynth* algorithm uses the returned step with an idle resource to schedule a duplicated copy of the operation being scheduled. Next, the scheduling algorithm calls the dynamic CSE algorithm as explained next.

7.5 Dynamic CSE Algorithm

Once operation chosen by the scheduler has been moved and scheduled, the dynamic CSE algorithm comes into play. The dynamic CSE algorithm, listed in Algorithm 10, calls the function *GetOperationsWithCSE* to determine the the list of operations *cseOpsList* that have a common sub-expression with the scheduled operation *op*. The *GetOperationsWithCSE* function (not shown here) examines only the remaining operations in the available list to get *cseOpsList* since these are the only operations whose data dependencies are satisfied.

For each operation *cseOp* in *cseOpsList,* if the basic block of *cseOp* is dominated by the basic block of *op* *after* scheduling, then the common sub-expression in *cseOp* is replaced with the result from *op* by calling the *ApplyCSE* function (lines 2 to 4 in Algorithm 10). The *ApplyCSE* function (not described here) also updates the data dependencies in G_{DFG} due to the elimination of the computation in *cseOp*. The functions *GetOperationsWithCSE* and *ApplyCSE* are part of the basic CSE algorithm.

7.6 Design Traversal Algorithms

The design traversal algorithms perform two tasks: get the next basic block to schedule and get the next scheduling step within this basic block. We present the algorithms for these two tasks in the following sections.

7.6.1 Algorithm to Get the Next Scheduling Step

The scheduler calls the function *GetNextSchedulingStep* to get the steps to schedule in the design. The algorithm for this function is listed in Algorithm 11. This

Algorithm 11 : GetNextSchedulingStep($step_k$)
Returns: Next Step to schedule *nextStep* after $step_k$

```
1:  $currBB \leftarrow BB_{Steps}(currStep)$
2:  $nextStep \leftarrow$ NextStep($currBB$,$currStep$)
3:  if ($nextStep = \phi$) then     /* Last step in currBB reached */
4:     $nextStep \leftarrow$ BalanceBranches($G_{HTG}$, $currBB$)
5:  end if
6:  if($nextStep = \phi$) then     /* nextStep is NULL after branch balancing */
       /* Traverse to next basic block */
7:     $nextBB \leftarrow$ GetNextBasicBlock($currBB$, $G_{CFG}$)
8:     if ($nextBB \neq \phi$) then
9:         nextStep← FirstStep(nextBB)   /* First step in nextBB */
10:    end if
11: end if
12: return nextStep
```

algorithm takes as input the current scheduling step *step* and returns the next step *(nextStep)* in the design to schedule. The algorithm starts by determining the current basic block, *currBB,* that the scheduling step, *step,* is in. *nextStep* is then the scheduling step after the current *step* in *currBB* (lines 1 and 2 in the algorithm). The algorithm then checks if *nextStep* is null; this happens when the current scheduling step, *step,* is the last scheduling step in *currBB.* In this case, the algorithm should traverse the design graph and get the next basic block in the design to schedule. However, it is at this point that we employ a novel technique that balances the branches of conditionals by inserting new scheduling steps in conditional branches that have fewer scheduling steps [GDGN03a]. This is done by a call to the function *BalanceBranches* (line 4); this function is discussed in the Section 7.7.

If the branch balancing function does not return a newly created scheduling step, i.e., *nextStep* is still null (line 6), then the algorithm proceeds to get the next basic block, *nextBB,* in the design by calling the *GetNextBasicBlock* function. If *nextBB* is not null, then *nextStep* is the first scheduling step ($step_1$) in *nextBB* (lines 7 to 9 in Algorithm 11). Finally, the *GetNextSchedulingStep* algorithm returns the next scheduling step *nextStep. If nextStep* is still null, then this indicates to the scheduler that all the basic blocks in the design (and the scheduling steps in them) have been scheduled and thus, the scheduler terminates as well.

7.6.2 Algorithm to Get the Next Basic Block to Schedule

The algorithm to get the next basic block to schedule in the design is given in Algorithm 12. This algorithm traverses the basic blocks in the design CFG in a topological manner starting at the initial node of the G_{CFG}. This algorithm takes the current basic block *currBB* as input and maintains a queue of basic blocks *(bbQueue)* that it uses to determine the next basic block to schedule.

Algorithm 12 : **GetNextBasicBlock**($currBB$, G_{CFG})
Returns: Next Basic Block to schedule *nextBB*
Static: Basic Block Queue *bbQueue*
/* Traverses basic blocks in G_{CFG} in topological order */

```
1: for all $bb_i \in$ SUCCs(currBB) do
2:    PREDs($bb_i$) ← PREDs($bb_i$) - $currBB$
3:    if (PREDs($bb_i$) = $\phi$)) then
4:       EnqueueAtTail(bbQueue, $bb_i$)
5:    end if
6: end for
7: nextBB ← DequeueHead(bbQueue)
8: return nextBB
```

The algorithm starts by inspecting each successor basic block bb_i of *currBB* by calling the function *SUCCs(currBB).* It removes *currBB* from the predecessor basic block list of bb_i (*PREDs* (bb_i)). If bb_i has no more predecessors, i.e., all predecessors of bb_i have been visited and scheduled, bb_i is added to the tail of *bbQueue* by calling the *EnqueueAtTail* function. Note that this algorithm is the same as the topological ordering algorithm. Also, note that in practice the functions *SUCCs* and *PREDs* also take G_{CFG} as input.

Finally, the *GetNextBasicBlock* returns the basic block in the front or head of *bbQueue* (lines 7 and 8 in Algorithm 12). When the last basic block in the design has been reached, *bbQueue* is empty and thus, the algorithm returns a null result. Subsequently, the algorithm to get the next scheduling step terminates by returning a null result and this indicates to the scheduling heuristic that it has finished scheduling the design. Note that, although not shown in Algorithm 12), the *GetNextBasicBlock* algorithm does not traverse the *backward* control flow edge of a loop, i.e., the edge that iterates over the loop body. For loops, the loop head is scheduled first, followed by the loop body and then the loop tail.

7.7 Dynamic Branch Balancing during Scheduling

Dynamic branch balancing is a technique employed during scheduling to insert scheduling steps in *unbalanced* conditional branches of an If-HTG. To enable new opportunities for conditional speculation, branch balancing has to be performed *dynamically* during scheduling. There are two opportunities for doing branch balancing during scheduling: (i) in the IR Walker function, i.e., while getting the next scheduling step to schedule; this is known as branch balancing during design traversal (BBDDT), and (ii) in the code motion function(s); this is known as branch balancing during code motions (BDDCM). These correspond to the calls to the function *BalanceBranches* in Algorithms 12 and 8 respectively.

The algorithm for the *BalanceBranches* function is listed in Algorithm 13. This algorithm takes as input the current basic block under consideration *currBB* and if possible inserts a new scheduling step in *currBB* and returns this new scheduling

Algorithm 13 : **BalanceBranches**(G_{HTG}, $currBB$)
Returns: New scheduling step *newStep*
Balance branches by inserting scheduling steps in the shorter conditional branch

```
 1: $newStep = \emptyset$
 2: if (|SUCCS(currBB)| = 1) then      /* If currBB has only one successor bb */
 3:    succBB ← NextTrue(currBB)
 4:    if (IsJoin(succBB) = true) then      /* if succBB is a join bb */
 5:          currBranchHTG ← GetCurrentBranch($G_{HTG}$, $currBB$)
 6:          compBranchHTG ← GetComplementBranch($G_{HTG}$, $currBB$)
 7:       if (compBranchHTG is scheduled) then
 8:          if (NumOfStepsInBBs(currBranchHTG) <
               NumOfStepsInBBs(compBranchHTG)) then
 9:             newStep ← CreateNewStepInBB(currBB)
10:          end if
11:       end if
12:    end if
13: end if
14: return newStep
```

step. The algorithm starts by determining if *currBB* has only one successor and if that successor is a join basic block (lines 2 to 4). This means that *currBB* is part of a conditional block (If-HTG). The algorithm then gets the compound HTG node that *currBB* is in, namely, *currBranchHTG,* and the compound HTG node in the complementary or mutually exclusive branch of the If-HTG, namely, *compBranchHTG.* This is done by calling the functions, *GetCurrentBranch* and *GetComplementBranch* respectively (not given here). If *compBranchHTG* is already scheduled and if the number of scheduling steps in the longest path through *compBranchHTG* is larger than the corresponding scheduling steps through *currBranchHTG,* then the algorithm creates and inserts a new scheduling step in *currBB* by calling the function *CreateNewStepInBB* (lines 7 to 10 of Algorithm 13). The functions to get the number of steps in a branch of a If-HTG and to create a new scheduling step are not given here.

Note that, we consider only a scheduled *compBranchHTG,* in order to have an accurate picture of the resource utilization in the basic blocks in the mutually exclusive branch of the If-HTG before adding any more scheduling steps to the design. The number of steps in a basic block may change after scheduling because of code compaction, i.e., several operations in the basic block may be compacted into a few steps and executed concurrently.

Hence, the *BalanceBranches* function presented above can be called by the IR walker when it has finished scheduling the last basic block in the last branch of an If-HTG to be scheduled. Similarly, during code motion, when an operation is being duplicated into multiple basic blocks and the *TrailSynth* algorithm cannot find an idle resource in the last basic block *(lastBBInTrail)* of a trail, then too the

BalanceBranches function is called to insert a new scheduling step in *lastBBInTrail* (see Algorithm 8).

The *IsTrailblazeOpPossible* function calls a variant of the *BalanceBranches* function (see Algorithm 7). This variant, *IsBranchBalancingPossible,* is similar to the *BalanceBranches* function in that it checks if a basic block is part of an unbalanced If-HTG and therefore, if a new scheduling step can be inserted into this basic block. But the *IsBranchBalancingPossible* function does not actually insert the scheduling step, since its calling function *IsTrailblazeOpPossible* is only evaluating whether an operation can be duplicated and accommodated into a basic block.

Branch balancing is a powerful technique that enables code motions such as conditional speculation without increasing the length of the longest path through a conditional block. This can lead to shorter schedule lengths for the design and improve resource utilization in the conditional block. Also, if profiling information is available, this technique can be modified so that it adds new scheduling steps only in basic blocks on control paths that are *less* likely to be taken.

7.8 An Illustrative Example of the Scheduler

In this section, we will walk through an example to understand how the scheduling heuristic works and particularly to show how code motions are employed by the heuristic. Consider the example in Figure 7.4(a) and consider that the resources allocated to schedule this design are one adder and one subtracter. The first node in the design of this example is basic block bb_1. The scheduler starts by scheduling on the adder in basic block bb_1. The available operations list will contain the operations 1 and 2. Lets say that among these available operations, operation 1 from basic block bb_3 is chosen for scheduling. The code motion technique, *TrailSynth,* determines that it has to speculate this operation in order to schedule it in bb_1. The resultant design is shown in Figure 7.4(b). In this figure, operation 1 is speculatively executed as operation 4 in bb_1 and the result of operation 4, variable *A,* is still written back to variable a in operation 1 in basic block bb_3. This is to ensure that variable a gets updated with the result *A* only if the condition *cond*1 evaluates to "true".

Next, the scheduler receives basic block bb_3 to schedule from the design traversal algorithms in Algorithm 12 (since bb_2 does not have any operations in it). The only operation available for scheduling in bb_3 is operation 3. However, this operation requires conditional speculation to be scheduled in basic block bb_3 and we only allow operation duplication when the basic block in which the operation is being duplicated has already been scheduled (see the function *IsTrailblazeOpPossible* in Algorithm 7). Since basic block bb_4 has not yet been scheduled, operation 3 is not included in the available operations list. Thus, only the copy operation, operation 1, is scheduled into basic block bb_3.

The scheduler receives basic block bb_4 to schedule next, as per Algorithm 12. First, operation 2 is scheduled on to the adder. This time for the subtracter, the available list contains operation 3, since now the basic block that it will be duplicated into, namely bb_3, has already been scheduled. Hence, operation 3 is conditionally speculated as operations 5 and 6 in basic blocks bb_3 and bb_4. The resultant design is as shown in

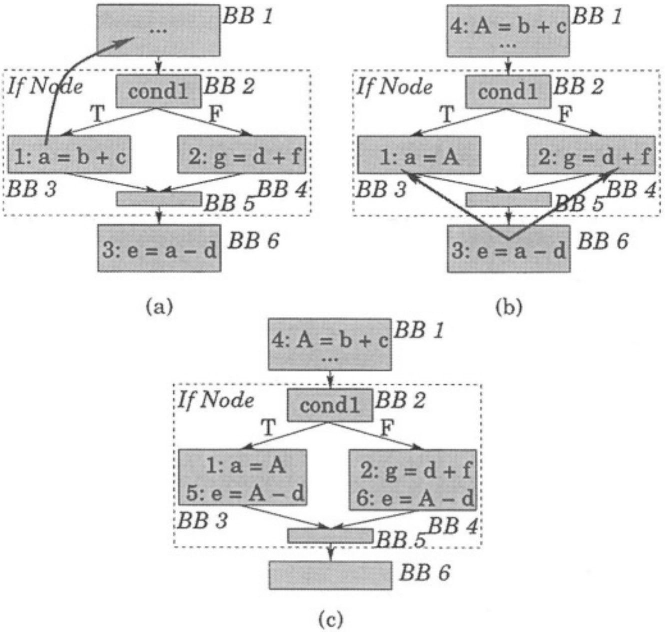


Figure 7.4: ***(a) HTG representation of an example (b) Operation* 1 *is speculatively executed as operation* 4 *in* bb_1 *(c) Operation* 3 *is conditionally speculated into basic blocks* bb_3 *and* bb_4*. Variable* a *is dynamically renamed with the speculatively calculated value* A *in operation* 5.**

Figure 7.4(c). In basic block bb_3, operation 5 directly uses the speculatively calculated value A of operation 1 by employing dynamic renaming (see Section 4.2.2). Finally, since basic blocks bb_5 and bb_6 are empty and there are no more unscheduled operations, the scheduling heuristic terminates.

This illustrative example demonstrates the working of the scheduling heuristics presented so far. In the next section, we show how synthesis transformations such as chaining can be incorporated into the various algorithms of the scheduling heuristic.

7.9 Incorporating Chaining into the Scheduler

To incorporate chaining into the scheduler framework, we have to modify the main scheduling heuristic, the available operations algorithm, and the code motion algorithm *(TrailSynth).* These modifications are discussed in the next two sections.

7.9.1 Incorporating Chaining into the Scheduling Heuristic

Chaining can be incorporated into the scheduling heuristic as shown in Algorithm 14. This modified priority-based list scheduling heuristic keeps track of not only the

Algorithm 14 : **PriorityListScheduling(** $\mathcal{DG}, \mathcal{R}, CMs$**)**
/* Incorporating Chaining into the Priority-based List Scheduling Heuristic */
/* If scheduling on a step with chaining enabled fails, then the same step is scheduled again without chaining enabled */

```
1:  $Pr \leftarrow$ CalculatePriority( $G_{DFG}$ )
2:  $prevStep \leftarrow step \leftarrow$ FirstStep ( FirstBB($G_{CFG}$) )
3:  while $step \neq \phi$ do
4:     $doChaining \leftarrow$ false
       /* Determine whether to do chaining */
5:     if ($BB_{Steps}(step) \neq BB_{Steps}(prevStep)$) then
6:        $doChaining \leftarrow$ true
7:        $chainingSteps \leftarrow$ GetChainSteps ($step, G_{CFG}$)
8:     end if
9:     for all resources res $\in \mathcal{R}$ do
10:       /* Can chaining be done on res, i.e., is res idle in chainingSteps */
11:       if (($doChaining$ = true) and (IsResUsed(res, $chainingSteps$) = true)) then
12:          Continue to next resource in foreach loop
13:       end if
14:       $\mathcal{A} \leftarrow$ GetAvailableOperations($\mathcal{DG}, step, res, CMs, chainingSteps$)
15:       if($\mathcal{A} \neq \emptyset$) then
16:          CalculateCostOfAvailOps ($\mathcal{A}, Pr$)
17:          Pick Operation $op \in \mathcal{A}$ with lowest cost
18:          TrailblazeOp($op, res, step, G_{HTG}, G_{CFG}, CMs, \mathcal{R}$)
19:          Mark $op$ as scheduled
20:          ApplyDynamicCSE($\mathcal{A}, op, \mathcal{DG}$)
21:       end if
22:    end for
23:    $prevStep \leftarrow step$
24:    $step \leftarrow$ GetNextSchedulingStep($G_{HTG}, step$)
25: end while
```

current scheduling step, but also the previous scheduling step, *prevStep*. Chaining across conditional boundaries is attempted if the current scheduling *step* is in a different basic block than that of *prevStep* (lines 5 and 6 in Algorithm 14). This is because chaining of operations within a basic block, i.e., with no control flow between them, is done within the same scheduling step.

When chaining across conditional boundaries is enabled, i.e., *doChaining = true,* the scheduler calls the function *GetChainSteps* to get all the steps in previous basic blocks that the current step has to be chained with (*chainingSteps*). The *getChainSteps* function (not shown here) traverses back up all the control paths in G_{CFG} leading up to the current basic block, looking for steps scheduled in the same cycle as the current scheduling step. If chaining is not enabled, then *chainingSteps* is empty.

With chaining enabled, the scheduler skips over any resource *res* in the resource list that is used in any of the steps in *chainingSteps* (lines 11 to 13 in Algorithm 14). It

```
Algorithm 15 : GetAvailableOperations($\mathcal{DG}$, $step$, $res$, $CMs$, $chainingSteps$)
Returns: Available Operations List $\mathcal{A}$
/* Incorporating Chaining into the available operations algorithm: lines 6 to 11 are the
chaining-specific additions */

    /* Find all unscheduled operations in $G_{CFG}$ that can be scheduled on $res$ */
 1: CollectUnscheduledOps($res$, $step$, $BB_{Steps}(step)$, $G_{DFG}$, $G_{CFG}$, $\mathcal{A}$)
    /* Remove all operations that cannot be moved to $step$ using $CMs$ */
 2: for all $op_i \in \mathcal{A}$ do
 3:   if (IsTrailblazeOpPossible($op_i$, $step$, $G_{HTG}$, $G_{CFG}$, $CMs$, $\mathcal{R}$) = false) then
 4:     $\mathcal{A} \leftarrow \mathcal{A}$ - $op_i$
 5:   end if
 6:   if ($chainingSteps \neq \emptyset$) then
        /* Can $op_i$ be chained with operations in $chainingSteps$ */
 7:     $TotalRunTime \leftarrow T_{res}(res) + \sum$ Execution time of ops in longest
                                              dependency chain of $op_i$ in $chainingSteps$
 8:     if ($TotalRunTime > C_{CLK}$) then
 9:       $\mathcal{A} \leftarrow \mathcal{A}$ - $op_i$
10:     end if
11:   end if
12: end for
13: return $\mathcal{A}$
```

employs the function *I sResU sed* to determine if a resource is used in a set of steps. If a resource is available for scheduling, then the scheduling heuristic proceeds as before and schedules the operation with the lowest cost from the list of available operations (if any). However, although not shown in this algorithm, if the scheduling heuristic fails to schedule anything on *step* with chaining enabled, then it tries to schedule on *step* again, albeit without chaining; that is, the *foreach* resource loop is repeated with *doChaining = false.*

The *Available Operations* algorithm also requires a modification to enable chaining across conditional boundaries as shown by the updated algorithm in Algorithm 15. If *chainingSteps* is not empty, then this algorithm determines the operations in the *chainingSteps* (if any) that each operation op_i in $\mathcal{A}$ is dependent on. The execution time of the longest chain of these dependency operations is determined and summed with the execution time of the resource being scheduled, *res.* If this *TotalRunTime* is larger than the clock period allocated to the design, then op_i cannot be scheduled in *step* since there exists a chain of operations in *chainingSteps* that will not finish executing in time for op_i to execute as well. This is given in lines 6 to 11 of the algorithm in Algorithm 15.

The final modification required in the scheduling framework to enable chaining operations across conditional boundaries is in the code motion algorithm (*TrailSynth*) as explained in the next section.

Algorithm 16 : ChainOpWithPrevSteps (*op*, *stepInBB*)
/* Chains operation *op* with the operations in the scheduling steps that are scheduled in the same cycle as *stepInBB*. Also inserts wire-variables into all the chaining trails */

```
1:  chainingSteps ← GetChainSteps(stepInBB, $G_{CFG}$)
2:  chainingTrailList ← GetChainingTrails(stepInBB, $G_{CFG}$)
3:  LeftWv and RightWv ← New wire-variables for left and right operands of op
4:  for all chainTrail in chainingTrailList do
5:     depOpList ← FindDependentOps(op, chainTrail)
6:     for all depOp in depOpList do
7:        if (depOp writes to left operand of op) then
8:           Wv ← LeftWv
9:        else   /* depOp writes to right operand of op */
10:          Wv ← RightWv
11:       end if
12:       Make depOp write to wire-variable Wv
13:       Insert copy operation from Wv to original result variable of depOp in its
          scheduling step
14:    end for
15: end for
```

7.9.2 Incorporating Chaining into the *TrailSynth* Code Motion Technique

Chaining is incorporated in the *TrailSynth* algorithm presented earlier in Algorithm 8 by checking if the scheduling step (*stepInBB*), in which the operation is inserted, is chained across conditionals. This is done for the scheduling steps both in the target basic block and in basic blocks where the operation is duplicated. If *stepInBB* is chained across conditionals, then the function, *ChainOpWithPrevSteps,* outlined in Algorithm 16 is called. This function inserts wire-variables into all the chaining trails (*chainingTrailList*) that lead up to the *stepInBB.* As explained in Section 6.6.1, chaining trails consist of the basic blocks that have a control-flow path to the basic block of the current scheduling *step* and have a scheduling step scheduled in the same cycle as the current scheduling *step.* The list of chaining trails are obtained by calling the function *GetChainingTrails*; this function is not presented here.

The chaining heuristic in Algorithm 16 first creates the new wire-variables, *LeftWv* and *RightWv,* for the left and right operands respectively, of the operation (***op***) being scheduled (line 3). Then, for each chaining trail, *chainTrail,* in the list of chaining trails, *chainingTrailList,* the heuristic calls the function *FindDependentOps* (lines 4 and 5). This function finds the list the operations (*depOpList*) in *chainTrail* that the current operation ***op*** has a dependency with. Each dependent operation, *depOp,* in this list is checked and if *depOp* writes to the left operand of ***op*****,** then the result of *depOp* is written to *LeftWv* instead. Conversely, if *depOp* writes to the right operand of ***op*****,** its result is written to *RightWv* instead. Also, a copy operation from *LeftWv* or *RightWv* to the original variable that *depOp* wrote to, is inserted after *depOp* in

Algorithm 17 : ShiftLoopBody(LoopNode)
/* Loop Shifting Algorithm: shifts one scheduling step in a loop body */

```
firstBB ← FirsiBB(LoopNode→ loopBody)
stepToShift ← FindStepToShift(firstBB)
BB(stepToShift) ← BB(stepToShift) - stepToShift
lastBB ←LastBB(LoopNode→loopBody)
lastBB ← lastBB ∪ stepToShift
loopHeadBB ← LastBB(LoopNode→loopHead)
loopHeadBB ← loopHeadBB ∪ Copy(stepToShift)
Reschedule(LoopNode)
```

the same scheduling step as *depOp*. This procedure is outlined within the *foreach* loop in lines 6 to 14 in Algorithm 16.

In this way, this chaining heuristic operates on the scheduling step, *stepInBB,* of each trail that the scheduled operation is duplicated into or moved into and inserts write operations to wire-variables as needed. Note that, if the *depOpList* is empty for any chaining trail in *chainingTrailList,* the chaining heuristic will insert two simple copy operations. One operation copies the original left operand of ***op*** to its left operand wire-variable (*LeftWv*) and the second operation copies the original right operand of ***op*** to *RightWv* (as explained earlier in the example in Figure 6.13). Hence, all the chaining trails will now have writes to these wire-variables and the scheduled operation ***op*** reads the wire-variables instead of its original operands. A dead code elimination pass performed after scheduling can determine if any of these copy operations can be eliminated.

7.10 Loop Shifting Algorithm

As explained in Chapter 6, loop shifting is an incremental loop pipelining technique that operates at the end of the scheduling phase. Loop shifting moves a set of concurrent operations from the end of the loop body to the beginning of the loop body and makes a copy in the loop header. It then reschedules the loop body. Our loop shifting algorithm is listed in Algorithm 17. This algorithm takes the loop HTG node to be shifted as input *(LoopNode)* and shifts one scheduling step from the beginning of the loop body to its end.

Note that the sub-parts (loop head, body, and tail) of a loop can be accessed easily in the HTG representation by referring to the members *loopHead, loopBody* and *loopTail* of *LoopNode.* Recall that the loop head and loop tail each contain one basic block, whereas the loop body is a compound HTG node that may contain other compound nodes including if-then-else blocks and other loops.

The loop shifting algorithm starts by looking for a scheduling step to shift. To do this, it calls the function *FindStepToShift* with the first basic block in the loop body as argument. This function, listed in Algorithm 18, is a recursive traverses the loop body starting at *currBB*. It first calls the function *FirstNonCondStep* for *currBB*. This function returns a NULL step if *currBB* is empty (due to past shift

Algorithm 18 : FindStepToShift(*currBB*)
Returns: The scheduling step to shift
/* Recursive function that returns a step to shift from the longer conditional branch */

```
stepToShift ← FirstNonCondStep(currBB)
if ($stepToShift = \emptyset$) then
  Find $nextBB \in SUCCS(currBB)$ with the maximum NumSteps(nextBB)
  stepToShift ← FindStepToShift(nextBB)
end if
return stepToShift
```

operations) or if *currBB* only has scheduling steps with conditional Boolean checks. If *FirstNonCondStep* does not find a scheduling step, the *FindStepToShift* function recursively traverses the basic blocks in the loop body till it finds a scheduling step in one of them. If a basic block has several successor basic blocks (branches), the algorithm traverses to the branch with the larger number of scheduling steps.

Once the *FindStepToShift* function returns a scheduling step *stepToShift,* the *ShiftLoopBody* algorithm removes this step from its basic block and adds it to the last basic block in the loop body (lines 3 to 5 in Algorithm 17). A copy of *stepToShift* is also added to the loop head (lines 6 and 7). We then reschedule the loop by calling the function *Reschedule.* Note that, by adding or removing a scheduling step, we mean that the operations in that step are added or removed from a basic block.

In the worst case, this algorithm may end up traversing all the basic blocks in a loop. In practice, it usually traverses not more than 2 to 3 basic blocks. Rescheduling the loop, on the other hand, can be computationally expensive. However, in practice, only the shifted operations have to be repacked in the schedule. In our experiments, we find that the run times of our synthesis tool, on a 1.6 Ghz PC running Linux, range from 1-3 usecs (user seconds) with no loop shifting, 2 to 6 usecs with loop shifting and 5 to 10 usecs with loop unrolling. In contrast, the logic synthesis tool takes between 2 to 8 *hours* for synthesizing these designs.

7.11 Summary

In this chapter, we presented the various scheduling algorithms that direct the application of the transformations in our parallelizing high-level synthesis methodology and have been implemented in the *Spark* framework. We first presented the software architecture of the scheduler in Section 7.1. We then presented the various algorithms that form part of this architecture. In Section 7.2, we presented the main list scheduling heuristic that calls all the other algorithms in the scheduler. The algorithm to collect the list of available operations was presented in Section 7.3, followed by the *TrailSynth* algorithm in Section 7.4. *TrailSynth* is the *Trailblazing* algorithm modified for high-level synthesis. We then presented the dynamic CSE algorithm in Section 7.5. The design traversal algorithms were presented in Section 7.6. This was followed by an algorithm for dynamically balancing conditional branches during scheduling in Section 7.7. We then walked through an example to demonstrate how the various scheduling algorithms work. In Section 7.9, we presented the enhancements required in the scheduling algorithms to incorporate chaining of operations across conditional boundaries. Finally, in Section 7.10, we presented an algorithm for loop shifting that incrementally exposes parallelism across loop iterations and reschedules the loop body after pipelining. The contributions of this chapter are a set of algorithms and heuristics that judiciously guide the various parallelizing compiler and synthesis transformations, so as to improve the quality of synthesis results.

8

RESOURCE BINDING AND CONTROL SYNTHESIS

8.1 Introduction

After a design has been scheduled, a resource binding pass maps the operations and variables in the design to functional units and registers respectively. This is followed by a control synthesis pass that generates the controller implementing the schedule as well as the select signals for the multiplexers connected to the functional units. In this chapter, we describe our resource binding methodology that minimizes the complexity of the interconnect, and a control synthesis pass that generates a controller based on a finite state machine style.

8.2 Resource Binding

Based on the number of functional units allocated to the design, resource-constrained scheduling determines the control step during which each operation in the design graph executes. A *resource binding* pass maps the operations in each control step to specific functional units and the variables in the design to registers.

The resource binding problem can be defined as [KM92]:

Definition 8.1. *Given a resource allocation* α *of the resource list* $\mathcal{R}$ *as defined by Definition 3.9, a* resource binding *of a design graph* $\mathcal{DG}$ *is a mapping* $\beta : Vops \mapsto (\mathcal{R} \times Z^{+})$*, where* $\beta(vop_i) = (res_k, j)$ *if operation* $vop_i \in Vops$ *is being executed by the j-th instance of resource type* $res_k \in \mathcal{R}$, $1 \leq k \leq \alpha(res_k)$*. Otherwise,* $\beta(vop_i)$ *is undefined.*

However, the choice of operation to functional unit and variable to register binding has a profound effect on the interconnect of the design [GDWL92]. It is possible to reduce interconnect by careful resource binding, as explained in the next section.

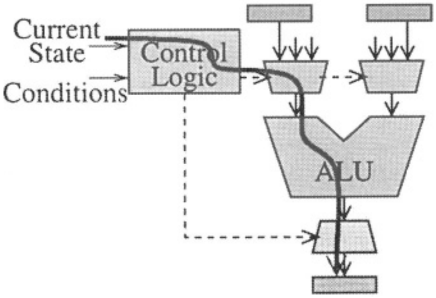


Figure 8.1: ***Typical critical paths in control-intensive designs pass through the control logic, the steering logic, the functional units and terminate in registers.***

8.2.1 Interconnect Minimization by Resource Binding

The speculative nature of the code motions – particularly conditional speculation that duplicates and executes operations conditionally that would have otherwise executed unconditionally – increases the complexity of the multiplexers and associated control logic by many-fold. The control logic selects the inputs to the functional units based on the current state of the finite state machine (FSM) and the evaluation of the conditions. We collectively refer to the multiplexers, de-multiplexers and the control logic as *steering logic* or *interconnect.*

The increase in the complexity of the interconnect leads to an increase in the total design area. Also, the longest combinational paths in the design or the *critical paths* often pass through the steering logic. An example of a typical critical path in a design is shown in Figure 8.1. The critical path originates in the control logic that generates the select signals for the multiplexers and passes through the multiplexers, the functional unit, the de-multiplexers and terminates in a register.

In a bid to control the increase in interconnect complexity, we developed an interconnect minimization strategy based on resource binding. This resource binding methodology first binds operations with the same inputs or outputs to the same functional unit. The variable to register binding then takes advantage of this by mapping variables that are inputs or outputs to the same functional units to the same register. This reduces the number of registers connected to the inputs and outputs of functional units, thereby, reducing the size of the multiplexers and de-multiplexers connected to them. We describe this methodology in the next two sections.

8.2.2 Modeling Interconnect Minimizing Resource Binding

To present the interconnect minimizing resource binding problem, we first define variable lifetimes as follows:

Definition 8.2. *We are given a list of variables, $V_{var} = \{var_i;\ i = 1, 2, ..., n_{vars}\}$ that are read and written by each operation $vop_j \in Vops$ in G_{DFG}. For each variable, $var_i \in V_{var}$, we create a list of 2-tuples of the birth time $t_{birth}(var_i)$ and death time,*

$t_{death}(var_i)$ of the variable. A new 2-tuple is created every time the variable is written: $t_{birth}(var_i)$ is time step when it is written and $t_{death}(var_i)$ is the last time step in which its value is read (before being written again). The lifetimes of a variable are then intervals defined by the difference between all the death and birth times of the variable. Thus, for each 2-tuple, the lifetime of the variable is $t_{lifetime}(var_i) = t_{death}(var_i) - t_{birth}(var_i)$. Similarly, for variables that are read across iterations of a loop, we create two lifetimes that span from the time step of creation of the variable in the loop till the end of the loop and then from the beginning of the loop till the time step of the last read of the variable.

The interconnect minimizing resource binding problem can be defined as:

Definition 8.3. *We are given a data flow graph, $G_{DFG}(Vop, E_{data})$ and corresponding control flow graph, $G_{CFG}(Vbb, E_{control})$, such that the operations in the data flow graph have been scheduled as per Definition 3.18. We are also given a set of resources Res of n_{res} resource types whose numbers are upper bound by $\{a_k;\, k = 1, 2, ..., n_{res}\}$. The scheduled design can be used to determine the lifetime of the variables as defined in Definition 8.2. The resource binding problem can then be defined as:*

- ❐ *Find a mappingfor the set ofconcurrent operations in each time step in each basic block $BB = \{vbb_i;\ i = 1, 2, ..., n_{bbs}\}$ in G_{CFG} to the resources in $\mathcal{R}$, such that the number of inputs to the multiplexers connected to the resources are minimized.*

- ❐ *Similarly, find a mapping for each variable that is written in the data flow graph G_{DFG} to a register, such that variables that have overlapping lifetimes are not mapped to the same register and the number of inputs to the multiplexers connected to the resources are minimized.*

Note that, operations and variables that are mutually exclusive can be bound to the same functional unit and register respectively. Also, the scheduled design used as the starting point in the above definition already ensures that the number of operations of any resource type, $\{k = 1, 2, ..., n_{res}\}$ does not exceed the number of resources of that type, a_k.

8.2.3 Resource Binding: An Illustrative Example

The interconnect required to connect functional units to each other and to registers can be reduced by combining operations that have the same inputs and/or same outputs. This leads to fewer registers connected to the ports of the functional units and hence, fewer inputs to the multiplexers. To understand this, consider the classical example of a design and one possible hardware implementation, shown in Figures 8.2(a) and (b) [GDWL92]. The hardware shown in this figure is obtained by randomly binding operations 1 through 4 to functional units *A*0 and *A*1. However, we can do better. If we exchange the functional units that operations 3 and 4 are bound to, we can eliminate a multiplexer from one of the inputs of each functional unit, as shown in Figure 8.3(a). The intuition behind this exchange is that operations 1 and 4 have the input variable c in common and hence, binding them to the same functional unit means that just one

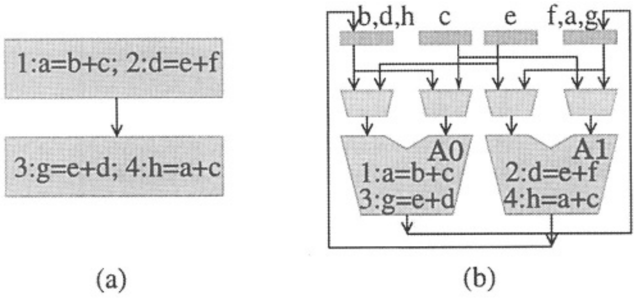


Figure 8.2: ***(a) Two consecutive sets of concurrent operations (b) An example of binding that leads to a large number of interconnections.***

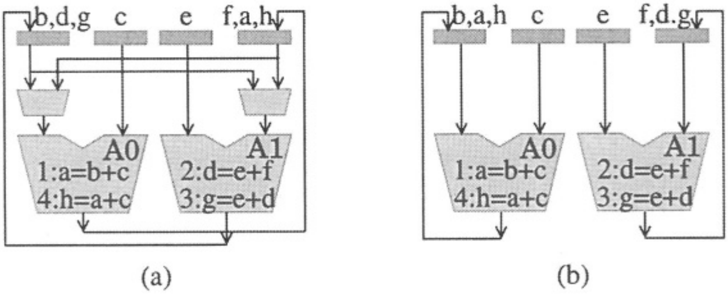


Figure 8.3: ***Reducing interconnect by improved (a) operation binding (b) variable binding.***

register can be used to store the input. The same is true for operations 2 and 3 that have variable e in common. Recall that two operations can be mapped to the same functional unit if they are mutually exclusive or execute in different control steps.

Variables that are input or output to the *same* port of a functional unit can be bound to the same register, provided their lifetimes do not overlap or they are mutually exclusive. This further reduces the number of registers connected to the ports of a functional unit, hence, leading to smaller interconnect at the port. Thus, for the example in Figure 8.3(a), variables b and a are inputs to the same port of functional unit $A0$ and similarly, variables f and d are inputs to same port of $A1$. Hence, b and a can be bound to the same register as can f and d. Similarly, we can switch the binding of the output variables g and h. The resulting hardware is shown in Figure 8.3(b).

In the next two sections, we formulate solutions for the interconnect minimizing operation to functional unit and variable to register binding problems.

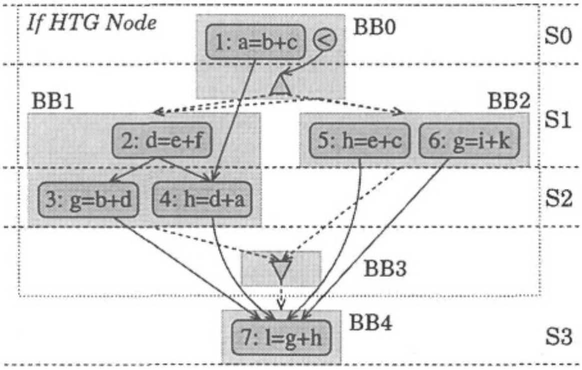


Figure 8.4: ***A scheduled design used to demonstrate the operation and variable binding formulations***

8.2.4 Operation to Functional Unit Binding

The operation binding problem defined in Definition 8.3 can be summarized as: given a scheduled design, *HTG,* and a set of resources, *Res,* map each operation to a functional unit in *Res,* such that the interconnect is minimized.

We formulate this problem by creating an *operation compatibility graph* for each type of resource in the resource list. A node is created in the graph corresponding to each operation in the design that can be mapped to the resource type under consideration. *Compatibility edges* are created between nodes corresponding to operations that are scheduled in either different control steps or execute under a different set of conditions. Note that, mutually exclusive operations (and their variables) scheduled in the same time step are compatible with each other.

Consider the example in Figure 8.4. This design has been scheduled with two single cycle adders and a single cycle comparator. The operation binding graph for the "adder" resource in this example is shown in Figure 8.5(a). Each node in this graph corresponds to an operation from the example. Solid edges denote compatibility edges and dashed edges denote incompatibility between operations. Two nodes are considered incompatible if they cannot be mapped to the same functional unit because they execute concurrently. In this example, operations 5 and 6 are incompatible because they both execute concurrently in basic block bb_2. Similarly, operations 3 and 4 are incompatible as well. All the other operations in this example are compatible with each other, since they execute either in different control steps or under different conditions.

Any set of cliques that forms a cover of this graph constitutes a valid binding of operations to functional units. However, since the number of resources are constrained, the number of cliques in the clique cover cannot exceed the number of functional units allocated to the design. Furthermore, to bias the operation to functional unit binding for reducing interconnect, we introduce weights on the edges. Initially, all the edges are assigned a weight of zero. We then add additional edge weights between operations for each instance of common inputs or outputs between them, so as to make them more

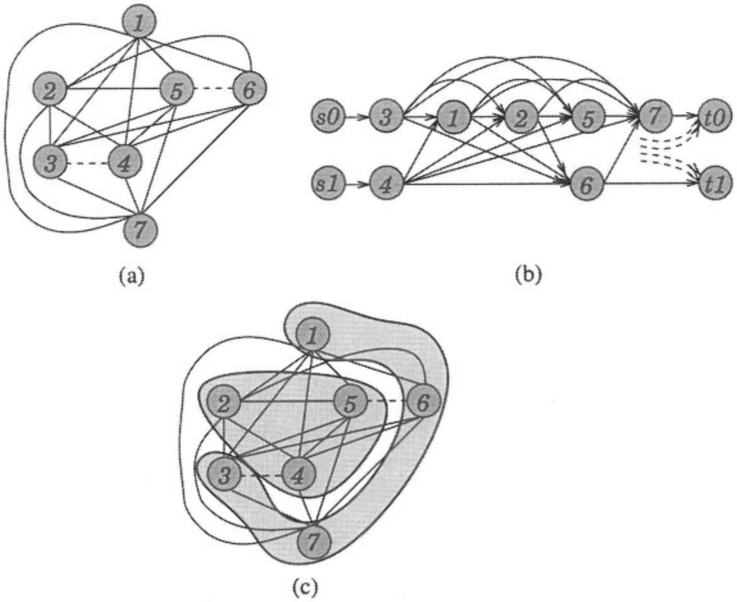


Figure 8.5: ***(a) Operation compatibility graph of the design in Figure 8.4 for the adder resource (b) Corresponding multi-commodity network (c) A clique cover obtained by finding the max-cost flow through the network.***

likely to be bound to the same functional unit. The graph problem then becomes one of finding a *maximally weighted* clique cover of this weighted compatibility graph. To solve this problem, we formulate it as a multi-commodity network flow problem [TTC90]. A *max-cost flow* through this network represents a valid maximally weighted clique cover [CP96, Sto92].

To convert the operation compatibility graph into a multi-commodity network graph, we first create one *source* node and one *sink* node for each instance of the resource type under consideration. Hence, for the compatibility graph in Figure 8.5(a), we create two source and two sink nodes for each of the two adders allocated to this design. We then pick a control step in the schedule that uses all the resources of the type under consideration. If there is no such control step, it implies that the resource allocation for the design is too high. In such a situation, the control step with the highest utilization of the current resource type is selected. From each source node, we add a directed edge to one operation in this control step, such that each operation in the control step has only one incoming edge from a source node. Also we add edges from each operation node in the graph to both the sink nodes. All the edges in the graph are made directed in such a manner that there are no cycles in the graph (this is possible since the execution times of the operations form intervals). The resulting graph is shown in Figure 8.5(b). In this graph, for the sake of clarity, we did not show the edge weights and the edges from all the nodes to the sink nodes (t_0 and t_1).

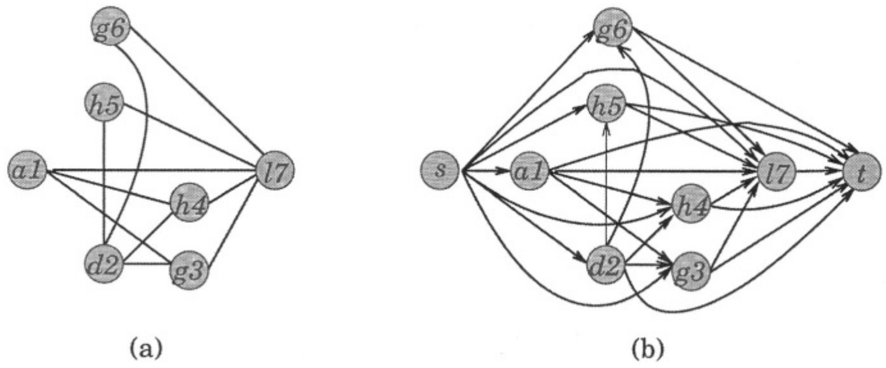


Figure 8.6: ***(a) Variable compatibility graph (b) Corresponding network flow graph***

A *max-cost flow* through this multi-commodity network then represents a valid maximally weighted clique cover for the operation compatibility graph [TTC90]. We determine this flow by negating all the weights in the graph and then finding a min-cost flow of value equal to the number of resources/source nodes. Nodes left uncovered are put into the compatible clique that leads to the maximum increase in total weight of the cover. This solution represents a valid operation to functional unit binding which minimizes interconnect. The resulting clique cover for the example from Figure 8.4 is shown in Figure 8.5(c): operations 1,3,6 and 7 are mapped to one adder and operations 2,4 and 5 to the other.

We note that Chang et al. [CP96] have used the same formulation for module allocation but their objective is to minimize power consumption. Stok [Sto92] has used a similar formulation for register allocation in order to eliminate superfluous data transfers between registers.

8.2.5 Variable to Register Binding

We perform variable binding after operation binding and thus, take advantage of the decisions made by operation binding phase to minimize the number of inputs to each port of the functional units. The variable to register binding problem is also formulated as a network flow problem, except that we do not place a constraint on the number of registers. This is because we are willing to allocate more registers in the interest of reducing interconnect.

We start by creating a *variable compatibility graph.* There is a node for each instance of a *write* to a variable in the design. If a variable is written twice, each write gets a new node in the graph. Compatibility edges are added between nodes corresponding to variables that (a) do not have overlapping lifetimes, and (b) whose lifetimes extend over mutually exclusive control paths. The *lifetime* of a variable is denoted by a *birth* time and a *death* time corresponding to when the variable is written and read respectively as explained in Definition 8.3. Variables in loops have multiple split lifetime intervals [Sto92]. The variable compatibility graph for the example in Figure 8.4 is shown in Figure 8.6(a).

We assign an initial weight of zero to each edge. Then edge weights are added between compatible variables for each instance of them being inputs or outputs to the same port of the same functional unit. A maximally weighted clique cover of this compatibility graph represents a valid variable to register binding that minimizes interconnect.

To find this clique cover, we formulate the problem as a *min-cost max-flow* network problem [TTC90]. A *source* and a *sink* node are added to this weighted compatibility graph. Directed edges are added from the source node to each variable node in the graph and from each variable node to the sink node. All the edges between the variable nodes are made directional in such a way so as to not create cycles. The resulting network flow graph for the example in Figure 8.6(a) is shown in Figure 8.6(b). The weights have been omitted from this figure for clarity.

On solving the network flow for the example in Figure 8.6(b), we get a resultant flow of 2. Variables a_1, g_3, g_6 and l_7 are bound to one register and d_2, h_4 and h_5 to another register. Chang et at. have used a similar approach to variable binding, albeit with the objective of reducing power [CP95]. Stok [SP91] used the same formulation for operation binding with an objective to minimize interconnect.

8.3 Control Synthesis in the PHLS Framework

To complete the high-level synthesis process, a control unit has to be synthesized that implements the schedule. This control unit generates the signals that drive the functional units, interconnect (multiplexers, de-multiplexers) and registers in the data path – the sequence of execution is as per the generated schedule.

Although there are several styles of controller architectures to choose from [DM94, KGM95], by far finite state machines (FSMs) are the most popular for digital design. This is also the controller architecture we have chosen in our methodology. We now describe the construction of the finite state machines from the scheduled HTG followed by the subsequent generation of register transfer level (RTL) VHDL. The output VHDL code can be synthesized using commercial logic synthesis tools such as Synopsys *Design Compiler* [DC].

8.3.1 State Assignment

Finite state machine generation from scheduled designs starts with *state assignment* of the operations. This is the process of assigning states to each scheduling step in which concurrent operations are scheduled. State assignment in purely data-flow designs is trivial. Each scheduling step is assigned a state of its own. However, in designs with control, first a *state transition graph* is generated based on the control flow in the scheduled HTG. Operations that execute during the same time, but in scheduling steps that are on mutually exclusive control paths, can either be assigned the same state or can be assigned a state based on the control path that they are in. The former method is known as *global* slicing and the latter as *local* slicing [TWR+88].

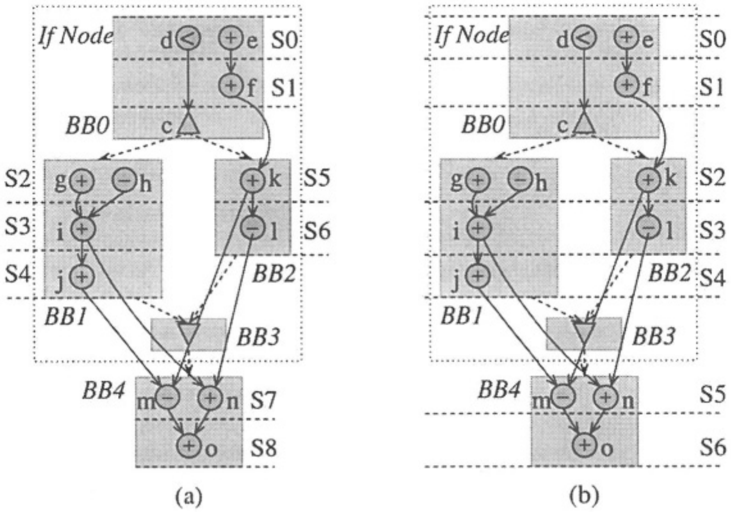


Figure 8.7: *State assignment by (a) Local slicing (b) Global slicing*

State assignment by these two techniques is demonstrated by an example in Figures 8.7(a) and (b). In these figures, the states are demarcated by dashed lines and marked as $S0$, $S1$, $S2$ and so on. The *local slicing* method of state assignment has been used for the example in Figure 8.7(a). Here, each set of concurrent operations in a scheduling step is viewed as a single slice and assigned a unique state. Hence, operations g and h in basic block bb_1 are assigned state S_2 and operation k in basic block bb_2 is assigned state S_5. However, in global slicing, concurrent operations in scheduling steps that are on mutually exclusive conditional branches but which execute in the same time step in the scheduled design, are assigned the same *global slice* and hence, the same state. Figure 8.7(b) presents the state assignment for the same example using global slicing. With global slicing, operations g and h in basic block bb_1 and operation k in bb_2 are all assigned the same state S_2.

Since global slicing assigns states across mutually exclusive control paths, it results in fewer states in the control unit. However, *status registers* are required to store the information about which set of mutually exclusive operations to execute in the state. On the other hand, local slicing requires as many states as the sum of the time steps in each basic block, leading to larger state machines. However, no status registers are required.

Weng and Parker [WP92] have done a comprehensive study on these two types of controllers and found that using global slicing with status registers always produces designs with lower area. Our own experiments support these results. We find that the larger number of states required for local slicing leads to poorer finite state machine optimization. Furthermore, since local slicing executes all sets of concurrent operations in different states, the mutual exclusivity information that can be potentially used for further interconnect optimization is lost to the logic synthesis tool.

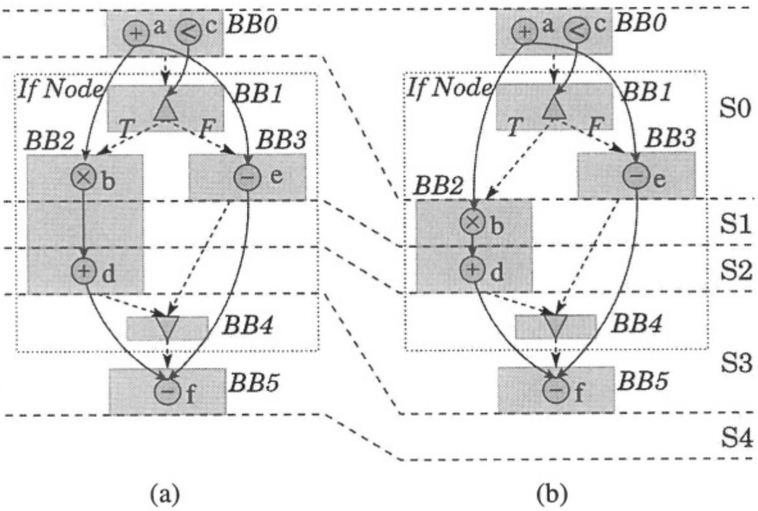


Figure 8.8: *(a) FSM state assignment for a sample scheduled HTG. Multiplier is 2-cycle and all other resources are single cycle. (b) New FSM state assignment for the same example but cycle time has been doubled. Operation e is chained with operation a across the conditional check of c.*

8.3.2 Modeling the Finite State Machine Controller

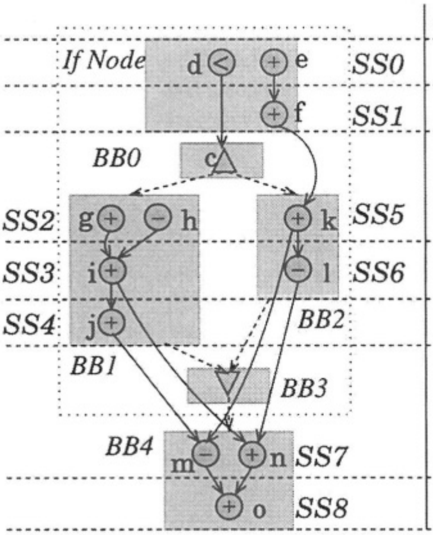
A finite state machine can be represented by a quintuple [GDWL92, DM94, HP81]:

$$< \mathcal{S}, I, O, f : \mathcal{S} \times I \rightarrow \mathcal{S}, h : \mathcal{S} \times I \rightarrow O > \tag{8.1}$$

where $\mathcal{S} = \{S_i\}$ is a set of states, $I = \{i_j\}$ is a set of input values and $O = \{o_k\}$ is a set of output values. f and h are next-state and output functions that map a cross product of $\mathcal{S}$ and I into $\mathcal{S}$ and O, respectively. This means that given a current state from $\mathcal{S}$ and an input value from I, the function f produces the next state in $\mathcal{S}$. Similarly, the function h produces an output in O using the same inputs. We also use the notion of an initial state $S_0 \in \mathcal{S}$ that the FSM can be initialized to by a synchronous reset.

Our FSM generation algorithm produces a state assignment for each set of concurrent operations, i.e., for each scheduling step. The start and end times of each step have already been determined by the scheduler. The algorithm also determines the state transitions between the states based on the current state of the controller and the active Boolean conditions. Hence, in the FSM quintuple above, the function f uses the current state (from $\mathcal{S}$) and the set of active conditions (Boolean checks) as the input I, and produces the next state. Similarly, the output function h reduces to the set of operations (scheduling steps) to be executed based on the current state and current set of active conditions. We describe a back-end code generator that generates VHDL conforming to this model in Section 8.3.4.

To understand the impact of scheduling and chaining on state assignment, consider the scheduled HTG in Figure 8.8(a). In this example, the adder and the subtracter take



(a)

States	Sched Step, Conds, Next SStep
S0	<SS0,1, SS1>
S1	<SS1,1, SS2>
S2	<SS2, c, SS3>, <SS5, !c, SS6>
S3	<SS3, c, SS4>, <SS6, !c, SS7>
S4	<SS4,c, SS7>
S5	<SS7,1, SS8>
S6	<SS8,1, -- >

(b)

Figure 8.9: ***(a) An example of a scheduled hierarchical task graph (b) State table containing the scheduling steps that execute in each state as a 3-tuple: <Scheduling Step, Conditions, Next Scheduling Step>***

one cycle to execute while the multiplier takes 2 cycles to execute. Hence, no operation is executed in state S_2 in basic block bb_2 because of the multi-cycle multiply operation b. Now, if we double the clock period allocated to this design and reschedule, we obtain the scheduled HTG shown in Figure 8.8(b). In this HTG, operation e is chained with operation a, since the add and subtract now execute in half a cycle (we ignore multiplexing time). Hence, their scheduling steps are chained across the conditional check and execute in the same state, S_0. Also, the multiplier now executes in one cycle in state S_2 in basic block bb_2.

8.3.3 Finite State Machine Construction

The finite state machine construction algorithm traverses the design in a top-down manner starting from the *Start* node of the top-level design HTG. It assigns a 3-tuple to each scheduling step (i.e. set of concurrent operations) in each basic block. This 3-tuple comprises of a pointer to the scheduling step, the set of conditions under which the step executes and a pointer to the next scheduling step. A *state table* stores the list of all the scheduling steps that execute in each state in the form of these 3-tuples. States are assigned to the scheduling steps by using the global slicing method. The state assigned to the first scheduling step in a basic block is one more than the larger of the states assigned to the last scheduling steps in all its preceding non-empty basic blocks.

The next scheduling step in a basic block is the step after the current scheduling step in the basic block. For the last scheduling step of a basic block, the next scheduling step is the first non-empty scheduling step in a subsequent basic block. If the last scheduling step in a basic block is a conditional check, then no entry is made in the state table and no 3-tuple is created for it. Conditional checks always appear in a scheduling step of their own. The next scheduling step of the last scheduling step of a basic block is empty (ϕ) if the basic block has no successor basic blocks, i.e., if it is last basic block in the design HTG.

We demonstrate the 3-tuple assignment by using the example in Figure 8.9(a). This example is a scheduled HTG with scheduling steps $SS0$ to $SS8$ in basic blocks bb_0 to bb_4. The FSM construction algorithm traverses the design starting at the first basic block bb_0 of the design level HTG and assigns a 3-tuple corresponding to each scheduling step. The states $S0$ to $S6$ are assigned using the global slicing method described in the previous section. So, in bb_0, scheduling step $SS0$ executes in state $S0$. Hence, the state table in Figure 8.9(b) has an entry in state *S0:* $< SS0, 1, SS1 >$, where $SS0$ is the scheduling step, 1 implies no condition, and $SS1$ is the next scheduling step.

The last scheduling step in basic block bb_0 is a conditional branch. The actual branch condition has been calculated in an earlier time step and this is only a Boolean check. Hence, it is scheduled in state $S2$ along with the operations that will be executed based on this Boolean check, i.e., scheduling steps $SS3$ and $SS6$. For this reason, we also do not assign a 3-tuple to this conditional check. At the fork or conditional check of a conditional block, the state assignment algorithm first traverses the basic blocks on the "true" control path and then those on the "false" control path (the ordering of control path traversal does not matter). The scheduling steps in basic block bb_2 have a condition list consisting of $!c$. Note that, the next *non-empty* scheduling step for the last scheduling steps in basic blocks bb_1 and bb_2 is $SS7$ in basic block bb_4.

The algorithm then traverses to basic block bb_4. The first scheduling step in this basic block, $SS7$, is assigned the state $S5$. This is one state more than the larger of the states of the last scheduling steps in basic blocks bb_1 and bb_2. The last scheduling step in bb_4 has an empty (ϕ) next scheduling step.

The state table in Figure 8.9(b) now contains the list of all the scheduling steps that execute in each state, the conditions under which they execute and the next scheduling step (and therefore, state) to transition to. Note that, in our FSM construction, there are no idle states in the shorter of the two branches of conditional blocks. So in the example in Figure 8.9, after the execution of the last scheduling step $SS6$ in basic block bb_2 in state $S3$, the state machine skips over state $S4$ and directly goes to state $S5$. This ensures that if the shorter control path through a conditional is taken, then the design finishes execution faster.

An algorithm that constructs the finite state machine as described above is outlined in Algorithm 19. This algorithm iterates over the scheduling steps in a basic block and makes an entry in the state table for the 3-tuple of each scheduling step, the conditions under which the step executes and the next scheduling step to execute. The input to this algorithm on its first call is the first/start basic block of the design graph. After assiging states to the basic block, the algorithm recursively assigns states to all successor basic blocks of the current basic block.

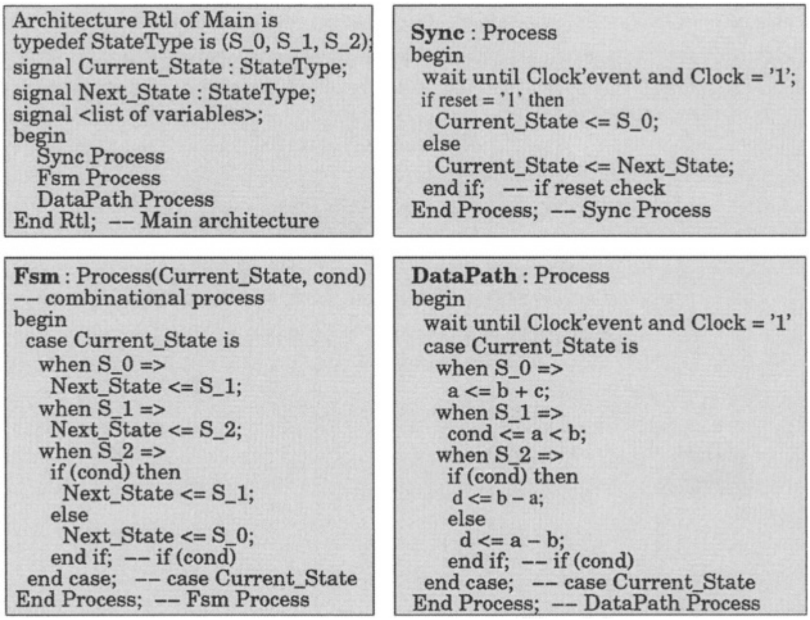
Algorithm 19 : AssignState(*bb*, *currState, condList*)
/* The finite state machine construction algorithm: assigns states to scheduling steps in basic block *currBB*. *condList* is the current list of conditions */

```
 1: if (IsJoin(bb) = true) then
 2:    currState $\leftarrow$ max(states of all preceding basic blocks)
 3:    condList -> remove(lastCondition)
 4: end if
 5: for all step in bb do
 6:    if (step does not have a conditional check) then
 7:       nextStep $\leftarrow$ GetNextSchedulingStep(step)
 8:       new3Tuple $\leftarrow$ < step, condList, nextStep >
 9:       StateTable[state] $\rightarrow$ add3Tuple(new3Tuple)
10:       step $\leftarrow$ nextStep
11:       state $\leftarrow$ state + 1
12:    else     /* step contains a condition */
13:       newCond $\leftarrow$ condition in step
14:    end if
15: end for
16: newTrueCondList $\leftarrow$ newFalseCondList $\leftarrow$ condList
17: if (newCond $\neq \phi$) then
18:    newTrueCondList $\rightarrow$ add(newCond)
19:    newFalseCondList $\rightarrow$ add(NOT(newCond))
20: end if
21: AssignState (NextTrue(bb), state, newTrueCondList)
22: AssignState (NextFalse(bb), state, newFalseCondList)
```

A 3-tuple is inserted in the state table for each non-conditional check step (lines 6 to 9). If a step with a conditional check is found, this condition is used later to create the new list of conditions *newTrueCondList* and *newFalseCondList* (lines 16 to 20). These new conditions are used when the function recursively assigns states to the basic blocks on the "true" and "false" control paths of the current basic block (lines 21 and 22 in Algorithm 19).

As shown in lines 1 to 4, the algorithm first checks if a basic block is a join basic block, i.e., the merge point of the branches of a conditional block. If it is, then the state is updated to the maximum of the states in the preceding non-empty basic blocks. Also, the last condition added to the condition list is removed (line 3) since the control has now moved out of the conditional block. Additionally, although not shown here, the algorithm assigns state to a join basic block only after checking that both the branches of the conditional have been scheduled (breath first traversal). This is similar to the *GetNextBasicBlock* algorithm presented earlier in Chapter 7.

This algorithm also handles loops since the loop body is executed if the conditional check of the loop evaluates to true. Hence, the loop conditional check is added to the list of conditions for the loop body. However, no condition is added for the basic blocks on the loop exit, i.e., the "false" path that branches out from the loop condition check. These details have been omitted from the algorithm shown in Algorithm 19 for brevity.



```
Architecture Rtl of Main is
typedef StateType is (S_0, S_1, S_2);
signal Current_State : StateType;
signal Next_State : StateType;
signal <list of variables>;
begin
  Sync Process
  Fsm Process
  DataPath Process
End Rtl; -- Main architecture
```

```
Sync : Process
begin
  wait until Clock'event and Clock = '1';
  if reset = '1' then
    Current_State <= S_0;
  else
    Current_State <= Next_State;
  end if; -- if reset check
End Process; -- Sync Process
```

```
Fsm : Process(Current_State, cond)
-- combinational process
begin
  case Current_State is
    when S_0 =>
      Next_State <= S_1;
    when S_1 =>
      Next_State <= S_2;
    when S_2 =>
      if (cond) then
        Next_State <= S_1;
      else
        Next_State <= S_0;
      end if; -- if (cond)
  end case; -- case Current_State
End Process; -- Fsm Process
```

```
DataPath : Process
begin
  wait until Clock'event and Clock = '1'
  case Current_State is
    when S_0 =>
      a <= b + c;
    when S_1 =>
      cond <= a < b;
    when S_2 =>
      if (cond) then
        d <= b - a;
      else
        d <= a - b;
      end if; -- if (cond)
  end case; -- case Current_State
End Process; -- DataPath Process
```

Figure 8.10: *Synthesizable Register Tranfer Level VHDL (a) Main architecture body with all three processes (b) The Current state assignment process with synchronous reset (c) The Next state calculation process (d) The Data path operation process*

The FSM generated for the loop is such that if the conditional check is true, the next state is the first scheduling step in the loop body. If the conditional check is false (i.e., the loop exits), the next state is the first scheduling step in the basic block *after* the loop.

Once the state table has been constructed by the algorithm presented above, the back-end code generator can then generate the VHDL code, as explained in the next section.

8.3.4 Synthesizable VHDL Generation

Designers usually prefer to partition the finite state machine into several components. In the context of *Progammable Logic Array* (PLA) based FSM designs, typically a *sequencing* PLA for next state generation and a command PLA for primary output generation are used [Pau89]. In FSM design using VHDL, three functional blocks are used, namely, the next-state conditioning logic, the state-register (or state-variable) creation and the output conditioning logic [Raj95].

We have found this approach to be best suited for synthesizable VHDL generation, since it cleanly partitions the controller from the data path. After construction of the finite state machine from the scheduled hierarchical task graph, the *Spark* framework outputs VHDL in the form of three processes (as shown in Figure 8.10):

- *Synchronous Process:* This process assigns the next state to the current state register of the FSM. It also has an asynchronous *reset* which resets the state machine to a known state (the initial idle state). We have chosen a synchronous process model that is triggered at the rising edge of the clock. This process is standard for all the VHDL code generated by the *Spark* framework.

- *FSM Process:* Based on the current state and the prevailing conditions in the data path, this combinational process determines the next state of the FSM and stores it in a *Next State* register. This process is generated using the state table constructed by the algorithm in the previous section.

- *Data Path Process:* This process executes the operations in the data path based on the current state. Since we are using a global slicing methodology, the operations executed also depend on which branch of the mutually exclusive conditional block is active. This process is also generated using the state table (as per the scheduled HTG).

The data path process is also triggered at the rising edge of the clock. An alternate style would have been to trigger it by a change in the current state. However, this leads to all the data path activation signals being latched on the current state. This can cause glitches and discrepancies between simulation and synthesis results.

There are several advantages of partitioning the FSM and data path in this manner. Data paths are generally regular and either use components picked from a library (either standard cell or custom) or synthesized by module compilers [GG00, CKS+98]. So designers often avoid trying to flatten the structural hierarchy inherent in the data path. On the other hand, designers prefer to flatten the control FSM and try to optimize it as much as possible, since critical paths typically end up lying on forwarding paths. Furthermore, frequently, there are data path components that are custom designed. So during the early design phase, a functional equivalent is used in the RTL synthesis process. The custom designed component can then be plugged into the netlist at a later stage. This process partitioning thus facilitates easy application of different constraints and synthesis commands for the control FSM and the data path.

Besides these four processes, the VHDL code for the multiplexing logic that feeds into the functional units is placed after all the processes in the architecture description of a component. We have experimented with several styles for this multiplexing logic and have settled on a style that has a process for each input port of a functional unit. The select signals for the multiplexer are determined based on the control state and the conditions under which a data value feeds into the input port of the functional unit. The select signals are thus used to choose between the various data inputs.

An example of the VHDL output for one input port of a comparator is given below:

```
res_CMP_4_in0_MUXES:PROCESS(CURRENT_STATE, regNum1, hT0,
                        regNum0, hT1, hT39, regNum6)
  variable mux_select : std_logic_vector(7 downto 0);
BEGIN
  mux_select := "00000000";
  if (CURRENT_STATE(1) ='1') then
```

```
      mux_select := "00000001";
    end if;
    if (CURRENT_STATE(3) = '1' and hT0) then
      mux_select := "00000010";
    end if;
    if (CURRENT_STATE(4) = '1' and hT0 and hT1 and NOT(hT39)) then
      mux_select := "00000100";
    end if;
    if (CURRENT_STATE(5) = '1' and hT0 and hT1 and NOT(hT39)) then
      mux_select := "00001000";
    end if;
    case mux_select is
      when "00000001" =>
        res_CMP_4_in0 <= regNum1;
      when "00000010" =>
        res_CMP_4_in0 <= regNum0;
      when "00000100" =>
        res_CMP_4_in0 <= regNum1;
      when "00001000" =>
        res_CMP_4_in0 <= regNum6;
      when others =>
        res_CMP_4_in0 <= 0;
    END CASE;
  END PROCESS;         -- res_CMP_4_in0_MUXES END PROCESS;
```

8.3.5 Related Work

The conditional resource sharing methods employed in high level synthesis tools lead to the critical paths being in the controller [HW94]. Rao et al. [RK94] have studied how the choice of centralized versus distributed controllers impact the control logic generated. Distributed controllers are attractive in asynchronous systems due to the absence of a global clock and because global control signals introduce delays that reduce the inherent parallelism of an asychronous system [ELL00].

Roy et al. [RV96] describe the generation of synthesizable controller models from communicating VHDL processes. Kim et al. [KKP91] group operations scheduled in various time steps into states based on their mutual exclusivity information and thereafter, generate a centralized controller.

Finite state machine controllers are the most popular type of controllers for digital design. However, several other types controllers have also been explored in high-level synthesis tools [DM94, KGM95]. Cathedral-II [ZSRM90] uses a hardwired microcoded controller, in which *a microprogram ROM* (read only memory) stores the micro-instructions and *a sequencer block* controls the execution sequence of the program. The sequencer block is synthesized and non-sequential (branch) addresses are generated by a dedicated logic block. *Programmable Microcoded* controllers are popular in the DSP community when building ASIPs (Application Specific Instruction set Processors). In this architecture, the sequencer has several modes which control the type of conditional branch that can be executed. Using these modes, the microinstructions in the ROM can perform conditional jumps [KGM95, BR98]. In a *Programmable*

Macrocoded controller, the ROM contains *macro*-instructions that can take more than one cycle to execute and which control the execution of several operations [Lee88].

8.4 Summary

In this chapter, we presented the resource binding and control synthesis methodology employed by our high-level synthesis approach. We first presented the motivation for a resource binding methodology that minimizes interconnect in designs with complex control flow. We then presented a formulation for this problem. In Sections 8.2.4 and 8.2.5, we presented network flow solutions for the operation and variable binding problems respectively. In Section 8.3, we presented the finite-state machine (FSM) based control synthesis methodology used in our PHLS framework. We presented a model for the FSM controller and an algorithm for constructing the FSM in Sections 8.3.2 and 8.3.3. In Section 8.3.4, we described the back-end code generator in *Spark* and how it generates synthesizable VHDL from the scheduled and resource-bound design. The contributions of this chapter are (a) the formulation and solution of an interconnect minimizing resource binding methodology that specifically targets designs with complex control flow, and (b) an FSM based control synthesis methodology and an algorithm for the construction of the FSM controller.

Part III

SPARK: Implementation, Scripts and Design Examples

9

SPARK: IMPLEMENTATION, USAGE, AND SYNTHESIS SCRIPTS

In this chapter, we present the *Spark* software tool in which we have implemented our parallelizing high-level synthesis methodology. We first discuss implementation details and briefly describe how the tool can be used. We then describe how synthesis scripts can be used to guide the transformations and heuristics applied by *Spark* during synthesis. In particular, we use the synthesis scripts to study the effects on synthesis results of the various speculative code motions, the branch balancing techniques, and different ways of calculating operation priorities.

9.1 Implementation of the SPARK PHLS Framework

We implemented the parallelizing high-level synthesis methodology presented in this book consisting of the scheduling heuristics, the pre-synthesis transformations and the synthesis and parallelizing compiler transformations in a prototype framework called *Spark.* The *Spark* software tool is over 100,000 lines of C++ code and uses the EDG (Edison Design Group) C/C++ front-end parser [EDG].

Spark takes a behavioral description in ANSI-C as input – currently with the restrictions of no pointers, no function recursion, and no irregular control-flow jumps. In addition to this, the designer has to specify the list of resources allocated to synthesize the design in a hardware resource library along with their timing information and the bit-widths of the data types used in the *C* input. Thus, resource allocation and module selection are done by the designer and are given as input to the synthesis tool through the hardware resource library. *Spark* provides the ability to control and thus, experiment with the various code transformations and heuristics employed during synthesis using synthesis scripts and command-line options.

Figure 9.1 presents an overview of the *Spark* framework. After parsing the design description, we capture it using the 3-layered intermediate representation (IR) described in Chapter 3. Recall that the three layers in this representation are hierarchical task graphs (HTGs), control flow graphs (CFGs), and data flow graphs (DFGs).

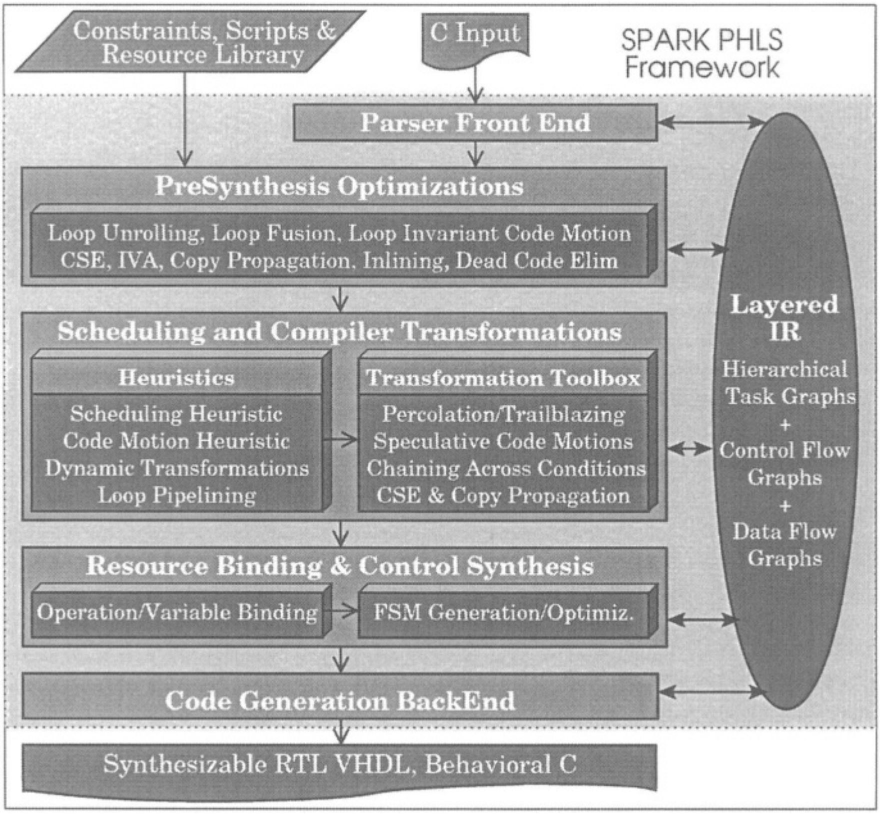


Figure 9.1: *Overview of the SPARK Parallelizing High-Level Synthesis Framework*

Since the HTGs store structural information about the control structures (if-then-else, loops, et cetera) in the code, they are useful for efficiently applying coarse-grain and global code transformations such as loop unrolling, loop pipelining, speculative code motions, *Trailblazing,* chaining across conditional boundaries, et cetera. CFGs are useful for design traversal during scheduling and DFGs capture data dependencies among operations that is useful during scheduling, code motions, and chaining.

The *Spark* tool first applies a range of pre-synthesis transformations as shown in Figure 9.1 and then proceeds to the scheduling phase. We built this phase in a modular style using a transformations toolbox that contains a range of transformations and techniques such as speculative code motions and *Trailblazing* that were discussed earlier in this book. We implemented the scheduling heuristics presented in Chapter 7 in this scheduler framework. These heuristics can be guided using the synthesis scripts as explained in the next section.

After the scheduling phase, *Spark* executes the interconnect minimizing resource binding and control synthesis passes discussed in Chapter 8. This is followed by a back-end code generation pass that generates register-transfer level (RTL) VHDL and behavioral *C*. The RTL VHDL is synthesizable using standard commercial logic synthesis tools such as Synopsys Design Compiler [DC] and Xilinx ISE [Xil]. The behavioral *C* represents the scheduled and optimized design description and can be used for functional verification with the input *C* code.

9.2 Using command-line options and scripts to direct synthesis and optimizations in SPARK

The algorithms and code transformations implemented in the *Spark* framework can be guided using command-line options and scripts that are read at the start of execution. Details of these options and scripts can be found in Appendix A or the user manual on the *Spark* download page [SPA].

Most of the pre-synthesis passes can be controlled by command-line options. For example, the command-line option for loop-invariant code motion is "-hli", for CSE is "-hcs", for dead-code elimination is "-hdc". The command-line options also dictate if resource binding is done ("-hb"), VHDL is generated ("-hvf') and whether C code is generated ("-hcc").

The synthesis scripts are used for turning certain transformations on or off and for guiding which heuristics are employed during synthesis. For example, specific code motions can be disallowed using the synthesis scripts by setting the variable controlling the code motions to *false*. Similarly, we can choose to apply or not apply dynamic branch balancing during scheduling using flags in the script file.

Whereas details about how to use these synthesis scripts are given in Appendix A, in the rest of this chapter we present an insight into how we used this scripting ability to experiment with new transformations and develop heuristics to guide these transformations. We study the interdependencies between the speculative code motions, heuristics that guide conditional speculation, different ways of specifying the priority of operations, and compare loop unrolling with loop shifting.

Note that, the characteristics of the design examples used for presenting results in the rest of the chapter are presented later in Chapter 10.

9.3 Study of the Interdependencies between the Code Motions

We developed several speculative code motions in this work that reorder operations in the design description to maximize performance. However, a designer may choose to disable particular code motions to reduce their impact on the hardware costs of the circuit. To aid the designer in making these decisions, we study the impact of enabling and disabling individual code motions on the quality of synthesis results. We also study the interactions between code motions.

We present three sets of experiments. In the first set, we enable all the speculative and non-speculative code motions and then disable one code motion at a time and

Table 9.1: ***Scheduling results for the MPEG-1** $pred1$ **and** $pred2$ **designs when individual code motions are*** disabled ***one at a time.***

Allowed Code Motions	$MPEG$-1 $pred1$ # States	$MPEG$-1 $pred1$ Long Path	$MPEG$-1 $pred2$ # States	$MPEG$-1 $pred2$ Long Path
All Code Motions	37	899	67	2127
- Early Cond Exec	37(0 %)	899(0 %)	68(+1.5 %)	2191(+3.0 %)
- Speculation	40(+8.1%)	972(+8.1%)	75(+11.9%)	3099(+45.7%)
- Cond Speculation	56(+51.4%)	1862(+107%)	104(+55.2%)	4178(+96.4%)

Table 9.2: ***Scheduling results for the MPEG-2** $dpframe$ **and GIMP** $tiler$ **designs when one code motion is disabled at a time.***

Allowed Code Motions	$MPEG$-2 $dpframe$ # States	$MPEG$-2 $dpframe$ Long Path	$GIMP$ $tiler$ # States	$GIMP$ $tiler$ Long Path
All Code Motions	47	563	31	2534
- Early Cond Exec	49(+4.3%)	571(+1.4%)	31(0 %)	2534(0 %)
- Speculation	59(+25.5%)	811(+44%)	38(+22.6%)	3144(+24.1%)
- Cond Speculation	56(+19.1%)	599(+6.4%)	52(+67.7%)	4534(+78.9%)

evaluate the impact on synthesis results. In the second set of experiments, we disable all the speculative code motions and thereafter, enable only one code motion at a time and again evaluate the impact on synthesis results. In the third set of experiments, we disable all the code motions and then enable two code motions at a time. Each of these code motions can be enabled or disabled by setting flags in the synthesis script file (see Section A.4.2 in Appendix A). For our experiments, we use four designs as case studies, namely, MPEG-1 $pred1$, MPEG-1 $pred2$, MPEG-2 $dpframe$ and the GIMP *tiler* transform. The characteristics of these designs are presented in Chapter 10. In all the experiments presented in this section, loop-invariant code motion, CSE, dynamic CSE, and branch balancing are enabled.

9.3.1 Disabling One Code Motion at a Time

In Tables 9.1 and 9.2, we present the results for the first set of experiments. In the first row of these tables, we list the scheduling results when all the code motions are enabled. The second row lists results when only early condition execution is disabled. In the third row, only speculation is disabled and in the last row only conditional speculation is disabled. Note that early condition execution employs reverse speculation for moving operations. The columns in these tables list the number of states and cycles on the longest path through the design. The percentage reduction of each row *over the first row* (the case when all code motions are enabled) is given in parentheses.

The results in these two tables (Tables 9.1 and 9.2) demonstrate that disabling early condition execution has no impact on the scheduling results for the $pred1$ and *tiler*

Table 9.3: *Scheduling results for the MPEG-1 pred1 and pred2 designs when only one code motion is* enabled *at a time.*

Allowed	*MPEG*-1 *pred*1		*MPEG*-1 *pred*2	
Code Motions	# States	Long Path	# States	Long Path
Non-Speculative CMs	60	1937	111	4409
+ Early Condition Exec	61(+1.7%)	1937(0 %)	113(+1.8%)	4409(0 %)
+ Speculation	52(-13.3%)	1670(-13.8%)	96(-13.5%)	3794(-13.9%)
+ Cond Speculation	41(-31.7%)	1036(-46.5%)	77(-30.6%)	2529(-42.6%)

Table 9.4: *Scheduling results for the MPEG-2 dpframe and GIMP tiler designs when only one code motion is* enabled *at a time.*

Allowed	*MPEG*-2 *dpframe*		*GIMP tiler*	
Code Motions	# States	Long Path	# States	Long Path
Non-Speculative CMs	72	863	66	5944
+ Early Condition Exec	71(-1.4%)	859(-0.5%)	63(-4.5%)	5544(-6.7%)
+ Speculation	56(-22%)	599(-30.6%)	44(-33.3%)	3834(-35.5%)
+ Cond Speculation	59(-18%)	811(-6.0%)	43(-34.8%)	3644(-38.7%)

designs. On the other hand, the cycles on the longest path increase by 3 % for *pred*2 and 1.4 % for *dp frame* when early condition execution is disabled. Clearly, this code motion has little impact on scheduling results.

However, it is disabling speculation and conditional speculation that has the most profound impact on the results. Disabling speculation leads to an increase in the cycles on the longest path ranging from 8.1 % for *pred*1 to 45.7 % *pred*2. Disabling conditional speculation leads to the largest increases for the *pred*1, *pred*2 and *tiler* designs: from 78 to 107 %. These results clearly demonstrate that by far speculation and conditional speculation are the most useful code motions.

9.3.2 Enabling One Code Motion at a Time

The results for the second set of experiments are listed in Tables 9.3 and 9.4. For these experiments, we started by first enabling only the non-speculative code motions, i.e., code motions only within basic blocks and across hierarchical blocks (first row of the two tables). We then enabled only one code motion at a time. First we enabled only early condition execution (second row), then only speculation (third row) and finally, only conditional speculation (fourth row). Once again the percentage reduction of each row over the results in the first row is given in parentheses.

From the results in Tables 9.3 and 9.4, we again see that speculation and conditional speculation have the largest impact on scheduling results. The improvements in cycles on the longest path with speculation enabled range from 13 to 35 % and with conditional speculation enabled range from 6 to 46 %. Early condition execution leads

Table 9.5: *Scheduling results for the MPEG-1 pred1 and pred2 designs when individual code motions are enabled two at a time (CMs = Code motions, ECE = Early condition execution, Spec = Speculation, Cond Spec = Conditional Speculation).*

Allowed Code Motions	*MPEG*-1 *pred*1		*MPEG*-1 *pred*2	
	# States	Long Path	# States	Long Path
Non-Speculative CMs	60	1937	111	4409
+ ECE + Spec	56(-6.7 %)	1862(-3.9 %)	104(-6.3%)	4178(-5.2%)
+ ECE + Cond Spec	40(-33.3%)	972(-49.8%)	75(-32.4%)	2401(-45.5%)
+ Spec + Cond Spec	37(-38.3%)	899(-53.6%)	67(-39.6%)	2127(-51.8%)
All Code Motions	37(-38.3%)	899(-53.6%)	67(-39.6%)	2127(-51.8%)

Table 9.6: *Scheduling results for the MPEG-2 dpframe and GIMP tiler designs when individual code motions are enabled two at a time.*

Allowed Code Motions	*MPEG*-2 *dpframe*		*GIMP tiler*	
	# States	Long Path	# States	Long Path
Non-Spec CMs	72	863	66	5944
+ ECE + Spec	56(-22.2 %)	599(-30.6 %)	52(-21.2 %)	4534(-23.7 %)
+ ECE + Cond Spec	59(-18.1 %)	811(-6.0 %)	38(-42.4 %)	3144(-47.1 %)
+ Spec + Cond Spec	47(-34.7 %)	563(-34.8 %)	32(-51.5 %)	2634(-55.7 %)
All Code Motions	47(-34.7 %)	563(-34.8 %)	31(-53.0 %)	2534(-57.4 %)

to modest improvement in results for two of the four designs: *dpframe* and *tiler*. Hence, there is a discrepancy in the effectiveness and impact of early condition execution on the scheduling results as compared to the previous section, We found that this discrepancy is because the improvements due to early condition execution get masked by other transformations such as conditional speculation and speculation.

9.3.3 Enabling Multiple Code Motions at a Time

Now we enable two code motions at a time and study their impact on scheduling results. Tables 9.5 and 9.6 list the results when we first disable all the speculative code motions (first row), then enable only early condition execution and speculation (second row), then enable only early condition execution and conditional speculation (third row), then enable only speculation and conditional speculation (fourth row), and finally enable all the code motions (fifth row).

The results in these two tables coupled with the results from Tables 9.3 and 9.4 from the previous section tell us what combinations of code motions have an interdependency. Hence, when early condition execution is enabled with speculation (second row in Tables 9.5 and 9.6), we get *worse* results than if speculation is enabled alone (third row in Tables 9.3 and 9.4). However, when early condition execution is enabled with conditional speculation (third row in Tables 9.5 and 9.6), we get *better* results than if conditional speculation is enabled alone (fourth row in Tables 9.3 and 9.4). For the *dpframe* design we get the same results for speculation and conditional speculation with or without early condition execution.

These results show that early condition execution interacts positively with conditional speculation and negatively with speculation. Also, the results in the last rows of Tables 9.5 and 9.6 demonstrate that when all the code motions are enabled, we get the best synthesis results. Thus, the net interaction with all the code motions enabled is positive. Similarly, when speculation and conditional speculation are enabled together, we get better synthesis results for all four designs than when only one of them is enabled.

9.4 Study of the Impact of Dynamic Branch Balancing

As we saw in the previous section, the speculative code motions can significantly improve the quality of synthesis results. However, whenever an operation is moved in the design using a speculative code motion, it has an impact on the hardware circuit. Often the control and multiplexing costs (and therefore the area) increase. We found that of all the code motions, conditional speculation by far is the most expensive code transformation from the point of view of hardware costs. This is because conditional speculation duplicates operations and hence, leads to more complex multiplexer and control logic.

This led us to develop several techniques to guide and aid conditional speculation. Of these, the most important are a set of *dynamic branch balancing* techniques that create new opportunities for employing conditional speculation [GDGN03a]. Branch balancing was discussed in detail in Chapter 6. There are two branch balancing techniques: (a) *Branch balancing during design traversal* (BBDDT): this technique inserts scheduling steps while traversing the design during scheduling, and (b) *Branch balancing during code motions* (BBDCM): this technique inserts steps if it enables code motions (specifically conditional speculation).

In Tables 9.7 and 9.8, we list the results for our experiments with the two branch balancing techniques. The first row lists the results for when all the speculative code motions are enabled *except* conditional speculation. We call this the baseline case. The second row has conditional speculation (CS) enabled along with the rest of the code motions. In the third row, all the code motions including CS are enabled along with the branch balancing during design traversal (BBDDT). The fourth row lists the results for when all the code motions are enabled along with the branch balancing during code motion (BBDCM). The fifth row has both the branch balancing algorithms enabled along with all the code motions. The percentage reductions of each row over the baseline case (first row) are given in parentheses. These techniques can be enabled and disabled in the synthesis scripts as explained in Section A.4.1 in Appendix A.

The results in Tables 9.7 and 9.8 demonstrate that the branch balancing techniques create many more opportunities to employ conditional speculation. When conditional speculation is enabled alone (second row in both tables), the improvements in number of states and cycles on the longest path are in the range of 8 to 12 % and 3 to 10 % for the *pred*1, *pred*2 and *dpframe* designs. However, for the *tiler* design, conditional

Table 9.7: *Scheduling results for the MPEG-1 $pred1$ and $pred2$ designs when the branch balancing during design traversal (BBDDT) and branch balancing during code motions (BBDCM) techniques are applied in conjunction with conditional speculation (CMs = Code Motions, CS = Conditional Speculation).*

Allowed Code Motions	$MPEG$-1 $pred1$ # States	$MPEG$-1 $pred1$ Long Path	$MPEG$-1 $pred2$ # States	$MPEG$-1 $pred2$ Long Path
All CMs - CS	56	1862	105	4179
+ Cond Spec (CS)	49(-12.5%)	1667(-10.5%)	95(-9.5%)	3856(-7.7%)
+ CS + BBDDT	39(-30.4%)	1027(-44.8%)	72(-31.4%)	2447(-41.4%)
+ CS + BBDCM	49(-12.5%)	1667(-10.5%)	94(-10.5%)	3792(-9.3%)
+ CS+BBD(DT+CM)	37(-33.9%)	899(-51.7%)	67(-36.2%)	2127(-49.1%)

Table 9.8: *Scheduling results for the MPEG-2 $dpframe$ and GIMP $tiler$ designs when the branch balancing techniques are applied with conditional speculation.*

Allowed Code Motions	$MPEG$-2 $dpframe$ # States	$MPEG$-2 $dpframe$ Long Path	$GIMP$ $tiler$ # States	$GIMP$ $tiler$ Long Path
All CMs - CS	56	599	52	4534
+ Cond Spec (CS)	51(-8.9 %)	579(-3.3 %)	33(-36.5%)	2734(-39.7%)
+ CS + BBDDT	52(-7.1 %)	587(-2.0 %)	32(-38.5%)	2634(-41.9%)
+ CS + BBDCM	49(-12.5%)	571(-4.7 %)	34(-34.6%)	2834(-37.5%)
+ CS +BBD(DT+CM)	47(-16.1%)	563(-6.0 %)	31(-40.4%)	2534(-44.1%)

speculation leads to reductions of 36 and 39 % in the number of states and cycles on the longest path. For the other three designs, the results improve significantly when the two branch balancing techniques are applied to aid conditional speculation. The total improvements with both the branch balancing techniques enabled range from 16 to 40 % in the number of states and 6 to 51 % in the cycles on the longest path for the four designs (last row in both tables).

The results in Tables 9.7 and 9.8 also demonstrate that the two branch balancing algorithms are complementary to some extent. Whereas the BBDDT technique is more effective for the *pred*1, *pred*2 and *tiler* designs, the BBDCM technique is more effective for the *dpframe* design. Also, we can see from the results in the two tables that the best result for all the designs is obtained when both the BBDDT and BBDCM algorithms are employed. These results demonstrate that each of the branch balancing algorithms create different and unique opportunities for employing conditional speculation.

Table 9.9: *Scheduling results for the MPEG-1* $pred1$ *and* $pred2$ *designs when priority of an operation is calculated as the* Sum *and one more than the* Maximum *of the priorities of all its dependent operations.*

Allowed Code Motions	*MPEG*-1 *pred*1 # States	 Long Path	*MPEG*-1 *pred*2 # States	 Long Path
Sum Priority	38	907	69	2143
Max Priority	37(-2.6 %)	899(-0.9 %)	67(-2.9 %)	2127(-0.7 %)

Table 9.10: *Scheduling results for the MPEG-2* $dpframe$ *and GIMP* $tiler$ *designs when priority of an operation is calculated as the* Sum *and one more than the* Maximum *of the priorities of all its dependent operations.*

Allowed Code Motions	*MPEG*-2 *dpframe* # States	 Long Path	*GIMP tiler* # States	 Long Path
Sum Priority	48	567	31	2534
Max Priority	47(-2.1 %)	563(-0.7 %)	31(0 %)	2534(0 %)

9.5 Different Ways of Calculating Priority

The *Spark* scheduler uses a priority-based list scheduling algorithm to schedule the design. Available operations are ordered on the basis of their priority and the operation with highest priority is chosen for scheduling (see Section 7.2).

We experimented with two different ways of calculating priority using our synthesis scripts (see Section A.4.1 in Appendix A):

1. Priority of an operation vop is the sum of the priorities of all the operations that use the result of vop. This is called the *Sum* method.

2. Priority of an operation vop is one more than the maximum of the priorities of all the operations that use the result of vop. This is called the *Max* method.

The results for experiments with these two different ways of calculating priority are listed in Tables 9.9 and 9.10. The first and second rows in these tables list the results when priority is calculated using the *Sum* and the *Max* method respectively. These results demonstrate the *Max* method leads to marginally better results than the *Sum* method. The *Max* method reduces the number of states and the cycles on the longest path by up to 2.9 % and 0.9 % respectively.

9.6 Recommended Synthesis Scripts

Based on the experiments presented in this book, we recommend invoking the *Spark* tool with the synthesis script and the command-line options given in Sections A.6 and

A.7 of Appendix A. These options produce the best synthesis results for the multimedia and image processing designs we have used.

In this synthesis script, we have enabled all the code motions, enabled both the branch balancing algorithms, enabled dynamic CSE, and enabled the *Max* method of calculating priority. The command-line options that are enabled are: loop-invariant code motion, common sub-expression elimination, copy and constant propagation, dead code elimination, scheduling, generation of synthesizable RTL VHDL, interconnect-minimizing resource binding and generation of statistics about cycle count (-hec).

The only things that change for the synthesis of microprocessor functional blocks is that we have to enable chaining of operations across conditional boundaries, give a large enough clock period so that all the operations can be chained into one or a few cycles and give the appropriate (usually full) loop unrolling options. The modifications to the synthesis script and command-line options for the synthesis of microprocessor blocks are given in Section A.8 of Appendix A. Note that, since we use *Spark* as a design exploration tool for the synthesis of microprocessor blocks, the large clock period is used only to enable packing of all operations into one or more cycles. The designer can then use the schedule and VHDL generated by *Spark* and replace the functional units with custom units that execute within the clock period of the microprocessor.

9.7 Summary

In this chapter, we presented the *Spark* framework in which we implemented our parallelizing high-level synthesis methodology. We described the implementation of *Spark* and briefly explained the usage of the software tool. We then demonstrated the utility of the scripting ability available in *Spark* to experiment with the different optimizations and heuristics. In Section 9.3, we presented experiments when the various speculative code motions are enabled and disabled individually and in pairs. In Section 9.4, we then presented results for two branch balancing techniques that increase the number of opportunities to employ conditional speculation. In Section 9.5, we compared two different ways of calculating the priority of operations. We found that playing with the synthesis scripts enables us to find the combination of techniques that leads to the best synthesis results. The contributions of this chapter are a presentation of the implementation of the *Spark* framework and a demonstration of the utility of the synthesis scripts for guiding the synthesis process.

10

DESIGN EXAMPLES

10.1 Introduction

In this chapter, we present the quality of results achieved by the *Spark* high-level synthesis framework on a set of design examples from the multimedia and image processing domains. We first detail the effects of each transformation presented in this book using a set of four designs that are representative of the multimedia and image processing domains. These four designs serve as *case studies* to understand the effect of individual transformations on the quality of synthesis results.

To quantify the effectiveness of the transformations presented in this book beyond the four case studies, we apply the transformations on a set of twelve designs derived from various multimedia and image processing applications. For these twelve designs, we perform a comparative analysis of the pre-synthesis transformations, the speculative code motions and the dynamic CSE technique.

10.1.1 Designs used for the Four Case Studies

The four designs we used as case studies are derived from three moderately complex real-life applications representative of the multimedia and image processing domains, namely, the MPEG-1 algorithm [SPA], the MPEG-2 algorithm [LPMS97, Med] and the GIMP image processing tool [Gim]. The designs we have chosen from these applications consist of two functions from the *Prediction* block of the MPEG-1 algorithm, one function from the *Motion Estimation* block of the MPEG-2 encoder algorithm and one function from the GIMP image processing tool. The MPEG-1 functions used are the *pred*1 and *pred*2 functions, the MPEG-2 function is the *dpframe* function and the GIMP function is the *tile* function (with the scale function inlined) from the "tiler" transform[1].

Table 10.1 lists the characteristics of the four designs used as case studies in terms of the number of if-then-else conditional blocks (If HTGs), loops (Loop HTGs), non-

[1] Note that this floating point function has been arbitrarily converted to an integer function for the purpose of our experiments. This does not affect the nature of the data and control flow, but only the data that is processed.

Table 10.1: *Characteristics of the four case studies used in our detailed experiments, along with the resources allocated for scheduling them.*

Benchmark	#	#	#	#	# Resources					
	Ifs	Loops	BBs	Ops	+−	<<	==	[]	*	/
MPEG-1 pred1	4	2	17	123	2	2	2	2	1	-
MPEG-1 pred2	11	6	45	287	2	2	2	2	1	-
MPEG-2 dpframe	18	4	61	260	4	2	2	2	1	-
GIMP tiler	11	2	35	150	3	2	2	2	1	1

Table 10.2: *Characteristics of the twelve designs used for presenting the overall synthesis results, along with the resources allocated for scheduling them.*

Benchmark	#	#	#	#	# Resources					
	Ifs	Loops	BBs	Ops	+−	<<	==	[]	*	/
GIMP nlfilt	16	0	37	127	3	1	1	-	1	1
GIMP polar	27	0	78	251	2	1	3	-	1	1
GIMP sobel	8	2	31	117	4	2	2	5	1	1
GIMP spread	9	9	59	170	2	3	3	2	1	1
GIMP sparkle_fspike	20	2	60	140	3	2	3	2	1	1
GIMP sparkle_rpnt	8	1	25	77	1	1	1	1	1	1
GIMP sel_gauss	10	5	40	99	2	3	2	1	1	1
GIMP unsharp_blue	3	15	67	151	2	1	1	1	1	1
XVID adapt_quant	6	3	25	69	2	2	2	1	1	-
G721 update	35	2	99	209	2	2	2	1	1	-
MPEG2 deinterlace	6	3	30	67	2	2	2	2	-	-
MPEG2 conv420to422	20	4	81	325	4	4	2	1	-	-

empty basic blocks and the total number of operations in the input description. The number of If HTGs, Loop HTGs, and basic blocks is indicative of the control complexity of the design. All these designs have doubly nested loops.

Table 10.1 also lists the type and quantity of each resource allocated to schedule these designs for all the experiments presented in the following sections. The resources indicated in this table are; $+-$ does add and subtract, $==$ is a comparator, $*$ a multiplier, / a divider, [ ] an array address decoder and $<<$ is a shifter. The multiplier ($*$) executes in 2 cycles and the divider (/) in 5 cycles. All other resources are single cycle.

10.1.2 Larger Set of Designs used for Validating Our Synthesis Approach

We use twelve more designs to validate the utility of our synthesis approach on a larger set of designs. These designs are listed in Table 10.2. These designs consist of 8 designs derived from the GIMP image processing tool [Gim], 2 designs from the MPEG-

2 algorithm [LPMS97, Med], 1 design from the G721 voice compression algorithm [LPMS97, Med] and 1 design from the XVID MPEG-4 video decoding tool [Xvi].

Table 10.2 lists the characteristics of the twelve designs in terms of the number of if-then-else conditional blocks (If HTGs), loops (Loop HTGs), non-empty basic blocks and the total number of operations in the input description. This table also lists the type and quantity of each resource allocated to schedule these designs for the experiments presented in Section 10.6.2.

10.1.3 Metrics for Scheduling and Logic Synthesis Results

We present two sets of results for the experiments in this chapter: the *scheduling results* as reported by the *Spark* high-level synthesis tool and the *logic synthesis results* obtained by synthesizing the RTL VHDL generated by *Spark* using the Synopsys *Design Compiler* logic synthesis tool.

The scheduling results are in terms of the number of states in the finite state machine controller and the cycles on the longest path (i.e. execution cycles). The longest path through an if-then-else conditional block is the cycles on the longer branch and for loops, the longest path length of the loop body is multiplied by the number of loop iterations. For all the designs used in our experiments, the loop bounds are known.

We use the LSI-10K synthesis library for logic synthesis of the RTL VHDL and the components are allocated from the Synopsys *DesignWare Foundation* library. The logic synthesis results are presented in terms of three metrics: the critical path length (in nanoseconds), the unit area (in terms of synthesis library used) and the maximum delay through the design. The critical path length is the length of the longest combinational path in the netlist as reported by static timing analysis tool. This length dictates the clock period of the design. The maximum delay is the product of the longest path length (in cycles) and the critical path length (in ns) and signifies the maximum input to output latency of the design. The area is the sum of the combinational and non-combinational area as reported by the logic synthesis tool.

Recall that, the *Spark* framework treats each function call as a resource and creates a functional unit corresponding to it in hardware. For example, the function *calc_forward_motion* is called from the two functions *pred*1 and *pred*2 and hence, is a component or functional unit that is embedded in these hardware blocks. These "called" functions contribute towards the schedule length and the number of states in the controller of the calling function.

10.2 Results for Pre-Synthesis Optimizations

For the results presented in this section, we start with a "baseline" case that has all the speculative code motions enabled, along with the compiler passes of copy propagation, constant propagation and dead code elimination that are applied both before and after scheduling. This baseline case, thus, represents a design that has already been optimized to a great extent by the speculative code motions (see Section 10.3). Using this baseline case, we demonstrate how the pre-synthesis transformations can further improve the synthesis results.

10.2.1 Function Inlining

In our discussion of source level transformations, we left out one important coarse grain source-to-source transformation, namely, *function inlining*. Function inlining is a transformation that replaces a call to a function by an instance of the function itself. This transformation is usually applied to increase the scope of application of other compiler transformations. Although this transformation has not been implemented in the *Spark* framework, we applied it manually to the MPEG-1 designs and to the tiler transform from the GIMP. To demonstrate the effectiveness of function inlining, we present scheduling results for these designs.

Both the functions, *pred*1 *and pred*2, of the MPEG-1 design call the function "calcid" at the start of doubly nested loops. For this reason and because *calcid* is a small function that consists of only straight-line code (no control), it is ideal for inlining. Similarly, we inline the function "scale" that is called by the *tiler* transform in the GIMP tool at the start of a doubly nested loop.

Table 10.3 lists the scheduling results before and after inlining the *calcid* function into the *pred*1 and *pred*2 functions and the *scale* function into the *tiler* function. From the results in this table, we observe that inlining leads to improvements of between 12 to 32 % in the number of states in the FSM controller and between 7 to 14 % in the cycles on the longest path through the designs. These improvements are because inlining the function calls for these designs increases the opportunities available to the parallelizing transformations to compact the code.

The logic synthesis results for these experiments are given by the bar charts in Figure 10.1 for the three designs. The metrics mapped here are the cycles on the longest path, the critical path length, the total delay (cycles x critical path length) and the unit area. All the values are normalized by the values for the non-inlined case.

Table 10.3: ***Results before and after inlining the calcid function for the MPEG-1 pred*1 *and pred*2 *functions and the scale function for the GIMP tiler function. After inlining, the operations from the inlined functions share the resources with the existing operations in the calling functions.***

Transform	*MPEG*-1 *pred*1		*MPEG*-1 *pred*2	
Applied	# States	# cycles	# States	# cycles
Baseline	46	2144	96	4891
+ inlining	40(-13.0%)	1824(-14.9%)	84(-12.5%)	4187(-14.4%)

Transform	*GIMP tiler*	
Applied	# States	# cycles
Baseline	62	4231
+ inlining	42(-32.3%)	3931(-7.1%)

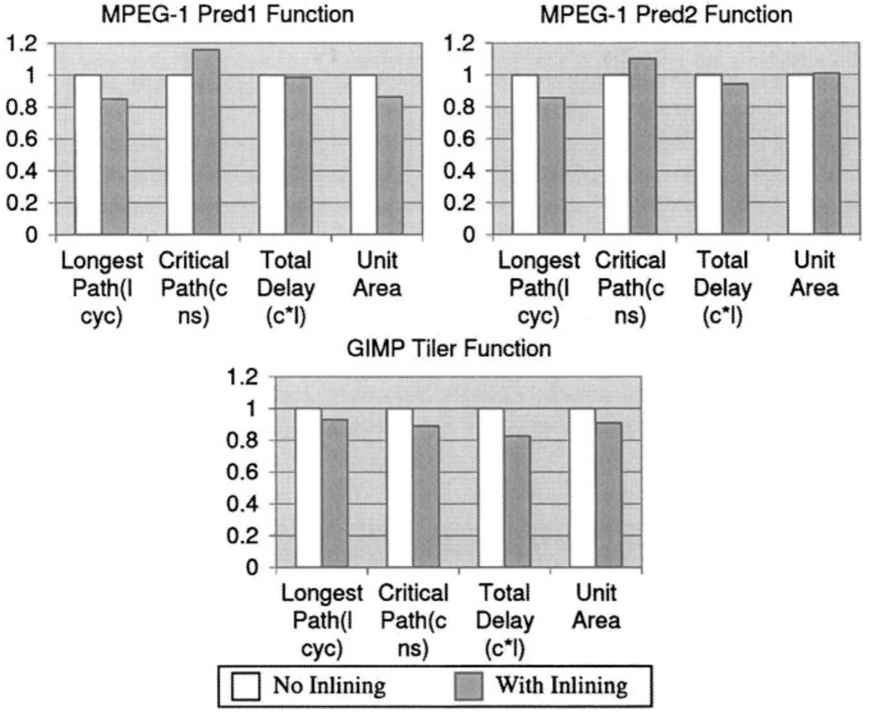


Figure 10.1: ***Effects of function inlining on logic synthesis results for the MPEG-1 pred*1 *and pred*2 *functions and the GIMP tiler function.***

We see from these logic synthesis results that inlining leads to a higher (about 10 to 15 %) critical path length for the *pred*1 and *pred*2 designs. This is because after inlining, the number of operations in the design increases due to the addition of the operations from the inlined function. Hence, a larger number of operations are now mapped to the same number of resources as before. This leads to more complex multiplexers and associated control logic. However, the total delay through the synthesized circuits for the three designs decreases from between 2 to 20 % with area remaining almost constant. Indeed, we found that when even more optimizations are enabled, the results are far better for the inlined designs over the non-inlined designs.

Based on these results, for the rest of the experiments presented in this chapter, we use the inlined versions of the *pred*1, *pred*2 and *tiler* functions as the "baseline" case. These inlining decisions have been made by inspecting the design and based on experimentation when it became clear that inlining would significantly enhance the opportunities to apply the transformations in the *Spark* toolkit.

Table 10.4: ***Results after applying pre-synthesis transformations on the MPEG-1*** $pred1$ ***and*** $pred2$ ***functions***

Transformation	$MPEG$-1 $pred1$		$MPEG$-1 $pred2$	
Applied	# States	Long Path	# States	Long Path
Baseline	40	1824	84	4187
Base + Loop Inv CM	51(+27.5 %)	1396(-23.5 %)	100(+19.1 %)	3266(-22 %)
Base + CSE	36(-10 %)	1504(-17.5 %)	73(-13.1 %)	3482(-16.8 %)
Base + LICM + CSE	40(0 %)	1091(-40.2 %)	74(-11.9 %)	2575(-38.5 %)

Table 10.5: ***Results after applying the pre-synthesis transformations on the MPEG-2*** $dpframe$ ***and the GIMP*** $tiler$ ***designs.***

Transformation	$MPEG$-2 $dpframe$		$GIMP$ $tiler$	
Applied	# States	Long Path	# States	Long Path
Baseline	53	672	42	3931
Base + Loop Inv CM	59(+11.3 %)	654(-2.7 %)	48(+14.3 %)	3163(-19.5 %)
Base + CSE	50(-5.7 %)	602(-10.4 %)	31(-26.2 %)	2831(-28 %)
Base + LICM + CSE	49(-7.6 %)	571(-15 %)	31(-26.2 %)	2534(-35.5 %)

10.2.2 Loop-Invariant Code Motion and CSE: Scheduling Results

Tables 10.4 and 10.5 list the scheduling results obtained after the application of the pre-synthesis transformations to the four designs. The results in the first row are for the baseline case (all code motions enabled along with copy propagation and dead code elimination); the second row for when only loop-invariant code motion (LICM) is applied to this baseline design, the third row for when only common sub-expression elimination (CSE) is applied and the fourth row for when both LICM and CSE are applied. The percentage reductions of each row over the *baseline* case are given in parentheses.

The results in the second row of these two tables show that when loop-invariant code motion alone is applied, the number of states in the controller increases by 11 to 27 %, while the cycles on the longest path through the design decrease between 2 to 23 %. The reduction in cycles is because loop-invariant operations are moved out of the loop, which in turn leads to the execution of fewer operations per iteration of the loop body. However, the operations that are moved outside the loop body require more states to execute and often this increase in the number of states outside the loop is greater than the decrease in the number of states required to execute operations within the loop. We will explore the trade-off this creates between area increase versus performance increase in the next section.

We see from the third row in Tables 10.4 and 10.5 that when CSE is applied in addition to the transformations in the baseline case, the number of states decrease between 5 to 26 % and cycles on the longest path decrease between 10 to 28 %. Clearly, there

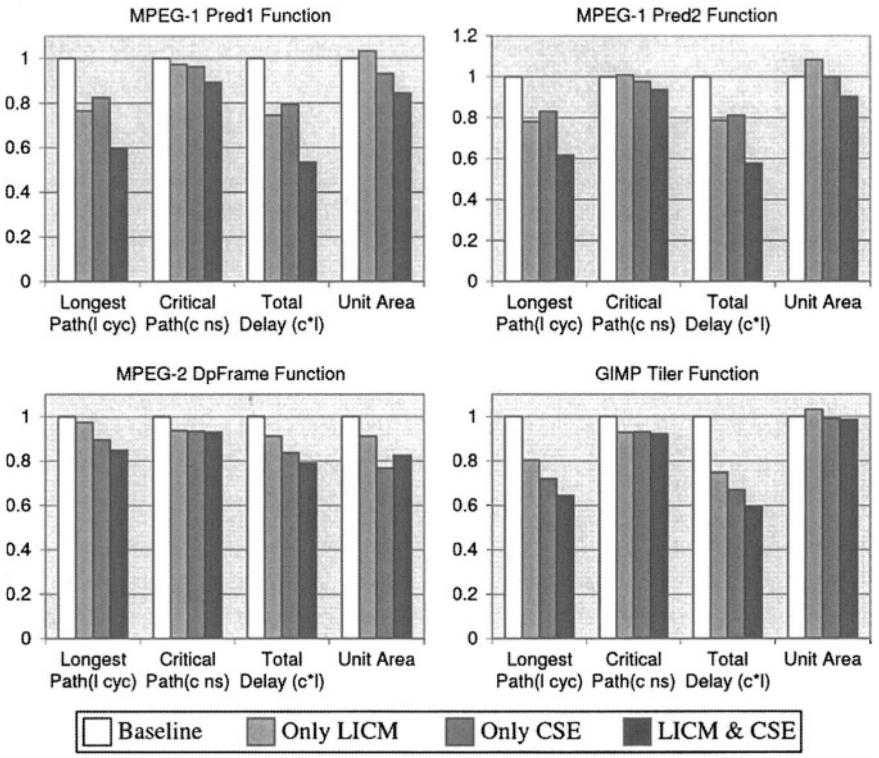


Figure 10.2: ***Effects of the pre-synthesis transformations, loop-invariant code motion (LICM) and common sub-expression elimination (CSE), on logic synthesis results for the four designs used as case studies.***

exist numerous opportunities to apply CSE in off-the-shelf code for these industrial applications.

The results in the last row of these tables show that when both loop-invariant code motion and CSE are applied, the improvements in the cycles on the longest path are additive to some extent. Also, CSE is able to counter the increase in the number of states caused by LICM. Hence, compared to the baseline case, after applying both the transformations we obtain reductions between 0 to 26 % in the number of states and between 15 to 40 % in the cycles on the longest path.

10.2.3 Loop-Invariant Code Motion and CSE: Logic Synthesis Results

The results after logic synthesis of the VHDL generated by *Spark* corresponding to the pre-synthesis experiments are presented in the graphs in Figure 10.2. The bars in these graphs represent the baseline case (1st bar), when only LICM is applied (2nd bar), when CSE is applied (3rd bar) and finally, when both LICM and CSE are applied (4th bar). All the metrics mapped are normalized with respect to the baseline case.

These results show that the critical path length remains fairly constant or reduces marginally when these transformations are applied. This is important because it signifies that the clock period in the design does not increase. The constant critical path length coupled with a decrease in the cycles on the longest path leads to significant reductions in the total delay through the circuits for the four designs. Clearly, LICM leads to larger improvements in delay than CSE (i.e. there are more opportunities for applying LICM in these codes).

The larger controller size (as given by the number of states) after applying LICM can lead to a larger circuit area by between 5 to 10 % over the baseline case. Applying CSE with LICM, however, brings this area increase under control. With both transformations, we achieve between a 20 to 50 % reduction in the total delay through the design, with a 5 to 15 % reduction in area.

Loop-invariant code motion has two opposing effects on the synthesized designs. On one hand, it reduces the cycles on the longest path through the design by executing fewer operations within the loop body. On the other hand, LICM also leads to a bigger FSM controller. Also, because transformations such as LICM and the speculative code motions increase resource utilization, the complexity of the steering logic (multiplexers and de-multiplexers) increases, thereby, leading to an increase in area. In contrast, CSE leads to a reduction in the size of the controller. Also, since CSE eliminates redundant operations, the number of operations mapped to the functional units reduces, hence reducing area.

10.3 Results for Speculative Code Motions

For the experiments presented so far, we have used a baseline design that has been optimized using the speculative code motions. In this section, we study the effectiveness of the individual speculative code motions on scheduling and logic synthesis results.

10.3.1 Scheduling Results for the Speculative Code Motions

Tables 10.6 and 10.7 list the number of states in the FSM and cycles on the longest path for the four designs under consideration. The rows in these tables present results with each code motion enabled incrementally, i.e., these signify the "allowed code motions" while determining the available operations (see Section 7.2) and do *not* represent an ordering of code motions.

We first allow code motions only within basic blocks (first row) and then, in the second row, we also allow code motions across hierarchical blocks, i.e., across entire if-then-else conditionals and loops. Thus, the second row lists the results after applying only non-speculative code motions. Speculation is enabled in addition to the non-speculative code motions for the experiments presented in the third row, and in the fourth row, early condition execution is enabled as well. Finally, in the last row, conditional speculation is also enabled; this row represents the case where all the code

Table 10.6: *Scheduling results after applying the speculative code motions the MPEG-1 pred1 and pred2 functions.*

Allowed Code Motions	*MPEG*-1 *pred*1 # States	*MPEG*-1 *pred*1 Long Path	*MPEG*-1 *pred2* # States	*MPEG*-1 *pred2* Long Path
Within basic blocks	71	2009	125	4555
+across hier blocks	60(-14.3 %)	1937(-3.6 %)	111(-11.2 %)	4409(-3.2 %)
+early cond exec	61(+1.7 %)	1937(0 %)	113(-1.8 %)	4409(0 %)
+speculation	56(-8.2 %)	1862(-3.9 %)	104(-8.0 %)	4178(-5.2 %)
+cond speculation	40(-28.6 %)	1091(-41.4 %)	74(-28.8 %)	2575(-38.4 %)
Total Reduction	**42.9 %**	**45.7 %**	**40.8 %**	**43.5 %**

Table 10.7: *Scheduling results after applying the speculative code the dpframe and tiler designs.*

Allowed Code Motions	*MPEG*-2 *dpframe* # States	*MPEG*-2 *dpframe* Long Path	*GIMP tiler* # States	*GIMP tiler* Long Path
Within basic blocks	77	911	69	6144
+across hier blocks	72(-6.5 %)	863(-5.3 %)	66(-4.3 %)	5944(-3.3 %)
+early cond exec	71(-1.4 %)	859(-0.5 %)	63(-4.5 %)	5544(-6.7 %)
+speculation	58(-18.3 %)	607(-29.3 %)	54(-14.3 %)	4734(-14.6 %)
+cond speculation	49(-15.5 %)	571(-5.9 %)	31(-42.6 %)	2534(-46.5 %)
Total Reduction	**36.4 %**	**37.3 %**	**55.1 %**	**58.8 %**

motions are enabled. The percentage reductions of each row over the *previous* row are given in parentheses.

As each code motion is enabled, we see significant reductions in both the number of FSM states and the cycles on the longest path. Enabling code motions across hierarchical blocks lead to improvements of between 4 to 14 % in the number of states and between 3 to 5 % in the number of cycles. The largest improvements in performance (cycles) are obtained by employing speculation and conditional speculation. Speculation and conditional speculation lead to improvements between 3 to 29 % and between 5 to 46 % in the number of cycles respectively (over the results from the previous rows). The number of states also decrease by 8 to 18 % and 15 to 42 % for these code motions. In contrast, early condition execution, which uses reverse speculation to move operations down into conditional branches, leads to only marginal or no improvements for the four designs.

The results in Tables 10.6 and 10.7 demonstrate that speculative code motions lead to substantial improvements in the latency of the design and complexity of the controller. The total reduction in cycles on the longest path and number of states achieved with all the code motions enabled over code motions only within basic blocks

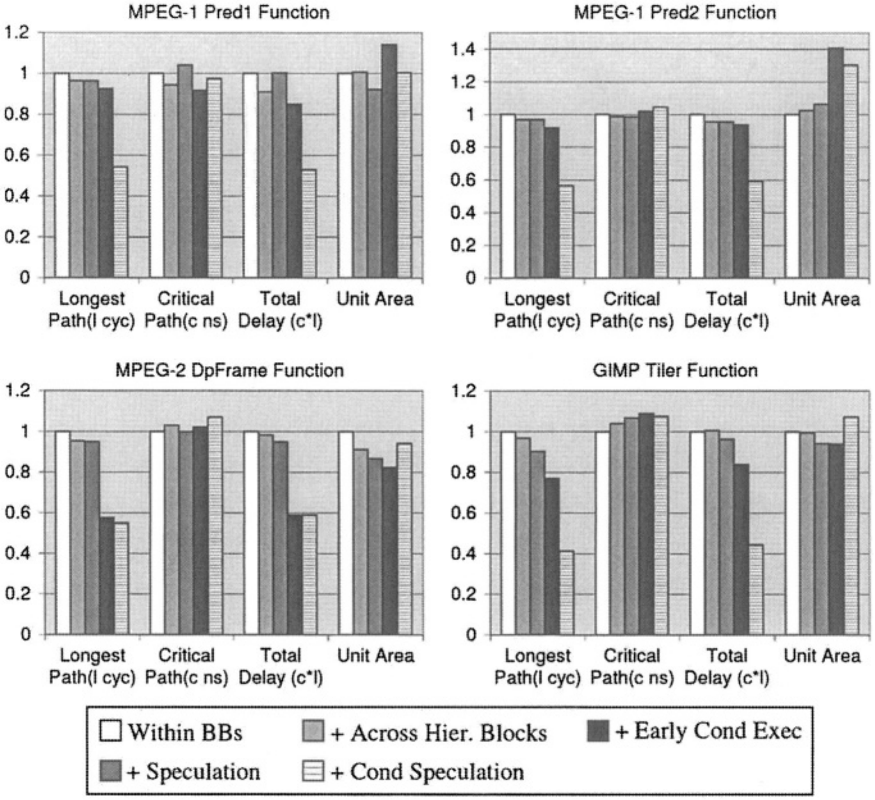


Figure 10.3: ***Effects of Speculative Code Motions on logic synthesis results for the four case studies.***

ranges between 36 % to 58 % (last row in the tables). Note that, when code motions within basic blocks and across hierarchal blocks are enabled, the priority-list scheduling heuristic we have presented reduces to the classical *non-speculative* list scheduling approaches presented in previous works [DM94, GDWL92].

10.3.2 Logic Synthesis Results for the Speculative Code Motions

The logic synthesis results for these experiments are presented in Figure 10.3. The bar charts in this figure plot the same logic synthesis metrics as the previous section. The bars in these charts present results for code motions within basic blocks only (first bar), for code motions across hierarchical blocks enabled as well (second bar), early condition execution enabled also (third bar), speculation enabled as well (fourth bar), and finally with conditional speculation enabled as well (fifth bar). Thus, the fifth bar represents the results with all the speculative and non-speculative code motions enabled and the second bar represents the results when only the non-speculative code motions are enabled.

These results show that the critical path length remains fairly constant as these code motions are enabled. This means that the clock period does not increase by applying these code motions. The constant critical path length, coupled with large decreases in cycles on the longest path, leads to large decreases in the total delay through the circuit.

However, code motions such as speculation and conditional speculation can lead to a large increase in area, as we can see from the area results in the bar charts in Figure 10.3. This area increase is due to the increasing complexity of control and multiplexing logic caused by the code motions. This increasing complexity, in turn, is because the speculative code motions schedule the design in fewer cycles than the non-speculative code motions. Thus, resource utilization and resource sharing increases and this leads to increase in the complexity of the multiplexers and associated control logic. This complexity increase is particularly large due to conditional speculation because it duplicates operations and thus, more operations are mapped to the same number of resources as before.

10.4 Results for Dynamic CSE

In this section, we compare the effectiveness of the dynamic CSE transformation applied *during* scheduling with a traditional CSE pass applied *before* scheduling. Once again, scheduling and logic synthesis results are presented.

10.4.1 Scheduling Results for Dynamic CSE

The scheduling results for these experiments are presented in Tables 10.8, 10.9, 10.10, and 10.11 for the four designs. The first row in these tables lists results for the baseline case with all code motions enabled, along with copy propagation and dead code elimination. The second row is for when only CSE is applied as a pass before scheduling, the third row for when only dynamic CSE is applied during scheduling and finally, the fourth row presents results for when both CSE and dynamic CSE are applied. In all these experiments, dynamic copy propagation is done whenever possible (even when dynamic CSE is not applied). The percentage reductions of each row over the baseline case are also given in parentheses. These tables also give the number of registers required to bind the variables in the designs.

The results in these tables demonstrate that applying CSE alone can lead to improvements between 5 to 23 % in the number of states and between 10 to 28 % in the cycles on the longest path. In themselves, these improvements are significant. When dynamic CSE is applied, the improvements are even more dramatic for all the designs, as is evident by the results in the third row of Tables 10.8, 10.9, 10.10, and 10.11. Dynamic CSE is able to eliminate many more operations with common sub-expressions than traditional CSE can. Employing dynamic CSE during scheduling reduces schedule length (cycles) by 10 to 35 % and the number of states by 5 to 31 % over the baseline case. The last rows in these tables show that applying both CSE and dynamic CSE together leads to further improvements of a few cycles for the MPEG-2 *dpframe* design.

Table 10.8: *Scheduling results after applying CSE and Dynamic CSE for MPEG-1* $pred1$ *design.*

Transformation	$MPEG$-1 $pred1$		
Applied	# States	Long Path	# Regs
Baseline	40	1824	17
Base + CSE	36(-10 %)	1504(-17.5 %)	14(-17.6 %)
Base + Dyn CSE	32(-20 %)	1184(-35.1 %)	10(-41.2 %)
Base + CSE + DCSE	32(-20 %)	1184(-35.1 %)	10(-41.2 %)

Table 10.9: *Scheduling results after applying CSE and Dynamic CSE for MPEG-1* $pred2$ *design.*

Transformation	$MPEG$-1 $pred2$		
Applied	# States	Long Path	# Regs
Baseline	84	4187	31
Base + CSE	73(-13.1 %)	3482(-16.8 %)	25(-19.4 %)
Base + Dyn CSE	65(-22.6 %)	2906(-30.6 %)	18(-41.9 %)
Base + CSE + DCSE	65(-22.6 %)	2906(-30.6 %)	17(-45.2 %)

Table 10.10: *Scheduling results after applying CSE and Dynamic CSE for the MPEG-2* $dpframe$ *design.*

Transformation	$MPEG$-2 $dpframe$		
Applied	# States	Long Path	# Regs
Baseline	53	672	32
Base + CSE	50(-5.7 %)	602(-10.4 %)	31(-3.1 %)
Base + Dyn CSE	50(-5.7 %)	602(-10.4 %)	28(-12.5 %)
Base + CSE + DCSE	49(-7.5 %)	598(-11 %)	30(-6.3 %)

Table 10.11: *Scheduling results after applying CSE and Dynamic CSE for the GIMP* $tiler$ *design.*

Transformation	$GIMP$ $tiler$		
Applied	# States	Long Path	# Regs
Baseline	42	3931	14
Base + CSE	31(-26.2 %)	2831(-28 %)	15(+7.1 %)
Base + Dyn CSE	29(-31 %)	2631(-33.1 %)	15(+7.1 %)
Base + CSE + DCSE	29(-31 %)	2631(-33.1 %)	14(0 %)

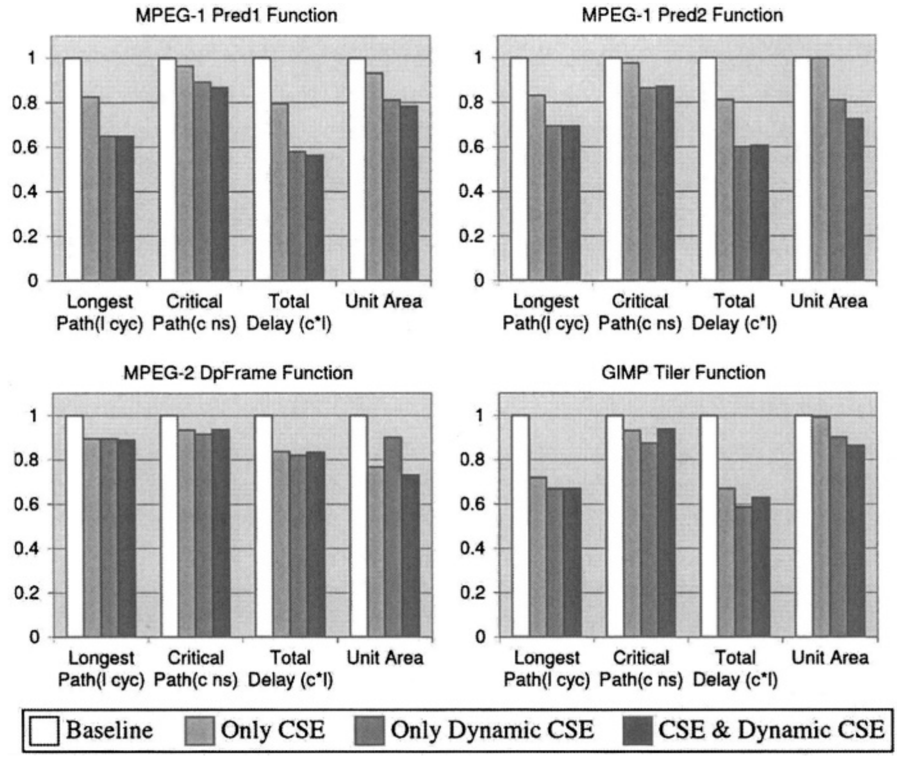


Figure 10.4: ***Effects of the pre-synthesis transformations, loop-invariant code motion (LICM) and common sub-expression elimination (CSE), on logic synthesis results for the four designs.***

Our experiments show another important result. Contrary to common belief, the results show that applying CSE and dynamic CSE leads to a *reduction* in the number of registers required (see last column in the four tables). This decrease can be attributed to three inter-related factors: (a) the reduced schedule lengths imply shorter variable lifetimes, especially for variables whose results are required for future loop iterations; (b) elimination of an operation by CSE means that instead of requiring two registers to store the two variables/operands that are read by the operation, only one register is required to store the result of the operation; and (c) when operations with the same sub-expression are eliminated, then they can reuse the result of only one of the operations. This saves on storing the results of several operations.

10.4.2 Logic Synthesis Results for Dynamic CSE

The logic synthesis results for the experiments using CSE and dynamic CSE are presented in the graphs in Figure 10.4. The values of each metric are mapped as before:

for when all the code motions are enabled but no CSE or dynamic CSE is applied (first bar), for when only CSE is applied (second bar), when only dynamic CSE is applied (third bar) and the last bar is for when both CSE and dynamic CSE are applied.

From the results in these graphs, we find that dynamic CSE leads to a shorter critical path length and smaller circuit area than applying only CSE. The decreases in critical path length, coupled with the reductions in cycles on the longest path we saw earlier, lead to dramatic reductions in the total delay when dynamic CSE is applied: from about 20 % (for *dpframe)* to 40 % (for *pred*1, *pred*2 and *tiler).* Also, when dynamic CSE and CSE are applied together, it consistently leads to lower area; sometimes up to 30 % less (for *pred*2 and *dpframe).* This decrease in area can be attributed to two factors. Firstly, the elimination of some operations due to CSE and dynamic CSE means that fewer operations are mapped to the functional units and this leads to reduced interconnect (multiplexers and de-multiplexers). Secondly, reductions in the controller size and the number of registers lead to further reductions in area.

The overall results in the graphs in Figure 10.4 demonstrate that enabling dynamic CSE reduces the total delay through the circuit by up to 40 % while at the same time reducing the design area. These improvements are better than applying only CSE before scheduling. Also, these results validate our belief that transformations applied dynamically during scheduling can exploit several new opportunities created by scheduling decisions and the movement of operations due to the speculative code motions.

10.5 Results for Chaining Across Conditionals

We developed the chaining across conditionals transformation primarily for the synthesis of microprocessor functional blocks [GKK$^+$02]. However, even in the domain of the multimedia and image processing applications discussed in this chapter, we find that there exist several opportunities to chain simple assign (copy) operations that occur in conditional blocks with operations that produce their values. This sometimes can generate a result one cycle earlier than it would otherwise would have been available (see Section 6.6).

In Tables 10.12 and 10.13, we compare the scheduling results of the baseline case for the four designs (first row) with the results for when chaining across conditionals is enabled (second row). We obtain marginal improvements: between 2 to 7 % in the number of states and only between 0.3 to 7 % in the cycles on the longest path.

The logic synthesis results for the four designs corresponding to these experiments are presented in the bar charts in Figure 10.5. The first bar is the baseline case, while the second bar is with chaining across conditionals enabled. These results demonstrate that chaining operations across conditionals leads to only marginally better results (and sometimes worse results).

Chaining operations across conditionals (even just variable copy operations) packs more operations into a cycle and thus leads to more complex multiplexers and associated control logic. The corresponding decrease in the controller size is modest. Clearly, chaining across conditionals is not a useful transformation for the class of multimedia and image processing applications considered by our work. However, this does not

Table 10.12: *Scheduling results for chaining operations across conditionals for the MPEG-1 designs.*

Transformation Applied	$MPEG$-1 $pred1$ # States	$MPEG$-1 $pred1$ # cycles	$MPEG$-1 $pred2$ # States	$MPEG$-1 $pred2$ # cycles
Baseline	40	1824	84	4187
+Chaining	37(-7.5 %)	1813(-0.6 %)	78(-7.1 %)	4151(-0.9 %)

Table 10.13: *Scheduling results after chaining operations across conditionals for the MPEG-2 and GIMP designs.*

Transformation Applied	$MPEG$-2 $dpframe$ # States	$MPEG$-2 $dpframe$ # cycles	$GIMP$ $tiler$ # States	$GIMP$ $tiler$ # cycles
Baseline	53	672	42	3931
+Chaining	50(-5.7 %)	622(-7.44 %)	40(-2.4 %)	3921(-0.3 %)

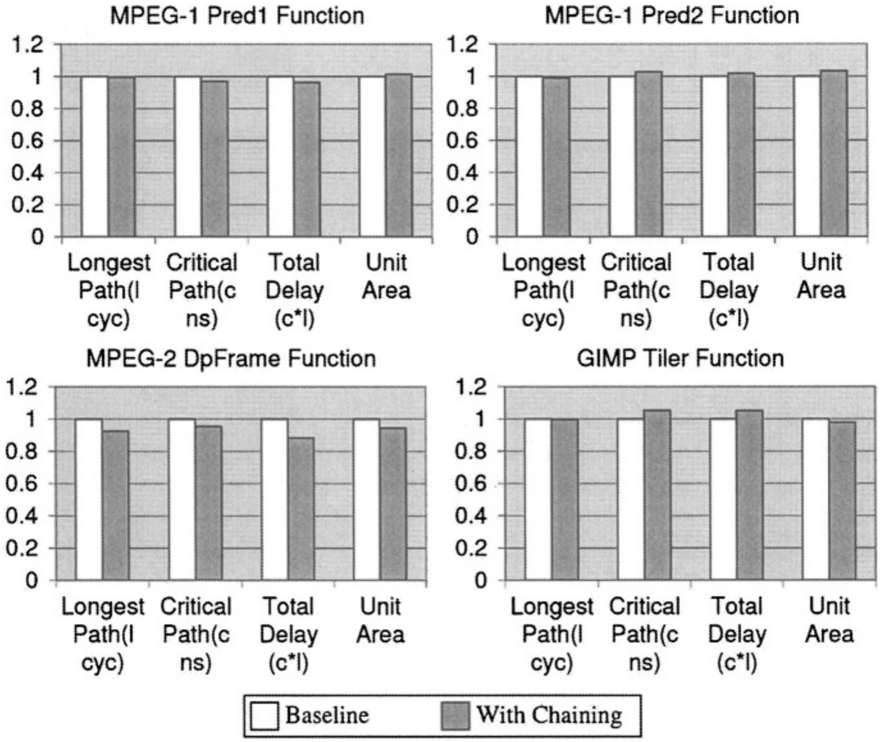


Figure 10.5: *Effects of chaining across conditionals on the logic synthesis results for the four designs.*

Table 10.14: *Overview of results obtained by applying the various transformations to the MPEG-1 designs.*

Transformation Applied	*MPEG*-1 *pred*1 # States	*MPEG*-1 *pred*1 Long Path	*MPEG*-1 *pred*2 # States	*MPEG*-1 *pred*2 Long Path
Non-Spec. CMs	60	2662	119	5927
+ Speculative CMs	40(-33.4 %)	1824(-31.5 %)	84(-29.4 %)	4187(-29.4 %)
+ LICM + CSE	40(0 %)	1091(-40.2 %)	74(-11.9 %)	2575(-38.5 %)
+ Dynamic CSE	37(-7.5 %)	899(-17.6 %)	67(-9.5 %)	2127(-17.4 %)
Total Reduction	**38.3 %**	**66.2 %**	**43.7 %**	**64.1 %**

Table 10.15: *Overview of results obtained by applying the various transformations to the dpframe and the tiler designs*

Transformation Applied	*MPEG*-2 *dpframe* # States	*MPEG*-2 *dpframe* Long Path	*GIMP tiler* # States	*GIMP tiler* Long Path
Non-Spec. CMs	79	1000	84	8131
+ Speculative CMs	53(-32.9 %)	672(-32.8 %)	42(-50 %)	3931(-51.7 %)
+ LICM + CSE	49(-7.5 %)	571(-15 %)	31(-26.2 %)	2534(-35.5 %)
+ Dynamic CSE	47(-4.1 %)	563(-1.4 %)	31(0 %)	2534(0 %)
Total Reduction	**40.5 %**	**43.7 %**	**63.1 %**	**68.8 %**

diminish the value of this technique; it is indispensable for the synthesis of microprocessor functional blocks (see Chapter 11).

10.6 Putting it all together

So far we analyzed the scheduling and logic synthesis results of the various transformations presented in this book. In this section, we present a comparative analysis of the pre-synthesis transformations, the speculative code motions and dynamics CSE. We first present this analysis for the four case studies used for the detailed experiments. We then demonstrate the utility of our transformations and transformation scripts over a large set of designs.

10.6.1 Overall Synthesis Results for the Four Case Studies

Tables 10.14 and 10.15 list the results after applying the speculative code motions, the pre-synthesis transformations, and dynamic CSE. In the experiments listed in the first row, we allowed code motions only within basic blocks and across hierarchical blocks, i.e., only the non-speculative code motions. In the second row, we enabled all the code

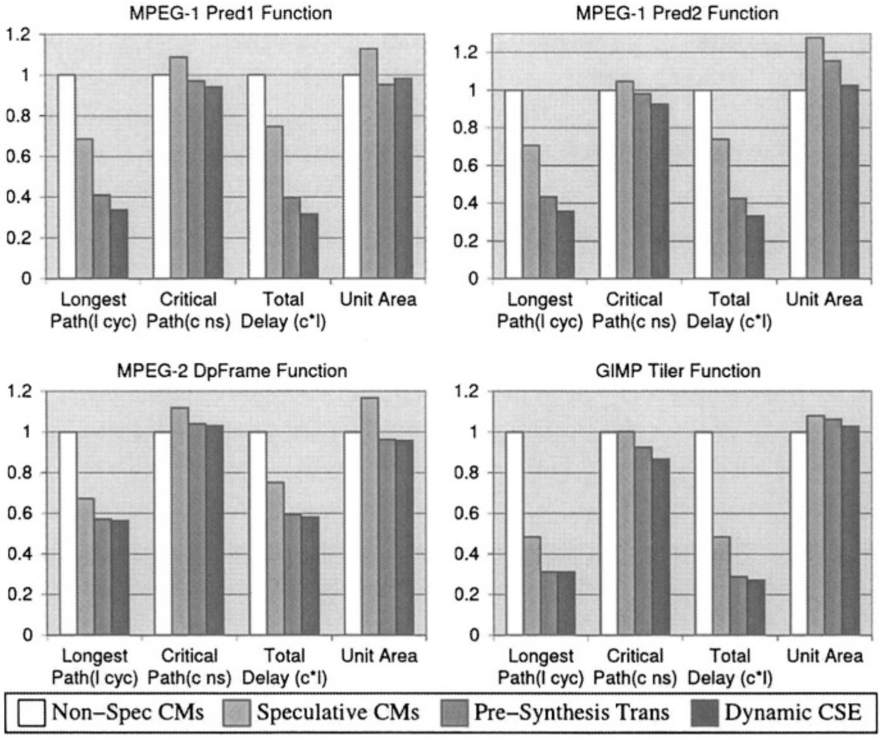


Figure 10.6: ***Final logic synthesis results for the*** *pred1*, *pred2*, *dpframe* ***and*** *tiler* ***designs with the speculative code motions, pre-synthesis transformations, and dynamic CSE***

motions, including the speculative code motions. In the third row, we applied the pre-synthesis transformations, namely, loop-invariant code motion (LICM) and common sub-expression elimination (CSE). In the fourth row, we also applied dynamic CSE during scheduling. The improvement of each row over the *previous* row is presented in parentheses. The last row lists the total improvements of all the transformations (fourth row) over the non-speculative code motions case (first row).

The results in these tables demonstrate that the speculative code motions and the pre-synthesis transformations result in performance improvements of up to 51 % and 40 % respectively. After applying the speculative code motions, the number of states reduce by 29 to 50 % and the number of cycles by 29 to 51 % and 40 to 51 %. The pre-synthesis transformations lead to *another* 0 to 26 % reduction in the number of states and 15 to 40 % reduction in the cycles on the longest path. Dynamic CSE is able to further improve the results by up to 9 % in the number of states and up to 17 % in cycles. Recall that CSE has already been applied as part of the pre-synthesis transformations. Hence, dynamic CSE is able to find several more opportunities for CSE during scheduling.

The logic synthesis results corresponding to these experiments are presented in Figure 10.6. The first bar corresponds to the case when the speculative code motions are not applied, the second bar has speculative code motions enabled, LICM and CSE are also applied for the results represented by the third bar, and finally, we apply dynamic CSE as well (fourth bar).

Once again, we can see from these graphs that the speculative code motions lead to improvements ranging from 25 to 50 % in the total delay through the circuit. Critical path increases marginally due to more complex multiplexing and control logic required to pack the operations in fewer cycles. This in turn leads to an increase in area by 10 to 25 %. However, when the pre-synthesis transformations are applied as well, both the area and the critical path length reduce. With pre-synthesis transformations and speculative code motions, the total delay through the circuit is 40 to 80 % less than over applying just the non-speculative code motions. The last bar shows that dynamic CSE leads to another improvement of 5 to 10 % in the total delay *and* area of the synthesized circuit. Keep in mind that the last 10 % improvements are usually the hardest to obtain.

10.6.2 Effect of Parallelizing Synthesis Transformations on Circuit Quality of a Large Set of Designs

So far we presented a detailed comparison of the synthesis results for individual compiler and synthesis transformations on four case studies: *pred*1, *pred*2, *dpframe* and *tiler*. In this section, we evaluate the effectiveness of the transformation scripts developed for the four case studies on a larger set of designs. We present logic synthesis results for twelve more designs for experiments that apply the speculative code motions, the pre-synthesis transformations and dynamic CSE. The characteristics of these twelve designs have been presented earlier in Section 10.1.2, along with the resources allocated to schedule them.

The overall synthesis results for these designs are presented in Figures 10.7 and 10.8. The first bar in the graphs in these figures plot the synthesis results corresponds to the case when the speculative code motions are not applied, the second bar has speculative code motions enabled, the pre-synthesis transformations are also applied for the results represented by the third bar, and finally, we apply dynamic CSE as well (fourth bar). The metrics mapped in these graphs are the cycles on the longest path, the critical path length, the total delay and the unit area.

The synthesis results for all these designs follow a similar pattern as the results presented in the previous section for the *pred*1, *pred*2, *dpframe* and *tiler* designs. By far, the speculative code motions and pre-synthesis optimizations have the maximum impact on the synthesis results. The overall improvements are in the range of 25 to 70 % for the total delay through the various designs. Also, once again for these 12 designs we see that the critical path length and area remain constant (or reduce marginally) when all the optimizations are applied.

The results in this section demonstrate the transformation scripts that we developed and tuned for the four case studies actually have utility to a larger set of designs.

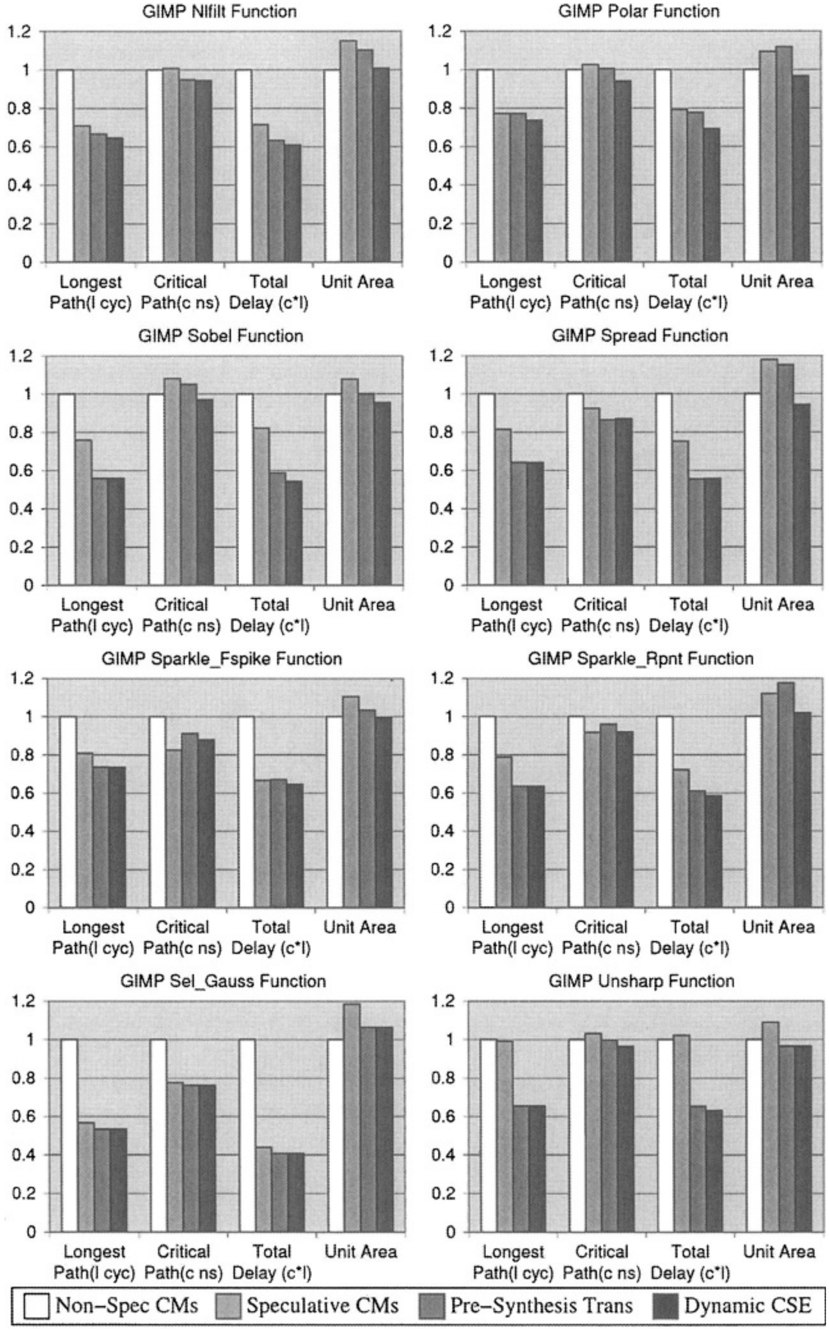


Figure 10.7: *Synthesis results for the GIMP transforms, nlfilt, polar, sobel, spread, sparkle_fspike, sparkle_rpnt, sel_gauss and unsharp, when the speculative code motions, pre-synthesis transformations, and dynamic CSE are applied.*

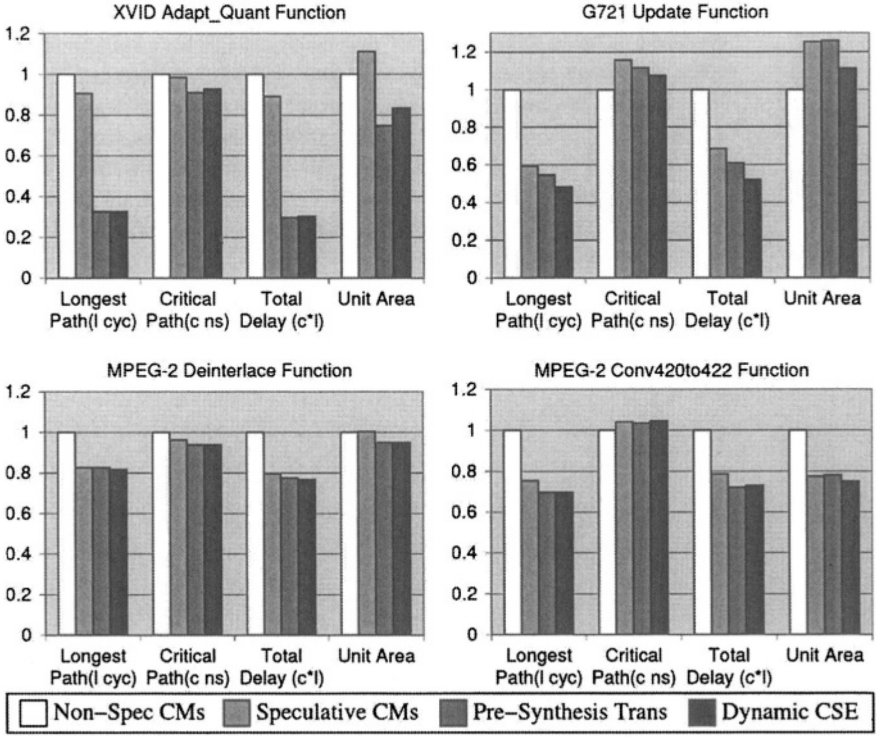


Figure 10.8: ***Synthesis results when the speculative code motions, pre-synthesis transformations, and dynamic CSE are applied to the functions:*** *XVID adapt_quant*, *G721 update* ***and the MPEG-2*** *deinterlace* ***and*** *conv420to422* ***functions.***

10.7 Study of Loop Unrolling and Loop Shifting

So far we presented synthesis results for the design examples based on transformations that look beyond conditional boundaries. In this section, we explore the next level of improvement that can be achieved using loop transformations that exploit parallelism across loop iterations. Specifically, we compare the impact of loop unrolling and loop shifting on the quality of synthesis results.

In the *Spark* synthesis framework, the number of unrolls and shifts for each loop is user-directed. *Spark* first unrolls the loop as specified by the designer and then schedules the design. After scheduling the design, the loop bodies are shifted, again as directed by the designer, and then rescheduled. In this section, we study the effects of different unrolling and shifting factors on hardware costs and circuit performance.

For the experiments presented below, we apply loop unrolling and loop shifting to inner loops of the three designs: MPEG-1 *pred*1 and *pred*2 and Gimp *tiler*. We start with a baseline that has **already been optimized** by all our previous transformations including the pre-synthesis transformations, the speculative code motions and the dynamic transformations.

Table 10.16: Scheduling results after unrolling the inner loops in *pred*1 and *pred*2.

Transform Applied	*MPEG*-1 *pred*1		*MPEG*-1 *pred*2	
	# States	# cycles	# States	# cycles
No Unrolls	37	899	67	2127
1 Unroll	48(+29.7%)	803(-10.7%)	79(+17.9%)	2031(-4.5%)
3 Unrolls	66(+37.5%)	723(-10%)	97(+22.8%)	1951(-3.9%)
Total Reduction	+78.4 %	-19.6 %	+44.8 %	-8.3 %

Table 10.17: Scheduling Results after unrolling the inner loop in *tiler*.

Transformation Applied	*GIMP tiler*	
	# States	# cycles
No Unrolls	31	2534
1 Unroll	52 (+67.7 %)	2284 (-9.9 %)
4 Unrolls	97 (+86.5 %)	1834 (-19.7 %)
Total Reduction	+212.9 %	-27.6 %

10.7.1 Scheduling and Logic Synthesis Results for Loop Unrolling

We first present the scheduling results for loop unrolling in Tables 10.16 and 10.17. The unroll factors are determined as follows: for a loop with an iteration count of *N*, we allow unrolling the loop by *M* times such that $N/(1 + M)$ is an integer and less than or equal to 1. The loops that are unrolled in *pred*1,*pred*2 and *tiler* have *N* equal to 8, 8 and 10 respectively. Hence, for *N*=8, possible values of *M* are 1, 3 and 7, and for *N*=10, *M* can be 1, 4 or 9.

The scheduling results are in terms of the number of states in the FSM controller and the cycles on the longest path. Longest path for loops is the cycles on the longest path through the loop body multiplied by the number of loop iterations. The first row in the two tables lists the results for the case when no loop unrolling is done, the second row for one loop unroll, and the third row for 3 loop unrolls for the *pred*1 and *pred*2 designs and 4 unrolls for *tiler*. The percentage reductions of each row over the previous row are given in parentheses. The last row gives the total reduction of the third row over the first row.

The results in the second row of Tables 10.16 and 10.17 show that with one unroll, we can achieve improvements ranging from 4 % to 10 % in the cycles on the longest path for the three designs. Unrolling the loop further (three times for the *pred*1 and *pred*2 designs and four times for *tiler*) leads to a further improvement of 10, 4 and 19.7 % respectively.

However, the results in Tables 10.16 and 10.17 also show that loop unrolling leads to a large increase in the size of the FSM controller (number of states). This is because

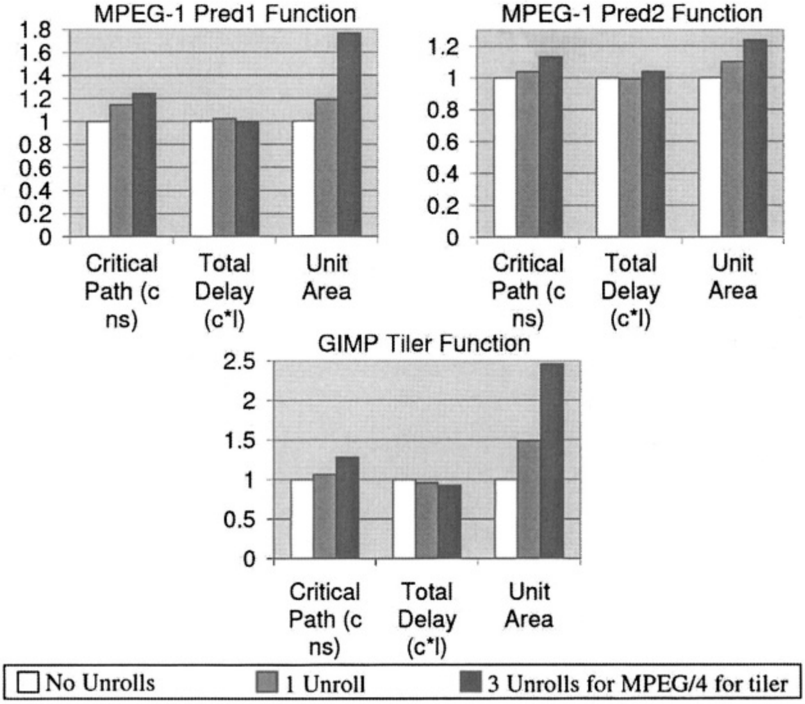


Figure 10.9: *Logic synthesis results after loop unrolling for pred1, pred2 and tiler*

when the loop body is duplicated, the number of control steps in the schedule increases, even though the number of executions of the loop body may reduce.

To study the impact on circuit area and delay, we performed logic synthesis on the RTL VHDL generated after scheduling, binding, and controller generation by the SPARK high-level synthesis tool. We used Synopsys Design Compiler with the TSMC 0.13 micron technology library. The logic synthesis results are presented in the graphs in Figure 10.9.

The results in these graphs show that when the loops are unrolled, the critical path lengths increase by 10 to 25 %. This increase works against the gains achieved in cycles through the longest path. As a result, the longest input to output delay or latency through the three designs remains almost constant as the loops are unrolled. However, there is a substantial increase in area – from 22 % to up to 150 %.

The increases in critical path length and area are due to the larger controller size and more complex steering logic (multiplexers, de-multiplexers and associated control logic). As the loops are unrolled, the number of operations in the design increases. Hence, a larger number of operations are mapped to the same number of resources. This increases resource utilization, which in turn leads to an increase in the size and complexity of the steering logic.

Table 10.18: *Scheduling results after loop shifting for the* $pred1$ *and* $pred2$ *designs. Further shifting does not improve scheduling results*

Transform Applied	*MPEG*-1 *pred*1		*MPEG*-1 *pred*2	
	# States	# cycles	# States	# cycles
No Shifts	37	899	67	2127
1 Shift	35(-5.4 %)	771(-14%)	65(-2.9 %)	1999(-6 %)
2 Shifts	36(+2.9 %)	779(+1 %)	66(+1.5 %)	2007(+0.4 %)
3 Shifts	35(-2.8 %)	715(-8.2 %)	65(-1.5 %)	1943(-3.2 %)
Total Reduction	-5.4 %	-20.5 %	-2.98 %	-8.7 %

Table 10.19: *Scheduling results after loop shifting for the* $tiler$ *design. Further shifting does not improve scheduling results*

Transform Applied	*GIMP tiler*	
	# States	# cycles
No Shifts	31	2534
1 Shift	29 (-6.5 %)	2334 (-7.9 %)
2 Shifts	29 (0 %)	2244 (-3.9 %)
3 Shifts	28 (-3.5 %)	2054 (-8.5 %)
Total Reduction	-9.7 %	-18.9 %

10.7.2 Scheduling and Logic Synthesis Results for Loop Shifting

Tables 10.18 and 10.19 list the scheduling results for the three designs as the inner loops are shifted, starting from no loop shifting (first row) to three shifts (fourth row). The percentage reductions of each row over the previous row are given in parentheses. The last row gives the total reduction of the fourth row (3 loop shifts) over the first row (no loop shifts).

The results in this table show that as the loops are shifted, the schedule length (cycles on the longest path) can sometimes increase. This happens when a set of concurrent operations is shifted from one branch of an already balanced conditional block. This means that, potentially, after shifting the scheduler is unable to compact the loop body to its size before shifting. However, in such a case, we can usually get back to the original schedule length by shifting once more; this time the scheduling step from the other branch of the conditional gets shifted.

If two consecutive shifts produce worse results, this indicates that we should stop shifting. The worse results mean that it is not possible to compact the loop body with any more shifted operations. Future work entails developing algorithmic techniques to determine the number of loop shifts. For now, we can experiment with different loop shifts due to low run times of our synthesis tool for fairly large designs.

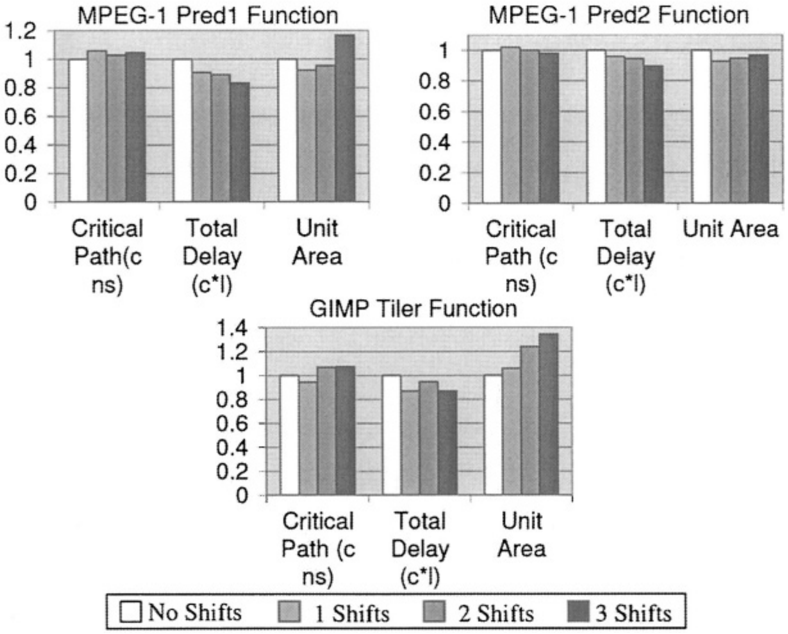


Figure 10.10: *Logic synthesis results after loop shifting for pred1, pred2 and tiler*

From the results in Tables 10.18 and 10.19, we can see that the best scheduling results are achieved for all the designs after shifting the loop 3 times. The total reductions (last row) range from 8 to 20 % in the cycles on the longest path and 2 to 9 % in the states in the FSM controller.

These scheduling results translate over to the critical path length and area results obtained after logic synthesis. These are presented in Figure 10.10. The bars in these graphs correspond to no shifts, 1 shift, 2 shift and 3 shifts. These logic synthesis results show that we can achieve 5-20 % improvements in the delay through the circuit by employing loop shifting, while incurring smaller increases in circuit area compared to loop unrolling. Recall that the base case (no shifts) represents a design that is already optimized by all the parallelizing compiler transformations presented earlier in this chapter.

10.7.3 Loop Unrolling and Shifting Results with Higher Resource Allocation

We ran our experiments again with a higher resource allocation for the MPEG designs (results are similar for the *tiler* design). We used 4 adders and 3 array decoders instead of 2 each, with other resources being the same as before (increasing the other resources did not affect scheduling results). The scheduling results for loop unrolling and shifting are listed in Tables 10.20 and 10.21 respectively. With a higher resource allocation, the

Table 10.20: *Loop unrolling with higher resource allocation: scheduling results for* $pred1$ *and* $pred2$

Transform Applied	$MPEG$-1 $pred1$ # States	$MPEG$-1 $pred1$ # cycles	$MPEG$-1 $pred2$ # States	$MPEG$-1 $pred2$ # cycles
No Unrolls	36	771	68	1935
1 Unroll	42(+16.7%)	611(-20.8%)	72(+5.9%)	1775(-8.3%)
3 Unrolls	56(+33.3%)	563(-7.9%)	86(+19.4%)	1727(-2.7%)
Total Reduction	+55.6 %	-27 %	+26.5 %	-10.7 %

Table 10.21: *Loop shifing with higher resource allocation: scheduling results for* $pred1$ *and* $pred2$

Transform Applied	$MPEG$-1 $pred1$ # States	$MPEG$-1 $pred1$ # cycles	$MPEG$-1 $pred2$ # States	$MPEG$-1 $pred2$ # cycles
No Shifts	36	771	68	1935
3 Shifts	36(0 %)	659(-14.5%)	66(-2.9%)	1823(-5.8%)
5 Shifts	36(0 %)	603(-8.5%)	66(0 %)	1767(-3.1%)
7 Shifts	37(+2.8%)	555(-8 %)	67(+1.5%)	1719(-2.7%)
Total Reduction	+2.8 %	-28 %	-1.5 %	-11.2 %

improvements are larger for both loop unrolling and loop shifting. Also, loop unrolling leads to a smaller increase in controller size because the unrolled operations get compacted more easily due to the higher resource allocation. Also, we are able to do more loop shifting: 7 shifts versus the earlier 3.

The logic synthesis results for these experiments with a higher resource allocation are presented in Figures 10.11 and 10.12. There results again demonstrate that loop unrolling leads to a large increase circuit area with only modest improvements in circuit delay. In contrast, loop shifting again leads to 5 to 25 % improvement in circuit delay with fairly constant circuit area. We also tried loop unrolling followed by loop shifting and found that the increase in circuit area and delay due to unrolling could not be compensated by loop shifting.

In conclusion, our experimental results show that loop shifting reduces delays by up to 20 % while area increases between 0-20 %. Also, these represent improvements over designs already optimized by the *Spark* synthesis framework using the entire range of parallelizing code motions and code transformations. In contrast, the control and multiplexing overheads of loop unrolling undo the gains achieved in schedule lengths. The length of the critical path (longest combinational path) and thus, circuit delay increases. Circuit area also increases by up to 150 %. Loop shifting, thus, represents a technique

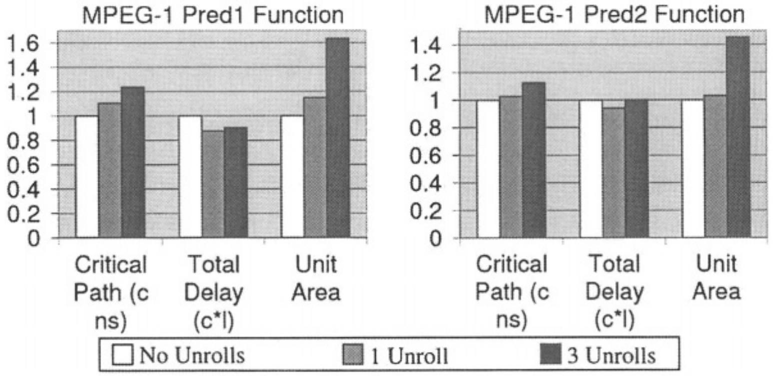


Figure 10.11: ***Logic synthesis results after loop unrolling using a higher resource allocation for the*** *pred*1 ***and*** *pred*2 ***designs***

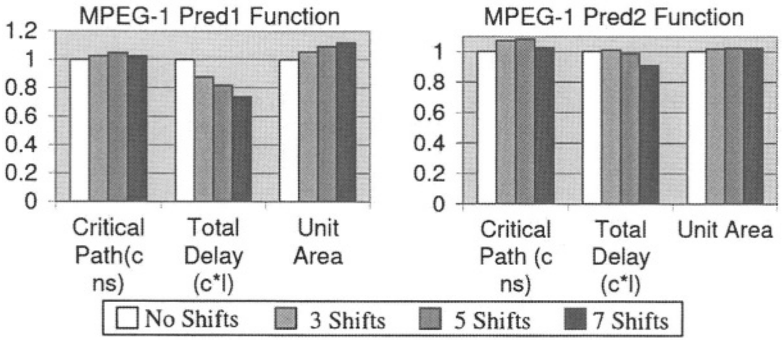


Figure 10.12: ***Logic synthesis results after loop shifting using a higher resource allocation for the*** *pred*1 ***and*** *pred*2 ***designs***

to incrementally exploit parallelism across loop iterations in designs with complex control flow without incurring the heavy penalties associated with loop unrolling.

10.8 Discussion and Conclusions

The overall improvements we obtain when we apply all the transformations in the *Spark* toolkit compared to when we apply only non-speculative code motions are up to 80 % in the total delay through the circuit with either no increase or a modest increase of 5 % in circuit area. Clearly, the optimizations we have developed, along with heuristics to guide them, are able to achieve high performance improvements with little or no degradation in circuit quality (area).

We have also done some preliminary analysis of the *power* requirements of the synthesized circuits using the Synopsys Power Compiler tool. We found that the switching power of the circuit follows the circuit area. Thus, as the various transformations are applied, the power profile looks similar to the area profile in the graphs in Figure 10.6. This, coupled with the speed up obtained after applying the transformations (delay reduces by 50 to 75 %), means that the *total energy* consumed by the circuit (= delay * power) reduces by the same amount as the delay when the optimizations in the *Spark* toolkit are applied.

We find that when an optimizing transformation is applied, there are two conflicting factors that come into play. As the resource utilization increases, the steering logic (multiplexers and de-multiplexers) connected to the functional units and the associated control logic increases. On the other hand, as the number of states in the controller decreases, the size and complexity of the controller decreases. We find that critical paths often originate in the controller, go through multiplexers, functional units and de-multiplexers, and finally, terminate in the registers that hold the results. Hence, optimizing transformations often lead to higher area and longer paths through the steering logic, but lower area and shorter paths through the FSM controller. Depending on the effectiveness of the transformation on the particular design being synthesized, one of these factors may overshadow the other. Also, the fact that the critical path length remains fairly constant as these optimizing transformations are applied is an important result because the critical path length dictates the minimum clock period for the design.

10.9 Summary

In this chapter, we presented scheduling and logic synthesis results for the various compiler, parallelizing compiler and synthesis transformations that have been implemented in the pre-synthesis and scheduling phases of the *Spark* framework. We first presented detailed experiments for each individual transformation for four case studies. The characteristics of these four case studies are described in Section 10.1.1. These designs are functions from control-intensive portions of the MPEG-1 and MPEG-2 multimedia applications and some image transforms from the GIMP image processing tool.

For these case studies, we first presented the scheduling and logic results for the pre-synthesis optimizations in Section 10.2. We then presented results for the speculative code motions in Section 10.3 and in Section 10.4, for the dynamic CSE technique. We presented results for operation chaining across conditional boundaries in Section 10.5. In Section 10.6, we presented a comparative study of the effectiveness of the pre-synthesis, speculative code motion and dynamic transformations first for the four case studies and then for a large set of designs derived from various multimedia and image processing applications. Finally, we studied the impact of loop unrolling and loop shifting on the MPEG-1 and GIMP designs.

The contribution of this chapter is a comparative study of the effectiveness of each individual transformation on both scheduling results and logic synthesis results. Also, we demonstrated the utility of our optimizations and algorithms for a larger set of designs.

11

CASE STUDY: SYNTHESIS OF AN INSTRUCTION LENGTH DECODER

11.1 Introduction

High-level synthesis has traditionally focused on the automated synthesis of ASIC (application specific integrated circuits) designs. Typical target architectures are multi-cycle designs with latencies in the 10s and 100s of cycles. These designs are usually area constrained and pipelining is generally the preferred means to improve system performance. Consequently, the classical high-level synthesis problem is one of transforming a behavioral description of an application through the scheduling and binding tasks, under constraints on the number of resources, into a multi-cycle schedule of operations.

In this chapter, we discuss a new target for high-level synthesis, namely, low latency functional blocks in microprocessors. High performance microprocessor designs are partially characterized by functional blocks consisting of a large number of operations that are packed into very few cycles (often a single-cycle) with little or no resource constraints but with tight bounds on the cycle time. Extreme parallelization and conditional and speculative execution of operations are essential to meet the processor performance goals. However, this is a tedious task for which classical high-level synthesis (HLS) formulations are inadequate and thus rarely used.

High performance microprocessor designs are typically considered to lie on the other end of the spectrum where much of the HLS optimizations in scheduling, resource binding and allocation do not find extensive use. There exist a good number of functional blocks within microprocessors, which are most naturally and succinctly described by a behavioral description. However, the lack of responsiveness to design constraints in HLS formulations leads to little or no use of traditional HLS tools in such high-performance functional blocks. The chief problem is that one major microprocessor design challenge – especially in the high end – is of identifying maximum parallelism and creating additional parallelization opportunities above and beyond those afforded by the algorithmic specification, and then packing all the resulting operations in a safe manner in the smallest number of cycles and in the shortest cycle time. Pure

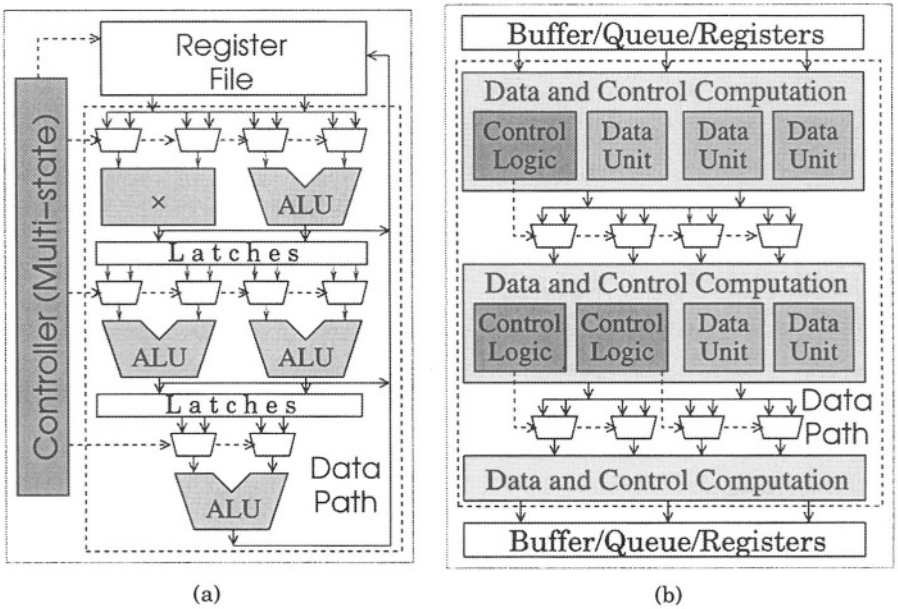


Figure 11.1: *Generic architectures (a) for ASIC designs and (b) for high performance microprocessor blocks.*

pipelining is of limited value since functional block latencies are critical in the presence of significant control in the behavior.

To understand the differences between the microprocessor synthesis domain and the ASIC synthesis domain, consider the generic architectures of microprocessor blocks and ASICs shown in Figure 11.1. ASICs, as shown in Figure 11.1 (a), are typically multi-cycle and pipelined, consisting of several functional units, steering logic (multiplexers), a controller (often a finite state machine) and a register file. Intermediate results are usually stored in latches and inter-stage forwarding paths may exist in the data path. On the other hand, as shown in Figure 11.1(b), microprocessor blocks are often single cycle and have several small computation blocks that operate in tandem and whose results are steered by control logic and in turn used by the control logic to generate control signals for other computation blocks. Inputs and outputs to these type of blocks are stored in memory elements such as buffers and queues.

Keeping these differences in mind, we used the *Spark* framework to provide a working environment for the microprocessor block designer to explore alternative designs and speed up the overall design process. Our work in this area has been motivated by the parallels between the nature of microprocessor functional blocks and the type of multimedia designs we have been looking at. Designs from both these domains have a complex mix of control and data and the quality of synthesis results depend on the amount of parallelism that can be extracted from the behavioral description. The challenge for us has been to identify and isolate a set of compiler and high-level transformations that are useful for the synthesis of microprocessor blocks. These are carefully guided by heuristics that maximize parallelism specifically by targeting loop constructs and efficiently chains operations to pack them into a few cycles. This, in essence, is the contribution of the *Spark* synthesis framework to the domain of microprocessor functional block design.

As a case study in understanding the complexity and challenges in the use of high-level synthesis for this domain, we walk the reader through the detailed design of an *instruction length decoder* block derived from the *Pentium*®-family of processors. The choice of this block is made for a few reasons: (a) it is moderately complex and yet small enough that a detailed walk through the design – in an attempt to understand the challenges in application of HLS to high-performance microprocessor block designs – is possible; (b) the synthesis of this design employs several parallelizing transformations that validate the underlying motivation for building *Spark;* (c) designs of this nature are most naturally described by a behavioral description rather than a structural model, making them ideal for HLS.

Previous work in high-level synthesis for microprocessor designs is limited. Brayton et al. [BCM+88] synthesized the 801 processor – a small processor with a simple data path. Gupta et al. [GKWB00] presented a synthesis strategy for long latency functional units that are then embedded into VLIW processors. In as far as we know, there has been no prior work on synthesizing the type of low latency functional blocks we explore in this chapter.

The rest of this chapter is organized as follows: the next section outlines synthesis transformations that are useful for microprocessor blocks and how they are employed by the *Spark* framework. Sections 11.3 and 11.4 describe the instruction length decoder and the steps employed in synthesizing this design. We conclude with a discussion and an outline of future work.

11.2 Synthesis Transformations for Microprocessor Blocks

Due to the presence of complex control in the behavioral descriptions of microprocessor blocks, we employ parallelizing transformations that move and reorder operations beyond conditionals and loops. The most useful parallelizing transformations of this type are a number of loop transformations and speculative code motions such as speculation and conditional speculation. In ASIC synthesis, the effectiveness of these code motions is often limited by the number of resources allocated to the design – a constraint that is more lax for microprocessor blocks.

The scope for application of code motions can be further increased by loop transformations such as *loop unrolling.* Loop unrolling was developed to enable software compilers to perform optimizations across loop iterations and facilitate global code optimizations [BGS94, Muc97]. However, loop unrolling can lead to code explosion; so, loops are unrolled one iteration at a time, followed by code compaction by parallelizing transformations, until no further improvements can be obtained. Loops are seldom unrolled fully.

On the other hand, for microprocessor functional blocks, loops are only a programming convenience and latency constraints generally dictate the amount of unrolling a loop has to undergo. For instance, if a design is targeted to, say, three clock cycles, it implies that *all* the operations within *all* the iterations of the loop have to be executed within these three cycles. Hence, when this design is mapped to hardware, it will generate a design in which the loop is, in essence, unrolled within these three cycles. Loops in single cycle designs must, of course, be unrolled completely.

We have already shown in Sections 5.3 and 5.4 in Chapter 5 that when loops are unrolled completely, we can apply another important technique, namely, *loop index variable elimination.* The combination of these two transformations can significantly increase the amount of parallelism exposed to the synthesis transformations and also, eliminate the data dependencies between the loop index variable and the operations within the loop body.

Another transformation that is important to the synthesis domain of microprocessor functional block is chaining of operations, especially *across conditional boundaries.* This transformation enables operations to be packed back-to-back within one or a few cycles with steering logic such as multiplexers between them. Chaining operations across conditional boundaries has also been discussed earlier in Chapter 6.

We implemented the various compiler and synthesis transformations required for the synthesis of microprocessor functional blocks have in the *Spark* framework. The rich set of tunable transformations in *Spark* enable the framework to aid in exploration of several alternative designs. Although *Spark* can apply the various transformations automatically, it also allows the designer to control the various passes and the degree of parallelization through script files. For example, the designer may specify which loops to unroll and by how much. This enables *Spark* to provide design alternatives that may not be obvious to a designer from the design's behavioral description. In the next few sections, we show how we used *Spark* to explore the architecture of a functional block from a modern microprocessor.

11.3 A Case Study: Instruction Length Decoder

An important component of the *Pentium®* microprocessor architecture is the *instruction length decoder* [Pen]. This component determines the starting byte and the length of each instruction from a stream of bytes it receives from the instruction cache. We consider an implementation of this microprocessor architecture in which the instructions are of a variable length ranging from 1 to 11 bytes and the decoder has to examine from 1 to 4 bytes to determine an instruction's length. Instead of processing a stream of bytes, the decoder examines a set of bytes in an instruction buffer at every cycle.

The instruction length decode works as shown in Figure 11.2. The decoder determines the length contribution of the first byte ($LengthContribution_1$), and checks to see whether it needs to examines the next byte as well (*Need_2nd_Byte*). If it does, then it determines $LengthContribution_2$ of the second byte, and checks to see if it needs the third byte, and so on. In this way, say, the ILD calculates that the first instruction is two bytes long, then it must determine the length of the next instruction that starts at the third byte in the instruction buffer, by (potentially) looking at the next four

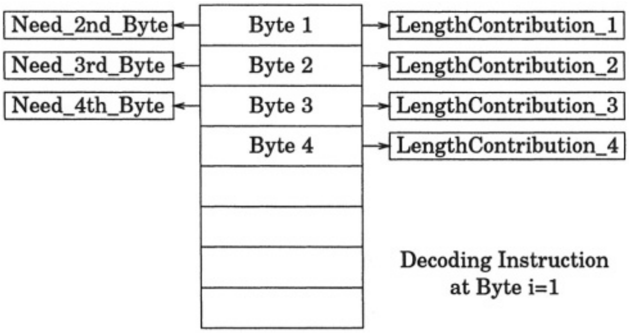


Figure 11.2: ***Instruction Length Decoder (ILD) of the Pentium microprocessor: calculating the length of the first instruction. Up to four bytes have to be processed to determine the length of an instruction.***

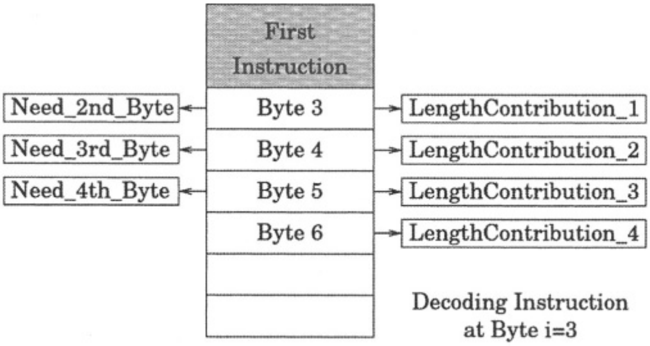


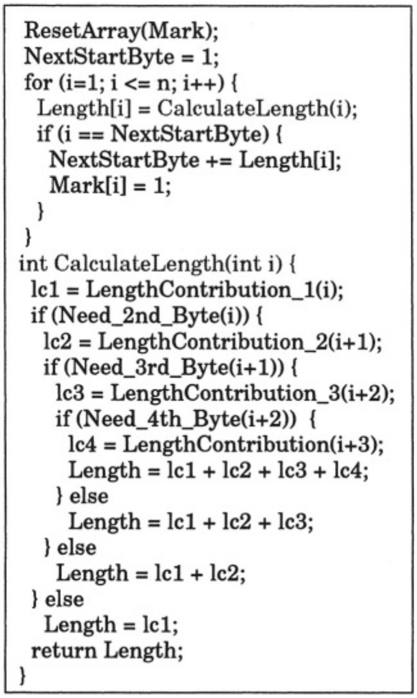
Figure 11.3: ***ILD: Calculating the length of the second instruction.***

bytes, as shown in Figure 11.3. This continues until the length of all the instructions in the buffer are determined.

A representation of this behavior in "C" is shown in Figure 11.4. In this code, a loop (indexed by i) iterates over the entire instruction buffer (of size n). If the start of the current instruction *NextStartByte* is the current byte i, then, it marks this as the starting point of an instruction ($Mark[i] = 1$) and calculates the length of the instruction at that byte by calling the function $CalculateLength(i)$. This function is the same as the behavior described above[1]. The final output of this description is a bit vector ($Mark[1..n]$) that contains a 1 at only those bit positions where an instruction starts.

There are several simplifications in this model of the ILD [Pen]. Since the ILD is decoding a stream of instructions arriving from memory, the behavioral description should have an infinite outer loop, which synthesis should break into chunks of n iterations each. Also, consider that an instruction starts at the $(n-1)$th byte. Then

[1] We assume a zero length contribution from the $n+1$ to $n+3$ bytes



```
ResetArray(Mark);
NextStartByte = 1;
for (i=1; i <= n; i++) {
  Length[i] = CalculateLength(i);
  if (i == NextStartByte) {
    NextStartByte += Length[i];
    Mark[i] = 1;
  }
}
int CalculateLength(int i) {
  lc1 = LengthContribution_1(i);
  if (Need_2nd_Byte(i)) {
    lc2 = LengthContribution_2(i+1);
    if (Need_3rd_Byte(i+1)) {
      lc3 = LengthContribution_3(i+2);
      if (Need_4th_Byte(i+2)) {
        lc4 = LengthContribution(i+3);
        Length = lc1 + lc2 + lc3 + lc4;
      } else
        Length = lc1 + lc2 + lc3;
    } else
      Length = lc1 + lc2;
  } else
    Length = lc1;
  return Length;
}
```

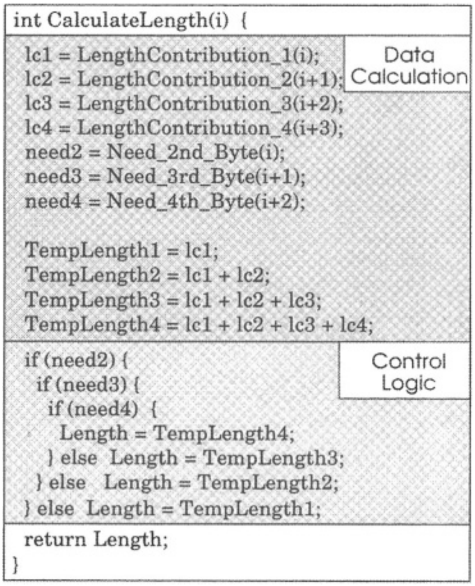
Figure 11.4: ***Behavioral "C" code for the instruction length decoder.***

the length calculation may need to check bytes from the next set of bytes that fill the buffer. So, the intermediate length calculation information must be saved across buffer decodes and passed to the next cycle. These simplifications are made to keep the discussion focused on the important code transformations used and do not alter the nature and applicability of the transformations presented here.

The processor architectural requirements imply that the whole buffer must be decoded in one cycle. Hence, a designer may choose to compute as much as possible in parallel and then, do the instruction marking after all the information has been calculated. The following section describes how *Spark's* synthesis methodology achieves this kind of a single cycle architecture for the decoder starting from a natural behavioral description.

11.4 Transformations applied by *Spark* to Synthesize the Decoder

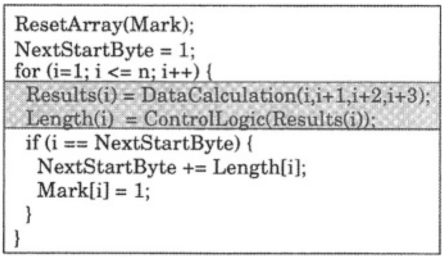
In order to achieve a single cycle architecture for the ILD design, the sequence of transformations applied by *Spark* follows the methodology outlined in Chapter 4. However, it is the scope of application of the transformations that changes. This means that we



```
int CalculateLength(i) {
  lc1 = LengthContribution_1(i);
  lc2 = LengthContribution_2(i+1);
  lc3 = LengthContribution_3(i+2);
  lc4 = LengthContribution_4(i+3);
  need2 = Need_2nd_Byte(i);
  need3 = Need_3rd_Byte(i+1);
  need4 = Need_4th_Byte(i+2);

  TempLength1 = lc1;
  TempLength2 = lc1 + lc2;
  TempLength3 = lc1 + lc2 + lc3;
  TempLength4 = lc1 + lc2 + lc3 + lc4;
  if (need2) {
    if (need3) {
      if (need4) {
        Length = TempLength4;
      } else  Length = TempLength3;
    } else   Length = TempLength2;
  } else  Length = TempLength1;
  return Length;
}
```

Figure 11.5: *All the data operations in the CalculateLength function are* speculatively *executed.*

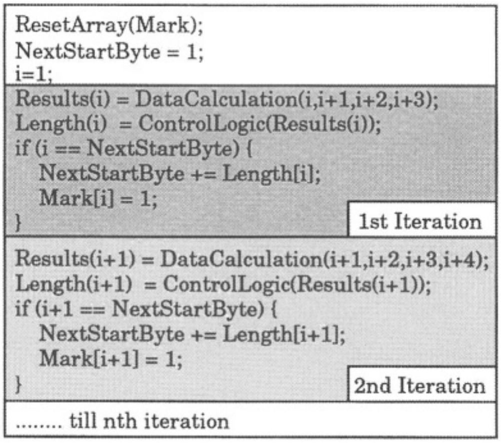


```
ResetArray(Mark);
NextStartByte = 1;
for (i=1; i <= n; i++) {
  Results(i) = DataCalculation(i,i+1,i+2,i+3);
  Length(i)  = ControlLogic(Results(i));
  if (i == NextStartByte) {
    NextStartByte += Length[i];
    Mark[i] = 1;
  }
}
```

Figure 11.6: *The instruction length calculation function is* inlined *into the main function.*

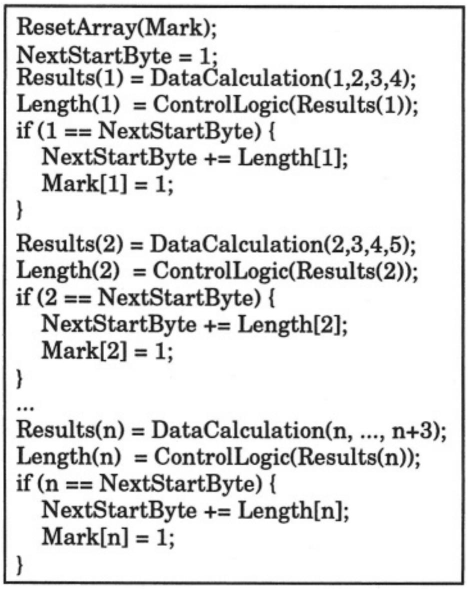
inline functions whenever possible, unroll loops fully, and apply the speculative code motions with an *unlimited* resource allocation.

Function inlining refers to replacing a call to a function or a subroutine with the body of the function or subroutine [ASU86]. This transformation allows the optimization of the inlined function with the rest of the code. Function inlining has not been implemented in *Spark.* Hence, we apply it manually to the source code. For the ILD design, this means that the function *CalculateLength* is inlined into the main ILD function.



```
ResetArray(Mark);
NextStartByte = 1;
i=1;
Results(i) = DataCalculation(i,i+1,i+2,i+3);
Length(i)  = ControlLogic(Results(i));
if (i == NextStartByte) {
   NextStartByte += Length[i];
   Mark[i] = 1;
}                                        1st Iteration
Results(i+1) = DataCalculation(i+1,i+2,i+3,i+4);
Length(i+1)  = ControlLogic(Results(i+1));
if (i+1 == NextStartByte) {
   NextStartByte += Length[i+1];
   Mark[i+1] = 1;
}                                        2nd Iteration
........ till nth iteration
```

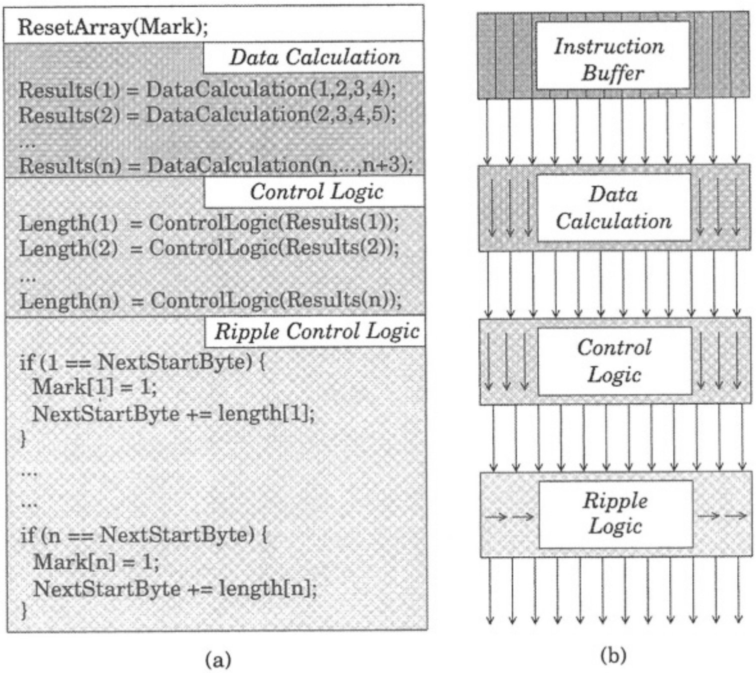
Figure 11.7: ***ILD: The loop with index variable i is*** unrolled ***fully; only two iterations are shown in this figure.***



```
ResetArray(Mark);
NextStartByte = 1;
Results(1) = DataCalculation(1,2,3,4);
Length(1)  = ControlLogic(Results(1));
if (1 == NextStartByte) {
   NextStartByte += Length[1];
   Mark[1] = 1;
}
Results(2) = DataCalculation(2,3,4,5);
Length(2)  = ControlLogic(Results(2));
if (2 == NextStartByte) {
   NextStartByte += Length[2];
   Mark[2] = 1;
}
...
Results(n) = DataCalculation(n, ..., n+3);
Length(n)  = ControlLogic(Results(n));
if (n == NextStartByte) {
   NextStartByte += Length[n];
   Mark[n] = 1;
}
```

Figure 11.8: ***Index variable elimination: Constant Propagation of loop index variable i after the loop has been fully unrolled.***

However, before we show the inlined version of this design, we will first demonstrate how the speculative code motions can speculatively compute all the data and control calculations in the function *CalculateLength.* This is shown in Figure 11.5; the length contributions due to the bytes, i through $i + 3$, are calculated speculatively



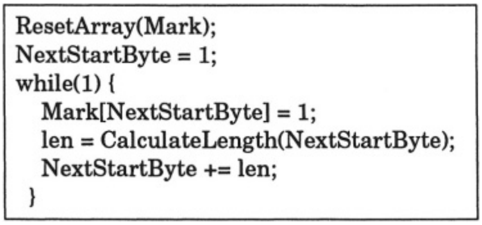
```
ResetArray(Mark);
                                   Data Calculation
Results(1) = DataCalculation(1,2,3,4);
Results(2) = DataCalculation(2,3,4,5);
...
Results(n) = DataCalculation(n,...,n+3);
                                   Control Logic
Length(1) = ControlLogic(Results(1));
Length(2) = ControlLogic(Results(2));
...
Length(n) = ControlLogic(Results(n));
                                   Ripple Control Logic
if (1 == NextStartByte) {
  Mark[1] = 1;
  NextStartByte += length[1];
}
...
...
if (n == NextStartByte) {
  Mark[n] = 1;
  NextStartByte += length[n];
}
```

Figure 11.9: *(a) Speculative calculation of all instruction lengths assuming an instruction starts at each byte (b) the final ILD architecture produced by the* Spark *framework.*

and so are the control variables *need*2 to *need*4 that determine which bytes contribute to the length of the current instruction. The lengths of the instruction for each case of these control variables (*TempLength*1 to *TempLength*3) are also speculatively computed. This results in a behavior where all the *data calculation* is performed up-front and speculatively, followed by a *control logic* structure that uses this data and assigns the correct result to the output. This control logic maps to multiplexers in hardware. Hereafter, for brevity of presentation, we will refer to these data and control components in the *CalculateLength* function as *DataCalculation* and *ControlLogic* respectively.

The *CalculateLength* function can be *inlined* into the main calling function as shown in Figure 11.6. As mentioned earlier, in practice, *Spark* performs inlining first, but speculation within the *CalculateLength* has been shown earlier to simplify explanation.

After inlining, the for-loop can be fully unrolled as shown by the code in Figure 11.7. To remove the dependency that still exists between the operations and the loop index variable i, we can propagate the constant assignment of $i = 1$ throughout the code and the loop index variable i can be eliminated. The resulting code is shown in



```
ResetArray(Mark);
NextStartByte = 1;
while(1) {
   Mark[NextStartByte] = 1;
   len = CalculateLength(NextStartByte);
   NextStartByte += len;
 }
```

Figure 11.10: *ILD: A succinct and natural behavioral description.*

Figure 11.8. This exposes further opportunities for early and speculative calculation of the lengths of instructions, as shown in Figure 11.9(a).

In the description in Figure 11.9(a), the lengths of the instructions are calculated by speculating that a new instruction starts at each byte. This leads to a design, where all the data for all the bytes is calculated concurrently, followed by a control logic unit that determines the length of the instructions if they were to start at each byte. Finally, a ripple control logic unit determines the actual start bytes of each instruction in the buffer. The hardware architecture corresponding to this code is shown in Figure 11.9(b). This is a maximally parallel architecture that can then be targeted for implementation in a single cycle.

Its important to note here that the sequence of application of the transformations is not as described above. This sequence has been chosen to aid in explanation of the case study. In practice, the pre-synthesis transformations, comprising of function inlining, loop unrolling, and loop index variable elimination, are applied first. The increased scope for parallelization that these transformations create is then exploited by the speculative code motions in the scheduling phase to achieve a single cycle schedule.

In this way, *Spark* starts with the “C” behavioral description of the ILD shown in Figure 11.4, and produces the register-transfer level VHDL code corresponding to the single cycle architecture shown in Figure 11.9. Although not demonstrated in this section, *Spark* chains the operations in the ILD together as described in Section 6.6 in Chapter 6 to achieve the single cycle architecture.

11.5 Future Work

While the case study of the instruction length decoder is instructive in understanding the code transformations needed for high performance microprocessor blocks, there are several areas of this high-level synthesis methodology that require further investigation. For instance, the behavioral description we used as a starting point for our work (Figure 11.4) may not be the simplest way to describe the design. A more natural and succinct way to describe the ILD’s behavior could be as shown in Figure 11.10. In this behavioral description, a stream of instructions are arriving in an infinite *while* loop. This means that we have to first split this infinite *while* loop into an outer infinite *while* loop and an inner *for* loop that operates on a fixed size buffer of bytes. Furthermore, if this fixed size buffer holds *N* bytes, then it is possible that an instruction at the end of a buffer requires a few more bytes from the next set of bytes that will load into the buffer to calculate the length of the instruction.

Similar, short behavioral descriptions can be used to describe several such low latency functional blocks in microprocessors. This leads us to future work of developing a new set of source-level transformations that can transform these type of descriptions into more easily synthesizable behavioral descriptions.

11.6 Summary

In this chapter, we demonstrated the way in which the *Spark* synthesis methodology can be used for the synthesis of low latency microprocessor functional blocks. In Section 11.1, we introduced the unique characteristics of microprocessor functional blocks and the demands that these characteristics place on the synthesis strategy, along with freedoms they give in terms of lax area constraints. In Section 11.2, we discussed the synthesis transformations that are required for synthesizing these function blocks and how the synthesis script employs these transformations. We then demonstrated the effectiveness of our methodology by presenting a case study of the instruction length decoder in the Intel Pentium, in Section 11.3. We then demonstrated (in Section 11.4) the step-by-step transformations applied by *Spark* to synthesize this design. In Section 11.5, we discussed future work to extend the methodology presented here. The contribution of this chapter is a synthesis methodology and the associated transformations for the synthesis of complex, low latency microprocessor functional blocks.

Part IV

Future Directions

12

CONCLUSIONS AND FUTURE WORK

12.1 Conclusions

In this book, we presented a synthesis methodology that optimizes the quality of synthesis results in the presence of complex control flow. Our synthesis methodology is based on employing aggressive coarse-grain and fine-grain parallelizing transformations to extract parallelism beyond conditionals and loops. This alleviates the effects of control constructs on the quality of synthesis results.

To improve controllability over the synthesis process and over the transformations applied to the design, we organized our optimizations into four groups: pre-synthesis transformations, scheduling transformations, dynamic transformations and basic compiler transformations. Furthermore, the synthesis framework is organized as a toolbox of transformations guided by heuristics that are fairly independent of the transformations. We also instrumented our synthesis framework with a scripting ability that provides knobs to experiment and tune the heuristics with and to identify the transformations that are useful in optimizing the overall circuit quality (delay and area).

Our synthesis methodology applies coarse-grain and fine-grain transformations during a pre-synthesis phase and a scheduling phase. These transformations consist of a range of compiler and parallelizing compiler optimizations, besides the traditional synthesis transformations. The parallelizing compiler transformations comprise of aggressive speculative code motions aided by transformations applied dynamically that take advantage of the movement of operations by the speculative code motions.

The speculative code motions extend the notion of speculation beyond just moving operations out of a conditional to moving and *duplicating* operations into conditional branches. Thus, these code motions re-order, speculate, and sometimes even increase the number of operations in a behavioral description so as to achieve higher quality of synthesis results and minimize the affects of control constructs on these results.

We implemented our parallelizing high-level synthesis methodology and the various transformations, along with the heuristics that guide them, in the *Spark* synthesis framework. *Spark* takes a behavioral description in ANSI-C as input and produces synthesizable RTL VHDL. This enables us to analyze the impact of the various transformations on the scheduling *and* logic synthesis results. Besides the speculative code motions and pre-synthesis transformations, we implemented a broad range of basic compiler and synthesis transformations in *Spark*. These include copy propagation, dead code elimination, operation chaining, ability to schedule on multi-cycle operations and so on. We also presented scheduling heuristics that guide the various transformations in our framework and an interconnect minimizing resource binding methodology.

We validated the utility of our synthesis methodology and the transformations and heuristics implemented in it by presenting results for experiments on several media designs. We first presented scheduling and synthesis results for experiments in which we applied each transformation individually on four designs. These four designs are derived from applications that are representative of the multimedia and image processing domains, namely, the MPEG-1, MPEG-2 and the GIMP applications. We then presented synthesis results for another set of twelve designs derived from various multimedia and image processing applications.

We found that the speculative code motions can improve performance and reduce controller size by up to 50 % when compared to list scheduling techniques that do not allow speculative code motions. Logic synthesis results show similar reductions in the total delays through the circuits, while maintaining critical path lengths fairly constant. Area of the synthesized netlist increases by up to 10-25 %, with most of the increases coming from operation duplication.

When pre-synthesis transformations (loop-invariant code motion and CSE) are applied before the scheduling phase improvements of up to 60 % can be obtained in the delay through the design with reductions of up to 20 % in the design area. Furthermore, these improvements are over a design that has already been optimized by the speculative code motions. Dynamic CSE leads to a further improvement of up to 10 % in performance. On average, with all the *Spark* optimizations enabled, we obtain performance and controller size improvements that are about 60 % with no increase in the critical path length and the area of the synthesized netlist.

Besides the multimedia and image processing domain, we also used the *Spark* synthesis framework to synthesize a functional block derived from a microprocessor. Functional blocks from the microprocessor domain are similar to multimedia designs in that they both have a complex mix of control and data operations. However, one important difference is that microprocessor blocks have to execute in one or a few cycles. Thus, microprocessor blocks are constrained by a low latency requirement and are allocated with virtually unlimited resources to achieve this latency.

We used the scripting facility in *Spark* to synthesize one such microprocessor functional block, namely, an instruction length decoder drawn from the *Pentium*®-family of processors. The synthesis methodology for low latency microprocessor functional blocks consists of a coordinated set of source-level and fine-grain compiler transformations that attempt to extract maximum parallelism in behavioral descriptions, specifically by targeting loop constructs in them. Also, this methodology employs techniques for efficient chaining of operations in order to pack all the operations into one cycle. The chief contribution of this work is the formulation of a domain-specific methodology for application of high-level synthesis techniques to a domain that rarely, if ever, finds use for it.

Lastly, we also demonstrated the utility of the scripting facility in *Spark* to experiment with different heuristics and transformations. Based on our experience, we suggest a recommended script for multimedia and image processing applications and another script for low latency designs from the microprocessor domain.

12.2 Future Work

The work presented in this book can be extended in the following directions:

- There is a need for more comprehensive cost models that incorporate the control and interconnect costs of the various code transformations. Scheduling heuristics can use these cost models to make area- or power-aware decisions while applying the code transformations.

- There are several other transformations that can be applied at the source-level [BGS94, Muc97] in the context of high-level synthesis such as function inlining, operation strength reduction [GMCG00], tree height reduction [NP91], and memory access and storage optimizations [Pan98, CWG+98, Sem01]. Although several of these transformations have been explored for high-level synthesis, their is limited understanding of the impact of these transformations on control and area costs for large designs with complex control flow.

- The impact of loop transformations on the quality of synthesis results needs further study. So far we studied loop shifting and partial loop unrolling to exploit inter-iteration parallelism [GDGN04]. We believe that coarse-grain loop optimizations such as loop fusion, loop interchange, and memory oriented loop transformations (i.e. transformations that reduce data transfers from memory) hold potential to lead to further improvements.

- The effects of the various code transformations on power are not well understood. Our preliminary experiments with RTL code power estimators indicate that the power profile follows the area of the final design. Since, after applying all the code optimizations, the area of the design is almost the same as that of the original code, we find that the power dissipation is also the same. Furthermore, since the code optimizations reduce the input-to-output delay through the design by more than 60 % on an average, this implies that the total energy consumed by the optimized design is also about 60 % less than the energy consumed by the initial design. In addition to this, it is possible to develop scheduling and post-scheduling transformations that specifically target reducing the energy and particularly, the peak power of the synthesized design.

- Several system-on-chip (SoC) platforms have emerged recently with programmable logic (such as FPGAs) on board. This programmable logic can be used as a programmable co-processor for reducing the computation load on the main processor core. In preliminary work, we demonstrated the utility of the *Spark* framework as part of a hardware-software co-design methodology to synthesize the computationally expensive kernels from an application to a programmable fabric on such a FPGA-based platform [LGD+03, GLD+03]. We compiled the remaining application code on to the processor core using the compiler suite available for the core. To generalize this methodology, an interface synthesis approach has to be developed that generates the interface through which hardware and software can communicate with each other.

➤ We demonstrated the utility of the *Spark* framework for the synthesis of microprocessor functional blocks. This work needs to be developed further and a comprehensive synthesis methodology is required for the synthesis of a class of designs from the microprocessor domain. This may require the development of a new set of transformations that target designs from this domain.

12.3 Summary

In summary, we proposed a solution that improves the state of the art by bringing a new range of code transformations to high-level synthesis. The *Spark* framework is a prototype that demonstrates that this technology can be successfully used to produce a high quality of results. We believe that our parallelizing high-level synthesis methodology can bring about an order of magnitude improvement in designer productivity by raising the level of abstraction in the microelectronic design process. Thus, our approach represents an important and big step in tackling the growing size and complexity of microelectronic systems.

Part V

Appendix

A

SPARK: USAGE, SYNTHESIS SCRIPTS, AND HARDWARE LIBRARY FILES

A.1 Command Line Interface

Spark can be invoked on the command-line by the command:

```
spark [command-line flags] filename.c
```

Some of the important command-line flags are:

Flag	Task Performed by Command-Line Flag
-h	Prints help
-hs	Schedule Design
-hvf	Generate RTL VHDL (output file is ./output/*filename_spark_rtl.*vhdl)
-hb	Do Resource Binding (operation to functional unit and variable to registers)
-hec	Statistics about states, path lengths are printed into RTL VHDL file
-hcc	Generate output C file of scheduled design in ./output/*filename_sparkout.c*
-hch	Chain operations across Conditional Boundaries
-hcp	Do Copy and Constant Propagation
-hdc	Do Dead Code Elimination
-hcs	Do Common Sub-Expression Elimination
-hli	Do Loop Invariant Code Motion
-hg	Generate graphs (output files are in directory ./output). Graphs are generated by default if scheduling is done, as: *filename_sched.*dotty
-hcg	Generate function call graph (./output/*CG_filename*.dotty)
-hq	Run quietly; no debug messages

Spark writes out several files such as the output graphs (see next section) and the backend VHDL and C files. All these files are written out to the subdirectory "output" of the directory from which *Spark* is invoked. This directory has to be created before executing *Spark.*

A.2 Viewing Output Graphs

The format that *Spark* uses for the output graphs is that of AT&TŠs *Graphviz* tool [Lab]. Output graphs are created for each function in the "C" input file. The output graphs generated are listed in the table below.

Graphs representing the original input file
CFG_filename_c_funcName.dotty
HTG_filename_c_funcName.dotty
DFG_filename_c_funcName.dotty
CDFG_filename_c_funcName.dotty

Graphs representing the scheduled design
CFG_filename_c_funcName_sched.dotty
HTG_filename_c_funcName_sched.dotty
DFG_filename_c_funcName_sched.dotty
CDFG_filename_c_funcName_sched.dotty

The key to the nomenclature used in tables above is:

Abbreviation	Description
CFG	*Control Flow Graph*: Basic blocks with control flow between them
HTG	*Hierarchical Task Graph*: Hierarchical structure of basic blocks
DFG	*Data Flow Graph*: Data flow between operations
CDFG	*Control-Data Flow Graph*: View of resource utilization in the CFG
filename	The name of "C" input file
funcName	The name of the function in input file

To view these output graphs, we use the *Graphviz* command line tool *dotty,* as follows:
dotty output /graph filename.dotty

A.3 Hardware Description File Format

Spark requires as input a hardware description file that has information on timing, range of the various data types, and the list of resources allocated to schedule the design (resource library). This file has to be named "filename.spark", where filename.c is name of the “C” input file. If a filename.spark does not exist, then the *Spark* executable looks for “default.spark”. One of these files *has to exist* for *Spark* to execute.

The various have sections in the .spark files are described in the next few sections. Note that, comments can be included in the .spark files by preceding them with “//”.

A.3.1 Timing Information

The timing section of the .spark file has the following format:

```
// ClockPeriod   NumOfCycles   TimeConstrained   Pipelined
[GeneralInfo]
10                    0               0              0
```

Of these parameters, only the clock period is used by the scheduler to schedule the design. The rest of the parameters have been included for future development and are not used currently. They correspond to the number of cycles to schedule the design in (timing constraint), whether the design should be scheduled by a time constrained scheduling heuristic, and whether the design should be pipelined.

A.3.2 Data Type Information

Each data type used in the “C” input file has to have an entry in the “[TypeInfo]” section of the .spark file, as shown below:

```
// typeName          lowerRange   upperRange
// or variableName   lowerRange   upperRange
[TypeInfo]
int                    -32767       32768
myVar                     0           16
```

This section specifies the range of the various data types that can be specified in a C description (such as int, char, float, unsigned int, signed int, long, double et cetera). The format is data type, lower bound range, and upper bound range. Also, the data value range of specific variables from the input C description can be specified in this section, as variable name, lower range, upper range. This is shown in the table above my the example of a variable “myVar” whose range is from 0 to 16.

A.3.3 Hardware Resource Information

The [Resources] section parameterizes each resource allocated to schedule the design as shown by the example below:
The example given above is:

- A resource called CMP (comparator).

//name	type	inpsType	inputs	number	cost	cycles	ns
[Resources]							
CMP	==,<,!=	i	2	1	10	1	10

- The operations ==, <, ! = can be mapped to this resource.
- It handles inputs of type integer (i).
- It has 2 inputs.
- There is one CMP allocated to schedule the design.
- Its cost is 10. The cost, although not used currently, can be integrated into module selection heuristics while selecting a resource for scheduling from among multiple resources.
- The CMP resource executes in 1 cycle.
- It takes 10 nanoseconds to execute.

This resource description section allows for multi-cycle resources but does not currently support structurally pipelined resources.

A.3.4 Loop Unrolling and Pipelining Parameters

// variable	maxNumUnrolls	maxNumShifts	percentThreshold	ThruputCycles
[RDLPParams]				
*	0	0	70	0
i	0	2	70	0

The [RDLPParams] section presents the parameters for loop unrolling and loop pipelining.

- Variable is the loop index variable to operate on ("*" means all loops)
- The second parameter specifies the number of times to unroll the loop.
- Number of times the loop should be shifted by the loop pipelining heuristic.
- Percentage threshold and throughput cycles are parameters used by the resource-directed loop pipelining (RDLP) heuristic implemented in our system.

The example in the second line of the "[RDLPParams]" section shown above says that the loop with loop index variable "i" should be shifted twice.

A.3.5 Other Sections in .spark files

The other sections in the .spark files are:

- ❒ [RDLPMetrics]: Controls the various parameters of the resource-directed loop pipelining (RDLP) heuristic.
- ❒ [SchedulerRules]: The file that *Spark* should read to get the scheduling scripts (rules and parameters). Default is: Priority.rules.
- ❒ [SchedulerScript]: The scheduling script to use: different scheduling heuristics can be employed by changing this entry. Default is “genericSchedulerScript”.
- ❒ [Verification]: Specifies the number of test vectors that should be generated for functional verification of output C with input C.

A.4 Scripting Options for Controlling Transformations

Spark allows the designer to control the transformations applied to the design description by way of synthesis scripts. In this section, we discuss the scripting options available to the designer.

The scripting options can be specified in the file given by the “[SchedulerRules]” section of the .spark file (see previous section). The default script file *Spark* looks for is “Priority.Rules”. This file has three main sections: the scheduler functions, the list of allowed code motions (code motion rules) and the cost of code motions. We discuss each of these in the next three sections. Note that in this file, “//” denotes that the rest of the line is a comment.

A.4.1 Scheduler Functions

An example of the scheduler functions section from a sample *Priority.rules* file is given in Figure A. 1. An entry in this section is of type “FunctionType=FunctionName”. A brief explanation for each function is given in the second part of the figure.

Of these we use the following flags for the experiments presented in this book: *DynamicCSE, PriorityType, BranchBalancingDuringCMs* and *BranchBalancingDuringTraversal.*

A.4.2 List of Allowed Code Motion

The [CodeMotionRules] section of the “Priority.rules” file has the list of code motions that can be employed by the scheduler. Each code motion can be enabled or disabled by setting the flag corresponding to it to “true” or “false”. An example of the list of allowed code motions sections is as given below.

```
// all the following can take values true or false
[CodeMotionRules]
RenamingAllowed=true                  // Variable Renaming allowed or not
AcrossHTGCodeMotionAllowed=true       // Across HTG code motion allowed or not
SpeculationAllowed=true               // Speculation allowed allowed or not
ReverseSpeculationAllowed=true        // Reverse Speculation allowed or not
EarlyCondExecAllowed=true             // Early Condition Execution allowed or not
ConditionalSpeculationAllowed=true    // Condition Speculation allowed or not
```

A.4.3 Cost of Code Motions

This section was developed to experiment with incorporating costs of code motions into the cost function based on which the operation to schedule is chosen. An example of this section is given below. Here all the code motions are assigned a cost of 1.

```
[CodeMotionCosts]
WithinBB                 1
AcrossHTGCodeMotion      1
Speculation              1
DuplicationUp            1
```

The total cost of scheduling an operation is determined as:
*Total Cost = - Basic Cost ofoperation * Cost Of Each Code Motion required to schedule the operation*

where basic cost of the operation is the priority of the operation (see Chapter 7) and cost of each code motion is as given in the "[CodeMotionCosts]" section of the Priority.rules file. Since this function generates a negative total cost, the operation with the lowest cost is chosen as the operation to be scheduled.

A.5 Sample default.spark Hardware Description file

```
//NOTE: do no put any comments within a section
// ClockPeriod  NumOfCycles  TimeConstrained   Pipelined
[GeneralInfo]
10      1       1       0

//typeName           lowerRange      upperRange
//or variableName    lowerRange      upperRange
[TypeInfo]
char                    0               8
signed_char             0               8
unsigned_char           0               8
```

```
short                   0               16
int                     0               32
unsigned_short          0               16
unsigned_int            0               32
long                    0               64
unsigned_long           0               64
long_long               0               128
unsigned_long_long      0               128
float                   0               32
double                  0               64
long_double             0               128
myVariableFromInput     0               4

// all cycles in resources have to be ns/ClockPeriod = cycles
//name  type  inpsType inputs number  cost    cycles  ns
[Resources]
ALU     +,-     i       2       1       10      1       10
MUL     *       i       2       1       20      2       20
CMP     ==,<    i       2       1       10      1       10
SHFT    <<      i       2       2       10      1       10
ARR     []      i       1       5       10      1       10

// variable  numUnrolls  numShifts  percentThreshold cycleThruput
[RDLPParams]
* 0 0 70 0

//unroll, shift, resetUnroll, and resetShift metrics
[RDLPMetrics]
UnrollMetric=RDLPGenericUnrollMetric
ShiftMetric=RDLPGenericShiftMetric
ResetUnrollMetric=RDLPGenericResetUnrollMetric
ResetShiftMetric=RDLPGenericResetShiftMetric

//lists file that has scheduler rules/functions
[SchedulerRules]
Priority.rules

//function that drives scheduler
[SchedulerScript]
genericSchedulerScript

// numOfTestVectors
[Verification]
20

[OutputVHDLRules]
```

```
PrintSynopsysVHDL=true
```

A.6 Recommended Priority.rules Synthesis Script file

We found the following choice of options in the synthesis script produces the best synthesis results for data-intensive designs with complex control flow.

```
//line format: <functiontype>=<functionvalue>
[SchedulerFunctions]
ScheduleRegionWalkerFunction=topDownBasicBlockNoEmpty
CandidateValidatorFunction=candidateValidatorPriority
CandidateMoverFunction=TbzMover
LoopSchedulingFunction=RDLP
CandidateRegionWalkerFunction=topDownGlobal
PreSchedulingStepFunction=preSchedulingPriority
PostSchedulingStepFunction=postSchedulingPriority
PreLoopSchedulingFunction=prepareForRDLP
PostLoopSchedulingFunction=constantPropagation
PreSchedulingFunction=initPriorities
ReDoHTGsForDupUp=false  // true or false - false is better
ReassignPriorityForCS=true      // true or false - true is better
PriorityType=max  // max, sum, maxNoCond - max is best
RestrictDupUpType=targetBBUnsched // or none or afterSchedOnce
BranchBalancingDuringCMs=true // true or false - true is better
BranchBalancingDuringTraversal=true   //  true  is better
DynamicCSE=true

[CodeMotionRules]
RenamingAllowed=true
AcrossHTGCodeMotionAllowed=true
SpeculationAllowed=true
ReverseSpeculationAllowed=true
EarlyCondExecAllowed=true
ConditionalSpeculationAllowed=true

// the higher the cost, the more profitable a code motion is
// total cost = basicCost * CostOfEachCodeMotion
[CodeMotionCosts]
WithinBB               1
AcrossHTGCodeMotion    1
Speculation            1
DuplicationUp          1
```

A.7 Recommended Command-line Options for Invoking *Spark*

We recommend the following command-line options for invoking *Spark:*

spark -hli -hcs -hcp -hdc -hs -hvf -hb -hec *filename*.c

The command-line options that are enabled are: loop-invariant code motion (-hli), common sub-expression elimination (-hcs), copy and constant propagation (-hcp), dead code elimination (-hdc), scheduling (-hs), generation of synthesizable RTL VHDL (-hvf), interconnect-minimizing resource binding (-hb) and generation of statistics about cycle count (-hec).

A.8 Options for Synthesizing Microprocessor Blocks

To synthesize microprocessor blocks, we have to enable operation chaining across conditional boundaries by using the command-line option *–hch* and we have to increase the clock period to a large number so that all the operations can be packed into one clock cycle. We arbitrarily increase clock period to 10000ns: this enables up to 1000 additions to be chained together (if each addition takes 10ns). The clock period can be set in the "[GeneralInfo]" section of the default.spark file (see Section A.3.1) and the timing of each operation in the design can be set in the "[Resources]" section (see Section A.3.3).

Also, we have to enable full loop unrolling. This can be done by setting the number of unrolls in the "[RDLPParams]" section of the default.spark file to the number of iterations of the loop to be unrolled (see Section A.3.4).

```
//line format: functiontype=functionvalue
[SchedulerFunctions]
 CandidateValidatorFunction=candidateValidatorPriority
 CandidateMoverFunction=TbzMover
 LoopSchedulingFunction=RDLP
 CandidateRegionWalkerFunction=topDownGlobal
 ScheduleRegionWalkerFunction=topDownBasicBlock
 PreLoopSchedulingFunction=prepareForRDLP
 PostLoopSchedulingFunction=constantPropagation
 PreSchedulingFunction=initPriorities
 PostSchedulingStepFunction=postSchedulingPriority
 PreSchedulingStepFunction=preSchedPriority
 ReDoHTGsForDupUp=false
 ReassignPriorityForCS=true
 RestrictDupUpType=targetBBUnsched
 PriorityType=max
 DynamicCSE=true
 BranchBalancingDuringCMs=true
 BranchBalancingDuringTraversal=true
```

```
CandidateValidatorFunction       // Candidate Validation Algorithm
CandidateMoverFunction           // use Trailblazing for code motions
LoopSchedulingFunction           // loop unrolling is done by RDLP
CandidateRegionWalkerFunction    // Algorithm to look for candidate operations
ScheduleRegionWalkerFunction     // schedule HTG top-down
PreLoopSchedulingFunction        // some initialization functions for loops
PostLoopSchedulingFunction       // optional post loop scheduling pass
PreSchedulingFunction            // Calculate priorities of operations before
                                 // scheduling
PostSchedulingStepFunction       // reverse speculate all unscheduled operations
PreSchedulingStepFunction        // the early condition execution algorithm
ReDoHTGsForDupUp                 // whether to reschedule HTGs for possible
                                 // duplication-up true or false
ReassignPriorityForCS            // Reassign priorities to favor operations within
                                 // basic blocks
RestrictDupUpType                // Restrict duplication-up
PriorityType                     // Calculate priority as max or sum of dependent
                                 // operations data dependencies
DynamicCSE                       // Whether Dynamic CSE is enabled or not
BranchBalancingDuringCMs         // Enable branch balancing during code motions
BranchBalancingDuringTraversal   // Enable branch balancing during design traversal
```

Figure A.1: *The scheduler functions section in a sample Priority.rules file.*

B

SAMPLE RUNS

B.1 A Sample Input C Program

```
int dest_cur[128];
void synthetic (int bytes, int x, int y)
{
 int val, data, a, col;

 for (col = 0; col < 10; col++)
  {
   data = dest_cur[col+bytes] ;
   val = data-y+2;
   if (col<x+2)
     a = x+2-col;
   else
    {
     a = col-x+2;
    }
   dest_cur[col+bytes+a] += val;
  } /* for (col = 0; col < 10; col++) */
}
```

B.2 Unbound VHDL output for the sample program

```
-- Automatically generated by the SPARK High-Level Synthesis System
-- Sat Jan 24 10:52:48 2004, source file : synthetic.c

-- 'SPARK' should be defined as the user package
PACKAGE spark_pkg is
  TYPE integer_vector IS ARRAY ( NATURAL RANGE <>) OF integer;
  TYPE boolean_vector IS ARRAY ( NATURAL RANGE <>) OF boolean;
  FUNCTION integer_wired_or ( arr_int : integer_vector ) RETURN integer;
  FUNCTION boolean_wired_or ( arr_bool : boolean_vector ) RETURN boolean;
  SUBTYPE wiredOrInt IS integer_wired_or Integer;
  SUBTYPE wiredOrBoolean IS boolean_wired_or boolean;
  TYPE ARRAY_127_0_32768_32767 is ARRAY(127 DOWNTO 0) of wiredOrInt range -32767 to 32768;
END spark_pkg;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;

PACKAGE BODY spark_pkg IS
  FUNCTION integer_wired_or ( arr_int : integer_vector ) RETURN integer is
```

```
    -- pragma resolution_method wired_or
    variable i : integer;
    variable returnVal : std_logic_vector(15 downto 0);
    variable arr_int_std_logic_vec : std_logic_vector(15 downto 0);
  BEGIN
    returnVal := (others => '0');
    for i in arr_int'range loop
      arr_int_std_logic_vec := conv_std_logic_vector(arr_int(i) , 16);
      returnVal := returnVal or arr_int_std_logic_vec;
    end loop;
    RETURN  conv_integer(returnVal);
  END integer_wired_or;

  FUNCTION boolean_wired_or ( arr_bool : boolean_vector ) RETURN boolean is
    -- pragma resolution_method wired_or
    variable i : integer;
    variable returnVal : boolean;
  BEGIN
    returnVal := FALSE;
    for i in arr_bool'range loop
      returnVal := raturnVal or arr_bool(i);
    end loop;
    RETURN returnVal;
  END boolean_wired_or;
end spark_pkg;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;
library work;
use work.spark_pkg.all;

ENTITY synthetic IS
port(
 bytes  : IN wiredOrInt range -32767 to 32768 ;
 x  : IN wiredOrInt range -32767 to 32768  ;
 y  :  IN wiredOrInt range -32767 to 32768 ;

-- global variables are
  dest_cur : INOUT ARRAY_127_0_32768_32767;
  CLOCK : IN std_logic ;
  RESET : IN std_logic ;
  done : OUT std_logic ) ;
END synthetic;

ARCHITECTURE rtl OF synthetic IS

   signal val : wiredOrInt range -32767 to 32768 ;
   signal data : wiredOrInt range -32767 to 32768 ;
   signal a : wiredOrInt range -32767 to 32768 ;
   signal col : wiredOrInt range -32767 to 32768 ;
   signal sT0_6 : wiredOrBoolean  ;
   signal sT1_8 : wiredOrInt range -32767 to 32768 ;
   signal sT2_8 : wiredOrInt range -32767 to 32768 ;
   signal sT3_10 : wiredOrInt range -32767 to 32768 ;
   signal sT4_10 : wiredOrBoolean ;
   signal sT5_11 : wiredOrInt range -32767 to 32768 ;
   signal sT6_14 : wiredOrInt range -32767 to 32768 ;
   signal sT7_16 : wiredOrInt range -32767 to 32768 ;
   signal sT8_16 : wiredOrInt range -32767 to 32768 ;
   signal sT9_16 : wiredOrInt range -32767 to 32768 ;
   signal sT10_11 :  wiredOrInt range -32767 to 32768 ;
   signal sTll_8:  wiredOrInt range -32767 to 32768  ;
   signal sT12_14 :  wiredOrInt range -32767 to 32768  ;
   signal sT13_11 :  wiredOrInt range -32767 to 32768 ;
   signal sT14_14 :  wiredOrInt range -32767 to 32768 ;
```

```
-- Statistics collected about this Schedule
-- minCycles = 51, maxCycles = 51, avgCycles = 51.0 in synthetic
-- Num of Ifs = 1, Nun of Loops = 1
-- Num of Non-Empty BBs = 9, Num of Ops = 20
-- Scheduled with the following resources
-- 2 ALU, 1 CMP, 1 ARR
-- Declarations of the 6 states in routine synthetic
subtype StateType is std_logic_vector(5 downto 0);
CONSTANT S_0 : std_logic_vector (5 downto 0) := "000001";
CONSTANT S_1 : std_logic_vector (5 downto 0) := "000010";
CONSTANT S_2 : std_logic_vector (5 downto 0) := "000100";
CONSTANT S_3 : std_logic_vector (5 downto 0) := "001000";
CONSTANT S_4 : std_logic_vector (5 downto 0) := "010000";
CONSTANT S_5 : std_logic_vector (5 downto 0) := "100000";
signal CURRENT_STATE : StateType;
signal NEXT_STATE : StateType;
BEGIN
  SYNC: PROCESS
  BEGIN
    wait until CLOCK’event and CLOCK = ’1’;
    if reset = ’1’ then
      CURRENT_STATE <= S_0;
      done <= ’0’;
    else
      CURRENT_STATE <= NEXT_STATE;
      if CURRENT_STATE /= S_0 and NEXT_STATE = S_0 then
        done <= ’1’;
      end if;
    end if;  -- if reset check
  END PROCESS;  -- SYNC Process

  FSM: PROCESS (CURRENT_STATE, sT4_10, sT0_6)
  BEGIN
    NEXT_STATE <= CURRENT_STATE;
    if CURRENT_STATE(0)  = ’1’  then
      NEXT_STATE <= S_1;
    elsif CURRENT_STATE(1)  = ’1’ then
      NEXT_STATE <= S_2;
    elsif CURRENT_STATE(2)  = ’1’  then
      if sT0_6 then
        NEXT_STATE <= S_3;
      else  --  sT0_6
        NEXT_STATE <= S_0;
      end if; -- conditions
    elsif CURRENT_STATE(3)  = ’1’  then
      if sT0_6 then
        if ST4_10 then
          NEXT_STATE <= S_4;
        else  --  sT4_10
          NEXT_STATE <= S_4;
        end if; -- conditions
      end if; -- conditions
    elsif CURRENT_STATE(4)  = ’1’  then
      if sT0_6 then
        if sT4_10 then
          NEXT_STATE <= S_5;
        else  --  sT4_10
          NEXT_STATE <= S_5;
        end if; -- conditions
      end if; -- conditions
    elsif CURRENT_STATE(5)  = ’1’ then
      NEXT_STATE <= S_1;
    END if;  --  if  (CURRENT_STATE)
  END PROCESS;  --  FSM Process

  DP: PROCESS

  BEGIN
```

```
      wait until CLOCK'event and CLOCK = '1';
      if reset = '1' then
        sT0_6 <= FALSE;
        sT4_10 <= FALSE;
      else   -- else of   if reset
        if  CURRENT_STATE(0)  = '1' then
          sT3_10  <=  (x + 2);
          col  <=  0;
        elsif CURRENT_STATE(1) = '1'  then
          sTll_8  <=  (col + bytes);
          sT12_14  <=  (col - x);
          sT0_6 <=  (col < 10);
        elsif CURRENT_STATE(2) = '1'  then
          if sT0_6 then
            sT13_ll  <=  (sT3_10 - col);
            sT14_14  <=  (sT12_14 + 2);
            sT4-10  <=  (col < sT3_10);
            data  <=  dest_cur(sTll_8);
          end if; -- conditions
        elsif CURRENT_STATE(3)  = '1'  then
          if sT0_6 then
            if sT4_10 then
              sT2_8  <=  (data - y) ;
              sT8_16  <=  (sTll_8  + sT13_ll);
            else  --  sT4_10
              sT2_8  <=  (data - y) ;
              sT8_16 <= (sT11_8 + sT14_14);
            end if; -- conditions
          end if; -- conditions
        elsif CURRENT_STATE(4)  = '1'  then
          if sT0_6 then
            if sT4_10 then
              val  <= (sT2_8 + 2);
              col  <=  (col + 1);
              sT9_16  <=  dest_cur(sT8_16);
            else  -- sT4_10
              val <= (sT2_8 + 2);
              col  <=  (col +1 ) ;
              sT9_16   <=   dest_cur(sT8_16);
            end if ; -- conditions
          end if; -- conditions
        elsif  CURRENT_STATE(5)  = '1'  then
          if sT0_6 then
            dest_cur(sT8_16) <= (sT9_16 + val) ;
          end if; -- conditions
        END if; -- if (CURRENT_STATE)
      end if; -- end of   if reset
    END PROCESS;  -- DP Process

END rtl;
```

B.3 Bound VHDL output for the sample program

```
-- Automatically generated by the SPARK High-Level Synthesis System
-- Sat Jan 24 10:51:54 2004, source file : synthetic.c

-- 'SPARK' should be defined as the user package
PACKAGE spark_pkg is
  TYPE integer_vector IS ARRAY ( NATURAL RANGE <>) OF integer;
  TYPE boolean_vector IS ARRAY ( NATURAL RANGE <>) OF boolean;
  FUNCTION integer_wired_or ( arr_int : integer_vector ) RETURN integer;
  FUNCTION boolean_wired_or ( arr_bool : boolean_vector ) RETURN boolean;
  SUBTYPE wiredOrInt IS integer_wired_or integer;
  SUBTYPE wiredOrBoolean IS boolean_wired_or boolean;
  TYPE ARRAY_127_0_32768_32767 is ARRAY(127 DOWNTO 0) of wiredOrInt range -32767 to 32768;
END spark_pkg;
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;

PACKAGE BODY spark_pkg IS
  FUNCTION integer_wired_or ( arr_int : integer_vector ) RETURN integer is
    -- pragma resolution_method wired_or
    variable i : integer;
    variable returnVal : std_logic_vector(15 downto 0);
    variable arr_int_std_logic_vec : std_logic_vector(15 dovnto 0);
  BEGIN
    returnVal := (others => '0');
    for i in arr-int'range loop
      arr_int_std_logic_vec := conv_std_logic_vector(arr_int(i), 16);
      returnVal := returnVal or arr_int_std_logic_vec;
    end loop;
    RETURN conv_integer(returnVal);
  END integer_wired_or;

  FUNCTION boolean_wired_or ( arr_bool : boolean_vector ) RETURN boolean is
    -- pragma resolution_method wired_or
    variable i : integer;
    variable returnVal : boolean;
  BEGIN
    returnVal := FALSE;
    for i in arr_bool'range loop
      returnVal := returnVal or arr_bool(i);
    end loop;
    RETURN returnVal;
  END boolean_wired_or;
end spark_pkg;

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;

ENTITY res_ALU IS
port(
  res_ALU_in0  : IN integer range -32767 to 32768;
  res_ALU_in1 : IN integer range -32767 to 32768;
  res_ALU_execOp : IN integer range 0 to 1;
  res_ALU_out : OUT integer range -32767 to 32768
);
END res_ALU;

ARCHITECTURE rtl of res_ALU IS
BEGIN
  res_ALU_out <= res_ALU_in0 +  res_ALU_in1   when (res_ALU_execOp = 0)
       else res_ALU_in0 -  res_ALU_in1;
END rtl; -- architecture of res_ALU

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;

ENTITY res_CMP IS
port(
  res_CMP_in0 : IN integer range -32767 to 32768;
  res_CMP_in1 : IN integer range -32767 to 32768;
  res_CMP_out : OUT boolean
);
END res_CMP;

ARCHITECTURE rtl of res_CMP IS
BEGIN
```

```
  res_CMP_out <= res_CMP_in0 < res_CMP_in1;
END rtl; -- architecture of res_CMP

library  IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;

ENTITY res_ARR IS
port(
  res_ARR_in0 :  IN integer range -32767 to 32768;
  res_ARR_out : OUT integer range -32767 to 32768
);
END res_ARR;

ARCHITECTURE rtl of res_ARR IS
BEGIN
  res_ARR_out <= res_ARR_in0;
END rtl; -- architecture of res_ARR

library  IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_signed.all;
library work;
use work.spark_pkg.all;

ENTITY synthetic  IS
port(
  bytes  :  IN wiredOrInt range -32767 to 32768;
  x :  IN wiredOrInt range -32767 to 32768;
  y :  IN wiredOrInt range -32767 to 32768;

-- global variables are
  dest_cur   :   INOUT ARRAY_127_0_32768_32767;
  CLOCK : IN std_logic ;
  RESET : IN std_logic ;
  done : OUT std_logic ) ;
END synthetic;

ARCHITECTURE rtl OF synthetic  IS

  COMPONENT res_ALU
  port(
    res_ALU_in0 :  IN integer range -32767 to 32768;
    res_ALU_in1 :  IN integer range -32767 to 32768;
    res_ALU_execOp :  IN integer range 0 to 1;
    res_ALU_out  : OUT integer range -32767 to 32768
  );
  END COMPONENT; -- end of component res_ALU

  COMPONENT res_CMP
  port(
    res_CMP_in0  :  IN  integer  range  -32767  to  32768;
    res_CMP_in1  :  IN  integer  range  -32767  to  32768;
    res_CMP_out  : OUT boolean
  );
  END COMPONENT; -- end of component res_CMP

  COMPONENT res_ARR
  port(
    res_ARR_in0 :  IN integer range -32767 to 32768;
    res_ARR_out :  OUT integer range -32767 to 32768
  );
  END COMPONENT;  -- end of component res_ARR

  signal sT0_6 : wiredOrBoolean  ;
  signal sT4_10  : wiredOrBoolean  ;
  signal res_ALU_0_in0 : integer range -32767 to 32768 := 0;
  signal res_ALU_0_in1  : integer range -32767 to 32768 := 0;
  signal res_ALU_0_execOp :  integer range 0 to 1 := 0;
  signal res_ALU_0_out : integer range -32767 to 32768 := 0;
  signal res_ALU_1_in0 : integer range -32767 to 32768 := 0;
  signal res_ALU_1_in1  : integer range -32767 to 32768 := 0;
  signal res_ALU_1_execOp : integer range 0 to 1 := 0;
  signal res_ALU_1_out : integer range -32767 to 32768 := 0;
  signal res_CMP_2_in0 : integer range -32767 to 32768 := 0;
  signal res_CMP_2_in1 : integer range -32767 to 32768 := 0;
  signal res_CMP_2_out  : boolean;
  signal res_ARR_3_in0 : integer range -32767 to 32768 := 0;
  signal res_ARR_3_out : integer range -32767 to 32768 := 0;
     -- Statistics collected about this Schedule
     -- minCycles = 51, maxCycles = 51, avgCycles = 51.0 in synthetic
     -- Num of Ifs = 1, Num of Loops = 1
     -- Num of Non-Empty BBs = 9, Num of Ops = 20
     -- Scheduled with the following resources
     -- 2 ALU, 1 CMP, 1 ARR, 1 dest_cur,
     -- Declarations of the 6 states in routine synthetic
  subtype StateType is std_logic_vector(5 downto 0);
  CONSTANT S_0 : std_logic_vector (5 downto 0) := "000001";
  CONSTANT S_1 : std_logic_vector (5 downto 0) := "000010";
  CONSTANT S_2 : std_logic_vector (5 downto 0) := "000100";
  CONSTANT S_3 : std_logic_vector (5 downto 0) := "001000";
  CONSTANT S_4 : std_logic_vector (5 downto 0) := "010000";
  CONSTANT S_5 : std_logic_vector (5 downto 0) := "100000";
  signal CURRENT_STATE : StateType;
  signal NEXT_STATE : StateType;
  -- Declarations of the 11 registers in synthetic
  signal regNum0 : integer range -32767 to 32768 := 0;
  signal regNum1 : integer range -32767 to 32768 := 0;
  signal regNum2 : integer range -32767 to 32768 := 0;
  signal regNum3 : integer range -32767 to 32768 := 0;
  signal regNum4 : integer range -32767 to 32768 := 0;
  signal regNum5 : integer range -32767 to 32768 := 0;
  signal regNum6 : Integer range -32767 to 32768 := 0;
  signal regNum7 : integer range -32767 to 32768 := 0;
  signal regNum8 : integer range -32767 to 32768 := 0;
  signal regNum9 : integer range -32767 to 32768 := 0;
  signal regNum10 : integer range -32767 to 32768 := 0;
  BEGIN
    res_ALU_instance_0 : res_ALU
    port map (
      res_ALU_in0 => res_ALU_0_in0,
      res_ALU_in1 => res_ALU_0_in1,
      res_ALU_execOp => res_ALU_0_execOp,
      res_ALU_out => res_ALU_0_out
    );
    -- end of port map of component res_ALU_instance_0

    res_ALU_instance_1 : res_ALU
    port map (
      res_ALU_in0 => res_ALU_1_in0,
      res_ALU_in1 => res_ALU_1_in1,
      res_ALU_execOp => res_ALU_1_execOp,
      res_ALU_out => res_ALU_1_out
    );
    -- end of port map of component res_ALU_instance_1

    res_CMP_instance_2 : res_CMP
    port map (
      res_CMP_in0 => res_CMP_2_in0,
      res_CMP_in1 => res_CMP_2_in1,
      res_CMP_out => res_CMP_2_out
    );
    -- end of port map of component res_CMP_instance_2
```

```
res_ARR_instance_3 : res_ARR
port map (
  res_ARR_in0 => res_ARR_3_in0,
  res_ARR_out => res_ARR_3_out
);
-- end of port map of component res_ARR_instance_3

SYNC: PROCESS
BEGIN
  wait until CLOCK'event and CLOCK = '1';
  if reset = '1' then
    CURRENT_STATE <= S_0;
    done <= '0';
  else
    CURRENT_STATE <= NEXT_STATE;
    if CURRENT_STATE /= S_0 and NEXT_STATE = S_0 then
      done <= '1' ;
    end if;
  end if;  --  if reset check
END PROCESS;    --  SYNC Process

FSM: PROCESS(CURRENT_STATE, sT4_10, sT0_6)
BEGIN
  NEXT_STATE  <= CURRENT_STATE;
  if CURRENT_STATE(0) = '1' then
    NEXT_STATE <= S_1;
  elsif CURRENT_STATE(1) = '1' then
    NEXT_STATE <= S_2;
  elsif CURRENT_STATE(2) = '1' then
    if sT0_6 then
      NEXT_STATE <= S_3;
    else -- sT0_6
      NEXT_STATE <= S_0;
    end if; -- conditions
  elsif CURRENT_STATE(3) = '1' then
    if sT0_6 then
      if sT4_10 then
        NEXT_STATE <= S_4;
      else -- ST4_10
        NEXT_STATE <= S_4;
      end if; -- conditions
    end if; -- conditions
  elsif CURRENT_STATE(4) - '1' then
    if sT0_6 then
      if ST4_10 then
        NEXT_STATE <= S_5;
      else -- ST4_10
        NEXT_STATE <= S_5;
      end if; -- conditions
    end if; -- conditions
  elsif CURRENT_STATE(5) = '1' then
    NEXT_STATE <= S_1;
  END if; -- if (CURRENT_STATE)
END PROCESS;  -- FSM Process

DP: PROCESS

BEGIN
  wait until CLOCK'event and CLOCK = '1';
  if reset = '1' then
    sT0_6 <= FALSE;
    sT4_10 <= FALSE;
  else  -- else of  if reset
    if CURRENT_STATE(0) = '1' then
      regNum0 <= res_ALU_1_out;
      regNum1 <= 0;
    elsif CURRENT_STATE(1) = '1' then
```

```
   regNum2 <= res_ALU_0_out;
   regNum3 <= res_ALU_1_out;
   sT0_6 <=res_CMP_2_out;
  elsif CURRENT_STATE(2) = ’1’ then
   if sT0_6 then
    regNum4 <= res_ALU_0_out;
    regNum5 <= res_ALU_1_out;
    sT4_10 <= res_CMP_2_out;
    regNum6 <= res_ARR_3_out;
   end if; -- conditions
  elsif CURRENT_STATE(3) = ’1’ then
   if sT0_6 then
    if sT4_10 then
     regNum7 <= res_ALU_1_out;
     regNum8 <= res_ALU_0_out;
    else -- sT4_10
     regNum7 <= res_ALU_1_out;
     regNum8 <= res_ALU_0_out;
    end if; -- conditions
   end if; -- conditions
  elsif CURRENT_STATE(4) = ’1’ then
   if sT0_6 then
    if sT4_10 then
     regNum9 <= res_ALU_1_out;
     regNum1 <= res_ALU_0_out;
     regNum10 <= res_ARR_3_out;
    else -- sT4_10
     regNum9 <= res_ALU_1_out;
     regNum1 <= res_ALU_0_out;
     regNum10 <= res_ARR_3_out;
    end if; -- conditions
   end if; -- conditions
  elsif CURRENT_STATE(5) = ’1’then
  END if; -- if (CURRENT_STATE)
 end if; -- end of if reset
END PR0CESS; -- DP Process

res_ALU_0_in0_MUXES:PROCESS(CURRENT_STATE, regNuml, sT0_6,
   regNum0, sT4_10, regNum2, regNum10)
 variable mux_select : std_logic_vector(13 downto 0);
BEGIN
 mux_select := "00000000000000";
 if (CURRENT_STATE(l) = ’1’)then
  mux_select := "00000000000001";
 end if;
 if (CURRENT_STATE(2) = ’1’ and sT0_6) then
  mux_select := "00000000000010";
 end if;
 if (CURRENT_STATE(3) = ’1’ and sTO_6 and sT4_10) then
  mux_select := "00000000000100";
 end if;
 if (CURRENT_STATE(4) = ’1’ and sT0_6 and sT4_10) then
  mux_select := "00000000001000";
 end if;
 if (CURRENT_STATE(3) =’1’ and sT0_6 and NOT(sT4_10)) then
  mux_select := "00000000010000";
 end if;
 if (CURRENT_STATE(4) =’1’ and sT0_6 and NOT(sT4_10)) then
  mux_select := "00000000100000";
 end if;
 if (CURRENT_STATE(5) =’1’ and sT0_6) then
  mux_select := "00000001000000";
 end if;
 case mux_select is
  when "00000000000001" =>
   res_ALU_0_in0 <= regNum1;
  when "00000000000010" =>
   res_ALU_0_in0 <= regNum0;
```

```
    when  "00000000000100"  =>
      res_ALU_0_in0 <= regNum2;
    when  "00000000001000"  =>
      res_ALU_0_in0  <=  regNum1;
    when  "00000000010000"  =>
      res_ALU_0_in0  <=  regNum2;
    when  "00000000100000"  =>
      res_ALU_0_in0  <=  regNum1;
    when  "00000001000000"  =>
      res_ALU_0_in0  <=  regNum10;
    when others =>
      res_ALU_0_in0 <= 0;
  END CASE;
END PROCESS;          -- res_ALU_0_in0_MUXES END PROCESS;

res_ALU_0_in1_MUXES: PROCESS(CURRENT_STATE, bytes, sT0_6, regNum1,
     sT0_6, ST4_10,  regNum4,  regNum5,  regNum9)
  variable mux_select  :  std_logic_vector(13 downto 0);
BEGIN
  mux_select  :=  "00000000000000";
  if  (CURRENT_STATE(1)  =  '1')  then
    mux_select  :=  "00000000000001";
  end if;
  if  (CURRENT_STATE(2)  =  '1'  and  sT0_6)  then
    mux_select  :=  "00000000000010";
  end if;
  if (CURRENT_STATE(3)  = '1'  and  sT0_6 and sT4_10) then
    mux_select  :=  "00000000000100";
  end if;
  if  (CURRENT_STATE(4)  =  '1'  and  sT0_6  and  sT4_10)  then
    mux_select  :=  "00000000001000";
  end if;
  if (CURRENT_STATE(3) = '1' and sT0_6 and NOT(sT4_10)) then
    mux_select  :=  "00000000010000";
  end if;
  if (CURRENT_STATE(4) = '1' and sT0_6 and NOT<sT4_10)) then
    mux_select  :=  "00000000100000";
  end if;
  if (CURRENT_STATE(5) = '1' and sT0_6) then
    mux_select  :=  "00000001000000";
  end if;
  case mux_select is
    when  "00000000000001"  =>
      res_ALU_0_in1 <= bytes;
    when  "00000000000010"  =>
      res_ALU_0_in1 <= regNum1;
    when  "00000000000100"  =>
      res_ALU_0_in1  <= regNum4;
    when  "00000000001000"  =>
      res_ALU_0_in1 <= 1;
    when  "00000000010000"  =>
      res_ALU_0_in1 <= regNum5;
    when  "00000000100000"  =>
      res_ALU_0_in1  <= 1;
    when "00000001000000" =>
      res_ALU_0_in1  <=  regNum9;
    when others =>
      res_ALU_0_in1  <=  0;
  END CASE;
END PROCESS;         -- res_ALU_0_in1_MUXES END PROCESS;

res_ALU_0_execOpMUXES: PROCESS(CURRENT_STATE, sT0_6, sT4_10)
  variable mux_select  :  std_logic_vector(13 downto 0);
BEGIN
  mux_select := "00000000000000";
  if  (CURRENT_STATE(1)  =  '1')  then
    mux_select  :=  "00000000000001";
  end if;
  if (CURRENT_STATE(2) = '1' and sT0_6) then
    mux_select := "00000000000010";
  end if;
  if (CURRENT_STATE(3) = '1' and sT0_6 and sT4_10) then
    mux_select := "00000000000100";
  end if;
  if (CURRENT_STATE(4) = '1' and sT0_6 and sT4_10) then
    mux_select := "00000000001000";
  end if;
  if (CURRENT_STATE(3) = '1' and sT0_6 and NOT(sT4_10)) then
    mux_select := "00000000010000";
  end if;
  if (CURRENT_STATE(4) = '1' and sT0_6 and NOT(sT4_10)) then
    mux_select := "00000000100000";
  end if;
  if (CURRENT_STATE(5) = '1' and sT0_6) then
    mux_select := "00000001000000";
  end if;
  case mux_select is
    when "00000000000001" =>
      res_ALU_0_execOp <= 0 ;
    when "00000000000010" =>
      res_ALU_0_execOp <= 1 ;
    when "00000000000100" =>
      res_ALU_0_execOp <= 0 ;
    when "00000000001000" =>
      res_ALU_0_execOp <= 0 ;
    when "00000000010000" =>
      res_ALU_0_execOp <= 0 ;
    when "00000000100000" =>
      res_ALU_0_execOp <= 0 ;
    when "00000001000000" =>
      res_ALU_0_execOp <= 0 ;
    when others =>
      res_ALU_0_execOp <= 0;
  END CASE;
END PROCESS;          -- res_ALU_0_execOp_MUXES END PROCESS;

res_ALU_1_in0_MUXES: PROCESS(CURRENT_STATE, x, regNum1, sT0_6,
     regNum3, sT4_10, regNum6, regNum7)
  variable mux_select : std_logic_vector(13 downto 0);
BEGIN
  mux_select := "00000000000000";
  if (CURRENT_STATE(0) = '1') then
    mux_select := "00000000000001";
  end if;
  if (CURRENT_STATE(1) = '1') then
    mux_select := "00000000000010";
  end if;
  if (CURRENT_STATE(2) = '1' and sT0_6) then
    mux_select := "00000000000100";
  end if;
  if (CURRENT_STATE(3) = '1' and sT0_6 and sT4_10) then
    mux_select := "00000000001000";
  end if;
  if (CURRENT_STATE(4) = '1' and sT0_6 and sT4_10) then
    mux_select := "00000000010000";
  end if;
  if (CURRENT_STATE(3) = '1' and sT0_6 and NOT(sT4_10)) then
    mux_select := "00000000100000";
  end if;
  if (CURRENT_STATE(4) = '1' and sT0_6 and NOT(sT4_10)) then
    mux_select := "00000001000000";
  end if;
  case mux_select is
    when "00000000000001" =>
      res_ALU_1_in0 <= x;
    when "00000000000010" =>
```

```
      res_ALU_1_in0 <= regNum1;
    when "00000000000100" =>
      res_ALU_1_in0 <= regNum3;
    when "00000000001000" =>
      res_ALU_1_in0 <= regNum6;
    when "00000000010000" =>
      res_ALU_1_in0 <= regNum7;
    when "00000000100000" =>
      res_ALU_1_in0 <= regNum6;
    when "00000001000000" =>
      res_ALU_1_in0 <= regNum7;
    when others =>
      res_ALU_1_in0 <= 0;
  END CASE;
END PROCESS;         -- res_ALU_1_in0_MUXES END PROCESS;

res_ALU_1_in1_MUXES: PROCESS(CURRENT_STATE, x, sT0_6, sT4_10, y)
  variable mux_select : std_logic_vector(13 downto 0);
BEGIN
  mux_select := "00000000000000";
  if (CURRENT_STATE(0) = ’1’) then
    mux_select := "00000000000001";
  end if;
  if (CURRENT_STATE(1) = ’1’) then
    mux_select := "00000000000010";
  end if;
  if (CURRENT_STATE(2) = ’1’ and sT0_6) then
    mux_select := "00000000000100";
  end if;
  if (CURRENT_STATE(3) = ’1’ and sT0_6 and sT4_10) then
    mux_select := "00000000001000";
  end if;
  if (CURRENT_STATE(4) = ’1’ and sT0_6 and sT4_10) then
    mux_select := "00000000010000";
  end if;
  if (CURRENT_STATE(3) = ’1’ and sT0_6 and NOT(sT4_10)) then
    mux_select := "00000000100000";
  end if;
  if (CURRENT_STATE(4) = ’1’ and sT0_6 and NOT(sT4_10)) then
    mux_select := "00000001000000";
  end if;
  case mux_select is
    when "00000000000001" =>
      res_ALU_1_in1 <= 2;
    when "00000000000010" =>
      res_ALU_1_in1 <= x;
    when "00000000000100" =>
      res_ALU_1_in1 <= 2;
    when "00000000001000" =>
      res_ALU_1_in1 <= y;
    when "00000000010000" =>
      res_ALU_1_in1 <= 2;
    when "00000000100000" =>
      res_ALU_1_in1 <= y;
    when "00000001000000" =>
      res_ALU_1_in1 <= 2;
    when others =>
      res_ALU_1_in1 <= 0;
  END CASE;
END PROCESS;         -- res_ALU_1_in1_MUXES END PROCESS;

res_ALU_1_execOpMUXES: PROCESS(CURRENT_STATE, sT0_6, sT4_10)
  variable mux_select : std_logic_vector(13 downto 0);
BEGIN
  mux_select := "00000000000000";
  if (CURRENT_STATE(0) = ’1’) then
    mux_select := "00000000000001";
  end if;
  if (CURRENT_STATE(1) = '1') then
    mux_select := "00000000000010";
  end if;
  if (CURRENT_STATE(2) = '1' and sT0_6) then
    mux_select := "00000000000100";
  end if;
  if (CURRENT_STATE (3) = '1' and sT0_6 and sT4_10) then
    mux_select := "00000000001000";
  end if;
  if (CURRENT_STATE(4) = '1' and sT0_6 and sT4_10) then
    mux_select := "00000000010000";
  end if;
  if (CURRENT_STATE (3) = '1' and sT0_6 and NOT(sT4_10)) then
    mux_select := "00000000100000";
  end if;
  if (CURRENT_STATE(4) = '1' and sT0_6 and NOT(sT4_10)) then
    mux_select := "00000001000000";
  end if;
  case mux_select is
    when "00000000000001" =>
      res_ALU_1_execOp <= 0 ;
    when "00000000000010" =>
      res_ALU_1_execOp <= 1 ;
    when "00000000000100" =>
      res_ALU_1_execOp <= 0 ;
    when "00000000001000" =>
      res_ALU_1_execOp <= 1 ;
    when "00000000010000" =>
      res_ALU_1_execOp <= 0 ;
    when "00000000100000" =>
      res_ALU_1_execOp <= 1 ;
    when "00000001000000" =>
      res_ALU_1_execOp <= 0 ;
    when others =>
      res_ALU_1_execOp <= 0;
  END CASE;
END PROCESS;          -- res_ALU_1_execOp_MUXES END PROCESS;

res_CMP_2_in0_MUXES: PROCESS (CURRENT_STATE, regNum1, sT0_6)
  variable mux_select : std_logic_vector(1 downto 0) ;
BEGIN
  mux_select := "00";
  if (CURRENT_STATE(1) = '1') then
    mux_select := "01";
  end if;
  if (CURRENT_STATE(2) = '1' and sT0_6) then
    mux_select := "10";
  end if;
  case mux_select is
    when "01" =>
      res_CMP_2_in0 <= regNum1;
    when "10" =>
      res_CMP_2_in0 <= regNum1;
    when others =>
      res_CMP_2_in0 <= 0;
  END CASE;
END PROCESS;          -- res_CMP_2_in0_MUXES END PROCESS;

res_CMP_2_in1_MUXES: PROCESS (CURRENT_STATE, sT0_6, regNum0)
  variable mux_select : std_logic_vector(1 downto 0);
BEGIN
  mux_select := "00";
  if (CURRENT_STATE(1)  = '1') then
    mux_select := "01";
  end if;
  if (CURRENT_STATE(2) = '1' and sT0_6) then
    mux_select := "10";
  end if;
```

```
      case mux_select is
        when "01" =>
          res_CMP_2_in1 <= 10;
        when "10" =>
          res_CMP_2_in1 <= regNum0;
        when others =>
          res_CMP_2_in1 <= 0;
      END CASE;
    END PROCESS;          -- res_CMP_2_in1_MUXES END PROCESS;

    res_ARR_3_in0_MUXES: PROCESS (CURRENT_STATE , sT0_6, dest_cur,
         regNum2, sT4_10, regNum8)
      variable mux_select : std_logic_vector(2 downto 0);
    BEGIN
      mux_select := "000";
      if (CURRENT_STATE(2) = '1' and sT0_6) than
        mux_select := "001";
      end if;
      if (CURRENT_STATE(4) = '1' and sT0_6 and sT4_10) then
        mux_select := "010";
      end if;
      if (CURRENT_STATE(4) = '1' and sT0_6 and NOT(sT4_10)) then
        mux_select := "100";
      end if;
      case mux_select is
        when "001" =>
          res_ARR_3_in0 <= dest_cur(regNum2);
        when "010" =>
          res_ARR_3_in0 <= dest_cur(regNum8);
        when "100" =>
          res_ARR_3_in0 <= dest_cur(regNum8);
        when others =>
          res_ARR_3_in0 <= 0;
      END CASE;
    END PROCESS;         -- res_ARR_3_in0_MUXES END PROCESS;

END rtl;
```

Bibliography

[AKPW83] R. Allen, K. Kennedy, C. Portfield, and J. Warren. Conversion of control dependence to data dependence. In *ACM Symposium on Principles of Programming Languages,* 1983.

[AN88a] A. Aiken, , and A. Nicolau. A development environment for horizontal microcode. *IEEE Transactions on Software Engineering,* May 1988.

[AN88b] A. Aiken and A. Nicolau. Perfect Pipelining: A new loop parallelization technique. In *European Symposium on Programming,* 1988.

[ANN95] A. Aiken, A. Nicolau, and S. Novack. Resource-constrained software pipelining. *IEEE Transactions on Parallel and Distributed Systems,* 6(12), December 1995.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles and Techniques and Tools.* Addison-Wesley, 1986.

[BC] Behavioral compiler. Synopsys.

[BCM$^+$88] R.K. Brayton, R. Camposano, G. De Micheli, R.H.J.M. Otten, and J. van Eijndhoven. *The Yorktown Silicon Compiler System,* chapter in Silicon Compilation. Addison-Wesley, 1988.

[Ber99] R.A. Bergamaschi. Behavioral network graph unifying the domains of high-level and logic synthesis. In *Design Automation Conference,* 1999.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys,* 26(4):345–420, 1994.

[BR98] M. Benmohammed and A. Rahmoune. Automatic generation of reprogrammable microcoded controllers within a high-level synthesis environment. In *IEE Proceedings-Computers and Digital Techniques,* 1998.

[BRT97] R. A. Bergamaschi, S. Raje, and L. Trevillyan. Control-flow versus data-flow-based scheduling: combining both approaches in an adaptive scheduling system. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* March 1997.

[Cam91] R. Camposano. Path–based scheduling for synthesis. *IEEE Transactions on Computer–Aided Design,* Jan. 1991.

[Cel] Celoxica. DK design suite. `http://www.celoxica.com`.

[CF87] R. Cytron and J. Ferrante. What's in a name? or the value of renaming for parallelism detection and storage allocation. In *International Conference on Parallel Processing,* 1987.

[CGR92] V. Chaiyakul, D.D. Gajski, and L. Ramachandran. Minimizing syntactic variance with assignment decision diagrams. Technical Report ICS-TR-92-34, UC Irvine, 1992.

[CGR93a] V. Chaiyakul, D. D. Gajski, and L. Ramachandran. High-level transformations for minimizing syntactic variances. In *Design Automation Conference,* 1993.

[CGR93b] V. Chaiyakul, D.D. Gajski, and L. Ramachandran. High level transformations for minimizing syntactic variances. In *Design Automation Conference,* 1993.

[CKS+98] A. Chowdhary, S. Kale, P. Saripella, N. K. Sehgal, and R. K. Gupta. A general approach for regularity extraction in datapath circuits. In *International Conference on Computer-Aided Design,* 1998.

[CLS93] L.-F. Chao, A. S. LaPaugh, and E. H.-M. Sha. Rotation scheduling: A loop pipelining algorithm. In *Design Automation Conference,* 1993.

[CP95] J.-M. Chang and M. Pedram. Register allocation and binding low power. In *Design Automation Conf.*, 1995.

[CP96] J.-M. Chang and M. Pedram. Module assignment for low power. In *European Design Automation Conference,* 1996.

[CW91] R. Camposano and W. Wolf. *High Level VLSI Synthesis.* Kluwer Academic, 1991.

[CWG+98] F. Catthoor, S. Wuytack, E. De Greef, F. Balasa, L. Nachtergaele, and A. Vandecappelle. *Custom Memory Management Methodology: Exploration of Memory Organisation for Embedded Multimedia System Design.* Kluwer Academic Publishers, 1998.

[DC] Design compiler. Synopsys Incorporated.

[DHB89] J. C. Dehnert, P. Y.-T Hsu, and J.P. Bratt. Overlapped loop support in the cydra 5. In *International Conference on Architectural Support for Programming Languages and Operating Systems,* 1989.

[DM94] G. De Micheli. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1994.

[DS] Forte Design Systems. Behavioral design suite. `http://www.forteds.com.`

[dS97] L.C.V. dos Santos. A method to control compensation code during global scheduling. In *Workshop on Circuits, Systems and Signal Processing,* 1997.

[dS98] L.C.V. dos Santos. *Exploiting instruction-level parallelism: a constructive approach.* PhD thesis, Eindhoven University of Technology, 1998.

[dSJ99] L.C.V. dos Santos and J.A.G. Jess. A reordering technique for efficient code motion. In *Design Automation Conference,* 1999.

[DT93] J. C. Dehnert and R. A. Towle. Compiling for the cydra 5. *IEEE Computer,* 7(1/2), 1993.

[Ebc87] K. Ebcioglu. A compilation technique for software pipelining of loops with conditional jumps. In *MICRO,* 1987.

[EDG] Edison Design Group (edg) compiler frontends. `http://www.edg.com.`

[ELL00] E. Eim, J.-G. Lee, and D.-I. Lee. Automatic process-oriented control circuit generation for asynchronous high-level synthesis. In *International Symposium on Advanced Research in Asynchronous Circuits and Systems,* 2000.

[EN89] K. Ebcioglu and A. Nicolau. A global resource-constrained parallelization technique. In *3rd International Conference on Supercomputing,* 1989.

[Fis81] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers,* July 1981.

[GDGN03a] S. Gupta, N.D. Dutt, R.K. Gupta, and A. Nicolau. Dynamic conditional branch balancing during the high-level synthesis of control-intensive designs. In *Design, Automation and Test Conference,* 2003.

[GDGN03b] S. Gupta, N.D. Dutt, R.K. Gupta, and A. Nicolau. Dynamically increasing the scope of code motions during the high-level synthesis of digital circuits. *Invited Paper in Special Issue of IEE Proceedings: Computers and Digital Technique: Best of DATE 2003,* 150(5), September 2003.

[GDGN03c] S. Gupta, N.D. Dutt, R.K. Gupta, and A. Nicolau. Loop shifting and compaction for the high-level synthesis of designs with complex control flow. Technical Report CECS-TR-03-14, UC Irvine, April 2003.

[GDGN03d] S. Gupta, N.D. Dutt, R.K. Gupta, and A. Nicolau. SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. In *International Conference on VLSI Design,* 2003.

[GDGN04] S. Gupta, N.D. Dutt, R.K. Gupta, and A. Nicolau. Loop shifting and compaction for the high-level synthesis of designs with complex control flow. In *Design, Automation and Test Conference,* 2004.

[GDWL92] D. D. Gajski, N. D. Dutt, A. C-H. Wu, and S. Y-L. Lin. *High-Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic, 1992.

[GE92] C.H. Gebotys and M.I. Elmasry. Optimal synthesis of high-performance architectures. *IEEE Journal of Solid-State Circuits,* March 1992.

[GG00] S. Gupta and R. K. Gupta. *The VLSI Handbook,* chapter ASIC Design. CRC Press and IEEE Press, 2000. Chapter 64.

[Gim] GNU Image Manipulation Program. `http://www.gimp.org`.

[Gir84] E. Girczyc. *Automatic Generation of Micro-sequenced Data Paths to Realize ADA Circuit Descriptions.* PhD thesis, Carleton University, 1984.

[Gir87] E. Girczyc. Loop winding - a data flow approach to functional pipelining. In *International Symposium of Circuits and Systems,* 1987.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[GKK+02] S. Gupta, T. Kam, M. Kishinevsky, S. Rotem, N. Savoiu, N.D. Dutt, R.K. Gupta, and A. Nicolau. Coordinated transformations for high-level synthesis of high performance microprocessor blocks. In *Design Automation Conference,* 2002.

[GKWB00] R. L. Gupta, A. Kumar, A. Van Der Werf, and G. N. Busa. Synthesizing a long latency unit within VLIW processor. In *Intl. Conf. on VLSI Design,* 2000.

[GL97] R. K. Gupta and S. Y. Liao. Using a programming language for digital system design. *IEEE Design and Test of Computers,* April 1997.

[GLD+03] S. Gupta, M. Luthra, N.D. Dutt, R.K. Gupta, and A. Nicolau. Hardware and interface synthesis of fpga blocks using parallelizing code transformations. In *To appear at the International Conference on Parallel and Distributed Computing and Systems,* November 2003.

[GMCG00] S. Gupta, M. Miranda, F. Catthoor, and R. Gupta. Analysis of high-level address code transformations for programmable processors. In *Design, Automation and Test in Europe,* 2000.

[GP92] M. Girkar and C.D. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. on Parallel & Distributed Systems,* Mar. 1992.

[GPN90] D.H. Gelernter, D. A. Padua, and A. Nicolau. *Languages and Compilers for Parallel Computing.* Morgan Kaufmann, 1990.

[GRS+02] S. Gupta, M. Reshadi, N. Savoiu, N.D. Dutt, R.K. Gupta, and A. Nicolau. Dynamic common sub-expression elimination during scheduling in high-level synthesis. In *International Symposium on System Synthesis,* 2002.

[GS90] R. Gupta and M. L. Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering,* April 1990.

[GSD+01] S. Gupta, N. Savoiu, N.D. Dutt, R.K. Gupta, and A. Nicolau. Conditional speculation and its effects on performance and area for high-level synthesis. In *International Symposium on System Synthesis,* 2001.

[GVM89] G. Goossens, J. Vandewlle, and H. De Man. Loop optimization in register-transfer scheduling for DSP-systems. In *Design automation conference*, 1989.

[GZD+00] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology.* Kluwer Academic Publishers, January 2000.

[Hay00] S. Haynal. *Automata-Based Symbolic Scheduling.* PhD thesis, University of California, Santa Barbara, 2000.

[HC89] R. Hartley and A. E. Casavant. Tree-height minimization in pipelined architectures. In *International Conference on Computer-Aided Design,* 1989.

[HD86] P. Y. T. Hsu and E. S. Davidson. Highly concurrent scalar processing. In *International Symposium on Computer Architecture,* 1986.

[HE95] U. Holtmann and R. Ernst. Combining MBP-speculative computation and loop pipelining in high-level synthesis. In *European Design and Test Conference,* 1995.

[Hea93a] S. Huang and et al. A tree-based scheduling algorithm for control dominated circuits. In *Design Automation Conference,* 1993.

[Hea93b] W.W. Hwu and et al. The superblock: An effective technique for vliw and superscalar compilation. *Journal of Supercomputing,* March 1993.

[HLH91] C.T. Hwang, T.H. Lee, and Y. C. Hsu. A formal approach to the scheduling problem in high level synthesis. *IEEE Transactions on CAD,* April 1991.

[HP81] F.J. Hill and G.R. Peterson. *Switching Theory and Logical Design.* Wiley, New York, 1981.

[HT83] C.Y. Hitchcock and D.E. Thomas. A method of automatic data path synthesis. In *Design Automation Conference,* 1983.

[HW94] S.C.-Y. Huang and W.H. Wolf. How datapath allocation affects controller delay. In *International Symposium on High-Level Synthesis*, 1994.

[IM91] M. Ishikawa and G. D. Micheli. A module selection algorithm for high-level synthesis. In *International Symposium on Circuits and Systems,* 1991.

[InaCs] Get2Chip Incorporated (now a Cadence subsidiary). G2C architectural compiler. `http://www.get2chip.com`.

[IPDP93] Z. Iqbal, M. Potkonjak, S. Dey, and A. Parker. Critical path optimization using retiming and algebraic speed-up. In *Design Automation Conference,* 1993.

[JA90] R. Jones and V. Allan. Software pipelining: A comparison and improvement. In *Proceedings of the Micro-23,* 1990.

[JCM94] M. Janssen, F. Catthoor, and H. De Man. A specification invariant technique for operation cost minimisation in flow-graphs. In *International Symposium on High-level Synthesis*, 1994.

[JDKR97] A. A. Jerraya, H. Ding, P. Kission, and M. Rahmouni. *Behavioral Synthesis and Component Reuse with VHDL.* Kluwer Academic Publishers, 1997.

[KA01] K. Kennedy and R. Allen. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann, 2001.

[KAH97] P. Kollig and B. M. Al-Hashimi. Simultaneous scheduling, allocation and binding in high level synthesis. *Electronics Letters,* August 1997.

[KCL$^+$99] R. Kennedy, S. Chan, S.-M. Liu, R. Io, P. Tu, and F. Chow. Partial redundancy elimination in SSA form. *ACM Trans. Progrm. Languages and Systems*, May 1999.

[KDJ94] P. Kission, H. Ding, and A.A. Jerraya. Structured design methodology for high-level design. In *Design Automation Conference*, 1994.

[KGM95] A. Kifli, G. Goossens, and H. De Man. A unified scheduling model for high-level synthesis and code generation. In *Proceedings of the European Design and Test Conference*, 1995.

[KKP$^+$81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *ACM Symposium on Principles of Programming Languages*, 1981.

[KKP91] J.J. Kim, F.J. Kurdahi, and N. Park. Automatic synthesis of time-stationary controllers for pipelined data paths. In *Proceedings of the International Conference on Computer-Aided Design*, 1991.

[KM88] D. Ku and G. De Micheli. HardwareC - A language for hardware design. Technical Report CSL-TR-90-419, Stanford University, 1988.

[KM90] D. Ku and G. De Micheli. Relative scheduling under timing constraints. In *Design Automation Conference*, 1990.

[KM92] D. C. Ku and G. De Micheli. *High Level Synthesis of ASICs Under Timing and Synchronization Constraints.* Kluwer Academic, 1992.

[Kna96] D. W. Knapp. *Behavioral Synthesis: Digital System Design using the Synopsys Behavioral Compiler.* Prentice-Hall, 1996.

[KT85] T. J. Kowalski and D. E. Thomas. The VLSI design automation assistant: What's in a knowledge base. In *Design Automation Conference,* 1985.

[Kuc78] D.J. Kuck. *The Structure of Computers and Computations,* John Wiley and Sons, 1978.

[KW99] A. Kountouris and C. Wolinski. High level pre-synthesis optimization steps using hierarchical conditional dependency graphs. In *Euromicro Confernce*, 1999.

[KW00] A.A. Kountouris and C. Wolinski. Hierarchical conditional dependency graphs as a unifying design representation in the CODESIS high-level synthesis system. In *Intl. Symposium on System Synthesis,* 2000.

[KYLL94] T. Kim, N. Yonezawa, J.W.S. Liu, and C.L. Liu. A scheduling algorithm for conditional resource sharing - a hierarchical reduction approach. *IEEE Transactions on CAD,* April 1994.

[Lab] AT&T Research Labs. Graphviz - Open source graph drawing software. `http://www.research.att.com/sw/tools/graphviz/`.

[Lam88] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *ACM SIGPLAN Conference Programming Languages Design Implementation*, 1988.

[Lee88] E.A. Lee. Programmable DSP architectures, Parts I, II. *IEEE ASSP Magazine*, October 1988.

[LG96] J. Li and R.K. Gupta. HDL optimizations using Timed Decision Tables. In *Design Automation Conference*, 1996.

[LG97] J. Li and R.K. Gupta. Decomposition of Timed Decision Tables and its use in presynthesis optimizations. In *International Conference on Computer Aided Design*, 1997.

[LGD$^+$03] M. Luthra, S. Gupta, N.D. Dutt, R.K. Gupta, and A. Nicolau. Interface synthesis using memory mapping for an FPGA platform. In *International Conference on Computer Design*, October 2003.

[LMD94] B. Landwehr, P. Marwedel, and R. Doemer. Oscar: Optimum simultaneous scheduling, allocation and resource binding based on integer programming. In *European Design Automation Conference*, 1994.

[LP91] D.A. Lobo and B.M. Pangrle. Redundant operator creation: A scheduling optimization technique. In *Design Automation Conference*, 1991.

[LPMS97] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, 1997.

[LRJ98] G. Lakshminarayana, A. Raghunathan, and N.K. Jha. Incorporating speculative execution into scheduling of control-flow intensive behavioral descriptions. In *Design Automation Conference*, 1998.

[LT81] G.W. Leive and D.E. Thomas. A technology relative logic synthesis and module selection system. In *Design Automation Conference*, 1981.

[LW92] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *International Symposium on Computer Architecture*, 1992.

[Man00] S. Mantripragada. *Branch Optimizations and Instruction Level Parallelism Exploitation for Dynamic Superscalar and VLIW Processors.* PhD thesis, University of California, Irvine, 2000.

[Mar86] P. Marwedel. A new synthesis for the mimola software system. In *Design Automation Conference*, 1986.

[MBVS97] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *International Symposium on Computer Architecture*, 1997.

[McF78] M. C. McFarland. The Value Trace: A database for automated digital design. Technical Report DRC-01-4-80, Carnegie-Mellon University, Design Research Center, 1978.

[MCJM98] M. Miranda, F. Catthoor, M. Janssen, and H. De Man. High-level address optimisation and synthesis techniques for data-transfer intensive applications. *IEEE Transactions on VLSI Systems, December* 1998.

[ME92] S.-M. Moon and K. Ebcioglu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. In *International Symposium on Microarchitecture*, 1992.

[Med] UCLA Mediabench benchmark suite. `http://cares.icsl.ucla.edu/MediaBench/`.

[MJS96] A. Mujumdar, R. Jain, and K. Saluja. Incorporating performance and testability constraints during binding in high-level synthesis. *IEEE Trans. on CAD,* 1996.

[MKMT90] G. De Micheli, D. C. Ku, F. Mailhot, and T. Truong. The Olympus synthesis system for digital design. *IEEE Design and Test of Computers,* pages 37–53, October 1990.

[MLC$^+$92] 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. In *International Symposium on Microarchitecture,* 1992.

[MPC90] M.C. McFarland, A.C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE,* February 1990.

[MRSC86] H. De Man, J. Rabaey, P. Six, and L. Claesen. Cathedral-II: A silicon compiler for digital signal processing. *IEEE Design & Test Magazine,* December 1986.

[Muc97] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[Nic85a] A. Nicolau. A development environment for scientific parallel programs. Technical Report TR 86-722, Department of Computer Science, Cornell University, 1985.

[Nic85b] A. Nicolau. Uniform parallelism exploitation in ordinary programs. In *International Conf. on Parallel Processing,* 1985.

[NN93] A. Nicolau and S. Novack. Trailblazing: A hierarchical approach to Percolation Scheduling. In *International Conference on Parallel Processing,* 1993.

[NN94] S. Novack and A. Nicolau. Mutation scheduling: A unified approach to compiling for fine-grain parallelism. In *Languages and Compilers for Parallel Computing*, 1994.

[NN96] S. Novack and A. Nicolau. An efficient, global resource-directed approach to exploiting instruction-level parallelism. In *Conference on Parallel Architectures and Compilation Techniques*, 1996.

[NP91] A. Nicolau and R. Potasman. Incremental tree height reduction for high level synthesis. In *Design Automation Conference*, 1991.

[NR00] M. Narasimhan and J. Ramanujam. On lower bounds for scheduling problems in high-level synthesis. In *Design Automation Conference*, 2000.

[OG86] A. Orailoglu and D.D. Gajski. Flow graph representation. In *Design Automation Conference*, 1986.

[Pan98] P.R. Panda. *Memory Optimizations and Exploration for Embedded Systems.* PhD thesis, University of California, Irvine, 1998.

[Pau89] P.G. Paulin. Horizontal partitioning of PLA-based finite state machines. In *Design Automation Conference*, 1989.

[Pen] Intel Inc., `http://developer.intel.com/design/pro/manuals/242691.htm`. *PentiumPro® Programmer's Reference Manual.* Chapter 11.

[PFH+95] P. Paulin, J. Frehel, M. Harrand, E. Berrebi, C. Liem, F Nacabal, and J.-C. Herluison. High-level synthesis and codesign methods: an application to a videophone codec. In *European design automation conference with EURO-VHDL*, 1995.

[PG86] B.M. Pangrle and D.D. Gajski. Slicer: A state synthesizer for intelligent silicon compilation. In *Proceedings of the International Conference on Computer-Aided Design*, 1986.

[PK89a] P. G. Paulin and J. P. Knight. Force-Directed Scheduling for the Behavioral Synthesis of ASIC's. *IEEE Transactions on CAD*, 8(6):661–678, June 1989.

[PK89b] P. G. Paulin and J. P. Knight. Scheduling and Binding Algorithms for High-Level Synthesis. In *Design Automation Conference,* 1989.

[PK91] I.-C. Park and C.-M. Kyung. Fast and near optimal scheduling in automatic data path synthesis. In *Design Automation Conference,* 1991.

[PLNG90] R. Potasman, J. Lis, A. Nicolau, and D. Gajski. Percolation based synthesis. In *Design Automation Conference,* 1990.

[PMH02] O. Penalba, J.M. Mendias, and R. Hermida. Maximizing conditional reuse by pre-synthesis transformations. In *Design, Automation and Test in Europe,* 2002.

[Pol88] C. D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, 1988.

[PP88] N. Park and A. Parker. Sehwa: A software package for synthesis of pipelines from behavioral specifications. *IEEE Transactions on Computer-Aided Design,* March 1988.

[PPM86] A.C. Parker, J. Pizarro, and M. Mlinar. MAHA: A program for datapath synthesis. In *Design Automation Conference*, 1986.

[PR92] M. Potkonjak and J. Rabaey. Maximally fast and arbitrarily fast implementation of linear computations. In *International Conference on CAD,* 1992.

[PR94] M. Potkonjak and J. Rabaey. Optimizing resource utlization using transformations. *IEEE Transactions on CAD,* March 1994.

[PS91] J. C. H. Park and M. Schlansker. On predicated execution. Technical Report HPL-91-58, Hewlett-Packard Software and Systems Laboratory, 1991.

[PSC96] M. Potkonjak, M.B. Srivastava, and A. Chandrakasan. Multiple constant multiplications: Efficient and versatile framework and algorithms for exploring common subexpression elimination. *IEEE Trans. on CAD,* Mar 1996.

[PSD+99] R. Pasko, P. Schaumont, V. Derudder, S. Vernalde, and D. Durackova. A new algorithm for elimination of common subexpressions. *IEEE Transactions on CAD,* Jan 1999.

[Raj95] S. Rajan. Practical state machine design using VHDL. *Integrated System Design Magazine,* Febuary 1995. http://www.isdmag.com/editorial/1995/fpgafeature9502.html.

[RB95] I. Radivojevic and F. Brewer. Analysis of conditional resource sharing using a guard-based control representation. In *International Conference on Computer Design*, 1995.

[RB96] I. Radivojevic and F. Brewer. A new symbolic technique for control-dependent scheduling. *IEEE Transactions on CAD*, January 1996.

[RCHP91] J. M. Rabaey, C. Chu, P. Hoang, and M. Potkonjak. Fast prototyping of datapath-intensive architectures. *IEEE Design & Test of Computers,* June 1991.

[RFJ95] M. Rim, Y. Fann, and R. Jain. Global scheduling with code-motions for high-level synthesis applications. *IEEE Transactions on VLSI Systems,* September 1995.

[RG81] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Annual Workshop on Microprogramming*, 1981.

[RK94] D.S. Rao and F.J. Kurdahi. Controller and datapath trade-offs in hierarchical RT-level synthesis. In *International Symposium on High-Level Synthesis*, 1994.

[RV96] N.N.J. Roy and R. Vemuri. Synchronous controller models for synthesis from communicating VHDL processes. In *International Conference on VLSI Design*, 1996.

[RYYT89] B. Rau, D. Yen, W. Yen, and R. Towle. The cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs. *IEEE Computer,* 22(1), 1989.

[Sem01] L. Semeria. *Applying Pointer Analysis to the Synthesis of Hardware from C.* PhD thesis, Stanford University, 2001.

[SGL96] V.C. Sreedhar, G. R. Gao, and Y.-F. Lee. A new framework for exhaustive and incremental data flow analysis using DJ graphs. *ACM SIGPLAN Conf. on PLDI,* 1996.

[SGL97] V.C. Sreedhar, G. R. Gao, and Y.-F. Lee. Incremental computation of dominator trees. *ACM Trans. Progrm. Languages and Systems,* March 1997.

[SLH90] M. D. Smith, M. S. Lam, and M. A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *International Symposium on Computer Architecture,* 1990.

[SP91] L. Stok and W.J.M. Philipsen. Module allocation and comparability graphs. In *IEEE International Sympoisum on Circuits and Systems, 1991.*

[SPA] SPARK parallelizing high-level synthesis framework website, `http://www.cecs.uci.edu/~spark`.

[Sto92] L. Stok. Transfer free register allocation in cyclic data flow graphs. In *European Conf. on Design Automation,* 1992.

[SyC] Synopsys Inc., `http://www.systemc.org`. *SystemC Reference Manual.*

[SZ90] A. Safir and B. Zavidovique. Towards a global solution to high level synthesis problems. In *European Design Automation Conference,* 1990.

[TDW+88] D.E. Thomas, E.M. Dirkes, R.A. Walker, J.V. Rajan, J.A. Nestor, and R.L. Blackburn. The system architect's workbench. In *Design Automation Conference*, 1988.

[TF70] G. S. Tjaden and M. J. Flynn. Detection and parallel execution of independent instructions. *IEEE Transactions on Computers,* 19(10), October 1970.

[Tri87] H. Trickey. Flamel: A high-level hardware compiler. *IEEE Transactions on Computer–Aided Design,* March 1987.

[TS86] C.J. Tseng and D.P. Siewiorek. Automated synthesis of data paths in digital systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* July 1986.

[TTC90] R. L. Rivest T. T. Cormen, C. E. Leiserson. *Introduction to algorithms.* MIT Press, Cambridge, MA, 1990.

[TWR+88] C.-J. Tseng, R.-S. Wei, S. G. Rothweiler, M. M. Tong, and A. K. Bose. Bridge: A versatile behavioral synthesis system. In *Design Automation Conference*, 1988.

[VSB99] S. Vernalde, P. Schaumont, and I. Bolsens. An object oriented programming approach for hardware design. In *IEEE Computer Society Workshop on VLSI*, April 1999.

[Wak99] K. Wakabayashi. C-based synthesis experiences with a behavior synthesizer, "Cyber". In *Design, Automation and Test in Europe*, 1999.

[Wal91] D.W. Wall. Limits of instruction-level parallelism. In *International Conference on Architectural Support for Programming Languages and Operating System* (*ASPLOS*), 1991.

[WMGB95] T.C. Wilson, N. Mukherjee, M.K. Garg, and D. K. Banerji. An ILP solution for optimum scheduling, module and register allocation, and operation binding in datapath synthesis. *VLSI Design*, 1995.

[Wol96] M.J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[WP92] J.-P. Weng and A. C. Parker. CSG: Control path synthesis in the ADAM system. In *International Workshop on High Level Synthesis*, 1992.

[WT89] R. Walker and D. Thomas. Behavioral transformation for algorithmic level IC design. *IEEE Transactions on CAD*, October 1989.

[WT92] K. Wakabayashi and H. Tanaka. Global scheduling independent of control dependencies based on condition vectors. In *Design Automation Conference*, 1992.

[WTH+92] W. Wolf, A. Takach, C.-Y. Huang, R. Manno, and E. Wu. The Princeton University behavioral synthesis system. In *Design automation conference*, 1992.

[WY89] K. Wakabayashi and T. Yoshimura. A resource sharing and control synthesis method for conditional branches. In *Proceedings of the International Conference on Computer-Aided Design*, 1989.

[Xil] Xilinx. ISE logic synthesis tools.

[Xvi] XviD MPEG-4 video de-/encoding solution. `http://www.xvid.org`.

[YSPJ97] T. Z. Yu, E. H.-M. Sha, N. Passos, and R. D.-C. Ju. Algorithms and hardware support for branch anticipation,. In *Great Lakes Symposium on VLSI*, 1997.

[ZSRM90] J. Zegers, P. Six, J. Rabaey, and H. De Man. CGE: Automatic generation of controllers in the CATHEDRAL-II silcion compiler. In *Design Automation Conference*, 1990.

Index