Code Generation and Optimization for Embedded Digital Signal Processors

by

Stan Yi-Huang Liao

S.B. (Elec. Engr.) Massachusetts Institute of Technology (1991)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1996

© Massachusetts Institute of Technology 1996. All rights reserved.

Author

Department of Electrical Engineering and Computer Science
January 22, 1996

Certified by

Srinivas Devadas
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by

Frederic R. Morgenthaler, Professor of Electrical Engineering
Chairman, Department Committee on Graduate Students

JUL 16 1996

ARCHIVES

LIBRARIES
Code Generation and Optimization for
Embedded Digital Signal Processors

by

Stan Yi-Huang Liao

Submitted to the
Department of Electrical Engineering and Computer Science
on January 22, 1996, in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Abstract
The advent of deep submicron processing technology has made it possible and
desirable to integrate a processor core, a program ROM, and application-specific
 circuitry all on a single IC. As the complexity of embedded software grows, high-
level languages such as C and C++ are increasingly employed in writing embedded
software. Consequently, high-level language compilers have become an essential tool
in the development of embedded systems.

Fixed-point digital signal processors are among the most commonly embedded
cores, due to their favorable performance–cost characteristics. However, these ar-
chitectures are usually designed and optimized for their application domain, and
pose challenges for compiler technology. Traditional compiler optimizations, though
necessary, are insufficient for generating efficient and compact code. Therefore, new
optimizations are required to produce code of the highest quality in a reasonable
amount of time. In this thesis the author presents techniques for code generation
and optimization that target embedded digital signal processors. These techniques
have proven to be effective in improving the performance and reducing the size of
compiled software. This thesis emphasizes optimization techniques; only by gaining a
deeper understanding of the problems involved can we then apply them to a wider
class of architectures.

Keywords—compiler optimizations, digital signal processors, embedded systems.

Thesis Supervisor: Srinivas Devadas
Title: Associate Professor of Electrical Engineering and Computer Science
To my beloved parents
Acknowledgments

Throughout the course of my undergraduate and graduate study at MIT, I have been indebted to many people—professors, fellow students, colleagues, friends, and family. Words can hardly express my most sincere gratitude to all who have helped me to grow during these golden years of study. I will attempt to do my best in the following.

First of all, I would like to thank Professor Srinivas Devadas, my thesis advisor and mentor since my sophomore year at MIT (1989). Srinivas has been an inspiration throughout these years I have worked with him. His energy, enthusiasm, wit, and cheerfulness have made study and research enjoyable.

I thank members of the Advanced Technology Group at Synopsys, Inc., especially Kurt Keutzer, Steve Tjiang, and Albert Wang. This thesis project commenced when I came to Synopsys for my first summer job in 1994, to work with them. Kurt introduced me to the problem of hardware–software co-design, and helped me to understand many of the issues involved and to clearly see the "big picture." Steve was instrumental in inducing me into the order of compiler-writers, and also provided the tools OLIVE and DAGWOOD which I used for tree matching and DAG matching. Albert, with his insight into combinatorial problems, elevated my understanding of several problems described in this thesis. Not only have Kurt, Steve, and Albert been great supervisors and colleagues, but also have they been good friends who encouraged me in those times when I became frustrated about research. Thanks also to Olivier Coudert and Richard Rudell for providing the efficient binate covering solver SCHERZO that I used to solve the code generation and code compression problems.

Collaboration and discussion with other members of the SPAM\textsuperscript{1} Project have been helpful in shaping this thesis work, and I would like to thank Randy Allen of Syn-

\textsuperscript{1}SPAM stands for the four institutions to which the members belong: Synopsys, Princeton, Aachen, and MIT.
opsys; Guido Araujo, Ashok Sudarsanam, and Professor Sharad Malik of Princeton University; Vojin Živoinović of Rheinisch-Westfälische Technische Hochschule Aachen, Germany; and Daniel Engels, George Hadjiyiannis, and Silvina Hanono, fellow students working with Srinivas; for their comments and participation in the research.

Since the days of UROP beginning in my sophomore year, I have enjoyed the companionship of members of the Eighth Floor VLSI Group, past and present, and I cherish the time I have been with the group. Thanks especially to Professor Jonathan Allen, Robert Armstrong, Sue Chafe, Mike Chou, Daniel Engels, George Hadjiyiannis, Silvina Hanono, Mattan Kamon, Ignacio McQuirk, José Monteiro, Amelia Shen, Luís Miguel Silveira, Ricardo Telichevesky, Filip Van Aelten, and Professor Jacob White for their assistance and encouragement. Professor Allen and Professor White served on my thesis committee and made invaluable suggestions for improvement of this thesis document. Daniel Engels, my office-mate, has also helped me with questions of form and style to make this thesis more reader-friendly. In addition, I thank my friends David Stephenson and Rajeev Surati, for their humor has turned dull days cheerful; and my former German teacher and friend Kermit Olson, who has continued to show his concern for my well-being after I left high school.

For as long as I have been in "the Hub of the Universe," the Boston Chinese Bible Study Group has been a home for me, and I wish to thank all the members, the fellowship with whom has enriched my life as a Christian. Special thanks are due to the Rev. John Tan, whose counsel I have often sought when I was perplexed about life and who has helped me in holding fast to my faith in LORD JESUS CHRIST. I am also deeply grateful to Paul and Chin-Fan Beckmann, Wan-Lin and Ru-Miao Hsu, Samuel Li, Frank and Christine Tang, and Jan-Ru and Tsing-Fang Tang, for their spiritual support.

Finally, I would like to give my wholehearted thanks to my family. To my sisters Joy and Grace, my brother Henryk, and my cousin Ying-Sheng Lee, I wish to express my appreciation for the care, attention, and comfort that I as the youngest in the family have constantly received from them. To my parents, Professor and Mrs. Mei-
Sung Liao, for their unceasing and unconditional love. In bringing me up and sending me to the United States to study, they have indeed sacrificed so much that I cannot ever possibly render to them anything comparable. I therefore respectfully dedicate this thesis to my parents as a token of gratitude. And thanks to the Most High God, whose lovingkindness endures forever and whose faithfulness is unto all generations!

◊◊◊

This thesis work was carried out at the Research Laboratory of Electronics of the Massachusetts Institute of Technology. Additional work was done at Synopsys, Inc., Mountain View, California. This research was sponsored in part by the Advanced Research Projects Agency under contract number DABT63-94-C-0053; in part by an NSF Young Investigator Award with matching funds from Mitsubishi and IBM Corporation; and in part by Synopsys, Inc., under Project SPAM. Their financial support for this work is gratefully acknowledged.
About the Author

Stan Yi-Huang Liao (廖逸晃) was born on March 22, 1972, in Pingtung, Taiwan, and came to the United States in 1985. He received his S.B. and S.M. degrees in Electrical Engineering and Computer Science, both from MIT, in June 1991 and September 1992. His research interests include logic synthesis, testing, verification, and high-level language compilers for hardware–software co-design of embedded systems.

Stan was a recipient of the Henry Ford II Scholarship (May 1991) for his academic achievements; the G. C. Newton Prize (May 1991) for the best undergraduate laboratory project; and the Best Paper Award in the MIT–ACM Student Conference (April 1991) for his work on the application of binary decision diagrams in sequential circuit testing. He has served as a teaching assistant for the undergraduate introductory course *Structure and Interpretation of Computer Programs* (6.001) and the graduate course *Computer-Aided Design of Integrated Circuits* (6.373) in the academic year 1992–93. Stan is a member of the Association for Computing Machinery (ACM), the Institute of Electrical and Electronics Engineers (IEEE), and Sigma Xi.

Recent Publications


For in much wisdom is much vexation,
and he who increases knowledge increases sorrow.

— Ecclesiastes 1:18

My son, beware of anything beyond these.
Of making many books there is no end,
and much study is a weariness of the flesh.

— Ecclesiastes 12:12
# Contents

Abstract ............................................. 3

Acknowledgments ...................................... 7

About the Author ....................................... 11

1 Introduction ........................................ 19
   1.1 Hardware–Software Co-Design ................. 21
   1.2 Reducing Code Size ............................ 24
   1.3 Summary of Contributions of This Thesis .... 27

2 Related Research and Approach of This Thesis .... 31
   2.1 Retargetable Compilation ....................... 32
   2.2 Traditional Compiler Research ................ 34
   2.3 Research in Code Generation for Embedded Processors 36
      2.3.1 MIMOLA .................................. 36
      2.3.2 CHESS .................................... 37
      2.3.3 FLEXWARE ............................... 39
   2.4 Approach of This Thesis ....................... 40
   2.5 Experimental Framework ...................... 41

3 Code Generation ................................... 45
   3.1 Tasks of a Code Generator .................... 47
3.2 Motivating Example .............................................. 50
3.3 Discovering Complex Instructions ............................... 53
  3.3.1 Basic Formulation ........................................ 53
  3.3.2 Transforming the Subject DAG ............................ 56
3.4 Data Transfers for One-Register Machines .................... 56
  3.4.1 Previous Work ............................................ 58
  3.4.2 Assumptions ............................................. 60
  3.4.3 Definitions ............................................. 60
  3.4.4 Binate Covering Formulation ............................. 66
  3.4.5 Fundamental Adjacency Clauses ......................... 67
  3.4.6 Clauses for U-Cycles ..................................... 70
  3.4.7 Self-Loops ............................................. 73
  3.4.8 Simple and Composite U-Cycles ......................... 76
  3.4.9 Clauses for Matches .................................... 77
  3.4.10 Clauses for Reloads and Spills ......................... 78
  3.4.11 Leaves and Roots of the DAG ........................ 80
  3.4.12 Summary of the Binate Covering Formulation ........... 82
  3.4.13 Optimality of the Binate Covering Formulation ........ 83
3.5 Extensions of the Binate Covering Formulation ............... 84
  3.5.1 Mode Optimization ...................................... 84
  3.5.2 Data Transfers for Multiple Register Classes .......... 87
3.6 Summary and Future Work ...................................... 91

4 Storage Assignment ............................................. 93
  4.1 Processor Model and Notations .............................. 96
  4.2 Simple Offset Assignment .................................. 96
    4.2.1 Example ............................................. 97
    4.2.2 Assumptions in SOA ................................ 99
    4.2.3 Approach to the Problem ............................ 101
CONTENTS

4.2.4 Access Sequence and Access Graph .......................... 101
4.2.5 SOA and Maximum Weight Path Covering .................. 103
4.2.6 Complexity Analysis ........................................... 108
4.2.7 A Heuristic Algorithm for SOA .............................. 110
4.2.8 Analysis of the Heuristic Procedure ....................... 112
4.2.9 A Branch-and-Bound Procedure for MWPC ................ 116

4.3 General Offset Assignment ....................................... 118
4.3.1 Example of GOA .................................................. 118
4.3.2 Formulation of GOA ............................................. 118
4.3.3 A Heuristic Algorithm for GOA .............................. 121

4.4 Offset Assignment for a Procedure ............................ 124
4.4.1 Example .......................................................... 130

4.5 Experimental Results .............................................. 132

4.6 Summary and Future Research ................................. 139

5 Code Compression ..................................................... 141
5.1 Previous Work ...................................................... 142
5.2 Our Approach ....................................................... 143
5.3 Preliminaries ......................................................... 146
5.3.1 Data Compression ............................................... 146
5.3.2 Definitions, Conventions, and Assumptions ................ 147

5.4 Examples .......................................................... 148
5.4.1 Example Illustrating the First Method ..................... 149
5.4.2 Example Illustrating the Second Method .................... 151

5.5 Proposed Compression Methods ................................. 153
5.5.1 Method I .......................................................... 153
5.5.2 Method II ......................................................... 157

5.6 An Algorithm for Code Compression .......................... 160
5.6.1 Generation of Potential Dictionary Entries ................. 160
5.6.2 Substitution and Dictionary Generation ........................................ 162
5.7 Refinements to the Proposed Algorithm ........................................... 167
   5.7.1 Eliminating Inefficiently Used Entries ....................................... 167
   5.7.2 Generating More Potential Dictionary Entries ............................... 170
5.8 Experimental Results ........................................................................ 171
5.9 Performance Considerations ............................................................ 173
   5.9.1 Extension of Covering Formulation for Performance ...................... 177
5.10 Summary and Future Research ........................................................ 178

6 Conclusion ......................................................................................... 181
   6.1 Future Work .................................................................................. 183

A Covering Problems ............................................................................ 185
   A.1 Set Covering .................................................................................. 186
   A.2 Binate Covering ............................................................................. 188
   A.3 Solving Covering Problems ............................................................ 189

B Other Optimizations .......................................................................... 191
   B.1 Interprocedural Analysis and Optimizations .................................... 191
   B.2 Static Allocation ............................................................................. 193
   B.3 Efficient Use of the Hardware Stack ................................................. 196
   B.4 Global Mode Optimization .............................................................. 199

Bibliography ......................................................................................... 203
Code Generation and Optimization for
Embedded Digital Signal Processors
Chapter 1

Introduction

Recent years have seen an amazing growth in the electronics and semiconductor industry. Consumer electronics products such as personal digital assistants, multimedia systems, and cellular phones are becoming ubiquitous. Electronic systems also find much utility in many and diverse fields: the automobile industry (e.g., brake systems and engine control), medical applications (e.g., electroencephalogram (EEG), electrocardiogram (EKG), and imaging), business equipment (e.g., smart copier and printer), among countless others.

Due to their enormous market demand, the manufacturing of electronic systems is very cost-sensitive. Moreover, many applications (e.g., cellular phones) have stringent requirements on power consumption, for the sake of portability. For these reasons, manufacturers profit from integrating an entire system on a single integrated circuit (IC) [Goossens 92] [Depuydt 93].

This desired high level of integration has been made possible by the advent of deep submicron processing technology (e.g., the 0.25-micron G10™ Technology of LSI Logic [LSI 95]), which allows for the integration of several million usable gates on a single die. The design of a cellular/PCS radio system for use in handsets and in PCMCIA wireless modem cards or wireless subsystems in PCs, shown in Figure 1-1, demonstrates the immense complexity of circuits that can be integrated on a single chip today.
Figure 1-1 A single-chip cellular/PCS radio system core for deployment in handsets and in PCMCIA wireless modem cards or wireless subsystems in PCs. From G10™ Technology for Communication Applications Product Brief, courtesy of LSI Logic Corporation.
1.1 HARDWARE–SOFTWARE CO-DESIGN

Such a high level of integration afforded by processing technology has, on the other hand, brought new challenges in the design of digital systems. Designing an entire system with custom integrated circuit is now neither economical nor practical. As time-to-market requirements place greater burden on designers for fast design cycles, programmable components are introduced in the system and an increasing amount of system functionality is implemented in software relative to hardware [Devadas 94, page 393]. These programmable components, called embedded processors or embedded controllers, can be general-purpose microprocessors, off-the-shelf digital signal processors (DSPs), in-house application-specific instruction-set processors (ASIPs), or microcontrollers. Systems containing programmable processors that are employed for applications other than general-purpose computing are called embedded systems.

The advantages of incorporating software components are twofold. First, whenever a software solution offers acceptable performance, it is usually preferred to hardware, because software design does not require the synthesis of custom control and data-paths, but rather uses predefined processors. Only the most time-critical tasks need to be implemented in hardware. Second, since embedded processors are field programmable, software design is more flexible than hardware design, and design errors, late specification and design changes, and product evolution can be accommodated more easily [Van Praet 94] [Paulin 95]. Moreover, various applications of the same genre may share the same hardware structure, with differences reflected in the software component, and it is possible to use the same mask for all of these applications. Since the aggregate volume of a number of applications is higher than the individual volumes, using software components substantially reduces the cost of manufacturing.

1.1 Hardware–Software Co-Design

As a result of the advantages of software, modern integrated systems are often composed of a heterogeneous mixture of hardware and software components. For
Figure 1-2 A heterogeneous system-on-a-chip. With the DSP core are integrated the program ROM, some application-specific circuitry (ASIC), A/D and D/A converters, serial/parallel converters, and direct memory access.

instance, Figure 1-2 presents one such heterogeneous system, consisting of a digital signal processor (DSP) core, a program ROM, RAM, application-specific circuitry (ASIC), and other interface and peripheral circuitries. Because of the trend towards this design style, developers of computer-aided design (CAD) tools are faced with the challenge of providing circuit designers with tools that can support the design of such systems. There have been proposals for the hardware–software co-design of digital systems (e.g., [Gupta 93], [Kalavade 93], and [Thomas 93]). The simplified view of a generic co-design methodology is shown in Figure 1-3.

In this design flow, the designer first determines which parts of the functionality of the system will be implemented in hardware and which parts in software. He then proceeds to design each of the hardware and the software components. The system is simulated and evaluated with a hardware–software co-simulator. If the results of the simulation meet design specifications (e.g., correctness and timing constraints), then the design is accepted. Otherwise, the designer may re-partition the original algorithmic specification and reiterate the same process.
Figure 1-3 A generic hardware–software co-design methodology. The designer first determines which functions of the system will be implemented in hardware and which ones in software. He then proceeds to design each of the hardware and software components. Simulation is used to evaluate and verify the design. The designer re-partitions the system and repeats the design process if necessary.
Under this methodology, tools for code generation and hardware-software co-simulation have become essential parts of the designer's tool-box. Specifically, as we will see in Section 1.2, compilers for software written in high-level languages are indispensable in the design of embedded systems.

This thesis addresses several issues involved in developing compiler optimizations for fixed-point digital signal processors, which are increasingly used as the processor component in embedded systems due to their favorable performance-cost characteristics [Beckmann 95]. Unlike work on traditional compilers which focuses primarily on performance, our work gives the same emphasis to code size as to performance; this emphasis will be justified in the section that immediately follows.

1.2 Reducing Code Size

Notwithstanding the large number of gates provided by new processing technology, it is still desirable to minimize the size of an integrated circuit, because the cost of the IC is most closely linked to its size. In fact, semiconductor costs vary exponentially with die size—smaller die sizes are conducive to higher yields [Holt 95]. In heterogeneous systems such as that shown in Figure 1-2, it is not unusual for the single largest factor of the area to be the ROM storing the program code for the embedded processor core. While reducing the size of the application-specific circuitry via logic synthesis and optimization techniques is absolutely necessary, its incremental value is smaller in cases where the application-specific circuitry comprises a relatively small percentage of the overall circuit area. On the other hand, there is great potential for cost reduction through diminishing the size of the program ROM. Also, there are often strong real-time performance requirements on the embedded software; hence, there is a necessity for producing high-performance code as well.

Alternatively, a given target die size for a product may limit the size of the ROMs and therefore the size of the code. In many embedded-system projects, the ROM space estimated at the beginning becomes insufficient later in the development
1.2 REDUCING CODE SIZE

phase or during program maintenance. In order to avoid excessive design modification,
designers usually have to work diligently to reduce the code size, sometimes at the
expense of removing certain features of the product [Ganssle 92].

Besides code size and performance, there are other important constraints and
requirements for embedded systems, most notably data memory and power dissipation. However, it is often easier to satisfy constraints on data memory than those on
program memory, since different data of the program can share the same portion of
data memory if their life-times are disjoint. As for power dissipation, we observe that
code that executes more quickly also consumes less energy, and if we can lower the
clock frequency while meeting throughput requirements, power dissipation diminishes
as well. Although there has been work on code generation specifically for low power
(e.g., [Tiwari 94]), we believe that performance is the main factor affecting power
dissipation. Hence, we will focus our efforts on code size and performance.

The traditional approach to high-quality embedded software has been to write
the code in assembly language. As the complexity of embedded systems grows,
nevertheless, programming in assembly language and optimization by hand are no
longer practical, except for time-critical portions of the program that absolutely require
it. Recent statistics from Dataquest indicate that high-level languages such as C and
C++ are gradually replacing assembly language [Keutzer 95], because using high-level
languages greatly lowers the cost and time of development, as well as the maintenance
costs of embedded systems. Also, less effort is required to reuse software written in
high-level languages.

Programming in a high-level language may, however, incur a code-size penalty.
One reason is that compiler optimization techniques (for examples see [Aho 86])
have classically focused on code speed and not code density, and most available
compilers optimize primarily for speed of execution. Although some optimizing
transforms such as local common subexpression elimination can improve both speed
and size at the same time, in many cases there is a speed–size trade-off. For example,
subroutine calls take less space than in-line code, but are generally slower due to
call overheads. Other optimizations that have potential speed–size trade-offs include induction variable elimination (or strength reduction) [F Allen 72], loop unrolling [Lam 88], and partial redundancy elimination [Morel 79]. Where execution speed is not critical, minimizing the code size is usually profitable. A second reason for the code-size penalty is that compiler-optimization techniques have typically been limited to approaches which can be executed quickly (less than $O(n^2)$) because programmers require fast compilation times during development. For this reason the numerous NP-hard optimization problems associated with code optimization are rarely faced directly, as they often are in computer-aided design of hardware, but are usually approached with simple linear-time heuristics. Present commercial DSP compilers, which employ standard software compilation techniques, often produce code of quality that leaves much room for improvement [Živojinović 94] [Goossens 96].

Thus the central theme of this thesis is that a new goal for code optimization has emerged: the generation of the most dense code with the highest performance, obtainable within any reasonable compilation time. By *reasonable* we mean the amount of time that a hardware designer can tolerate using existing CAD tools for hardware synthesis, which often face NP-hard problems squarely and solve them either exactly or using powerful, super-linear heuristics. In the context of embedded-system design, the software compiler plays much the same role as hardware synthesis tools—since the compiled code will eventually become part of the hardware. Hence, longer compilation time is available to us.

We have two important premises. First, the methodology required for code generation will be most useful if it can be easily adapted to generating code for different processors. This property, commonly called *retargetability*, will be discussed in Chapter 2. Second, the methodology will require, in addition to traditional code optimizations, new techniques specifically for the classes of processors that are commonly found in embedded systems: off-the-shelf fixed-point DSP architectures and application-specific instruction processors used in DSP applications. These seem to be the most challenging targets for the following reasons. The emphasis in digital signal
processing has for many years been on performance, and these processors have traditionally been programmed in assembly language and hand-optimized. In addition, DSP architectures are generally designed with little regard for compilers [Lee 88] [Lee 89]. Due to the architects' quest for good performance and low cost, irregular data-paths and limited addressing capabilities are often present in DSP architectures. Although programmers may rely on optimized library routines for such common tasks as convolution, correlation, and fast Fourier transform, compiler optimizations are necessary for the relatively unstructured portions of the code. There is a clear trend that more complex tasks will involve much unstructured code [J Allen 85], partly due to the use of special-purpose algorithms and partly due to the need for the processor to handle events and interrupts.

Therefore, the goal of this thesis is to develop techniques for code generation and optimization for these processors. For our experiments we have used existing off-the-shelf DSP processors. Although these are not ASIPs per se, they share a large number of characteristics with ASIPs that make code generation for them a difficult task: the presence of irregular data-paths and nonhomogeneous, specialized registers [Lanneer 95].

1.3 Summary of Contributions of This Thesis

This section briefly summarizes the contributions of this thesis to the area of code generation and optimization for embedded digital signal processors. A survey of relevant compiler literature and of other related research, as well as the approach of this thesis, is given in Chapter 2. We also discuss the notion of retargetability and justify the approach of this thesis from the standpoint of compiler optimizations.

In Chapter 3 we present a new formulation for code generation based on binate covering. This formulation takes into consideration the effects of scheduling on data transfer costs. The most important contribution of Chapter 3 is a new theory of code generation for noncommutative one-register machines. Central to this theory is the notion
of *worms* and *worm-partitions*, which place scheduling constraints on an expression directed acyclic graph (DAG). Based on properties of worm-partitions, we derive a compact set of clauses to describe the set of all legal worm-partitions. The variables in these clauses appear in other clauses to relate scheduling with the selection of instruction patterns and register transfers. Unlike previous work (e.g., [Aho 77]), our formulation clearly and succinctly describes the scheduling constraints along with accumulator spills and reloads, and allows for more-flexible cost functions. We also propose extensions of this formulation to tackle multiple-register machines and other optimization objectives such as the *node optimization problem* of [Liao 95a].

In Chapter 4 we address the problem of storage assignment that one frequently encounters in DSP architectures. Unlike general-purpose register machines, DSP architectures typically have limited addressing modes and storage assignment has a great impact on the size as well as performance of the generated code. We present algorithms for effectively exploiting the auto-increment and decrement features of most DSP architectures. In our approach to this problem, we first consider the problem of using a single address register to address all variables in a procedure. We then extend our formulation to allow for the use of multiple address registers. As the experimental results demonstrate, using multiple address registers often yields more compact code.

In Chapter 5 we present a methodology for compressing the object code, based on textual substitution. This methodology trades off speed for size. However, as we shall see, because a program spends most of its time in a relatively small part of the code, compressing the rest of the code has little impact on the overall performance while substantially reducing the code size. We present an algorithm for discovering common sequences of instructions throughout the program, and generating a dictionary as well as determining which occurrences of the sequences should be substituted in the program. Our formulation of the problem, based on *set covering*, naturally takes into consideration the size–performance trade-offs, and the user can specify parameters for these trade-offs.
Finally, Chapter 6 concludes the thesis with a retrospective examination of what has been achieved in this thesis, and with directions for future research.

Appendix A gives a concise review of the combinatorial optimizations problems known as set covering and binate covering. Although these problems are NP-hard, average-case efficient algorithms have been proposed. By casting the problems of code selection and code compression as covering problems we benefit from these algorithms.

Appendix B describes several other optimizations based on interprocedural analysis that can be applied to further improve the quality of the generated code. These optimizations, though simple in nature, may cooperate to contribute substantially to reducing code size and improving performance, and, therefore, should not be neglected.
Chapter 2

Related Research and Approach of This Thesis

In response to the need for software-compilation tools that comprise part of a hardware-software co-design environment, there has been increasing research interest in code generation for embedded processors, as evinced by the First Workshop on Code Generation for Embedded Processors (Schloss Dagstuhl, Germany, 1994) [Marwedel 95]. Various researchers have taken different approaches to the difficult problem of developing compilers that have the following properties:

1. The code generated by the compiler must be of the highest quality attainable within a reasonable amount of time.

2. The compiler should be easily adapted to varying target architectures.

We have given reasons for the importance of generating code with both good performance and compact size in Section 1.2. This should be achieved with aggressive problem formulations and powerful algorithms at the expense of longer compilation time. The second property, commonly called retargetability, has attracted the attention of several research projects. The need for a retargetable compiler arises from the fact that, in the hardware-software co-design iteration (see Figure 1-3, page 23), the target architecture may itself be changing, especially for designs involving application-specific instruction-set processors (ASIPs). It is impractical, if not impossible, to rewrite the software compiler each time when features are added to or removed from the
present architecture, or when a new architecture is chosen. A compiler that takes a machine description of the target architecture and automatically retargets itself is a much desired tool.

Nevertheless, code quality and retargetability are often at odds with each other. If we were to quantify these properties, we might observe an inverse relationship between the two. This is because program optimizations are indispensable for generating high-quality code and many of these optimizations are machine-dependent to a large extent. Optimizations suitable to one architecture may be inapplicable to another. Furthermore, even if two architectures share a similar feature, adapting an optimization technique for this feature from one to the other may be a nontrivial task and require much manual work, due to interaction with other machine idiosyncrasies.

In Section 2.1 we will first expound the notion of retargetability, which carries a variety of connotations and is used in all senses by various researchers. We will then present a brief survey of the traditional compiler research in Section 2.2 and of recent work on code generation for embedded processors in Section 2.3. In Sections 2.4 and 2.5 we state the approach of this thesis and give a detailed description of the framework in which our designs and experiments were carried out.

### 2.1 Retargetable Compilation

The term retargetable compilation has been widely used and describes a broad range of capabilities. It is instructive to examine the various levels of retargetability. We divide the spectrum into three categories and detail the extent to which each of these categories applies.

- **Automatically retargetable.** The (somewhat farfetched) ideal automatically retargetable compiler takes in a description of the target architecture in its structural form and generates code for the target. By structural form we mean a form that is suitable also for the synthesis of hardware. Thus the code generation process requires no user intervention. In practice, however, this level of
2.1 RETARGETABLE COMPILATION

Retargetability seems to be applicable to a limited family of architectures within which variations are well characterized, e.g., number of registers in a register file, bit-widths, and available functions of an execution unit. Essentially, all possible target architectures that the compiler is intended to be used for are already built in.

- **User retargetable.** Here, the user specifies the target architecture to a compiler generator by describing the instruction set and the actions of each instruction. Such a description is usually called *behavioral*. The compiler generator takes the description as input, and outputs a compiler for the target architecture. This method has been used with some success for the code selection phase of existing compilers.

  A widely used behavioral description methodology is based on *grammars* and their derivatives (e.g., tree-matcher generators). For example, the input to the code-generator generator T WIG [Aho 89] consists of a set of patterns with a cost function for each pattern and a sequence of actions corresponding to emitting code for matches with this pattern. Another approach is that of nML (Section 2.3.2) which captures a programmer’s model of the processor at the level of a data book specification by a hybrid of high-level structural information and a full description of the instruction set.

  Optimizations considered by these methodologies, however, are typically limited to instruction selection. More complex machine-specific optimizations are difficult to handle in such an environment.

- **Developer retargetable.** This level of retargetability is sometimes simply termed *portability*. One way to handle machine specific optimizations that go beyond instruction selection is to permit the developer to modify the compiler to target the given architecture. Clearly the dividing line between retargeting and essentially writing a new compiler for each architecture is rather thin here. We believe that for a compiler to be considered retargetable in this scenario, no
new processor-dependent optimization capabilities are added to the compiler during retargeting. Instead, the developer is using the processor-dependent architectural information to tailor the built-in optimization algorithms, and sequence them in the most effective order for that architecture.

2.2 Traditional Compiler Research

This section presents a survey of traditional compiler literature that pertains to retargetability and optimizations. By traditional we mean compiler research geared towards general-purpose computing rather than specialized architectures such as digital signal processors.

Portability has always been a concern of compiler researchers since the inception of high-level programming languages and compilers [Conway 58]. Therefore, almost all compilers are organized in two major phases [Wulf 75] [Aho 86] (which may, of course, consist of smaller phases). The first phase, called the front-end, consists of lexical analysis and parsing. The task of the front-end is to translate a program from its source-language form to an intermediate form that is largely language-independent and machine-independent. The second phase, called the back-end, translate the program in the intermediate form to assembly code (or object code if the assembler is considered part of the back-end) for the target architecture. In such an organization, portability is achieved in the sense that, whenever a new architecture is to be targeted, only the back-end (instead of the entire compiler) needs to be rewritten.

The table-driven code generation technique proposed by Glanville and Graham [Glanville 78] [Graham 80] was one of the first efforts to advance the notion of portability by describing the code generator in a grammar and automatically generating the code generator, in a manner much analogous to the parser generator YACC. Thus, another level of abstraction is created whereby the compiler writer may describe his code generator more easily and does not have to write it in a general-purpose high-level language. Ganapathi and Fischer refined this basic grammar-based approach by
using affix grammars with attributes [Ganapathi 85]. A survey of various grammar-based code generation methods is presented in [Ganapathi 82].

An alternative to parsing is to use pattern-matching techniques on expression trees. Cattell designed a framework in which code generators based on pattern-matching are automatically derived from a machine description language [Cattell 80]. The landmark paper by Aho et al. [Aho 89] established the foundation in which several modern dynamic-programming code-generator generators find their origin. One such code-generator generator is IBURG [Fraser 92], employed in the compiler framework LCC [Fraser 95]. The dynamic-programming methodology yields locally optimal code (i.e., within the expression tree). However, local optimality is often insufficient when the piece of code is placed in the context of the entire procedure, and procedure-wide and program-wide optimizations are required to further improve the code.

There is a wealth of optimization techniques in the literature of compiler research (see, for example, proceedings of the annual Conferences on Programming Language Design and Implementation (PLDI) and of the Symposia on Principles of Programming Languages (POPL)). Most machine-independent optimizations, such as constant propagation, global common subexpression elimination, dead code elimination, have been very well understood [F Allen 72]. Other optimization problems, such as register allocation, are closely tied to the target architecture. Thus, these problems are more difficult to formulate precisely. A common approach to such problems is to use a simpler model that can approximate a wide class of architectures. For instance, for the register allocation problem, the usual starting point is a uniform register set which provides a good approximation for most RISC architectures, and for which several algorithms have been devised (e.g., [Chaitin 81], [F Chow 90], and [Callahan 91]). Adapting the algorithms to a particular target may, however, still require much manual effort, since each architecture has its features that need special treatment if they are to be exploited.

Several researchers have, on the other hand, attempted to automatically derive machine-specific optimizations from a description of the target machine. Davidson and
Fraser have proposed a methodology for automatically deriving peephole optimizers from a machine description that consists of a set of code-transformation templates [Davidson 82] [Davidson 84]. One of the limitations of this methodology is that the two- or three-instruction "window" sometimes fails to discover optimizations that are obscured by the presence of unrelated instructions within the window. Another notable work is that of Giegerich, who improved upon the Davidson–Fraser approach by taking machine-level data-flow information into consideration [Giegerich 83]. These methods operate at a late stage of the compilation process (i.e., object-to-object transformation) and are not capable of capturing the optimizations at earlier stages, such as register allocation and scheduling.

More recently, Bradlee proposed a retargetable scheduler MARION that is used mainly for RISC architectures [Bradlee 91]. It uses a machine description that models pipelines and superscalar instruction issues, and schedules instructions accordingly to effectively utilize the features or to reduce conflicts. Other works on retargetable compilers include that of the PowerPC compiler [Shipnes 94], which is used for the various members of the PowerPC family.

2.3 Research in Code Generation for Embedded Processors

We have selected for review and critique three representative research projects in the area of code generation for embedded systems: MIMOLA [Marwedel 84], CHESS [Lanneer 95], and FLEXWARE [Paulin 95]. The proceedings of the Dagstuhl Workshop [Marwedel 95] contains a collection of papers documenting several other contributors' efforts.

2.3.1 MIMOLA

The MIMOLA design system was originally conceived as a design environment for hardware structures, using the MIMOLA hardware description language [Zimmermann 79]. It later evolved into an environment for hardware–software co-design and includes
a retargetable microcode compiler [Marwedel 84] [Marwedel 93]. The microarchitecture structure is described in the language MIMOLA as before, and the description is translated into an intermediate representation called TREEMOLA. In addition, the algorithm to be compiled into microcode is also written in MIMOLA, although using the behavioral subset (which is similar to the high-level language Pascal) instead of the structural subset. The code generator operates by pattern-matching between the algorithm and the machine structure.

One of the key features that distinguish the MIMOLA project from others is that the microcode compiler infers rules for code generation directly from a structural description (e.g., a net-list) of the target architecture, instead of a behavioral description (e.g., the instruction set). The advantage of this approach is that it provides a uniform mechanism of machine description for the various tasks in the entire design process: the same machine description is used for both the synthesis of the target architecture and the generation of microcode. The uniformity and consistency of representation, if successfully introduced, are very desirable properties because they help to attain the goal of automatic retargetability. To allow the basic MIMOLA system to handle a wider class of architectures, Leupers et al. have presented a method for extracting the instruction set from the structural description [Leupers 94b], and have retargeted their compiler for the TMS320C25 digital signal processor [Leupers 94c].

However, even though the MIMOLA methodology is interesting in its own right, the publications suffer from the fact they offer relatively few demonstrable results, and these few results are often not easy to evaluate and interpret. Hence, the effectiveness (in terms of code quality and of the amount of manual work involved) of the many techniques proposed (e.g., [Marwedel 93] and [Leupers 94a]) remains to be supported with more experimental results.

2.3.2 CHESS

CHESS [Lanneer 95] is a retargetable code generation environment for fixed-point digital signal processors and ASIPs; it was developed in the context of the CATHE-
DRAI II high-level synthesis system [Goossens 95]. Unlike MIMOLA which uses mostly-structural models, CHESS employs a mixed behavioral-structural model for processor representation.

The code generation process consists of six major phases: (1) optimizing transformations and flow graph refinement, (2) code selection, (3) register allocation, (4) bit alignment, (5) scheduling, and (6) code assembly. The first phase refines the operations in the source program so that they can be implemented with micro-operations of the processor. This is followed by the second phase that does the actual mapping of operations in the transformed program to partial micro-operations supported by the instruction set. A technique called data routing for the register allocation phase is used to determine where values are to be stored and how they are transferred between different storage elements. Bit alignment assures a correct bit-level behavior of the implementation. Finally, a complete schedule is constructed using a list-scheduling algorithm and object code is emitted.

The target machine is described using the language nML [Fauth 95]. A graphical representation of the architecture called the instruction-set graph (ISG) is then derived from the nML description. The ISG representation describes the connectivity and timing characteristic (i.e., transient vs. permanent) of the various resources (i.e., execution units, busses, and registers) along with encoding of micro-operations and restrictions on the encoding. Paths in the ISG correspond to valid instructions, and the code selection phase, instead of using pattern matching techniques, searches for paths in the ISG that can implement the operation in question and emits micro-operations. The register allocator, on the other hand, uses the connectivity and timing information to make routing and storage decisions.

The nML mechanism appears attractive because it allows the user to specify the target architecture in a way that parallels instruction-set descriptions found in a user’s manual. In contrast to MIMOLA, the machine description contains behavioral information as well as structural. This helps the code generator to recognize more optimization opportunities. However, there have been few convincing experimental
results thus far, and the authors of CHESS have not stated to what extent they have successfully retargeted their compiler.

2.3.3 FLEXWARE

FLEXWARE consists of two components: a code generator CODESYN [Paulin 94] and an instruction-set simulator INSULIN [Sutarwala 93]. FLEXWARE is a tool-set developed in response to the results of the survey conducted by Paulin et al. regarding trends and requirements in DSP design environments [Paulin 95].

In the code generation process of CODESYN, the source program is first translated into an intermediate form, called BDS (BNR Data Structure). The graph-rewrite phase transforms complex constructs (e.g., if-then-else) in the BDS into constructs available in the target machine. A pattern-matching phase using the dynamic-programming paradigm of [Aho 89] selects instructions from the instruction set to implement the functions represented by the subject graph. Global scheduling and register allocation then follow. Finally, the micro-operation compaction, assembly, and linking stages produce the object code.

The machine description for CODESYN consists of three components: the instruction set, the available resources and their classification, and the micro-instruction format. The description of the instruction set is composed of graph patterns representing the individual instructions along with their timing properties; these graph patterns are utilized in the pattern-matching phase. Resources (i.e., register and memory) are described in terms of their connectivity and relationship (e.g., the points-to relation) with one another. Furthermore, they are classified according to their functionality, which guides the register allocation process [Liem 94]. The micro-instruction format is used for compaction, assembly, and linking.

Although the approach of Paulin et al., unlike MIMOLA and CHESS, does not use a unified machine representation, it is, among these three approaches, the most conducive to good code quality. The reported experimental results claim that the size of the generated code is within 20% of hand-crafted code size. The benchmarks used
in their experiments, however, are quite small (fewer than 40 lines of C code) and not diversified. Therefore, it is not clear how the results would extrapolate for larger and more realistic examples.

2.4 Approach of This Thesis

One of the most conspicuous shortcomings of the above-cited works, and of the research on code generation for embedded processors at large, is that experimental results are few and difficult to compare. This is partly due to the inevitable circumstance that each research project has its own development environment and many methods are particular to their respective environments. Because of the lack of a standard set of benchmarks, it is not obvious how to independently evaluate the effectiveness of the various techniques proposed. Furthermore, aggravating the situation is the fact that many subproblems encountered are not formulated in a well-defined manner, and in many cases ad hoc solutions are only lightly sketched without substantial contribution to the understanding of the problems. Hence, it seems infeasible to apply the proposed techniques in other contexts.

Based on this observation and our own experience, we conclude that optimizations are indeed at odds with retargetability. Marwedel's notion of automatic retargetability, when taken to the extreme, appears to be out of reach if high-quality code is desired. We believe that it is still possible to achieve a certain degree of retargetability by limiting ourselves to a family of architectures, within which variations can be formally characterized. To this end, we will need to study existing architectures and ask ourselves why certain features were designed, and what optimizations, if applicable and feasible, are necessary to exploit such features. This naturally leads to a study of the design of architectures in which compiler support plays an important role [P Chow 94].

Therefore, in contrast to MIMOLA and CHESS cited in Section 2.3 which primarily emphasize retargetability, this thesis approaches the software compilation problem
from the standpoint of the generation of high-quality code, and, in particular, program optimization techniques. It is not the intention of this thesis to de-emphasize the importance of retargetability. Rather, we believe that retargetability should not be gained at the expense of code quality. By focusing on code optimization techniques we will gain a deeper understanding of what architectural features are amenable to compilation and what features are difficult for the compiler to utilize. Only then can we design machine-description mechanisms to facilitate quick retargeting of the optimization techniques.

2.5 Experimental Framework

In our experimental framework, we use the SUIF Compiler [Wilson 94] as the front-end to translate source programs from C into the Stanford University Intermediate Form. (SUIF is the name of the compiler as well as the abbreviation for the intermediate form.) This thesis focuses on several optimization problems of the back-end. The overall compiler organization is illustrated in Figure 2-1 (page 42), and the problems addressed by this thesis are highlighted. It is worth pointing out that the organization of our compiler is similar to that of FLEXWARE, in that different tasks require different kinds of machine descriptions.

A program written in C is first translated into SUIF, which is a largely machine-independent representation of the program. Machine-independent optimizations, such as global common subexpression elimination and dead code elimination, are performed at this stage. Then, the program is translated via a preliminary code generation stage into another intermediate form called TWIF. The preliminary code generated is produced from a rule-based machine description written in OLIVE [Tjiang 94] or DAGWOOD [Tjiang 95]. OLIVE, a descendent of TWIG [Aho 89] and IBURG [Fraser 92], is a language for writing tree-matchers based on dynamic programming. If tree-covering for preliminary code generation is desired, OLIVE allows for compact specifications of code generators. Otherwise, as in Chapter 3, we use DAGWOOD to specify instruc-
Figure 2-1 Overall compiler organization. The source program undergoes several stages of translation and optimizations. This thesis addresses the problems of preliminary code generation, storage assignment, and code compression.
tion patterns and generate matches for the construction of the corresponding binate covering problem.

TWIF serves as a secondary intermediate form that captures some machine-dependent information such as (most of) the instruction set, while remaining largely machine-independent in form (e.g., call graph representation of the program and control-flow graph representation of the procedures). The purpose of this secondary intermediate form is to support optimizations which are to some extent machine-dependent, but whose basic formulations and algorithms can be shared across a range of architectures. These optimizations include those based on global data-flow analyses (e.g., global mode optimization discussed in Appendix B.4), refinements to the schedule produced by the preliminary code generation phase, traditional register allocation, storage assignment (Chapter 4), and interprocedural analyses and optimizations (Appendix B.1).

The instruction set of TWIF need not correspond exactly to that of the target machine. Certain macros and pseudo-instructions can be created for the sake of convenience during the optimizations. For example, the simple model we have used in the discussion of storage assignment does not precisely reflect the TMS320-C25 architecture. The latter, instead of encoding the current address register in the instruction word, uses a register ARP (Figure 3-3 on page 51) to indicate the current address register and encodes the next address register in the instruction word. Machine idiosyncrasies such as these present inconveniences to the optimizations. Our approach is therefore to use an instruction set containing macros and pseudo-instructions that are amenable to analyses and optimizations.

It is then the task of the final code generation phase to translate macros and pseudo-instructions into actual target machine instructions. Also, because interprocedural analysis (Section B.1) is performed in the TWIF intermediate form, this phase assumes the task of resolving symbolic addresses of global variables and procedures as well. Along with this step we can perform peephole optimizations (e.g., [McKeeman 65], [Lamb 81], and [Davidson 82]) to eliminate redundancy that may have been neglected
in earlier phases or may have arisen from the translation of macros and pseudo-instructions. With peephole optimization the compiler attempts to find small sequences (using a sliding peephole) in the assembly or object code and either remove useless instructions in the sequences or replace the sequences with shorter ones. Finally, if further reduction in code size is desired, code compression can be applied to the object code.

Our work on code generation (Chapter 3) and on storage assignment (Chapter 4) has been implemented in this framework. At the time of this writing, however, the SUIF Compiler still lacks many of the standard machine-independent analyses and optimizations. Therefore, for our experimentation in code compression (Chapter 5), which is very sensitive to optimizations of the earlier phases, we have chosen to use the native TMS320C25 optimizing compiler to generate assembly code, instead of the yet-incomplete compiler based on SUIF.
Chapter 3

Code Generation

While several compiler researchers have proposed alternative program representations such as the program dependence graph (PDG) [Ferrante 87] and the value dependence graph (VDG) [Weise 94] [Ruf 95], the most widely used representation of program structures to date has still been the control-flow graph. Although the PDG and the VDG provide more-powerful program analyses, it is relatively difficult to generate code from these representations and the incremental benefits they offer still remain to be realized. A control-flow graph representation, in contrast, more closely models the working of most processors, and data-flow analyses and optimizations are thoroughly understood under this representation. Alternatively, we may begin with control-flow graphs generated after PDG- or VDG-based analyses and need not concern ourselves with these earlier stages. Hence, throughout the thesis we will assume the control-flow graph representation.

A control-flow graph is a directed graph in which the vertices are called basic blocks and the edges denote possible flows of control. A basic block consists of a (maximal) sequence of instructions such that if any instruction of the sequence is executed, then so is every other instruction of the sequence. For the purpose of code generation, a basic block is usually represented by an expression directed acyclic graph (DAG), which naturally discovers local common subexpressions [Aho 86]. It is also possible to obtain larger and more complicated DAGs for traces [Fisher 81] [Ellis 85]
\[ p = c - g; \]
\[ t = b \ast p; \]
\[ d = a + t; \]
\[ u = g \ast h; \]
\[ e = t + u; \]
\[ f = u - i; \]

Figure 3-1 Constructing an expression DAG from a basic block. (a) C code for a basic block. (b) The corresponding expression DAG, assuming \( p, t, \) and \( u \) are not live upon exit of the basic block.

that cross basic block boundaries. The present chapter will focus on code generation for expression DAGs, regardless of how they are constructed.

Figure 3-1(a) shows some C code representing a basic block. Suppose the variables \( p, t, \) and \( u \) are not live on exit of the basic block. The expression DAG derived from this C code is shown in Figure 3-1(b). The leaves of the DAG, represented as squares, correspond to primary inputs, whose values are assumed to reside in the memory at the beginning of the evaluation of the DAG. The roots, represented as double-circles, correspond to primary outputs (i.e., variables live on exit), into whose memory locations values are to be stored. Every other vertex corresponds to some computation. The vertices corresponding to the variables \( p, t, \) and \( u \) are not connected to primary output vertices because they are not live on exit and it is not necessary (unless required during the evaluation of the DAG) to store these values into the memory.
3.1 Tasks of a Code Generator

Code generation is traditionally viewed as consisting of three main tasks: code selection, scheduling, and register allocation [Goossens 96]. Code selection is the task of mapping operators in the intermediate form into target machine operators. Scheduling is the task of ordering the instructions to make the program more efficient and/or smaller. Register allocation is the problem of deciding which values will reside in which registers at every point in the program. These are analogous to the main tasks of high-level synthesis: execution unit mapping, scheduling, and resource allocation.

The problem of code generation for expression DAGs has long been known to be computationally complex for many machine models, because the three main tasks of code generation are coupled and because the scheduling of DAGs is by itself difficult for most objective functions. The phase-coupling problem [Goossens 96] is especially severe in machines with few registers and irregular data-paths.

The standard heuristic to alleviate the problems caused by DAGs is to break a subject DAG into a forest of trees by cutting it at vertices with multiple fanouts. For example, consider the DAG in Figure 3-2(a) (page 48). The two multiply vertices have multiple fanouts, and at these vertices new primary outputs, $t$ and $u$, are created (Figures 3-2(b) and (c)). References to these vertices are then replaced by references to the newly created variables (Figures 3-2(d)–(f)).

The trees thus obtained are covered independently to arrive at a covering for the DAG. (In code-generation and technology-mapping terminology, a match refers to the association of a pattern, which corresponds to a sequence of instructions or a library gate, with a set of vertices in the DAG, and a covering refers to the selection from a set of matches to implement the functions represented by trees or DAGs.) Locally optimal code generation techniques are known for expression trees under a wide range of machines models and objectives. For example, Sethi and Ullman presented, for machines with uniform registers, an algorithm (the well-known SU-numbering algorithm) that uses the smallest number of registers to evaluate a tree [Sethi 70]. Aho
Figure 3-2 Cutting a DAG into trees at vertices with multiple fanouts. (a) Subject DAG. (b)-(f) Forest of trees resulting from cutting the DAG at vertices $t$ and $u$, each of which has two fanouts.
et al. [Aho 89] presented an algorithm based on dynamic programming that allows for complex instruction patterns. More recently, Araujo and Malik [Araujo 95] extended this scheme to handle architectures with irregular and limited register connectivity.

Independent covering of trees, however, may result in a suboptimal solution for the original DAG, because tree covering inherently precludes the use of complex instructions in cases where internal vertices are shared. An alternative to tree covering is to tackle the DAG directly and formulate the DAG-covering problem as a \textit{binate covering problem} [Rudell 89], a special case of integer linear programming, and solve the problem exactly or heuristically using branch-and-bound methods. (Appendix A gives a brief review of the set covering and binate covering problems.)

In the following sections we present a two-phase algorithm for code generation. The first phase, described in Section 3.3, is a basic formulation of code selection based on binate covering that is similar to technology mapping in logic synthesis. This phase ignores data transfer costs between vertices in the DAG, and is used to first obtain a preliminary code selection where patterns that match more than one vertex in the given subject binary DAG are selected. (The intermediate form uses machine-independent, primitive operators, which are in general unary or binary.) Unlike the heuristic formulation with trees, a good heuristic procedure for solving the binate covering problem is likely to elude the difficulties faced by trees (i.e., complex instructions with fanouts from internal vertices).

After the first phase, the covered binary DAG is transformed into another DAG of which each vertex now corresponds to a machine operator that may take more than two operands. The second phase of instruction selection determines the locations of the operands and the destination of each vertex, taking into account data transfer costs. Of particular interest is the case of one-register machines, for which an optimal solution can be derived from that of the binate covering problem for the second phase. This will be discussed in detail in Section 3.4. We then propose extensions of this method to treat the mode optimization problem and multiple-register machines in Sections 3.5.1 and 3.5.2.
3.2 Motivating Example

Figure 3-3 shows a simplified model of the data-path of Texas Instruments' popular TMS320C25 architecture [TI 93] for fixed-point digital signal processing. The TMS320-C25 is an accumulator-based machine. In addition to the usual arithmetic–logic unit (ALU), there is a separate multiplier which takes inputs from the T register and the memory, and places the result in the P register. Note that there are no general-purpose registers; most computations involve the accumulator and another operand from the memory.

An important feature in this architecture and in many other DSP architectures is that certain instructions assume their operands to be in specific locations (registers or the memory) and deposit their results in specific registers. For example, the MPY instruction assumes that the multiplier and multiplicand come from the memory and the T register, and writes the result into the P register. Another example is the ADDT instruction, which adds an operand from the memory, shifted by the amount specified in the T register, to the accumulator. This is in contrast to RISC architectures, in which most operations involve general-purpose registers and these registers are usually interchangeable.

It is also not unusual to find complex instructions in DSPs. Typical examples include add-with-shift (e.g., TMS320C25 ADD and ADDT) and multiply-add (e.g., DSP56000 MAC [Motorola 90]). Utilizing these instructions is essential to generating compact and efficient code. The conventional heuristic of breaking up a DAG into trees prohibits the use of these complex instructions in the case where internal vertices are shared. To see this, consider the subject DAG and patterns shown in Figure 3-4 (page 52). Conventional tree-covering will first break up the DAG at vertex $v_1$, thereby prohibiting the use of pattern (d). Figure 3-5(a) shows the resulting tree-cover, consisting of matches $m_1$, $m_2$, and $m_3$. On the other hand, if we attempt to cover the DAG without first breaking it up into trees, then we may use pattern (d) to match vertices $v_2-v_1$ and $v_3-v_1$ (matches $m_4$ and $m_5$ shown in Figure 3-5(b)). With
3.2 MOTIVATING EXAMPLE

Figure 3-3  A simplified model of the TMS320C25 data-path. The TMS320C25 is an accumulator-based machine. In addition to the usual ALU, there is a separate multiplier which takes input from the T register and the memory, and places the result in the P register. There is a set of eight auxiliary registers used to address the data memory. The address generation unit (AGU) is used for address arithmetic. However, there is no general-purpose register file.
Figure 3-4  Example subject DAG and patterns. (a) Subject DAG. (b) Pattern for multiply. (c) Pattern for add. (d) Pattern for multiply-add.

Figure 3-5  Two coverings of the subject DAG. (a) Covering using only the multiply and add patterns. (b) Better covering with the multiply-add pattern.
this selection of matches, we do not need to implement functionality of \( v_1 \), because it is not explicitly used by any match. Given the costs of the patterns shown in Figure 3-4(b)–(d), this cover has a lower cost than that in Figure 3-5(a).

### 3.3 Discovering Complex Instructions

#### 3.3.1 Basic Formulation

The formulation in this section assumes that the target machine is such that data transfers between two registers or between a register and the memory have zero cost. The purpose of the first phase of DAG covering is to discover complex patterns on the subject DAG, and the formulation is reminiscent of the binate covering formulation for technology mapping in [Rudell 89].

Given a set of patterns that correspond to machine instructions, a subject DAG corresponding to a basic block or a trace is to be covered using these patterns. Each pattern has an associated cost that reflects the cost of the corresponding instruction or instructions. The DAG-covering problem is to select a set of matches with minimum cost to implement the primary output functions represented by the DAG.

There are three steps associated with DAG covering.

1. All matches of the patterns in the subject DAG are generated.

2. A binate covering problem is created that expresses the conditions leading to a legal cover.

3. A cover with minimum cost is obtained by solving the binate covering problem either exactly or heuristically.

Step 1 is a relatively straightforward pattern matching step. A Boolean variable \( m_i \) corresponds to each successful match of a pattern in the subject DAG \( D(V,E) \). Let the vertices in the subject DAG be \( v_j, 1 \leq j \leq |V| \). Each vertex \( v_j \in V \) can be covered by a set of matches \( m_{j1}, m_{j2}, ..., m_{jp_j} \), where \( p_j \) is the number of patterns that
can be used to cover $v_j$. For example, $v_2$ can be covered by one of two matches: either $m_2$ or $m_4$. Note that a match is said to cover only the vertex matched by the root vertex of the pattern, not other vertices of the pattern. For example, $m_4$ covers $v_2$, but not $v_1$. All matches $m_1$ through $m_5$ for the example subject DAG are marked in Figure 3-5.

Step 2 generates the binate covering problem with variables $m_i$ and two sets of disjunctive clauses:

- The first set consists of clauses of the form

$$m_{j1} + m_{j2} + \cdots + m_{jp_j} \quad (3.1)$$

for every internal vertex $v_j$ that fans out to root (primary output) vertices. This set of clauses represents the different ways that any particular vertex $v_j \in V$ can be covered using different matches. For the subject DAG of Figure 3-4(a) the covering matrix is shown in Figure 3-6. The first row of the matrix, $(m_2 + m_4)$, corresponds to vertex $v_2$ and indicates that $v_2$ needs to be covered, either by match $m_2$ or match $m_4$, as shown in Figure 3-5. Similarly, the next row, $(m_3 + m_5)$, indicates that either $m_3$ or $m_5$ needs to be selected to cover vertex $v_3$.

Note that the first set of clauses cover only those vertices that fan out to root vertices (in this example, $v_2$ and $v_3$), because the selection of a particular match will necessitate the selection of matches that cover vertices connected to its inputs. This is described by the second set of clauses that immediately follows.

- Matches are allowed to have vertices internal to the match feed vertices not in the match. For instance, in Figure 3-5 $v_1$ of match $m_4$ feeds $v_3$, which is not in the match.

Thus, for each match $m_i$, we must ensure that all the nonleaf inputs to the match are implemented, i.e., covered by some other match. By nonleaf inputs
we mean internal vertices in the DAG (in contrast to primary inputs) that serve as inputs to other vertices. For example, if we choose \( m_2 \) to cover \( v_2 \), we need to select a match to cover \( v_1 \), a nonleaf input to \( m_2 \).

Let the nonleaf inputs to match \( m_i \) be \( s_{i1}, s_{i2}, ..., s_{iT_i} \). For each \( i_k \), let \( W_{ik} \) be the set of matches that cover \( s_{ik} \). \( W_{ik} \) can be viewed as a disjunctive expression over the Boolean variables corresponding to the matches. Thus, selecting match \( m_i \) implies that we have to satisfy each of the \( W_{ik} \)‘s. We therefore write the clause

\[
m_i \Rightarrow W_{ik}, \quad 1 \leq k \leq T_i
\]  

(3.2)

which translates to the clauses

\[
\bar{m}_i + W_{i1} \\
\bar{m}_i + W_{i2} \\
\vdots \\
\bar{m}_i + W_{iT_i}
\]

Each match \( m_i \) generates \( T_i \) additional clauses if it has \( T_i \) nonleaf inputs.

In the covering matrix of Figure 3-6, the second set of rows corresponds to these additional clauses. For match \( m_2 \), we have to implement the nonleaf vertex \( v_1 \) as the output of some other match. This can be done using match \( m_1 \) alone; hence, we generate the clause \((\bar{m}_2 + m_1)\) corresponding to the third
row. Similarly, the fourth row is generated for match \( m_3 \), which if selected would require the selection of \( m_1 \) as well. If vertex \( v_1 \) could be covered by another match, say \( m_6 \), then these two rows would become \((\overline{m}_2 + m_1 + m_6)\) and \((\overline{m}_3 + m_1 + m_6)\).

The cost of a match \( \text{cost}(m_i) \) is simply the cost of its associated pattern. In Step 3, we select a set of columns from the covering matrix such that the cumulative cost of the columns is minimum, and such that all the disjunctive clauses are satisfied. In our example, we will find ourselves selecting \( m_4 \) and \( m_5 \) with a minimum total cost of \( 1 + 1 = 2 \); this corresponds to the covering of Figure 3-5(b). It is easily verified that selecting \( m_4 \) and \( m_5 \) satisfies all the clauses of Figure 3-6. Note that tree covering methods would not be able to discover the optimal solution of Figure 3-5(b) since the subject DAG would have been first broken up into three trees, which when covered independently would result in the covering of Figure 3-5(a) that has a cost of \( 1 + 1 + 1 = 3 \).

3.3.2 Transforming the Subject DAG

After complex patterns are selected, a new DAG is derived in which vertices now correspond to available operations in the target machine. We create a vertex for every match selected by the binate covering solver, and duplicate each edge that is incident to a vertex belonging to two or more matches. For example, after selecting the matches \( m_4 \) and \( m_5 \), we create two multiply-add vertices, as shown in Figure 3-7, for these matches. The edges \( e_5 \) and \( e_6 \) are duplicated so that each multiply-add vertex has its own copies (\( e_{5a} \) and \( e_{6a} \) for one, and \( e_{5b} \) and \( e_{6b} \) for the other). After transforming the subject DAG, we proceed to solve the problem of data transfers.

3.4 Data Transfers for One-Register Machines

To gain an understanding of how scheduling is coupled with code selection and register usage, let us first focus on one-register machines, or accumulator-based
architectures. Many fixed-point DSPs are in essence accumulator-based, with some architecture-specific idiosyncrasies. The one-register machine model therefore provides a good approximation for these architectures. In a one-register machine, accumulator spills to and reloads from the memory may account for a large fraction of the instructions. We must take this cost into account in order to find an optimal instruction selection.

The major complication in modeling accumulator spills and reloads is that the spilling of values depends on the chosen instruction schedule [Liao 95a] [Liao 95c]. However, since our instruction selection is not yet complete we do not know the schedule. Hence, we have to both choose the instructions and determine a (partial) schedule of these instructions at the same time. In this section we will develop a theory of code generation for one-register machines based on a compact binate covering formulation. This theory takes the impacts of scheduling into account by associating a Boolean variable with each edge in the subject DAG to indicate whether the source and destination vertices of the edge are placed adjacent in the final schedule. A set of clauses consisting of these adjacency variables implicitly enumerates all possible schedules, and a second set of clauses relates schedules to accumulator
spills and reloads. By solving the associated binary covering problem we arrive at an optimal solution for the original code generation problem.

### 3.4.1 Previous Work

In [Aho 77] Aho et al. presented optimal code generation algorithms (on DAGs) for two different models of one-register machines:

- Noncommutative machines, in which available operations are:

  - `acc ← op acc` (unary operator)
  - `acc ← acc op mem` (binary operator)
  - `acc ← mem` (reload)
  - `mem ← acc` (spill)

  where `acc` denotes the accumulator and `mem` denotes an operand residing in the memory.

- Commutative machines, in which available operations are:

  - `acc ← op acc` (unary operator)
  - `acc ← acc op mem` (memory-right binary operator)
  - `acc ← mem op acc` (memory-left binary operator)
  - `acc ← mem` (reload)
  - `mem ← acc` (spill)

We find these two models inadequate for the following reasons. First, in our application the given DAG may have ternary or higher-arity operators depending on the complex patterns chosen in the first step of binary covering (Section 3.3). Second, the noncommutative model of [Aho 77] does not take into account the commutativity of certain operators. It always requires the left operand of a binary operator in the accumulator. For example, in evaluating the expression \( (b + c) \), the value of \( b \) must be first loaded to the accumulator if it is not already there, and then added with \( c \); but not vice versa. However, if \( b \) and \( c \) are themselves values of expressions rather
than primary inputs, the accumulator may already contain \( c \) immediately before the evaluation of \( b + c \). Since addition is commutative, adding the accumulator with \( b \) is perfectly acceptable.

The commutative model, on the other hand, assumes that the first operand of any binary operation, whether commutative or not, may come from the memory as well as from the accumulator. However, we find that in most accumulator-based machines, noncommutative operations usually require the first operand to be in the accumulator and the second in the memory. Although sometimes it is possible to use alternative instructions to avoid the exchange of operands, it may be more expensive to do so. For example, suppose we wish to compute \( b - c \) and it happens that \( c \) is in the accumulator. Without having to exchange the operands, we might use the \textit{negate} operator, if it is available, to negate the contents of the accumulator (i.e., \( c \)) and then add \( b \) to it. But this would require two instructions instead of one, while the machine model assumes that every operation has a cost of one. Thus this model is not sufficiently expressive to take into account the availability and costs of commutative forms of certain operators.

We believe the best way to handle commutativity is to treat each operation independently, using a \textit{separate pattern} for each of the commutative forms of the operations wherever necessary, rather than assuming commutativity in the machine model. In addition, each pattern is allowed to have a different cost. A subset of matches will then be selected to minimize the total cost of operations, spills, and reloads.

We now proceed to present a new theory of code generation for the noncommutative one-register machine, based on a compact binary covering formulation. This theory takes into account the commutativity of each operator individually, instead of assuming commutativity in the machine model. The operators can be binary, ternary, or higher-arity operators. For the purpose of exposition we will concentrate on binary operators, although the techniques are readily generalized for operators of higher arity.
3.4.2 Assumptions

We will assume that for binary operators the address of the operand in the memory may be directly specified. If memory locations have to be addressed indirectly via address registers, and the instruction-set architecture does not have a register-plus-offset addressing mode, then a separate optimization pass, known as offset assignment, is carried out after instruction selection; this will be the topic of Chapter 4. In addition, we will assume that upon entrance into a basic block the accumulator contains no useful value, so that the value of a primary input needs to be loaded if it is to be used as an operand in the accumulator. We will also assume that algebraic identities of expressions other than the commutativity of individual operators are not exploited, and that it is more expensive to recompute a common subexpression than storing the value of the common subexpression into the memory and using it later. Subject DAGs are assumed to be connected in the sense that the undirected graph derived from the DAG by disregarding the directions of the edges is connected.

3.4.3 Definitions

Standard graph terminology will be used throughout this chapter. In addition, for a directed edge $e$ we will denote by $\text{src}(e)$ and $\text{dst}(e)$ the source vertex and the destination vertex of $e$. Until Section 3.4.11, when we speak of the subject DAG we mean the expression DAG (derived from Section 3.3.2) without the primary inputs, the primary outputs, and the edges emanating from or incident to these vertices; these vertices and edges will be treated specially in Section 3.4.11.

Definition 3.1 Let $H(V,E)$ be a directed graph. A u-cycle in $H$ is a subset of $V \cup E$ that would form a cycle if the edges were considered undirected. If $H$ contains a u-cycle, it is said to be u-cyclic; otherwise, it is u-acyclic.

We use the terms d-cycle, d-cyclic, and d-acyclic for the case where the directions of the edges are considered. For instance, the DAG in Figure 3-2(a) (page 48) is
Figure 3-8 Simple and composite u-cycles. U-cycles $C_1$ and $C_2$ are simple, but $C_3$ is composite because vertices B and C belong to $C_3$ but edge $(C, B)$ does not.

u-acyclic. (Recall that we are disregarding roots and leaves of the DAG.) If, on the other hand, the primary input $g$ were replaced by an internal vertex, then the DAG would become u-cyclic.

Definition 3.2 A u-cycle $C$ in a directed graph $H$ is said to be simple if the following holds: for every pair of vertices $u$ and $v$ of $C$, if there exists an edge $e$ in $H$ between $u$ and $v$, then $e$ is also an edge in $C$. Otherwise, the u-cycle is said to be composite.

Figure 3-8 shows some examples of simple and composite u-cycles. U-cycles $C_1$ and $C_2$ are simple. However, $C_3$ is composite, because the edge $(C, B)$ connects two vertices of $C_3$ but does not belong to the u-cycle.

Definition 3.3 Let a subject DAG $D(V, E)$ be given. A worm $w$ in $D$ is a subset of $V \cup E$ forming a directed path, possibly of zero length, such that the vertices in the path will appear consecutively in the schedule [Aho 77].

By consecutively we do not exclude the possibility of data transfers (i.e., spills and reloads) between the two vertices, which represent computations.
**Definition 3.4** Let $w$ be a directed path (worm). Every vertex of $w$ other than the first and the last is called an interior vertex (with respect to the worm).

**Definition 3.5** A worm-partition of $D$ is a set of disjoint worms [Aho 77].

An edge is said to be selected with respect to a worm-partition if it belongs to some worm in the partition. We can visualize a worm-partition by associating with it a directed graph $G$, which we call a worm-graph.

**Definition 3.6** Let $W$ be a worm-partition of a DAG $D(V,E)$. The worm-graph of $W$ is a directed multigraph $G(W,F)$, where

$$F = \{ \langle w,x,e \rangle \mid w,x \in W, e \in E, \text{src}(e) \in w, \text{dst}(e) \in x \}.$$ 

Recall that a multigraph is a graph in which multiple edges are permitted between any two vertices of the graph. Therefore, we use triples to represent edges of the worm-graph. The third element of a triple uniquely identifies an edge of the worm-graph in case two distinct edges of $F$ have the same source $w$ and destination $x$. Intuitively, each vertex of $G$ corresponds to a worm in $D$, and there is an edge between vertices $w$ and $x$ of $G$ whenever there is an edge in $D$ between some vertex of worm $w$ and some vertex of worm $x$. Given a worm-partition, we can derive $G$ from $D$ by successively imploding the selected edges (i.e., merging the vertices that are connected by selected edges). Henceforth we shall denote by $D$ the subject DAG, and by $G$ the induced worm-graph of a worm-partition of $D$. We may also speak of a worm-partition and its worm-graph interchangeably.

**Definition 3.7** A worm-partition is said to be legal if a valid schedule can be derived from $G$ such that the vertices of each worm appear consecutively in the schedule.

Figures 3-9 through 3-11 illustrate the notion of worms and worm-partitions, and their relation to scheduling. The vertices of the worm-graphs are shaded. In Figure 3-9, the selected edges are $e_2$, $e_4$, $e_5$, $e_6$, and $e_8$ and the worms are B, ADEH,
Figure 3-9 Worms and worm-graphs. (a) A DAG $D$ with a selection of worms. (b) Corresponding worm-graph $G_1$, in which each vertex represents a worm in $D$, and a schedule based on this worm-partition. For simplicity, the edges in $G_1$ are labeled with the corresponding edges of $D$ (instead of triples). Note that every topological sort of the worm-graph yields a valid schedule.
Figure 3-10  Worms and worm-graphs.  (a) DAG $D$ with a different selection of worms.  (b) Corresponding worm-graph $G_2$ and a schedule based on this worm-partition.
Figure 3-11 Worms and worm-graphs. (a) DAG $D$ with yet another selection of worms. (b) Corresponding worm-graph $G_3$. This worm-selection has no valid schedule because it contains a nontrivial d-cycle.
FG, and CI. The edges $e_1$, $e_3$, $e_7$, and $e_9$ connect vertices belonging to different worms; hence, they have corresponding edges in $G_1$. A schedule is derived by scheduling the vertices of the worm-graph $G_1$ and then expanding the worms back into vertices of $D$. Figure 3-10 shows the same DAG $D$ with a different selection of worms, its worm-graph $G_2$, and a schedule derived from $G_2$. Note that in each schedule the vertices of each worm are placed consecutively, and that schedules are not unique for a given worm-graph. In Figure 3-11, an illegal worm-partition is shown. This partition gives rise to a nontrivial $d$-cycle in $G_3$, and no schedule exists that places the vertices in each worm consecutively.

It is readily observed that a sufficient condition for a worm-partition to be legal is that $G$ be $d$-acyclic [Aho 77]. This is because a $d$-acyclic DAG has a topological sort, and every topological sort of $G$ gives a schedule for $D$. This condition, however, is not always necessary. In particular, self-loops in $G$ are allowed (see Theorem 3.4 on page 75).

3.4.4 Binate Covering Formulation

The binate covering problem for data transfers in one-register machines consists of two sets of Boolean variables. The first is the set of adjacency variables. An adjacency variable is associated with each edge $e_i$. For the sake of notational convenience, we will let $e_i$ also denote the adjacency variable associated with the edge. The variable $e_i$ takes the value of 1 if edge $e_i$ is selected, and 0 otherwise.

The second set is composed of the variables for matches, spills, and reloads. For example, Figure 3-12 shows a subject DAG and two patterns. Variables $m_1$ and $m_2$ are for the matches of the two patterns at vertex $v$ in the subject DAG. Similarly, match variables are created for the matches on every other vertex. We will denote by $\text{spill}(v)$ the spill variable for vertex $v$ and by $\text{reload}(e)$ the reload variable for edge $e$. If $\text{spill}(v)$ is set to 1, the result of vertex $v$ needs to be spilled into the memory; and if $\text{reload}(e)$ is set to 1, the value transmitted by edge $e$ needs to be reloaded from the memory before the operation that uses $e$ is executed.
The adjacency variables have a cost of 0, because they do not correspond to any emitted code. They are used to describe the set of all valid schedules, and to relate the schedule with instruction selection and with necessary spills and reloads. Each of the variables for matches, spills, and reloads has a cost equal to that of the corresponding instruction or instructions. We will now show how to construct the binate covering problem for one-register machines.

### 3.4.5 Fundamental Adjacency Clauses

Because the selection of an edge $e$ indicates that $\text{src}(e)$ and $\text{dst}(e)$ will be placed adjacently in the schedule, the following constraints, called fundamental adjacency constraints or fundamental constraints, must be satisfied:

**Fundamental adjacency constraints.** If a vertex has multiple fanouts, then at most one of the fanout edges may be selected. If a vertex has multiple fanins, then at most one the fanin edges may be selected.

Clearly the fundamental constraints are necessary for a worm-partition to be legal in any DAG. (Simply stated, in any schedule each vertex may have at most one
immediate predecessor and at most one immediate successor.) The clauses for the fundamental constraints are, therefore, for each vertex \( n \),

\[
\overline{e_i} + \overline{e_j}
\]  

(3.3)

for every pair of fanout edges \( e_i \) and \( e_j \) of \( n \), and for every pair of fanin edges \( e_i \) and \( e_j \) of \( n \). Thus, for example, if vertex \( v \) has three fanout edges \( e_1, e_2, \) and \( e_3 \), then we need to write three clauses (one for each pair):

\[
\overline{e_1} + \overline{e_2}
\]

\[
\overline{e_1} + \overline{e_3}
\]

\[
\overline{e_2} + \overline{e_3}
\]

The following theorem shows that these fundamental clauses are sufficient for u-acyclic DAGs; i.e., every worm-partition that satisfies the fundamental constraints is legal.

**Theorem 3.1** If the subject DAG \( D(V,E) \) is u-acyclic, then the fundamental clauses are sufficient. In other words, if a worm-partition satisfies the fundamental clauses, then it is a legal worm-partition.

**Proof** — Consider the process of deriving the worm-graph \( G \) from \( D \). Suppose \( D \) is u-acyclic. If we disregard the directions of \( E \), then all properties of (undirected) trees apply to \( D \) as well. In particular, selecting an edge \( e \) and merging \( \text{src}(e) \) and \( \text{dst}(e) \) reduces both \( |E| \) and \( |V| \) by one, whereby the property \( |E| = |V| - 1 \) still holds. By Theorem 5.2 of [Cormen 90, page 91], the resulting DAG remains u-acyclic. Therefore, by repeatedly imploding the selected edges of the worm-partition, no u-cycle, much less a d-cycle, will appear in \( G \). The worm-partition is hence legal.

If there are u-cycles in \( D \), however, then the fundamental clauses become insufficient. The most common instance is one of reconvergent paths (Figure 3-13). Two or more paths are said to be reconvergent if they have the same initial vertex and the same final vertex. Selecting all edges of either reconvergent path will result in
Figure 3-13 Reconvergent paths may lead to d-cycles in the worm-graph. (a) Two reconvergent paths in a DAG: ABCD and AED; all edges of the first path are selected. (b) A d-cycle in G due to the selection.

Figure 3-14 Interleaved sharing may also lead to d-cycles in the worm-graph. (a) Three vertices sharing inputs from three other vertices. Worms are selected that satisfy the fundamental clauses. (b) A d-cycle in G due to the selection of the worms.
a d-cycle in the worm-graph. Another example is shown in Figure 3-14 in which a
set of vertices share as children another set of vertices. Here, the edges ⟨A, D⟩, ⟨C, D⟩,
⟨C, F⟩, ⟨B, F⟩, ⟨B, E⟩, and ⟨A, E⟩ form a u-cycle in Figure 3-14(a). The selected edges
lead to the d-cycle shown in Figure 3-14(b). Although in both cases the worms se-
lected satisfy the fundamental constraints, these selections must be forbidden because
they create cyclic dependencies in the worm-graphs.

Note, on the other hand, that selecting an edge that is not part of any u-cycle in
D will not create a d-cycle in G. Thus we only need to focus on writing additional
clauses for u-cycles. In the following section we derive necessary and sufficient
conditions, in the presence of u-cycles, for a worm-partition to be legal.

3.4.6 Clauses for U-Cycles

Since u-cycles in D may lead to d-cycles in G, we need to add clauses to prevent
this from happening. Let C be a u-cycle of D, and arbitrarily choose a direction of
traversal on C as the forward direction, and label the edges as forward and backward
accordingly. For instance, in Figure 3-8 on page 61, the edges ⟨A, C⟩ and ⟨C, D⟩ are
forward with respect to u-cycle C₃, whereas ⟨A, B⟩ and ⟨B, D⟩ are backward.

**Theorem 3.2** If all forward edges (or all backward edges) in a u-cycle are selected, then
imploding the selected edges will result in a d-cycle. Conversely, if at least one forward edge
and at least one backward edge are not selected, then the u-cycle remains d-acyclic after
implusion of the selected edges.

**Proof — ⇒:** If all forward edges in a u-cycle are selected, then in the imploded u-
cycle only the backward edges remain. Since all the remaining edges of the imploded
u-cycle are in the same direction, the imploded u-cycle is also a d-cycle.

⇐: If at least one forward edge and at least one backward edge are not selected,
then the imploded u-cycle has at least two edges pointing to opposite directions;
hence, the imploded u-cycle remains d-acyclic.

■
For example, consider the DAG in Figure 3-13(a) (page 69). With respect to the u-cycle ABCDE, the forward edges are \( \langle A, B \rangle, \langle B, C \rangle, \) and \( \langle C, D \rangle \); and the backward edges are \( \langle A, E \rangle \) and \( \langle E, D \rangle \). The worm selected consists of all the forward edges, and hence a d-cycle results (Figure 3-13(b)). Similarly in Figure 3-14(a), with respect to the u-cycle AEBFCD, the forward edges are \( \langle A, E \rangle, \langle B, F \rangle, \) and \( \langle C, D \rangle \); the backward edges \( \langle A, D \rangle, \langle B, E \rangle, \) and \( \langle C, F \rangle \). The worms selected consist of all the backward edges, and again we see a d-cycle in the worm-graph (Figure 3-14(b)).

By virtue of Theorem 3.2, therefore, the following condition is necessary for a worm-partition to be legal:

**U-cycle constraints.** For each u-cycle in \( D \), edges of the same orientation may not all be selected.

Is it possible that, even if the selected edges satisfy this condition for every u-cycle, there is still a d-cycle in \( G \)? In other words, is it possible that a d-cycle in \( G \) arises from another cause than u-cycles in \( D \)? The following theorem shows that this is impossible, thereby establishing the sufficiency of the u-cycle constraints.

**Theorem 3.3** If \( G \) is d-cyclic, then there exists a u-cycle in \( D \) of which all the forward edges or all the backward edges are selected. Hence, if the u-cycle constraints derived from Theorem 3.2 are satisfied for every u-cycle in \( D \), then \( G \) is d-acyclic.

**Proof** — Let \( w_1, w_2, ..., w_k \) be the vertices of a d-cycle in \( G \), which also denote the corresponding worms in \( D \). By the definition of \( G \), there exist vertices \( v_1 \in w_1 \) and \( u_2 \in w_2 \) such that there is an edge \( e_1 = \langle v_1, u_2 \rangle \) between them. Similarly, there exist vertices \( v_i \in w_i \) and \( u_{i+1} \in w_{i+1} \), and edges \( e_i = \langle v_i, u_{i+1} \rangle \) for \( i = 2, ..., k - 1 \); and \( v_k \in w_k \), \( u_1 \in w_1 \), and \( e_k = \langle v_k, u_1 \rangle \) (Figure 3-15). Since \( v_i \) and \( u_i \) are vertices of the same worm, there is a path between them (in one direction or the other). Denote by \( P_i \) the path between \( v_i \) and \( u_i \), and designate the direction of the \( e_i \)'s as the forward direction. Now \([P_1, e_1, P_2, e_2, ..., P_k, e_k] \) form a u-cycle in \( D \). Furthermore, every backward edge of this u-cycle belong to one of the \( P_i \)'s. Because all edges of the \( P_i \)'s are selected edges,
Figure 3-15  D-cycles in the worm-graph arise solely from u-cycles in the subject DAG. If there is a d-cycle in the worm-graph $G$, then we can find a u-cycle in $D$ of which all the forward edges or all the backward edges are selected.
we conclude that all backward edges in this u-cycle are selected. This selection of
dges violates the u-cycle constraints for the u-cycle; therefore, the u-cycle constraints
are sufficient.

We can compactly write clauses to require that at least one forward edge and
one backward edge be selected, as follows. Let $f_1, f_2, ..., f_k$ be the adjacency variables
for the forward edges of a u-cycle in $D$, and $b_1, b_2, ..., b_l$ for the backward edges. The
clauses

$$ f_1 + f_2 + \cdots + f_k $$

$$ b_1 + b_2 + \cdots + b_l $$

will ensure that not all of the forward edges and not all of the backward edges are
selected. Otherwise, one of these clauses will evaluate to false. Hence, two clauses for
each u-cycle suffice. No new variables are introduced into the formulation, merely
additional clauses.

3.4.7 Self-Loops

One important exception needs to be made regarding self-loops in $G$ (which was not
addressed in [Aho 77]). Consider the fragment of a DAG shown in Figure 3-16, and
the selected edges $e_1$, $e_2$, $e_3$, and $e_4$. This selection contains all forward edges of u-
cycles ABCD and BCDE and gives rise to d-cycles (the edges $e_5$ and $e_6$ in Figure 3-16(b))
in the worm-graph $G$, in accordance with Theorem 3.2. Thus this selection appears
to violate the u-cycle clauses of Section 3.4.6.

A closer examination reveals, however, that a valid schedule can be constructed
that satisfies the property that vertices of the worm appear consecutively: simply
ABCDE. What allows for this schedule is the fact that the induced d-cycles in $G$ are
self-loops. In the following lemma and theorem, we will state this condition formally
and prove its necessity and sufficiency.
Figure 3-16  Self-loops are excepted from the u-cycle clauses. (a) Fragment of a DAG $D$ with a selected worm. (b) The induced self-loops in $G$. This worm would violate the u-cycle clauses. However, it is still a legal selection: the schedule ABCDE satisfies the property that vertices of the worm appear consecutively in the schedule.
Definition 3.8 An edge is called a reconvergent edge if the edge by itself constitutes a reconvergent path.

Lemma 3.1 Let \( l = (w,w,e) \) be an edge of \( G \) forming a self-loop. The corresponding \( u \)-cycle in \( D \) must consist of two reconvergent paths, and the edge \( e \) is a reconvergent edge and one of the two reconvergent paths.

Proof — The loop-edge \( l \) corresponds to a single edge \( e \) from some vertex \( u \) of the worm to some other vertex \( v \) of the same worm. In addition, there is a path \( P \) from \( u \) to \( v \) since \( u \) and \( v \) are vertices in the same worm. With respect to \( D \), \( u \) must be a (transitive) predecessor of \( v \), because otherwise \( D \) would not be \( d \)-acyclic. Thus we have the two reconvergent paths: the edge \( (u,v) \) and the path \( P \).

Theorem 3.4 If all \( d \)-cycles of a worm-partition \( G \) are self-loops, then \( G \) is legal. Conversely, if a \( d \)-cycle of \( G \) contains more than one vertex, then \( G \) is illegal.

Proof — Self-loops arise solely from the kind of reconvergent paths described in Lemma 3.1, with the longer path being part of a worm. Let \( u \) and \( v \) be the first and last vertices of such a pair of reconvergent paths as in Lemma 3.1. When we schedule the vertices of the worm consecutively, we will encounter \( u \) before \( v \), whereby the precedence relation required by the edge \( (u,v) \) is not violated. On the other hand, if there are two or more vertices in a \( d \)-cycle of \( G \), the attempt to schedule the vertices of one worm consecutively will be unsuccessful, because some vertex of the current worm depends on some vertex of another worm, which in turn depends on the current worm.

In light of Theorem 3.4, Clauses (3.4) and (3.5) are not required for self-loops. Instead, a clause consisting of a single variable requiring the reconvergent edge \( (u,v) \) not to be selected is prescribed—clearly, choosing the edge \( (u,v) \) would lead to a nontrivial \( d \)-cycle in \( G \), and this must be prevented. If, on the other hand, both reconvergent paths are single edges (e.g., when an operator takes both of its
Figure 3-17 Each u-cycle, simple or composite, needs to be taken into account individually. The worms shown in the figure satisfy the clauses for u-cycles C₁ and C₂, but they create a d-cycle in the worm-graph, since edges (G, D), (D, A), and (A, B) are the forward edges of C₃.

operands from the same vertex), then neither the u-cycle clauses nor the self-loop clause is necessary. The fundamental clauses (3.3) described in Section 3.4.5 ensure that at most one of these edges is selected.

3.4.8 Simple and Composite U-Cycles

We might reasonably conjecture that if the u-cycle constraints are satisfied for all simple u-cycles, then they would be satisfied for all composite u-cycles as well. Unfortunately, this is not the case. Figure 3-17 shows a counter-example.

Consider the worms chosen in Figure 3-17. Clearly the fundamental constraints are satisfied since the worms are disjoint. This worm-partition is legal with respect to either C₁ or C₂—in each case there is at least one unselected forward edge and one unselected backward edge. However, with respect to C₃, all of the forward edges
\( \langle G, D \rangle, \langle D, A \rangle, \) and \( \langle A, B \rangle \) are selected. Thus this selection of worms creates a d-cycle in the worm-graph.

Therefore, given a DAG we will have to find all u-cycles in it and prescribe clauses for each one. The set of cycles in an undirected graph may be viewed as a vector space over the “addition” operator \( \oplus \), defined as follows: for two cycles \( C_1(V_1, E_1) \) and \( C_2(V_2, E_2) \),

\[
C_1 \oplus C_2 = \langle V_1 \cup V_2, (E_1 \cup E_2) - (E_1 \cap E_2) \rangle.
\]

For instance, in Figure 3-17, \( C_3 = C_1 \oplus C_2 \). The cycle space of a graph \( G(V, E) \) has dimension \( \gamma = (|E| - |V| + 1) \), and given any basis (called a cycle basis) for this space, every cycle may be expressed as a sum of cycles in this basis [van Leeuwen 90]. A cycle basis can be generated in \( O(|V| \cdot (|E| - |V| + 1)) \) time using depth-first search [Paton 69].

Constructing the set of all u-cycles in a DAG may potentially involve the enumeration of all \( (2^{|V|} - 1) \) combinations of cycles in the basis. However, in our context the sum of two u-cycles may not always be two u-cycles. In particular, if two u-cycles do not share any common edge, then their sum is not a u-cycle. Therefore, we may partition the set of cycles in the basis into edge-disjoint subsets, and exhaustively enumerate combinations only within each subset. Since the connectivity of a typical program is low, the dimension of the cycle space is also small and this method is useful in practice. Theoretically, it is an interesting open question whether there exists a set of clauses which is equivalent to the set of u-cycle clauses and which can be constructed in polynomial-time regardless of the connectivity of the graph.

### 3.4.9 Clauses for Matches

Each vertex in the DAG needs to be implemented by some pattern. There may be a set of alternatives for commutative operations. Figure 3-18(a) shows a fragment of a DAG, and Figure 3-18(b) and (c) show two patterns matching at vertex \( v \); the two matches are denoted \( m_1 \) and \( m_2 \), either of which may be used to implement vertex
Figure 3-18 Matches, spills, and reloads. Vertex $v$ may be matched by either $m_1$ or $m_2$. Depending on which match is chosen, and depending on the selection of $e_1$ and of $e_2$, different spills and reloads are required. For example, if match $m_1$ is selected, then the value of vertex $t$ needs to be in the accumulator before $v$ is scheduled. Hence, if $e_1$ is selected also, $v$ will be scheduled immediately after $t$ and no spill or reload is necessary. On the other hand, if $e_1$ is not selected, before $v$ is scheduled the accumulator will have a value other than that of $t$; therefore, $t$ needs to be spilled and its value $e_1$ needs to be reloaded before $v$ is scheduled.

$v$. Hence, for each vertex we write a disjunctive clause consisting of all the match variables for the vertex. For the example of Figure 3-18, we would write:

$$m_1 + m_2$$  \hspace{1cm} (3.6)

to require one of these matches to be selected.

3.4.10 Clauses for Reloads and Spills

The main purpose of introducing an adjacency variable for each edge in the DAG is to relate them to reloads and spills. Depending on where an operation takes its operands from and which edges are selected, different spills and reloads may be required between computations. We now describe precisely how to write clauses to activate spills and reloads.

Consider the edge $(t,v)$ in Figure 3-18(a), whose corresponding adjacency variable is $e_1$. There are four cases to examine for this edge:
1. Match $m_1$ is used and $e_1 = 1$. Since $m_1$ requires its left operand from the accumulator, and $v$ will be scheduled immediately after $t$, no spill on $t$ or reload on the edge $e_1$ is necessary.

2. Match $m_1$ is used and $e_1 = 0$. In this case, a spill on $t$ is required, because a vertex other than $v$ immediately follows $t$ and destroys the contents of the accumulator, but this value is needed by $v$ later. Also, a reload is necessary immediately before $v$ is scheduled, because $m_1$ takes its left-operand from the accumulator.

3. Match $m_2$ is used and $e_1 = 1$. Even though $v$ immediately follows $t$, a spill is still required because $m_2$ takes its left-operand from the memory. No reload on $e_1$ is necessary.

4. Match $m_2$ is used and $e_1 = 0$. As in the previous case, only a spill is required.

Recall that $\text{spill}(v)$ denotes the transfer of the value of $v$ from the accumulator to the memory immediately after $v$ is computed, and $\text{reload}(e)$ denotes the reload of the value of $\text{src}(e)$ from the memory to the accumulator immediately before $\text{dst}(e)$ is scheduled. We can describe the above conditions by the following expressions:

$$m_1 \cdot \overline{e_1} \Rightarrow \text{spill}(t)$$  \hspace{1cm} (3.7)

$$m_1 \cdot \overline{e_1} \Rightarrow \text{reload}(e_1)$$  \hspace{1cm} (3.8)

$$m_2 \Rightarrow \text{spill}(t)$$  \hspace{1cm} (3.9)

which are conveniently rewritten in disjunctive form:

$$\overline{m_1} + e_1 + \text{spill}(t)$$  \hspace{1cm} (3.10)

$$\overline{m_1} + e_1 + \text{reload}(e_1)$$  \hspace{1cm} (3.11)

$$\overline{m_2} + \text{spill}(t).$$  \hspace{1cm} (3.12)

Likewise, for $e_2$, we write the following clauses:
\[ m_2 + e_2 + \text{spill}(u) \]  \hspace{1cm} (3.13)

\[ m_2 + e_2 + \text{reload}(e_2) \]  \hspace{1cm} (3.14)

\[ m_1 + \text{spill}(u). \]  \hspace{1cm} (3.15)

Similar clauses are prescribed for every other vertex with its incoming edges and all possible matches on it.

Scheduling spills and reloads is trivial for one-register machines. Because we know that the accumulator will be destroyed in the next operation, we should spill as soon as possible, namely immediately after the current operation. Reloads, on the other hand, should be scheduled as late as possible.

### 3.4.11 Leaves and Roots of the DAG

Unlike internal vertices whose values are first computed into the accumulator and then stored into the memory only if necessary, the value of a leaf must be loaded to the accumulator if an operator needs its value there. Also, a root vertex denotes a store into the memory location which its name designates, rather than a computation. Therefore, the edges that emanate from leaves and those that are incident to roots are not treated in the same way as internal edges. These edges will not be considered part of worms, and hence the fundamental adjacency clauses and the u-cycle clauses are not applicable to them. In other words, when we write the fundamental clauses and the u-cycle clauses we disregard the existence of these edges.

Instead, we will write clauses that take into consideration the loading of variables into the accumulator and the storing of the contents of the accumulator into memory locations:

1. For every edge \((u, v)\) such that \(v\) is a primary output, we require that the result of \(u\) be stored into the memory location of \(v\). Thus we add a clause consisting of only the spill variable \(\text{spill}(u)\). (This is equivalent to setting \(\text{spill}(u)\) to 1. For
3.4 DATA TRANSFERS FOR ONE-REGISTER MACHINES

Figure 3-19 Primary inputs require special treatment. (a) Subject DAG. (b) (c) Patterns matching at v. If $m_1$ is selected, the loading of a into the accumulator is required. If $m_2$ is selected, the value of a will be accessed from the memory.

Consistency of exposition, we will describe these in terms of clauses. Clauses of this type can be easily be eliminated in the reduction (by essentiality) step of the covering algorithm; see Appendix A). The only difference is that, instead of spilling into a temporary variable created during code selection, we spill this value to a variable that is live on exit.

2. For every edge $(u, v)$ such that $u$ is a primary input, we need to load the value of $u$ into the accumulator if the selected match on $v$ so requires. For instance, consider the DAG fragment shown in Figure 3-19(a), a variation of Figure 3-18(a). If $m_1$ is selected, then we will need to load the value of a into the accumulator before scheduling $v$. If $m_2$ is selected instead, then no load is necessary; the value of a will be accessed from the memory. Therefore, the clause

$$\overline{m_1} + \text{reload}(e_1)$$  \hfill (3.16)

suffices in this case. If both inputs of $v$ were to be taken from leaf vertices, we would add:

$$\overline{m_2} + \text{reload}(e_2).$$  \hfill (3.17)
3.4.12 Summary of the Binate Covering Formulation

We briefly summarize the clauses that constitute an instance of the binate covering problem for code generation:

1. Clauses describing the set of all legal worm-partitions.
   
   (a) Fundamental adjacency clauses (Clauses (3.3), Section 3.4.5).
   
   (b) U-cycle clauses with the exception of self-loops (Clauses (3.4) and (3.5), Section 3.4.6).
   
   (c) Clauses for reconvergent edges (Section 3.4.7).

2. Clauses for instruction selection.
   
   (a) Clauses requiring the implementation of a vertex using a match from a set of alternatives (Section 3.4.9).
   
   (b) Clauses relating instruction selection and scheduling to spills and reloads (Section 3.4.10).
   
   (c) Clauses for primary inputs and primary outputs (Section 3.4.11).

The binate covering formulation has an important property that is conducive to computational efficiency:

If the subgraph (of the subject DAG) rooted at vertex \( v \) is a tree, then the clauses related to this subgraph are independent of clauses for other parts of the subject DAG.

By independent we mean that these clauses by themselves constitute a (smaller) binate covering problem that can be solved independently of the larger problem without losing optimality. If we exploit this property when solving the binate covering problem, the time complexity in practice is exponential only in the number of edges that are coupled via the fundamental clauses or via the u-cycle clauses. (The complexity of covering problems is exponential in the number of variables and polynomial in the number of clauses.)
3.4.13 Optimality of the Binate Covering Formulation

We conclude the presented theory with a proof that an optimal solution to the binate covering problem derived in Sections 3.4.2–3.4.11 yields optimal solutions for the code generation problem for one-register machines. Recall that a worm-partition only determines a partial schedule. We will show that any total schedule derived from the partial schedule has the same optimal cost.

Given a solution of the binate covering problem, the evaluation of a vertex \( v \) consists of the following steps:

1. Reloading, if necessary, the fanin edge of \( v \) that requires the corresponding operand to be in the accumulator.

2. Computing \( v \) using the instructions associated with the selected pattern.

3. Spilling \( v \) into the memory if necessary.

We define the cost of evaluating a vertex \( v \) to be the cost of performing these steps. We will show that the necessity of reloading and spilling for a given vertex \( v \) is only determined by the worm-partition but not the total ordering of worms of the worm-partition in the final schedule.

**Theorem 3.5** Let \( G \) be the worm-partition constructed from an optimal solution of the binate covering problem for one-register machines. Then every total schedule derived from \( G \) is optimal.

**Proof** — Let \( w \) be a worm of \( G \). We first note that the cost of evaluating an interior vertex \( v \) of \( w \) cannot be affected by where the worm is located. This is because in any valid schedule \( S \) derived from \( G \), the immediate predecessor of \( v \) and the immediate successor of \( v \) remain unchanged—they are two other vertices of the same worm. Therefore, before the evaluation of \( v \), the state of the machine that concerns \( v \) (i.e., contents of the accumulator and the operands of \( v \)) is the same for any \( S \), and the spill/reload requirements (if any) do not change, either.
The only vertices that may possibly be affected are the first and the last vertices of the worm, which we denote by first(w) and last(w). By definition, the fanin edges of first(w) are either edges from primary inputs or unselected edges. Hence, evaluating first(w) always requires a reload. Suppose there exists a schedule S such that this reload is actually redundant, and let y be the immediate predecessor worm of w in S. This means the output of last(y) may be used immediately by first(w). We are then permitted to select the edge \( \langle \text{last}(y), \text{first}(w) \rangle \), without violating the fundamental or u-cycle constraints: the concatenation \( yw \) becomes a new legal worm because by construction \( w \) is scheduled immediately after \( y \). Thus we have found a different legal worm-partition that has a lower cost than \( G \); this contradicts the assumption that \( G \) is an optimal worm-partition. A similar argument applies to last(w).

Since the fundamental and u-cycle clauses implicitly enumerate all legal worm-partitions, which in turn implicitly encompass all schedules of \( D \) [Aho 77], every total schedule derived from \( G \) is optimal.

3.5 Extensions of the Binate Covering Formulation

We now consider two extensions of the binate covering formulation for data transfers presented in Section 3.4:

- Mode optimization problem.
- Data transfers in machines with multiple register classes.

These problems will be discussed in Sections 3.5.1 and 3.5.2.

3.5.1 Mode Optimization

In some processors, such as the TMS320 family, certain instructions are controlled by mode variables (or residual control in microprogramming terminology). Two simple examples are: the sign-extension mode variable, which affect arithmetic operations involving the accumulator, and the product-shift mode variable that controls the
number of bits by which the contents of the P register should be shifted before being transferred to the accumulator (see Figure 3-3 on page 51). It is a common technique to use mode registers to increase code density by reducing the number of bits required to encode instructions.

Since mode variables assert control beyond that encoded in an instruction, they must be first set to the correct values if the current values are not as desired for the next instruction to be executed. In [Liao 95a] the authors presented the mode optimization problem and a strategy to minimize the number of changes in mode settings and accumulator spills. In this section we propose a method to incorporate the mode optimization in the binate covering formulation.

The optimality theorem (Theorem 3.5) applies only if the instructions required to evaluate a vertex \( v \) consists solely of reloading, computation, and spilling. If instructions are controlled in part by mode variables, then the scheduling of the worm-graph does have an effect on the total cost of evaluation. For instance, consider the worm-graph shown in Figure 3-20(a) (page 86), and two different schedules given in Figures 3-20(b) and (c). Here we have a single mode variable with two mode values \( s \) and \( u \). Each worm \( w \) in Figure 3-20(a) is labeled with the beginning and ending mode values of that worm, i.e., the mode values required by \( \text{first}(w) \) and \( \text{last}(w) \). An asterisk inserted between two worms indicate that a mode change is required. For example, if \( w_2 \) is scheduled immediately after \( w_1 \), as in Figure 3-20(b), then we need to set the mode variable from \( s \) to \( u \). The schedule of Figure 3-20(b) requires three mode changes (in addition to intra-worm changes), whereas that of Figure 3-20(c) requires only one. Hence, when we construct a total schedule from the partial schedule given by the worm-graph in the presence of mode variables, we need to take this cost into account.

The branch-and-bound algorithm described in [Liao 95a] is readily applied, here on the worm-graph rather than the subject DAG. Because the size of the worm-graph (in terms of number of vertices) is substantially smaller than the underlying subject DAG, it takes relatively little time in practice to schedule the worms.
Figure 3-20 Scheduling a worm-graph in the presence of mode variables. (a) A worm-graph labeled with beginning and ending mode values. (b) A schedule requiring three mode changes (in addition to intra-worm changes). (c) A different schedule requiring only one mode change.
Figure 3-21 Taking into account the cost of mode-switching during the selection of worms. Edge $e_2$ connects two vertices requiring different mode values; therefore, it is more costly to select $e_2$. We write the clause $(\overline{e_2} + \text{mode}(e_2))$ to signify the required mode change.

We have thus far assumed that a worm-partition is determined by the binate-covering solver, and attempted to schedule the worms of the worm-partition. We may further refine this procedure by incorporating the mode-switching costs in the binate covering formulation itself, thereby biasing the solver towards choosing worms that have fewer mode changes within. This is accomplished by the addition of clauses for edges connecting two vertices that require different mode values. For example, consider the scenario of Figure 3-21. Other things being equal, it is preferable to select $e_1$ rather than $e_2$ because the latter carries a penalty of mode switching. Hence we add the clause:

$$\overline{e_2} + \text{mode}(e_2),$$

where $\text{mode}(e_2)$ is the Boolean variable denoting the insertion of the required mode-changing instruction.

3.5.2 Data Transfers for Multiple Register Classes

Data transfers in machines with multiple register classes are substantially more difficult to model than one-register machines. The main reason that our binate covering formulation yields optimal solutions for one-register machines is that the life-times of the accumulator and of the memory are easily estimated. The accumulator has a life-
Figure 3-22 Example DAG involving the P register and the accumulator. Vertex $t$ does not have to be spilled to the memory even though only one of its fanout edges may be selected.

time of one operation, since its contents are overwritten in every operation. Memory locations, on the other hand, have life-times of infinity. By virtue of these properties, the adjacency of operations in a given worm-partition completely determines whether and where spills and reloads are required. In contrast, in machines with multiple register classes, the life-time of each register is difficult to estimate—it may be anywhere between one and infinity.

As indicated in Section 3.2, in some DSP architectures instructions take operands from specific locations and deposit their results into specific registers. In this section we will use the $[1,\infty]$ model, as in [Araujo 95], in which every resource class (register or memory) is assumed to have either one or infinitely many elements. For those register classes that have an infinite number of elements, a separate allocation phase (e.g., using a graph-coloring register allocator [Chaitin 81]) is carried out afterwards.

As an example, consider the TMS320C25 architecture (see Figure 3-3 on page 51), which has three registers in the main data-path: the accumulator, the P register, and the T register. In the TMS320C25, only the multiply instruction writes to the P register;
3.5 EXTENSIONS OF THE BINATE COVERING FORMULATION

every other instruction writes to the accumulator. In addition, there are two versions of *add*: `ADD`, which adds the contents of a memory location to the accumulator, and `APAC`, which adds the contents of the P register to the accumulator. Therefore, to evaluate the DAG in Figure 3-22, for instance, we carry out the following operations:

1. Multiply $b$ and $c$; the product is in the P register.

2. Use `APAC` to add $a$ and the product; store the sum to $e$.

3. Use `APAC` to add $d$ and the product; store the sum to $f$.

Note that, from the perspective of worms, only the edge $e_1$ is selected. However, the fact that $e_2$ is not selected does not imply that the *multiply* needs to be spilled. Instead, the results of the *multiply* is allowed to remain in the P register after the evaluation of vertex $u$. Thus, in this case the clauses that require spills are too pessimistic. In other cases, however, the clauses may be too optimistic in not taking into account certain necessary data transfers.

Our current strategy for handling this problem is to use the binate covering problem to generate a partial schedule, derive a complete schedule from the partial schedule, and then remove the redundant data transfers and insert the necessary ones. These data transfers result from the inexactness of the binate covering formulation in the case of multiple register classes. To minimize the extent of this inexactness, we may write additional clauses that depend on the specific target architecture. For example, for the TMS320C25, we may write clauses to prevent some worms from being selected that are apparently legal by our definition but in reality leads to impossible code that needs to be mended later. To see this, consider the DAG shown in Figure 3-23(a) (page 90). The symbol `preg` in the patterns of Figure 3-23(b) denotes P register. Let $m_1$ be the match between vertex $s$ and pattern $p_3$, and $m_2$ be the match between vertex $v$ and pattern $p_4$. If the edges $e_3$ and $e_4$ are selected as a worm are part of a worm, and both $m_1$ and $m_2$ are used, then it is impossible to schedule the vertices $s$, $t$, and $v$ consecutively. This is because values represented vertices $r$ and
Figure 3-23 Selecting a worm, though legal, may lead to impossible code selection. (a) Subject DAG. (b) Patterns that can be used to implement add vertices. If we let \( v \) take its left input from the P register and \( s \) take its right input from the P register, and select the edges \( e_3 \) and \( e_4 \), then it is impossible to schedule \( s \), \( t \), and \( v \) consecutively because \( u \) and \( r \) have to be simultaneously alive in the P register.
$u$, which are both inputs to this worm, are alive simultaneously under this schedule, and both are assumed (by the selection of matches $m_1$ and $m_2$) to be in the P register, a one-element register class. Thus, we write the clause:

$$m_1 + m_2 + \bar{e}_3 + \bar{e}_4$$ (3.19)

to prevent this from happening. In general, if vertices $u$ and $v$ both take one of its inputs from the P register, and there is a unique path from $u$ to $v$ such that every vertex in this path other than $u$ and $v$ do not use the P register, then we write a clause of the form of (3.19). Note that if there are two or more paths from $u$ to $v$, then these paths must have reconvergent subpaths, of which the selection of all edges of either subpath is already precluded by the $u$-cycle clauses.

Also, unlike the one-register case, not selecting an edge does not always imply a spill. For instance, if we select $e_5$ instead of $e_4$ in Figure 3-23, we do not have to spill vertex $t$ if the eventual sequence of operations is $t u v$, since the evaluation of $u$ does not destroy the accumulator. Therefore, we modify the spill clauses of Section 3.4.10 to reflect this. Only when a vertex that writes to the accumulator have two or more fanouts does it need to be spilled. Of course, in the final schedule vertex $t$ may still need to be spilled and edge $e_4$ reloaded, because we are overly aggressive in assuming otherwise in the binate covering problem.

### 3.6 Summary and Future Work

In this chapter we have presented a two-phase strategy for code generation for expression DAGs. The first phase tackle the problem of selecting complex instructions, and the second phase solves the problem of scheduling and data transfers.

The problem formulation for the first phase is similar to the DAG-covering formulation for technology mapping in [Rudell 89]. For the second phase we have presented a new theory of code generation for one-register machines. This new theory, also based on the binate covering problem, is more general than that of Aho et al.
[Aho 77] and encompasses a wider class of one-register machines. Although we use the same notion of worms and worm-partitions, our exposition is more complete.

Our experience indicates that by tackling the code generation problem directly on the subject DAG, rather than breaking the subject DAG into a forest of trees, we may achieve code-size reductions of up to 10%, although for a typical DAG the improvements are not as good, only about 1–3% on the average, since many expression DAGs are quite loosely connected and the amount of sharing is only modest. In the context of code generation for embedded systems (in which the software will become silicon), however, this relatively small degree of improvement still justifies the use of more-complex formulations and algorithms, as the cumulative effects of many "small" optimizations may prove to be substantial (Appendix B).

The main contribution of the theory of Section 3.4 is that the set of all legal worm-partitions may be described by clauses with adjacency variables. These clauses are independent of the machine. For one-register machines it is the match, spill, and reload clauses that direct the binate covering process. For machines with multiple register classes, the clauses may be more difficult to write, and in some cases the cost of a solution to the binate covering problem may not reflect the actual cost of the generated code. Therefore, future work will involve the generation of clauses that more closely mirror the actual cost of the generated code.
Chapter 4

Storage Assignment

Many architectures (such as the VAX, Motorola MC68000, Texas Instruments TMS-320C25, and most embedded controllers and digital signal processors) provide register-indirect addressing modes with auto-increment and auto-decrement arithmetic. These addressing modes allow for efficient sequential access of memory and increase code density because they subsume address arithmetic instructions and result in shorter instructions in variable-length instruction architectures.

In particular, DSPs and embedded controllers are designed under the assumption that software that runs on them would make heavy use of auto-increment and auto-decrement addressing. In some cases, DSPs and controllers have such a restricted set of addressing modes that the set does not include a mode for indexing with an offset. For example, the TMS320C25 has an 8-word auxiliary register file (Figure 3-3 on page 51) that can be used to address the data memory. However, unlike most general-purpose architectures, the encoding of the instruction set does not allow for an offset to be specified in the instruction word. The memory locations accessible are those pointed to by one of these auxiliary registers. Therefore, it is necessary to allocate one or more registers and perform address arithmetic to access variables. If an address arithmetic is either addition or subtraction by one, then we may subsume it into auto-increment or auto-decrement modes to improve both the performance the the size of the generated code.
The placement of variables in storage has a significant impact on the effectiveness of subsumption, which is in turn dependent on the patterns in which variables are accessed [Liao 95b]. Therefore, we perform the actual assignment of locations to variables after code selection, thereby increasing opportunities to use efficient auto-increment and auto-decrement modes. In other words, we first use symbolic names for memory locations during the code selection phase (e.g., Chapter 3), and then resolve these memory references into indirect addressing. We formulate this delayed storage allocation as the offset assignment problem. Although some modern DSP architectures permit increments and decrements of values other than one (e.g., Motorola DSP56000 [Motorola 90]), it is usually costly to use this feature if the modifier value is frequently changing. This is because extra instructions are required to set the modifier value, which is typically stored in a register rather than encoded in the instruction word. (This feature is intended for traversing arrays with strides greater than one.) Therefore, we will focus on unit increments and decrements.

We will first consider a simpler problem which we call simple offset assignment (SOA). A solution to the SOA problem assigns optimal frame-relative offsets to variables of a procedure, assuming that the target machine has a single indexing register with only the indirect, auto-increment, and auto-decrement addressing modes. We begin by optimally solving the simple offset assignment problem for a basic block, and then propose a method to treat entire procedures. To this end, we represent a basic block by the sequence of variables in the order they are accessed in the basic block. We then summarize this sequence by an undirected graph (called the access graph) with weighted edges, and show that the SOA problem is equivalent to a graph covering problem, called the maximum weight path cover (MWPC) problem. By solving the MWPC problem we can obtain a solution to the SOA problem.

Bartley was the first to address the SOA problem and presented an approach based on finding a Hamiltonian path of maximum weight on the graph [Bartley 92]. However, several aspects of his formulation and implementation can be improved: the inefficiency of his algorithm arises from the reduction of SOA to the weighted
Hamiltonian path problem, and from the underlying representation of the problem. He considered complete graphs which usually contains much information unnecessary for the construction of an optimal solution to the original assignment problem. Also, his procedures for selecting an edge of the Hamiltonian path and detecting whether a cycle is created by a selection are inefficient, of \( O(|V|) \). As a result, his algorithm runs in \( O(|V|^3 + |L|) \) time, where \(|V|\) is the number of variables and \(|L|\) is the number of variable accesses.

In this chapter we provide a more formal treatment of the offset assignment problem. We show that the SOA problem is equivalent to a path covering problem of the access graph, and that the decision problem for SOA is NP-complete. We then present an \( O(|E| \log |E| + |L|) \) algorithm that produces empirically near-optimal solutions, where \(|E|\) is the number of edges in the access graph. Our extensive experimental results on larger examples (Section 4.5) indicate that access graphs are generally quite sparse and, therefore, our algorithm has a significant advantage over Bartley's. There are several similarities between the two approaches—both are based on access graphs and both use a greedy strategy in selecting edges in the graph. However, instead of considering complete graphs, we only retain edges that have nonzero weights. This allows for an \( O(1) \) procedure for testing whether selecting an edge causes a cycle, which is essential to reducing the overall complexity of the heuristic. To evaluate the effectiveness of our algorithm, we have also designed and implemented a branch-and-bound procedure. Based on certain properties of the problem, we develop a simple pruning condition that proves to be very efficient. Experimental results show that the simple heuristic achieves optimal or near-optimal solutions most of the time.

To model more realistic architectures, we also extend the SOA problem to the general offset assignment problem (GOA). In GOA, more than one address register may be used to address the variables. This problem is substantially more difficult than SOA due to the numerous ways variables can be accessed by different registers at different times. We present a simpler formulation of the problem and show how the algorithms
for SOA can be used to efficiently solve GOA. Since the SOA heuristic is used a core procedure for GOA, the reduction in complexity from \(O(|V|^3)\) to \(O(|E| \log |E|)\) is significant. Although we emphasize code size, our formulation of the storage assignment problem also lends itself naturally to application-specific performance optimization in the presence of trace information from actual applications.

4.1 Processor Model and Notations

For the purpose of exposition, we use a simple processor model that reflects the addressing capabilities of most DSPs. The model is an accumulator-based machine. Each binary operation involves the accumulator and another operand from the memory. Memory accesses can occur only indirectly via a set of address registers, \(AR0\) through \(AR(k-1)\). Furthermore, if an instruction uses \(AR_i\) for indirect addressing, then in the same instruction \(AR_i\) can be optionally post-incremented or post-decremented by one at no extra cost. If an address register does not point to the desired location, it may be changed by adding or subtracting a constant, via the instructions \(ADAR\) and \(SBAR\). Also, we use the \(LDAR\) instruction to initialize an address register. Since \(LDAR\) involves the address of a variable, its cost is typically higher than either \(ADAR\) or \(SBAR\). Thus, if the contents of an address register is known, \(ADAR\) and \(SBAR\) are preferred.

We use \(* (AR_i)\), \(* (AR_i) +\), and \(* (AR_i) -\) to denote indirect addressing through \(AR_i\), indirect addressing with post-increment, and indirect addressing with post-decrement, respectively. For instance, the instruction \(ADD * (AR0) +\) adds to the accumulator the contents of the memory location pointed to by \(AR0\) and post-increments \(AR0\).

4.2 Simple Offset Assignment

In this section we assume that only one address register, \(AR0\), is used to address all variables. We describe the optimization problem corresponding to assigning offsets
to variables in a frame in order to obtain the most compact and efficient code. This implies that we have to minimize the number of instructions whose sole function is setting AR0 to point to appropriate locations in the frame.

### 4.2.1 Example

As an example illustrating how storage assignment affects the size of the code, consider the C program in Figure 4-1(a) (page 98). Assume that the offset assignment to the various variables is as shown in Figure 4-1(b), which is based on first use. The assembly code for the C program is shown in Figure 4-1(c). The register transfers shown to the right of the assembly instruction sequence describe the effects of the instructions: the first column shows the effects in the main execution unit involving the accumulator and the variables, and the second column shows how the value of AR0 changes throughout the course of the basic block. The instructions SBAR and ADAR are used to change AR0 to point to the frame location accessed in the next instruction, if it is not already pointing to the desired location.

The first instruction LDAR AR0, &c initializes the address register AR0 to the address of variable c. The value of the variable c is then loaded into the accumulator, and AR0 is auto-incremented after the first LOAD instruction. At this time AR0 is now pointing to variable d. Since d is the next operand to be accessed, the next instruction, ADD, may use AR0 to address it immediately, without having to first change it. The auto-increment associated with this instruction changes AR0 to pointing to f, which is the operand for the next ADD instruction. So far the variables accessed are laid out in the frame in the order in which they are accessed, and auto-increment and auto-decrement may be used. Before we come to the STOR instruction which writes the result back to c, we need to use an explicit SBAR AR0, 2 instruction to set AR0 to point to c, because the address of f and that of c differ by two and auto-decrement cannot be used along with the previous ADD instruction. Similarly, for every other pair of accesses that does not refer to variables placed adjacently in the frame, either an ADAR or a SBAR instruction must be used. In total, ten such instructions are required.
c = c + d + f;

a = h - c;
b = b + e;
c = g - b;
a = a - c;

(a)

(b)

LDAR AR0, &c
LOAD *(AR0)+ acc ← c
ADD *(AR0)+ acc ← acc + d
ADD *(AR0) acc ← acc + f
SBAR AR0, 2
STOR *(AR0) c ← acc
ADAR AR0, 3
LOAD *(AR0) acc ← h
SBAR AR0, 3
SUB *(AR0) acc ← acc - c
ADAR AR0, 4
STOR *(AR0)+ a ← acc
LOAD *(AR0)+ acc ← b
ADD *(AR0)- acc ← acc + e
STOR *(AR0) b ← acc
ADAR AR0, 2
LOAD *(AR0) acc ← g
SBAR AR0, 2
SUB *(AR0) acc ← acc - b
SBAR AR0, 5
STOR *(AR0) c ← acc
ADAR AR0, 4
LOAD *(AR0) acc ← a
SBAR AR0, 4
SUB *(AR0) acc ← acc - c
ADAR AR0, 4
STOR *(AR0) a ← acc

(c)

Figure 4-1 Example illustrating effect of offset assignment on code size. (a) C code sequence. (b) Offset assignment based on order of first use. (c) Assembly code based on this assignment.
to execute the code of Figure 4-1(a), given the offset assignment of Figure 4-1(b). These instructions are highlighted in the assembly code in Figure 4-1(c).

Now consider the offset assignment of Figure 4-2(b) (page 100) for the same C code. As before, we use the instruction LDAR AR0, &a to initialize the address register AR0. This assignment gives rise to a shorter assembly code sequence (Figure 4-2(c)). Only four address-arithmetic instructions are required to execute the code of Figure 4-2(a).

In the following sections we will show how to obtain the offset assignment that minimizes the number of address-arithmetic instructions.

4.2.2 Assumptions in SOA

The simple offset assignment (SOA) problem consists of assigning a frame-relative offset to each of the local variables to minimize the number of address-arithmetic instructions required to execute a basic block. Since the objective is to minimize the number of address-arithmetic instructions, we define the cost of an assignment to be the number of such instructions. (When using a single address register, the number of LDAR instructions is a constant and, therefore, may be ignored. For multiple address registers, some LDAR instructions will be needed for every additional address register introduced; this cost is included in the setup cost described in Section 4.3.)

For the SOA problem we will make the following assumptions:

1. Every data object has a size of one word.
2. A single address register is used to address all variables in the basic block.
3. One-to-one mapping of variables to locations.
4. The basic block has a fixed evaluation order (schedule).
5. Special features such as address wraparound (e.g., modulo addressing) are not exploited.

We will consider the use of multiple address registers in Section 4.3.
c = c + d + f;
a = h - c;
b = b + e;
c = g - b;
a = a - c;

(a)

(c)

Figure 4-2 A better assignment leads to smaller code size. (a) C code sequence. (b) Better offset assignment. (c) Assembly code based on this assignment.
4.2.3 Approach to the Problem

Our approach to solving the SOA problem is to formulate it as a well-defined combinatorial problem of graph covering, called maximum weight path covering (MWPC). From a basic block we derive a graph, called an access graph, that summarizes the relative benefits of assigning each pair of variables to adjacent locations. By solving the MWPC problem, we can construct an assignment with minimum cost. We then show how to reduce an instance of the Hamiltonian path problem into an instance of SOA, demonstrating that a polynomial-time solution is unlikely to exist (unless, of course, P = NP). We will then present a heuristic algorithm and a branch-and-bound procedure for solving this problem.

4.2.4 Access Sequence and Access Graph

Given a code sequence C that represents a basic block, we can uniquely define an access sequence for the block. Given an operation \( z = x \text{ op } y \), the access sequence is \( x y z \). The access sequence for an ordered set of operations is simply the concatenated access sequences for each operation in the appropriate order. The access sequence for the basic block of Figure 4-2(a) is shown in Figure 4-3(a).

With the notion of the access sequence, it is easily seen that the cost of an assignment is equal to the number of adjacent accesses of variables that are not assigned to adjacent locations. For instance, four address arithmetic instructions are required under the offset assignment of Figure 4-2(b), since the following two-symbol substrings of the access sequence refer to variables assigned to nonadjacent locations: a b, b c, c d, and f c. In contrast, the same access sequence requires ten address arithmetic instructions under the offset assignment of Figure 4-1(b) because of the larger number of such two-symbol substrings.

We can summarize the pattern in which variables are accessed by means of a weighted, undirected graph. The access graph \( G(V, E) \) is derived from an access sequence as follows. Each vertex \( v \in V \) in the graph corresponds to a unique
Figure 4-3 An access sequence and its access graph. Each vertex in the access graph corresponds to a variable in the access sequence. An edge with weight $w$ is placed between vertices $u$ and $v$ if $u$ and $v$ are adjacent $w$ times in the access sequence.
variable. An edge \( e = (u, v) \in E \) between vertices \( u \) and \( v \) exists with weight \( w(e) \) if variables \( u \) and \( v \) are adjacent to each other \( w(e) \) times in the access sequence. Note that it makes no difference whether \( u \) is before or after \( v \), since we may either auto-increment or auto-decrement AR0 during any load, store, or any other instruction that accesses memory via AR0. The access graph for the basic block of Figure 4-2(a) is shown in Figure 4-3(b).

Thus, in terms of the access graph, the cost of an assignment is equal to the sum of the weights of those edges that connect variables assigned to nonadjacent locations. This is illustrated in Figure 4-4(a) and (b) (page 104), which correspond to the assignments in Figure 4-1(b) and Figure 4-2(b), respectively. The dark edges indicate that the variables are assigned to adjacent locations. The weights on the light edges are exactly the costs we have to pay for address-arithmetic instructions.

Having established the relationship between access graphs and assignments, we now proceed in the other direction—finding a minimum-cost assignment by selecting edges in an access graph.

### 4.2.5 SOA and Maximum Weight Path Covering

**Definition 4.1** A path \( P \) in \( G \) is an alternating sequence of vertices and edges \([v_1, e_1, v_2, e_2, \ldots, e_{m-1}, v_m]\) where \( e_i = (v_i, v_{i+1}) \in E \), and no \( v_i \) appears more than once in the sequence.

**Definition 4.2** Two paths are said to be disjoint if they do not share any vertices.

**Definition 4.3** A disjoint path cover (henceforth cover) of a weighted graph \( G(V, E) \) is a subgraph \( C(V, E') \) of \( G \) such that:

- For every vertex \( v \) in \( C \), \( \deg(v) \leq 2 \);

- There are no cycles in \( C \).

Note that the edges in \( C \) form a set of disjoint paths (some of which may contain no edges), hence the name.
Figure 4-4  Access graphs in which dark edges indicate that two variables are assigned to adjacent locations. (a) Graph corresponding to the first assignment. (b) Graph corresponding to the second assignment.
4.2 SIMPLE OFFSET ASSIGNMENT

Definition 4.4 The weight of a cover C is the sum of the weights of all edges of C. The cost of a cover C is the sum of the weights of all edges in G but not in C:

\[ \text{cost}(C) = \sum_{e \in G, e \notin C} w(e). \]  \hspace{1cm} (4.1)

Definition 4.5 An offset assignment A is said to be implied by a cover C(V, E') if edge \( \langle u, v \rangle \in E' \) implies variables u and v are adjacent in A.

Definition 4.6 (Maximum Weight Path Covering) Given an access graph G, find a cover C with maximum weight.

A cover with maximum weight is equivalent to one with minimum cost. We now show that solving the MWPC problem is equivalent to solving the simple offset assignment problem.

Lemma 4.1 Given a cover C of G, the cost of every offset assignment implied by C is less than or equal to the cost of the cover.

Proof — Let A be any assignment implied by C. As seen in Section 4.2.4, the cost of the assignment is equal to the sum of the weights of all edges \( \langle u, v \rangle \) such that \( |A(u) - A(v)| > 1 \), where A(u) denotes the offset of variable u under assignment A. By Definition 4.5, these edges are a subset of edges in G but not in C. (There may well exist vertices u and v such that \( |A(u) - A(v)| = 1 \) but \( \langle u, v \rangle \) is not in C.) Thus the cost of this assignment is at most equal to that of C.

Figure 4-5 (page 106) gives an example of a cover and an implied assignment with cost less than that of the cover. The edge \( \langle b, e \rangle \) is not in the cover; but it does connect two variables assigned to adjacent locations. Thus, the cost of the cover is 6, whereas the cost of this particular implied assignment is 4. Upon comparing with the cover in Figure 4-4(b), it is evident that this cover is not optimal.

Lemma 4.2 Given any offset assignment A and an access graph G, there exists a disjoint path cover C which implies A and which has the same cost as A.
Figure 4-5  The cost of a cover may be greater than the cost of an implied assignment. (a) A disjoint path cover with a cost of 6. (b) an implied assignment with a cost of 4. The difference is due to the edge (b, e): this edge is not selected but the two variables b and e are assigned adjacent locations.
4.2 SIMPLE OFFSET ASSIGNMENT

Proof — Given an assignment $A$, we construct a disjoint path cover $C$ as follows: for each pair of vertices $u$ and $v$ such that $|A(u) - A(v)| = 1$, we pick the edge $(u, v)$, if it exists in $G$, to be included in $C$. $C$ is a disjoint path cover because no vertex in $C$ has a degree greater than two (a variable can have at most two neighbors) and there are no cycles (we are not considering memory wrap-around). Furthermore, $C$ implies $A$ by construction. The edges in $G$ but not in $C$ are exactly those which connect two vertices with nonadjacent assignments, and thus the cost of $C$ is exactly equal to that of $A$.

Theorem 4.1 Every offset assignment implied by an optimal disjoint path cover is optimal.

Proof — Let $C$ be an optimal disjoint path cover with cost $c$. Suppose there is an assignment (not necessarily implied by $C$) with cost $c' < c$. Since an offset assignment implies the existence of a disjoint path cover with the same cost (Lemma 4.2), there is a disjoint path cover with cost $c'$ which is less than $c$. This contradicts our assumption that $C$ is an optimal cover. Hence, no assignment has a cost strictly less than $c$, and all assignments implied by $C$ have cost $c$ (Lemma 4.1).

Theorem 4.1 allows us to arrive at an optimal simple offset assignment by solving the corresponding maximum weight path covering problem. Intuitively, an edge denotes the number of times two variables are accessed immediately one after another and hence the number of address arithmetic instructions necessary if these two variables are not assigned to adjacent locations. Therefore, by selecting a cover with the maximum weight we minimize the number of address arithmetic instructions required.

Consider again Figure 4-4(b). The dark edges form an optimal disjoint path cover for the access graph. To construct an assignment from this cover, we simply traverse each path from one end to the other. By doing so we obtain the offset assignment shown in Figure 4-2(b). It makes no difference in which order we traverse the paths, since Theorem 4.1 guarantees that every assignment implied by the cover is optimal.
HAMPATH \leq_p SOA \leq_p MWPC

Figure 4-6 Relationship between HAMPATH, SOA, and MWPC, in the proof of the NP-completeness of SOA. The symbol \leq_p denotes polynomial-time reducibility.

4.2.6 Complexity Analysis

Our approach to solving the SOA problem is in essence to reduce it to the MWPC problem, solve the latter, and then construct a solution for the former. It is trivial to prove that the MWPC problem is NP-complete. This, however, does not mean that SOA is itself an NP-complete problem, since we might have reduced a problem in complexity class P to one in class NPC (the class of NP-complete problems). Just as one had to prove that the register allocation problem is as hard as the coloring problem to which is usually reduced [Chaitin 81], we need to show that the SOA problem is indeed an NP-hard problem. We will do so by constructing an access sequence from an instance of the (unweighted) Hamiltonian path problem (HAMPATH), such that optimally solving the offset assignment problem on the access sequence will yield a decision to the Hamiltonian path problem. The relationship (of reduction) among the three problems is illustrated in Figure 4-6.

We first show that, given an undirected graph, it is possible to construct an access sequence of length equal to twice the number of edges in the graph (whence the reduction is of polynomial time). We then prove that solving a decision problem for SOA on this access sequence yields a decision for the Hamiltonian path problem on the original graph.

**Lemma 4.3** Given an undirected, connected graph \( G(V, E) \), there exists an access sequence such that the corresponding access graph \( G' \) is isomorphic to \( G \) and each edge of \( G' \) has a weight of 2. Furthermore, this sequence can be constructed in \( O(|E|) \) time.
4.2 SIMPLE OFFSET ASSIGNMENT

Figure 4-7 Using depth-first search to construct an access sequence from an undirected graph. A depth-first search beginning with vertex a yields a sequence in which two variables $u$ and $v$ twice appear adjacent to each other if the edge $(u,v)$ exists in the graph.

Proof — Select any vertex $r$ in $G$ as the root vertex, and perform a depth-first search on $G$. During the depth-first search each edge $(u,v)$ is traversed exactly twice, once forward and once backward. Consider the sequence $T$ in which the vertices are visited (including backtracks). This sequence is an access sequence that gives rise to an access graph that is isomorphic to $G$. (An example is shown in Figure 4-7.) In addition, since each edge $(u,v)$ is traversed twice, vertices $u$ and $v$ are adjacent to each other in $T$ exactly twice as well. Depth-first search takes $O(|E| + |V|)$ time [Cormen 90, page 479], which is $O(|E|)$ for connected graphs.

Theorem 4.2 Given an access sequence $T$ and an integer $k$, the problem of deciding whether there exists an assignment for $T$ with cost less than or equal to $k$ is NP-hard.

Proof — We prove this by reduction from the Hamiltonian path problem. Let $G(V,E)$ be an undirected graph. We obtain an access sequence $T$ in polynomial-time as in
Lemma 4.3, with access graph $G'$ isomorphic to $G$. Each edge of $G'$ has a weight of 2. The weight of any disjoint path cover on $G'$ is at most $2 \cdot (|V| - 1)$, since every edge has the same weight of 2 and a cover can have at most $(|V| - 1)$ edges. This means the cost of any cover is at least $2 \cdot (|E| - |V| + 1)$. Now let $k = 2 \cdot (|E| - |V| + 1)$ and suppose there is an assignment $A$ for $T$ whose cost is less than or equal to $k$. By Lemma 4.2 there is a cover $C$ that has the same cost as $A$. This implies that the cost of $C$ is exactly $2 \cdot (|E| - |V| + 1)$, and, in turn, that $C$ has $(|V| - 1)$ edges. On the other hand, if $C$ has $(|V| - 1)$ edges, it must be a Hamiltonian path.

Conversely, if there does not exist an assignment $A$ with cost less than or equal to $k$, then by Lemma 4.1 there does not exist a cover with cost equal to $k$. This means every cover has fewer than $(|V| - 1)$ edges and therefore $G$ has no Hamiltonian path.

4.2.7 A Heuristic Algorithm for SOA

Because SOA and MWPC are NP-hard, a polynomial-time algorithm for solving these problems exactly is unlikely to exist. We therefore present a heuristic algorithm for MWPC that is similar to Kruskal's maximum spanning tree algorithm [Aho 74]. The heuristic is greedy in that it repeatedly selects an edge that seems best at each iteration. The heuristic algorithm is shown in Figure 4-8.

Given an access sequence $L$, SOLVE-SOA first calls ACCESS-GRAPH to construct the access graph $G(V, E)$ from $L$, sorts the edges in $E$ in descending order of weight, and initializes $C(V', E')$ to the empty solution. It then enters the main loop (lines 7–15), in which it chooses the first edge from the remaining edges and checks if the edge would produce a cycle or would increase the degree of any vertex in $V'$ to more than two. If the edge passes the test, it is included in the solution $C$; otherwise, it is discarded. There is no backtrack in the selection of edges. After all edges are examined, SOLVE-SOA calls CONSTRUCT-ASSIGNMENT to produce the offset assignment, which simply enumerates the disjoint paths in the cover and sequentially assigns an offset to each variable encountered in this enumeration. As our experimental results
4.2 SIMPLE OFFSET ASSIGNMENT

```plaintext
SOLVE-SOA(L)
{
  /* L = access sequence for basic block */
  G(V, E) ← ACCESS-GRAPH(L);
  E_sort ← sorted list of edges in E in descending order of weight;
  C(V', E') : V' ← V, E' ← { }; 
  while ( |E'| < |V| - 1 and E_sort not empty ) {
    choose e ← first edge in E_sort;
    E_sort ← E_sort - {e};
    if ((e does not cause a cycle in C) and
        (e does not cause any vertex in V' to have degree > 2))
      add e to E';
    else
      discard e;
  }
  /* Construct an assignment from E' */
  return CONSTRUCT-ASSIGNMENT(E');
}
```

Figure 4-8 Heuristic Algorithm for SOA.
in Section 4.5 demonstrate, this heuristic often produces a solution very close to the optimal solution.

As an example illustrating the heuristic algorithm, consider again the access graph shown in Figure 4-3(b) (page 102). We first select the edge \((a, c)\), which has the highest weight, and then \((e, b)\), \((b, g)\), and \((c, h)\). Now edges \((a, b)\), \((b, c)\), \((c, d)\), and \((c, f)\) must be rejected, because the selection of any one of these edge will either produce a cycle or cause a vertex to have degree greater than two. The remaining edge is \((d, f)\), which we also select. The resulting path cover is shown in Figure 4-4(b) (page 104), which is an optimal cover for this access graph.

4.2.8 Analysis of the Heuristic Procedure

With careful implementation, we can obtain a running time of \(O(|E| \log |E| + |L|)\) for the heuristic procedure described in Section 4.2.7, where \(|E|\) is the number of edges in the access graph and \(|L|\) is the length of the access sequence. Constructing the access sequence requires \(O(|L|)\) time, and \(O(|E| \log |E|)\) is due to the need to sort the edges in descending order of weight. The main loop of the algorithm (lines 7–15) runs for \(|E|\) iterations. Therefore, if the test on lines 10–11 takes constant time, then the total time for the main loop is bounded by \(O(|E|)\).

Testing whether an edge causes a vertex to have degree greater than two is trivial: we simply keep for each vertex a counter that is incremented whenever an incident edge is selected. Testing for cycles in constant time, however, requires a little more work. We accomplish this as follows.

At each step of the main loop of the algorithm (lines 7–15), the selected edges form a set of disjoint paths. To compactly represent a path, we use a data structure called path element which contains two pointers, each pointing to one of the two end-vertices of a path. The two end-vertices, in turn, have back-pointers to this path element. This is illustrated in Figure 4-9, which shows a portion of an access graph from which edges are being selected. At this point, the edges selected are \((g, a)\), \((a, c)\), \((c, b)\), \((d, e)\), and \((e, f)\). The corresponding path elements are shown. Suppose
Figure 4-9 Using path elements to determine if an edge causes a cycle. Path elements allows for fast testing of whether an edge connects two ends of a path.
we wish to consider the selection of edge \((d, f)\). Comparing the back-pointers of these two vertices reveals that they are the end-vertices of a path. Selecting this edge would cause a cycle and this edge must therefore be discarded. Let us consider the case of selecting edge \((g, c)\). It is not necessary to know the back-pointer of vertex \(c\), because selecting edge \((g, c)\), which causes \(c\) to have more than two incident edges selected, will immediately fail the first test.

In other words, we do not have to keep a record of back-pointers for vertices that are not at the ends of a path. This leads us to the procedure for updating path elements when an edge is selected. When an edge is selected, it connects two originally disjoint paths. We arbitrarily choose the path element of the first path, and update the pointer of the path element and the back-pointer of the other end of the second path. For example, let us select edge \((b, d)\) in Figure 4-9. Selecting this edge connects the two paths \(g \rightarrow a \rightarrow c \rightarrow b\) and \(d \rightarrow e \rightarrow f\). We choose that path element for path \(g \rightarrow a \rightarrow c \rightarrow b\) as the path element for the new path \(g \rightarrow a \rightarrow c \rightarrow b \rightarrow d \rightarrow e \rightarrow f\). Updating the data structure involves changing the pointer that originally pointed to vertex \(b\) to now point to vertex \(f\), and changing the back-pointer of vertex \(f\) to point to this path element. The back-pointers of vertices \(b\) and \(f\) are now irrelevant because they will not be used any longer. The resulting structure is shown in Figure 4-10.

The main advantage of this heuristic over that of Bartley's is due to the underlying representation of the problem. Bartley cast the SOA problem into one of \emph{weighted Hamiltonian path problem}. In certain cases, however, the optimal weighted Hamiltonian path may not correspond to the optimal solution in SOA. Figure 4-11 shows an example in which the weighted Hamiltonian path formulation fails to correspond to an optimal solution for SOA. In other cases, a Hamiltonian path may not even exist.

In order to circumvent these difficulties, Bartley added edges of zero weight between vertices that are not adjacent to each other in the access sequence, resulting in a complete graph. This unnecessarily sets the number of edges to \(O(|V|^2)\). In addition, because he used an adjacency-matrix representation, selecting the valid edge with the highest weight requires \(O(|V|^2)\) time. This led to an \(O(|V|^3 + L)\) procedure.
4.2 SIMPLE OFFSET ASSIGNMENT

Figure 4-10 Data structure resulting from selection of edge \((b, d)\).

Figure 4-11 Example in which an optimal weighted Hamiltonian path does not correspond to an optimal solution of SOA. (a) An optimal weighted Hamiltonian path with weight 8. (b) A maximum weight path cover with weight 13.
Our heuristic, in contrast, only keeps edges with positive weight. In practice the number of such edges is significantly smaller than \(O(|V|^2)\) (see Table 4.1 on page 133), and, therefore, our heuristic is more efficient.

4.2.9 A Branch-and-Bound Procedure for MWPC

We now describe a branch-and-bound procedure for solving the MWPC problem, for which a simple pruning condition proves to be very efficient.

Given an access graph, a partial solution consists of a set of selected edges. With respect to each partial solution there is a set of valid edges such that selecting one of these edges and adding it to the present partial solution will neither produce a cycle nor cause any vertex in the partial solution to have a degree higher than two. At each step we keep the valid edges in decreasing order of weight. We then select the edge that has not been visited, estimate the upper bound of the new partial solution. If the upper bound is lower than the weight of the current best solution, then we terminate the search at this edge. Otherwise, we recursively call the procedure, with the new partial solution and the new upper bound as arguments.

A simple upper bound can be obtained as follows. Suppose the access graph \(G\) has \(|V|\) vertices. A maximum-weight path cover can consist of at most \((|V| - 1)\) edges, because any subgraph of \(G\) with more than \((|V| - 1)\) edges is bound to contain a cycle. Thus, given a partial solution \(P\) with \(|P|\) edges, we can sum up the weights of the \((|V| - |P| - 1)\) weightiest edges that are valid with respect to \(P\). Although it may not be a very tight bound, we have found that this bound in practice reduces the search space considerably. Figure 4-12 summarizes the branch-and-bound procedure described here. At the top level SOLVE-MWPC-B&B is called with the empty partial solution, the sorted list of all edges of the access graph, the initial upper bound that is the sum of the \((|V| - 1)\) highest-weighted edges, and the empty solution as the initial best solution.

Note that on line 9 of the algorithm we do not need to test whether the weight of solution \(P\) is greater than that of solution \(B\), because if \(E\) is empty, then \(u\) is the
Figure 4-12 Branch-and-bound procedure for MWPC.
weight of \( P \) and line 15 of the recursive call from the previous step ensures that it is greater than the weight of \( B \).

### 4.3 General Offset Assignment

The main limitation of the SOA problem described in the previous section is the use of one single address register to address all variables. If the access graph is relatively dense, then there will be many edges which cannot be selected due to the constraints of a disjoint path cover. This section considers the generalization of SOA to the case where there are \( k \) address registers, \( AR_0 \) through \( AR(k-1) \). As we will see in Section 4.5, the use of multiple registers greatly reduces the number of address arithmetic instructions.

#### 4.3.1 Example of GOA

Consider again the C code, now shown in Figure 4-13(a). Suppose we allocate a second address register, \( AR_1 \), to address the variables \( b \) and \( c \). With the offset assignment shown in Figure 4-13(b), we obtain the assembly code in Figure 4-13(c). Note that, while we need an extra instruction \texttt{LDAR AR1, &c} to initialize the second address register, we have gained overall. Assuming that the cost of \texttt{LDAR} is 2, we have further reduced the cost of evaluating this sequence by 1 (compared with Figure 4-2).

#### 4.3.2 Formulation of GOA

We begin by observing that by partitioning the set of variables into two disjoint subsets and allocating an address register to each, we effectively obtain two access sequences that are subsequences of the original sequence. This is illustrated in Figure 4-14 (page 120), which also shows the corresponding access graphs.

Based on this observation, we will make the following additional assumptions in our formulation of GOA:
\[ c = c + d + f; \]
\[ a = h - c; \]
\[ b = b + e; \]
\[ c = g - b; \]
\[ a = a - c; \]

(a) \hspace{2cm} (b)

\begin{align*}
LDAR & \hspace{1cm} AR0, \&d & AR0 & \leftarrow \&d \\
LDAR & \hspace{1cm} AR1, \&c & AR1 & \leftarrow \&c \\
LOAD & *(AR1) & acc & \leftarrow c \\
ADD & *(AR0)+ & acc & \leftarrow acc + d & AR0 & \leftarrow \&f \\
ADD & *(AR0)+ & acc & \leftarrow acc + f & AR0 & \leftarrow \&h \\
STOR & *(AR1) & c & \leftarrow acc \\
LOAD & *(AR0)+ & acc & \leftarrow h & AR0 & \leftarrow \&a \\
SUB & *(AR1)+ & acc & \leftarrow acc - c & AR1 & \leftarrow \&b \\
STOR & *(AR0)+ & a & \leftarrow acc & AR0 & \leftarrow \&e \\
LOAD & *(AR1) & acc & \leftarrow b \\
ADD & *(AR0)+ & acc & \leftarrow acc + e & AR0 & \leftarrow \&g \\
STOR & *(AR0) & b & \leftarrow acc \\
LOAD & *(AR0) & acc & \leftarrow g \\
SUB & *(AR1) & acc & \leftarrow acc - b & AR1 & \leftarrow \&c \\
STOR & *(AR1) & c & \leftarrow acc \\
SBAR & \hspace{1cm} AR0, 2 & AR0 & \leftarrow \&a \\
LOAD & *(AR0) & acc & \leftarrow a \\
SUB & *(AR1) & acc & \leftarrow acc - c \\
STOR & *(AR0) & a & \leftarrow acc
\end{align*}

(c)

Figure 4-13 Example of general offset assignment using two address registers. (a) C code sequence. (b) Using AR1 for the variables \{b, c\}. (c) Assembly code based on two address registers.
Figure 4-14 Access subsequences and derived access graphs. (a) Original access sequence. (b)(c) Access subsequence induced by the subset \{a, d, e, f, g, h\} and the corresponding derived access graph. (d)(e) Access subsequence induced by the subset \{b, c\} and the corresponding derived access graph.
1. There is a fixed cost of introducing an additional address register. This setup
cost reflects the cost associated with initialization upon entry to the procedure
and re-initialization after return from a callee.

2. Each address register is used to point to a disjoint subset of variables.

Disjointness of the subsets is not absolutely required in a more aggressive formulation.
We may contrive situations where it may be beneficial to overlap the subsets of
variables addressed by different address registers. However, allowing for overlap
unnecessarily complicates the problem, for then we will have to determine, at each point
in time, what variable each address register points to. Because an offset assignment
is not yet determined, it is indeed difficult to estimate the costs of changing the
contents of an address register. On the other hand, the element of time becomes
irrelevant under the assumption of disjointness. Although the problem still remains
difficult, we find that the heuristics often yield good results.

With the above assumptions, we will state the general offset assignment problem
as follows.

**Definition 4.7** Let $L$ be the access sequence of the basic block, and $V$ be the set of variables
in $L$. The access subsequence generated by $W \subseteq V$ is the subsequence of $L$ consisting of
variables in $W$.

**Definition 4.8 (General Offset Assignment)** Given an access sequence $L$, the set of vari-
ables $V$, and the number of address registers $k$, find a partition of $V$, $\Pi = \{P_1, P_2, ..., P_m\}$,
where $m \leq k$, such that the total cost of the optimal simple offset assignment of the
corresponding access subsequences, plus the setup costs for using $m$ registers, is minimum.

### 4.3.3 A Heuristic Algorithm for GOA

An exact solution to this problem is clearly too expensive to compute, due to the large
number of possible partitions of the set of variables. We have designed a heuristic
procedure that yields good results in practice.
Our heuristic to construct the partition is to repeatedly select and remove a subset of variables from the base set. We select the subset such that allocating an additional address register for the subset would likely contribute the greatest reduction in cost. Figure 4-15 describes the heuristic procedure for solving the general offset assignment problem. In this procedure, \textit{SUBSEQ}(L, P) denotes the access subsequence of $L$ generated by $P$.

The function \texttt{SOLVE-GOA} returns a collection of disjoint ordered sets of variables which forms a partition of the set of all variables. The order within each subset gives an offset assignment; these assignments are combined to form the final solution. Given an access sequence $L$, \texttt{SOLVE-GOA} first computes the SOA of $L$ (which we call $H$), by invoking the \texttt{SOLVE-SOA} procedure of Section 4.2.7 (or \texttt{SOLVE-MWPC-B&B} of Section 4.2.9). If there is only one address register, then the solution found for SOA is also the solution for GOA. Otherwise, \texttt{SOLVE-GOA} calls \texttt{SELECT-VARIABLES} to choose a subset of the variables in $L$ and solves SOA on the derived subsequences $L_1$ and $L_2$. If the cost of this split along with the setup cost is more expensive than that of $H$, there is no benefit in introducing the new partition block and the current solution $H$ is returned. Otherwise, it is advantageous to introduce a new address register for this subset of variables, and \texttt{SOLVE-GOA} is recursively called for the remaining variables.

The procedure \texttt{SELECT-VARIABLES} selects a subset of variables for which a new partition block may be created. It is important to note that on line 13 of the algorithm in Figure 4-15 we are making the assumption that, if allocating a new address register for the subset $L_1$ returned by \texttt{SELECT-VARIABLES} does not reduce the cost, then further partitioning will not improve either. In other words, we assume that if there is a favorable subset of variables that can reduce the overall cost, \texttt{SELECT-VARIABLES} will find that subset at the first opportunity.

To develop good heuristics for this procedure, we observe the following:

1. If an access subsequence consists of two variables, then the cost for this access subsequence is just the setup cost. No switching cost is incurred. It is also possible to select more variables (typically between two and six); provided the
4.3 GENERAL OFFSET ASSIGNMENT

```
1 SOLVE-GOA(L, k)
2 {
3     /* L = access sequence of basic block */
4     /* k = number of address registers */
5     H ← SOLVE-SOA(L);
6     if (k == 1)
7         return {H};
8     P ← SELECT-VARIABLES(L);
9     L₁ ← SUBSEQ(L, P);
10    L₂ ← SUBSEQ(L, L - P);
11    H₁ ← SOLVE-SOA(L₁);
12    H₂ ← SOLVE-SOA(L₂);
13    if (setup-cost + cost(H₁) + cost(H₂) > cost(H))
14        return {H};
15    else
16        return {H₁} U SOLVE-GOA(L₂, k - 1);
17 }
```

Figure 4-15 Heuristic Algorithm for GOA.
graph for the access subsequence is sufficiently sparse, the cost for the new partition block will be kept low.

2. If a vertex in an access graph has more than two incident edges, the associated minimum penalty for retaining the vertex in the graph is the sum of the weights on all edges except the two with the largest weights. Hence, if a variable has a high penalty, then it may be beneficial to move it to another partition block. The vertices with high penalties correspond to variables that are accessed frequently. As in traditional register allocation [Chaitin 81] [F Chow 90] where we tend to keep busy variables in fast registers, here we desire to minimize address arithmetic instructions for busy variables by allocating extra address registers to address them.

Based on these observations, a simple heuristic for SELECT-VARIABLES is to choose a small subset of variables with the largest penalty and allocate a new address register for this subset. Our first experiments were based on selecting a fixed number \( p \) of variables for every iteration. Although the computational requirement for this heuristic is small, it is not always clear what \( p \) should be. As our experimental results in Section 4.5 show, the "best" \( p \) varies among examples. We may also try more aggressive strategies by varying \( p \) between iterations, i.e., to choose a different number of variables for each call to SELECT-VARIABLES. This will require much more computation, because accurate estimation of the effect of partitioning requires several calls to SOLVE-SOA. Our initial results of GOA, however, already show encouraging improvements.

4.4 Offset Assignment for a Procedure

Section 4.2 and Section 4.3 presented solutions to the simple and general offset assignment problems for a single basic block. It is relatively straightforward to extend the formulation to take into account the presence of control-flow. Because our formulation of GOA breaks down the problem into several instances of SOA, in this section we will, for the sake of clarity, focus on using only one address register, AR.
As in the basic SOA formulation, we wish to capture the patterns in which the variables are accessed throughout the procedure by counting the number of times each pair of variables is accessed consecutively. To this end, we will need the following information:

1. The expected number of times each basic block is executed.

2. The expected number of times control flows through each edge.

This information can be computed by either profiling an actual execution, or by assigning probabilities to each control-flow edge and solving a linear system of equations. (If code size is our only objective, then we would weigh each basic block and each control-flow edge equally.)

Let \( V \) be the set of variables the address register may point to, and let \( \text{first}(n) \) and \( \text{last}(n) \) denote the first variable and last variable accessed in basic block \( n \). In addition, let \( \text{freq}(n) \) and \( \text{freq}(f) \) denote the expected execution frequency of basic block \( n \) and of control-flow edge \( f \). We begin by building the access graph for each basic block \( n \), with the edges (of the access graph) properly weighted by the execution count \( \text{freq}(n) \). These are merged to form the access graph \( G \) for the entire procedure. Then, for each control-flow edge \( f = (n, m) \) (\( m \) a successor of \( n \)), we increase the weight of the edge \( (\text{last}(n), \text{first}(m)) \) (in \( G \)) by \( \text{freq}(f) \), or create such an edge with weight \( \text{freq}(f) \) if it does not already exist.

This access graph \( G \) is then covered by using either the heuristic or the branch-and-bound procedure described in Sections 4.2.7 and 4.2.9. Once a solution is found, we will determine the contents of the address register (AR) at the exit of each basic block, and place auto-increment, auto-decrement, address-arithmetic instructions (ADAR and SBAR), and address-register initialization instructions (LDAR) at the appropriate locations so that the number of instructions is minimized. To correctly account for the cost of placing the various operations, we need to take the following points into consideration:
1. Depending on the instruction set implementation, setting an address register from an unknown value may cost more than setting it from a known value. For example, the latter may involve computing the address of a variable from its offset and the frame pointer, whereas the latter requires only an address arithmetic instruction.

2. Adding a basic block on a critical edge may incur the charge of additional unconditional jump (JMP) instructions. A critical edge in a control-flow graph is an edge that emanates from a basic block with more than one successor and leads into another basic block with more than one predecessor [Dhamdhere 92] [Knoop 95].

Figure 4-16 illustrates the insertion of a new basic block on a critical edge and its impact on code size and performance. In Figure 4-16(a), the edge \( f_{23} \) is a critical edge because its source vertex \( n_2 \) has two fanouts and its destination vertex \( n_3 \) has two fanins. When emitting code for the basic blocks, we may place the code for \( n_3 \) immediately after \( n_1 \) and make \( n_3 \) the target of the BZ (branch-on-zero) instruction at the end of \( n_2 \). No JMP instruction is necessary to transfer control from \( n_1 \) to \( n_3 \). Now consider the insertion of a new basic block \( n_5 \) on the edge \( f_{23} \); the resulting control-flow graph is shown in Figure 4-16(b). After the insertion, \( n_1 \) and \( n_5 \) each have exactly one and the same successor, namely \( n_3 \). Because we may at best place \( n_3 \) immediately after \( n_1 \) or \( n_5 \) but not both, we must append a JMP instruction at the end of one of them, presumably the one that is executed less frequently. Like all transformations based on partial redundancy elimination [Morel 79] [Knoop 95], we need to account for not only the cost of the code for basic block \( n_5 \), but also that of the additional JMP instructions required.

The decision on what value an address register should contain at the end of a basic block \( n \) affects the successors of \( n \), which are in turn affected by the decisions at their other predecessors. Therefore, we will define an equivalence relation on the
Figure 4-16 Inserting a basic block on a critical edge. Edge $f_{23}$ is a critical edge. Inserting $n_5$ on $f_{23}$ changes the control-flow graph in (a) to that in (b). Because $n_3$ can be placed immediately after either $n_1$ or $n_5$ but not both, a JMP instruction needs to be appended at the end of $n_1$ or $n_5$; in this example it is appended to the latter.
edges of a control-flow graph as the transitive closure $R^*$ of a relation $R$, defined over the control-flow edges as follows:

For control-flow edges $f_1$ and $f_2$, $f_1 \ R \ f_2$ if and only if $f_1$ and $f_2$ have the same source basic block or have the same destination basic block.

With each equivalence class $F$ we may associate a bipartite graph $(N, M, F)$, where $N$ and $M$ are the sets of source and destination basic blocks of the edges in $F$. Note that it is possible that a basic block appears in both $N$ and $M$; in this case we would duplicate the basic block so that its occurrences in $N$ and $M$ are distinct. Figure 4-17 illustrates the notion of the equivalence relation introduced previously. Parts (b)–(f) show the bipartite graphs that result from the partition.

The problem that remains to be solved here for the bipartite graph $(N, M, F)$ is to assign to each basic block $n \in N$ an attribute $\text{out}(n) \in V$, which denotes the value of the address register upon exit of basic block $n$, such that the cost of the instructions used to modify the address register is minimized. At the same time, we need to determine the best location where such instructions are placed. Since the bipartite graphs generated in a typical program are very small (at most eight vertices), we can enumerate all possible solutions and select the best one. We can reduce the search space by restricting the possible values $\text{out}(n)$ to the set:

$$\bigcup_{m \in M} \{\text{first}(m)\} \cup \bigcup_{n \in N} \text{adj}[\text{last}(n)] \quad (4.2)$$

where $\text{adj}(v)$ denotes the set of variables that are adjacent to $v$ under the given assignment as well as $v$ itself. The first union in Expr. (4.2) gives the set of variables that the basic blocks in $M$ prefer upon entry, and the second union is the set of variables to which the address register may be made to point upon exit of some basic block in $N$, via auto-increment, auto-decrement, or no change. The rationale for this restriction is that it is never beneficial to expend an address arithmetic instruction to change the address register to point to a variable that is not subsequently used.
Figure 4-17  Equivalence classes of control-flow edges. (a) A control-flow graph. (b)–(f) Equivalence classes of edges and derived bipartite graphs.
4.4.1 Example

An example will serve to clarify the problems involved in determining the best locations as well as the best combination of operations that modify the address register. In the following examples we will assume that the cost of ADAR and SBAR is 1; LDAR, 2; and JMP, 2. Note that we are disregarding the cost of *(AR), *(AR)+, and *(AR)- here, because one of these instructions must be used to access the last variable of a basic block.

Consider the bipartite graph shown in Figure 4-18(a). The values of first, last, and freq for each basic block are given, from which we infer that freq(f_{12}) = 10, freq(f_{52}) = 90, and freq(f_{56}) = 20. Figure 4-18(b) gives the offset assignment, and Figures 4-18(c)–(e) show three possible solutions. The solution in Figure 4-18(c) corresponds to the case out(n_1) = out(n_5) = c. At the end of n_1, we may use an auto-decrement since the last variable accessed was a. For n_5, however, we must use SBAR 2 to modify the address register to point to c. Now, since on all edges leading into n_2 AR points to c, we need not change anything at the beginning of n_2. On the other hand, we need an ADAR 3 at the entry of n_6, because AR points to c on edge f_{56}. The cost of this solution is 130: 110 from n_5 and 20 from n_6.

The solutions in Figures 4-18(d) and (e) correspond to the case out(n_1) = c and out(n_5) = d. In Figure 4-18(d), we insert a new basic block on edge f_{52}, a critical edge. As we have seen in the previous section, we need a JMP instruction at the end of either n_1 or the new basic block. Since freq(n_1) < freq(f_{52}), we place the JMP instruction at the end of n_1. Now, AR points to c on entry to n_2 and to d on entry to n_6; therefore, no more address arithmetic instructions are required. The cost of this solution is 110: 20 from n_1 (JMP has a cost of 2) and 90 from n_7. An alternative to Figure 4-18(d) is to set AR to &c upon entry to n_2, as shown in Figure 4-18(e). Because the two edges leading into n_2 carry different values of AR, we need to use LDAR &c instead of ADAR or SBAR. This solution has a cost of 200, due to the LDAR &c instruction in n_2. Since freq(n_2) = freq(n_1) + freq(f_{52}), and the cost of LDAR is greater than or equal to the cost of either JMP or SBAR, the solution in Figure 4-18(d)
Figure 4-18  Determining the best location and combination of operations that modify the address register. (a) A bipartite graph \((N,M,F)\), where \(N = \{n_1,n_5\}\), \(M = \{n_2,n_6\}\), and \(F = \{f_{12},f_{52},f_{56}\}\). (b) Offset assignment. (c)-(e) Solutions.
is always superior to that in Figure 4-18(e). Given different frequencies of execution and different costs for these instructions, however, the solution in Figure 4-18(e) might become preferable. For instance, if the cost of JMP is increased to 3, and \( \text{freq}(m_1) > \text{freq}(f_{52}) \), then the solution in Figure 4-18(e) has a lower cost.

4.5 Experimental Results

We have implemented the heuristic algorithms of Sections 4.2 and 4.3, and also the branch-and-bound procedure described in Section 4.2.9 in order to evaluate the heuristic SOA algorithm. All the implementations handle not only basic blocks, but entire procedures, with the formulation described in Section 4.4. Our initial goal is to minimize static code size; hence, we weigh each basic block equally.

Table 4.1 exhibits a summary of the examples we used for offset assignment. The first five examples, CHENDCT through JREV, are core routines from a JPEG-MPEG implementation. The next eight, LOADGIF through 332DITHER, are graphics routines from the xv program. Following them are procedures from the GNU gzip program: GENTILEN through UNLZW. INITDES and UFCDIT are two procedures in the GNU implementation of the DES encryption algorithm [DES 77]. Finally, MD5C and DEQUAN were taken from an implementation of the RSA cryptosystem [Rivest 78].

The column labeled \(|V|\) shows the number of variables, including compiler-generated temporaries, in the procedure. The next column, \(|E|\), gives the number of edges in the initial access graph. It is easily seen that the access graphs are very sparse. As we have indicated in the beginning of this chapter, this sparsity is favorable to both our heuristic and branch-and-bound procedures. The column labeled "LB" shows the number of instructions in the generated code excluding those that manipulate the address registers (i.e., LDAR, SBAR, and ADAR). The next columns show the number of instructions when variables are assigned to locations based on
### Table 4.1 Summary of examples.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>V</th>
<th>E</th>
<th>LB</th>
<th>Decl</th>
<th>Ord</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Inst</td>
<td>Inst</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHEN DCT</td>
<td>24</td>
<td>63</td>
<td>561</td>
<td>718</td>
<td>1.280</td>
<td></td>
</tr>
<tr>
<td>ICHEN DCT</td>
<td>31</td>
<td>82</td>
<td>579</td>
<td>790</td>
<td>1.364</td>
<td></td>
</tr>
<tr>
<td>LEE DCT</td>
<td>26</td>
<td>82</td>
<td>616</td>
<td>836</td>
<td>1.357</td>
<td></td>
</tr>
<tr>
<td>ILEE DCT</td>
<td>48</td>
<td>131</td>
<td>686</td>
<td>974</td>
<td>1.420</td>
<td></td>
</tr>
<tr>
<td>JREV</td>
<td>29</td>
<td>141</td>
<td>3293</td>
<td>4524</td>
<td>1.374</td>
<td></td>
</tr>
<tr>
<td>LOAD GIF</td>
<td>126</td>
<td>150</td>
<td>1597</td>
<td>1797</td>
<td>1.125</td>
<td></td>
</tr>
<tr>
<td>AUTO CROP</td>
<td>23</td>
<td>53</td>
<td>506</td>
<td>585</td>
<td>1.156</td>
<td></td>
</tr>
<tr>
<td>AUTO CROP 24</td>
<td>128</td>
<td>290</td>
<td>1719</td>
<td>2113</td>
<td>1.229</td>
<td></td>
</tr>
<tr>
<td>SMOOTH X</td>
<td>27</td>
<td>120</td>
<td>621</td>
<td>795</td>
<td>1.280</td>
<td></td>
</tr>
<tr>
<td>SMOOTH Y</td>
<td>60</td>
<td>152</td>
<td>763</td>
<td>979</td>
<td>1.283</td>
<td></td>
</tr>
<tr>
<td>SMOOTH XY</td>
<td>50</td>
<td>102</td>
<td>513</td>
<td>671</td>
<td>1.308</td>
<td></td>
</tr>
<tr>
<td>DITHER</td>
<td>97</td>
<td>219</td>
<td>1345</td>
<td>1712</td>
<td>1.273</td>
<td></td>
</tr>
<tr>
<td>332 DITHER</td>
<td>62</td>
<td>143</td>
<td>823</td>
<td>1057</td>
<td>1.284</td>
<td></td>
</tr>
<tr>
<td>GEN BITLEN</td>
<td>18</td>
<td>45</td>
<td>344</td>
<td>420</td>
<td>1.221</td>
<td></td>
</tr>
<tr>
<td>HUFF BUILD</td>
<td>39</td>
<td>92</td>
<td>702</td>
<td>896</td>
<td>1.276</td>
<td></td>
</tr>
<tr>
<td>INFLATE C</td>
<td>36</td>
<td>62</td>
<td>623</td>
<td>779</td>
<td>1.250</td>
<td></td>
</tr>
<tr>
<td>INFLATE D</td>
<td>48</td>
<td>79</td>
<td>819</td>
<td>963</td>
<td>1.176</td>
<td></td>
</tr>
<tr>
<td>INFLATE S</td>
<td>15</td>
<td>19</td>
<td>241</td>
<td>284</td>
<td>1.178</td>
<td></td>
</tr>
<tr>
<td>LONG MATCH</td>
<td>35</td>
<td>66</td>
<td>454</td>
<td>532</td>
<td>1.172</td>
<td></td>
</tr>
<tr>
<td>SCAN TREE</td>
<td>16</td>
<td>33</td>
<td>191</td>
<td>223</td>
<td>1.168</td>
<td></td>
</tr>
<tr>
<td>UN LZW</td>
<td>34</td>
<td>68</td>
<td>771</td>
<td>909</td>
<td>1.179</td>
<td></td>
</tr>
<tr>
<td>INIT DES</td>
<td>38</td>
<td>63</td>
<td>888</td>
<td>1005</td>
<td>1.132</td>
<td></td>
</tr>
<tr>
<td>UFC DoT</td>
<td>18</td>
<td>52</td>
<td>280</td>
<td>386</td>
<td>1.379</td>
<td></td>
</tr>
<tr>
<td>MD 5C</td>
<td>10</td>
<td>19</td>
<td>2366</td>
<td>2643</td>
<td>1.117</td>
<td></td>
</tr>
<tr>
<td>DEC QUAN</td>
<td>73</td>
<td>129</td>
<td>790</td>
<td>961</td>
<td>1.216</td>
<td></td>
</tr>
<tr>
<td><strong>Cumulative</strong></td>
<td></td>
<td></td>
<td>22091</td>
<td>27552</td>
<td>1.247</td>
<td></td>
</tr>
</tbody>
</table>
order of declaration, and the ratio of this number to that shown in "LB", which also
forms the basis for the ratios in the tables to follow.

The number of instructions in "LB" serves as a lower bound against which we
can evaluate our results. However, this lower bound is not very tight, since it is
impossible to completely eliminate all such instructions under the assumption that
address registers will be used to address variables. Therefore, the ratios shown in
the tables to follow are on the conservative side—the actual lower bounds may be
somewhat higher.

Table 4.2 shows the experimental results of simple offset assignment using the
greedy heuristic we presented in Section 4.2.7, compared against assignment based
on declaration order. CPU times are measured in seconds on a SparcStation 20. On
the average, the greedy heuristic reduces the number of instructions by 4.9% (with
respect to the lower bound), or approximately 20% of address-arithmetic instructions.

Table 4.3 shows the experimental results of simple offset assignment using the
branch-and-bound procedure of Section 4.2.9, compared against the ratios obtained
using the greedy heuristic (shown in the column labeled "Greedy Ratio"). For all
examples the difference between the ratios are very small, and on the average the
branch-and-bound procedure only outperforms the heuristic by about 0.2%, while
taking somewhat longer to complete.

Table 4.4 shows the experimental results for general offset assignment with six
address registers \(k = 6\) in Figure 4-15), compared against the ratios based on the
branch-and-bound SOA. We use the greedy SOA heuristic for the function SOLVE-
SOA, because the heuristic performed very well in practice. As in previous tables,
the ratios shown in the column "GOA Ratio" are based on the simple lower bound.
We have also experimented with varying number of variables \(p\), between two and
six, selected in the procedure SELECT-VARIABLES of Figure 4-15; the best numbers are
shown in the column "Sel". The CPU times given are the total times for trying
different \(p\)'s. Also, although six address registers were allocated, not all of them were
used, for the setup costs may outweigh the benefits when too many address registers
<table>
<thead>
<tr>
<th>Procedure</th>
<th>LB Inst</th>
<th>Decl Ord Ratio</th>
<th>Greedy SOA</th>
<th>Inst</th>
<th>Ratio</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHENDCT</td>
<td>561</td>
<td>1.280</td>
<td>689</td>
<td>1.228</td>
<td>0.5s</td>
<td></td>
</tr>
<tr>
<td>ICHENDCT</td>
<td>579</td>
<td>1.364</td>
<td>753</td>
<td>1.300</td>
<td>0.5s</td>
<td></td>
</tr>
<tr>
<td>LEEEDCT</td>
<td>616</td>
<td>1.357</td>
<td>787</td>
<td>1.278</td>
<td>0.6s</td>
<td></td>
</tr>
<tr>
<td>LLEEDCT</td>
<td>686</td>
<td>1.420</td>
<td>907</td>
<td>1.322</td>
<td>0.7s</td>
<td></td>
</tr>
<tr>
<td>JREV</td>
<td>3293</td>
<td>1.374</td>
<td>4302</td>
<td>1.306</td>
<td>2.8s</td>
<td></td>
</tr>
<tr>
<td>LOADGIF</td>
<td>1597</td>
<td>1.125</td>
<td>1727</td>
<td>1.081</td>
<td>2.5s</td>
<td></td>
</tr>
<tr>
<td>AUTOCrop</td>
<td>506</td>
<td>1.156</td>
<td>571</td>
<td>1.128</td>
<td>0.8s</td>
<td></td>
</tr>
<tr>
<td>AUTOCrop24</td>
<td>1719</td>
<td>1.229</td>
<td>2050</td>
<td>1.192</td>
<td>3.0s</td>
<td></td>
</tr>
<tr>
<td>SMOOTHX</td>
<td>621</td>
<td>1.280</td>
<td>753</td>
<td>1.213</td>
<td>0.7s</td>
<td></td>
</tr>
<tr>
<td>SMOOTHY</td>
<td>763</td>
<td>1.283</td>
<td>929</td>
<td>1.218</td>
<td>0.8s</td>
<td></td>
</tr>
<tr>
<td>SMOOTHXY</td>
<td>513</td>
<td>1.308</td>
<td>626</td>
<td>1.220</td>
<td>0.7s</td>
<td></td>
</tr>
<tr>
<td>DITHER</td>
<td>1345</td>
<td>1.273</td>
<td>1658</td>
<td>1.233</td>
<td>2.0s</td>
<td></td>
</tr>
<tr>
<td>332DITHER</td>
<td>823</td>
<td>1.284</td>
<td>1014</td>
<td>1.232</td>
<td>1.2s</td>
<td></td>
</tr>
<tr>
<td>GENBITLEN</td>
<td>344</td>
<td>1.221</td>
<td>405</td>
<td>1.177</td>
<td>0.4s</td>
<td></td>
</tr>
<tr>
<td>HUFTBUILD</td>
<td>702</td>
<td>1.276</td>
<td>848</td>
<td>1.208</td>
<td>0.9s</td>
<td></td>
</tr>
<tr>
<td>INFLATEC</td>
<td>623</td>
<td>1.250</td>
<td>740</td>
<td>1.188</td>
<td>0.8s</td>
<td></td>
</tr>
<tr>
<td>INFLATED</td>
<td>819</td>
<td>1.176</td>
<td>938</td>
<td>1.145</td>
<td>1.1s</td>
<td></td>
</tr>
<tr>
<td>INFLATES</td>
<td>241</td>
<td>1.178</td>
<td>266</td>
<td>1.104</td>
<td>0.3s</td>
<td></td>
</tr>
<tr>
<td>LONGMATCH</td>
<td>454</td>
<td>1.172</td>
<td>523</td>
<td>1.152</td>
<td>0.5s</td>
<td></td>
</tr>
<tr>
<td>SCANFRE</td>
<td>191</td>
<td>1.168</td>
<td>223</td>
<td>1.168</td>
<td>0.2s</td>
<td></td>
</tr>
<tr>
<td>UNLZW</td>
<td>771</td>
<td>1.179</td>
<td>872</td>
<td>1.131</td>
<td>0.9s</td>
<td></td>
</tr>
<tr>
<td>INITDES</td>
<td>888</td>
<td>1.132</td>
<td>970</td>
<td>1.092</td>
<td>0.8s</td>
<td></td>
</tr>
<tr>
<td>UFCDIT</td>
<td>280</td>
<td>1.379</td>
<td>354</td>
<td>1.264</td>
<td>0.3s</td>
<td></td>
</tr>
<tr>
<td>MD5c</td>
<td>2366</td>
<td>1.117</td>
<td>2620</td>
<td>1.107</td>
<td>1.2s</td>
<td></td>
</tr>
<tr>
<td>DECQUAN</td>
<td>790</td>
<td>1.216</td>
<td>944</td>
<td>1.195</td>
<td>1.3s</td>
<td></td>
</tr>
<tr>
<td>Cumulative</td>
<td>22091</td>
<td>1.247</td>
<td>26469</td>
<td>1.198</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2 Results of SOA using greedy heuristic—compared against declaration order.
<table>
<thead>
<tr>
<th>Procedure</th>
<th>LB Inst</th>
<th>Greedy Ratio</th>
<th>B&amp;B SOA Inst</th>
<th>B&amp;B SOA Ratio</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHENDCT</td>
<td>561</td>
<td>1.228</td>
<td>688</td>
<td>1.226</td>
<td>0.7s</td>
</tr>
<tr>
<td>ICHENDCT</td>
<td>579</td>
<td>1.300</td>
<td>750</td>
<td>1.295</td>
<td>0.7s</td>
</tr>
<tr>
<td>LEE DCT</td>
<td>616</td>
<td>1.278</td>
<td>784</td>
<td>1.273</td>
<td>0.9s</td>
</tr>
<tr>
<td>ILEE DCT</td>
<td>686</td>
<td>1.322</td>
<td>905</td>
<td>1.319</td>
<td>1.8s</td>
</tr>
<tr>
<td>JREV</td>
<td>3293</td>
<td>1.306</td>
<td>4285</td>
<td>1.301</td>
<td>3.9s</td>
</tr>
<tr>
<td>LOADGIF</td>
<td>1597</td>
<td>1.081</td>
<td>1727</td>
<td>1.081</td>
<td>5.4s</td>
</tr>
<tr>
<td>AUTO CROP</td>
<td>506</td>
<td>1.128</td>
<td>571</td>
<td>1.128</td>
<td>0.9s</td>
</tr>
<tr>
<td>AUTO CROP 24</td>
<td>1719</td>
<td>1.192</td>
<td>2049</td>
<td>1.192</td>
<td>23.9s</td>
</tr>
<tr>
<td>SMOOTHX</td>
<td>621</td>
<td>1.213</td>
<td>752</td>
<td>1.211</td>
<td>1.8s</td>
</tr>
<tr>
<td>SMOOTHY</td>
<td>763</td>
<td>1.218</td>
<td>929</td>
<td>1.218</td>
<td>3.0s</td>
</tr>
<tr>
<td>SMOOTH XY</td>
<td>513</td>
<td>1.220</td>
<td>626</td>
<td>1.220</td>
<td>1.5s</td>
</tr>
<tr>
<td>DITHER</td>
<td>1345</td>
<td>1.233</td>
<td>1650</td>
<td>1.227</td>
<td>6.9s</td>
</tr>
<tr>
<td>332DITHER</td>
<td>823</td>
<td>1.232</td>
<td>1006</td>
<td>1.222</td>
<td>2.8s</td>
</tr>
<tr>
<td>GENBITLEN</td>
<td>344</td>
<td>1.177</td>
<td>403</td>
<td>1.172</td>
<td>0.4s</td>
</tr>
<tr>
<td>HUFFBUILD</td>
<td>702</td>
<td>1.208</td>
<td>844</td>
<td>1.202</td>
<td>1.3s</td>
</tr>
<tr>
<td>INFLATEC</td>
<td>623</td>
<td>1.188</td>
<td>736</td>
<td>1.181</td>
<td>0.9s</td>
</tr>
<tr>
<td>INFLATED</td>
<td>819</td>
<td>1.145</td>
<td>935</td>
<td>1.142</td>
<td>1.5s</td>
</tr>
<tr>
<td>INFLATES</td>
<td>241</td>
<td>1.104</td>
<td>266</td>
<td>1.104</td>
<td>0.3s</td>
</tr>
<tr>
<td>LONGMATCH</td>
<td>454</td>
<td>1.152</td>
<td>521</td>
<td>1.148</td>
<td>0.8s</td>
</tr>
<tr>
<td>SCAN TREE</td>
<td>191</td>
<td>1.168</td>
<td>223</td>
<td>1.168</td>
<td>0.3s</td>
</tr>
<tr>
<td>UNLZW</td>
<td>771</td>
<td>1.131</td>
<td>872</td>
<td>1.131</td>
<td>1.1s</td>
</tr>
<tr>
<td>INITDES</td>
<td>888</td>
<td>1.092</td>
<td>970</td>
<td>1.092</td>
<td>1.0s</td>
</tr>
<tr>
<td>UFC DoIT</td>
<td>280</td>
<td>1.264</td>
<td>354</td>
<td>1.264</td>
<td>0.4s</td>
</tr>
<tr>
<td>MD5 C</td>
<td>2366</td>
<td>1.107</td>
<td>2620</td>
<td>1.107</td>
<td>1.2s</td>
</tr>
<tr>
<td>DECQUAN</td>
<td>790</td>
<td>1.195</td>
<td>944</td>
<td>1.195</td>
<td>3.4s</td>
</tr>
<tr>
<td>Cumulative</td>
<td>22091</td>
<td>1.198</td>
<td>26410</td>
<td>1.196</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3 Results of SOA using branch-and-bound—compared against greedy heuristic.
### Table 4.4 Results of general offset assignment—compared against branch-and-bound SOA.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>LB Inst</th>
<th>B&amp;B SOA Ratio</th>
<th>GOA Inst</th>
<th>GOA Ratio</th>
<th>Reg</th>
<th>Sel</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHENDDCT</td>
<td>561</td>
<td>1.226</td>
<td>598</td>
<td>1.066</td>
<td>6</td>
<td>2</td>
<td>4.0s</td>
</tr>
<tr>
<td>ICHENDDCT</td>
<td>579</td>
<td>1.295</td>
<td>606</td>
<td>1.047</td>
<td>6</td>
<td>2</td>
<td>4.6s</td>
</tr>
<tr>
<td>LEE3DCT</td>
<td>616</td>
<td>1.273</td>
<td>699</td>
<td>1.134</td>
<td>5</td>
<td>4</td>
<td>5.0s</td>
</tr>
<tr>
<td>ILEE3DCT</td>
<td>686</td>
<td>1.319</td>
<td>756</td>
<td>1.102</td>
<td>6</td>
<td>4</td>
<td>7.1s</td>
</tr>
<tr>
<td>JREV</td>
<td>3293</td>
<td>1.301</td>
<td>3510</td>
<td>1.066</td>
<td>6</td>
<td>2</td>
<td>30.7s</td>
</tr>
<tr>
<td>LOADGIF</td>
<td>1597</td>
<td>1.081</td>
<td>1716</td>
<td>1.075</td>
<td>5</td>
<td>3</td>
<td>21.6s</td>
</tr>
<tr>
<td>AUTOCROP</td>
<td>506</td>
<td>1.128</td>
<td>571</td>
<td>1.128</td>
<td>1</td>
<td></td>
<td>4.5s</td>
</tr>
<tr>
<td>AUTOCROP24</td>
<td>1719</td>
<td>1.192</td>
<td>2016</td>
<td>1.172</td>
<td>5</td>
<td>3</td>
<td>33.1s</td>
</tr>
<tr>
<td>SMOOTHX</td>
<td>621</td>
<td>1.211</td>
<td>698</td>
<td>1.124</td>
<td>5</td>
<td>3</td>
<td>7.2s</td>
</tr>
<tr>
<td>SMOOTHY</td>
<td>763</td>
<td>1.218</td>
<td>866</td>
<td>1.135</td>
<td>6</td>
<td>2</td>
<td>10.0s</td>
</tr>
<tr>
<td>SMOOTHXY</td>
<td>513</td>
<td>1.220</td>
<td>589</td>
<td>1.148</td>
<td>3</td>
<td>3</td>
<td>7.5s</td>
</tr>
<tr>
<td>DITHER</td>
<td>1345</td>
<td>1.227</td>
<td>1560</td>
<td>1.160</td>
<td>4</td>
<td>5</td>
<td>18.3s</td>
</tr>
<tr>
<td>332DITHER</td>
<td>823</td>
<td>1.222</td>
<td>931</td>
<td>1.131</td>
<td>5</td>
<td>3</td>
<td>12.5s</td>
</tr>
<tr>
<td>GENBITLEN</td>
<td>344</td>
<td>1.172</td>
<td>387</td>
<td>1.125</td>
<td>4</td>
<td>4</td>
<td>3.0s</td>
</tr>
<tr>
<td>HUFTBUILD</td>
<td>702</td>
<td>1.202</td>
<td>787</td>
<td>1.121</td>
<td>5</td>
<td>2</td>
<td>9.2s</td>
</tr>
<tr>
<td>INFLATEC</td>
<td>623</td>
<td>1.181</td>
<td>719</td>
<td>1.154</td>
<td>2</td>
<td>4</td>
<td>5.1s</td>
</tr>
<tr>
<td>INFLATED</td>
<td>819</td>
<td>1.142</td>
<td>893</td>
<td>1.090</td>
<td>4</td>
<td>2</td>
<td>9.8s</td>
</tr>
<tr>
<td>INFLATES</td>
<td>241</td>
<td>1.104</td>
<td>266</td>
<td>1.104</td>
<td>1</td>
<td></td>
<td>4.3s</td>
</tr>
<tr>
<td>LONGMATCH</td>
<td>454</td>
<td>1.148</td>
<td>498</td>
<td>1.097</td>
<td>6</td>
<td>3</td>
<td>6.0s</td>
</tr>
<tr>
<td>SCANTRIE</td>
<td>191</td>
<td>1.168</td>
<td>213</td>
<td>1.115</td>
<td>5</td>
<td>2</td>
<td>3.1s</td>
</tr>
<tr>
<td>UNLZW</td>
<td>771</td>
<td>1.131</td>
<td>857</td>
<td>1.112</td>
<td>3</td>
<td>4</td>
<td>7.3s</td>
</tr>
<tr>
<td>INITDES</td>
<td>888</td>
<td>1.092</td>
<td>957</td>
<td>1.078</td>
<td>5</td>
<td>4</td>
<td>6.2s</td>
</tr>
<tr>
<td>UFCDOIT</td>
<td>280</td>
<td>1.264</td>
<td>317</td>
<td>1.132</td>
<td>5</td>
<td>2</td>
<td>2.7s</td>
</tr>
<tr>
<td>MD5c</td>
<td>2366</td>
<td>1.107</td>
<td>2468</td>
<td>1.043</td>
<td>2</td>
<td>2</td>
<td>6.1s</td>
</tr>
<tr>
<td>DEQUAN</td>
<td>790</td>
<td>1.195</td>
<td>931</td>
<td>1.178</td>
<td>2</td>
<td>5</td>
<td>8.5s</td>
</tr>
<tr>
<td>Cumulative</td>
<td>22091</td>
<td>1.196</td>
<td>24409</td>
<td>1.105</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 4-19 Comparison of declaration order, SOA, and GOA. For SOA, only the results using the greedy algorithm are shown.
are used. The number of address registers that are actually used is shown in the column "Reg". On the average, using multiple address registers further reduces the number of instructions by 9.1%, or another 46% of the address-arithmetic instructions that SOA could not eliminate. Since the lower bound is obviously loose, the results are in fact closer to optimal than shown in the tables.

4.6 Summary and Future Research

This chapter examined the problems of storage assignment that DSP compilers often encounter. When the target machine has very limited addressing capabilities, the layout of the variables in a procedure affects the size and performance of the generated code. Therefore, to obtain better code a compiler must analyze the patterns in which variables are accessed and intelligently assign frame-relative offsets to the variables. We have formulated the simple offset assignment problem and presented both heuristic and optimal procedures for the problem. We have also extended the problem to a more general form, called general offset assignment, in which more than one address register may be used.

There are several interesting problems that need further investigation. In the procedure SOLVE-GOA, we have focused on a particular scheme for building up the partition of variables, namely, allocating a new address register in each iteration. There may be other methods of determining the best partition given a number of available registers. In addition, the procedure SELECT-VARIABLES may be refined to take more parameters into consideration.

We can also extend the offset assignment problems to take into account other characteristics of variable accesses than merely those summarized by the access graph. For instance, it is not uncommon to find variables with disjoint life-times. We can reduce data memory requirements if we assign the same location to two variables with disjoint life-times. In the context of simple offset assignment, however, the sharing of locations means collapsing two or more vertices into one vertex in the access graph.
This may lead to vertices with too many incident edges, most of which cannot be selected; hence, merging variables with disjoint life-times may be detrimental to the goal of improving code size and performance we set out to achieve in this chapter. On the other hand, if we allocate an additional address register, as in general offset assignment, we may be able to circumvent this problem provided the access graph for the subset of variables addressed by this register is sufficiently sparse. Because the number of address registers is limited, it is not always possible to allow for merging of variables. Therefore, a more thorough analysis of variable accesses is needed. The tile tree [Callahan 91] offers a natural and powerful way of analyzing and summarizing variable usage, and has been successfully applied to the traditional register allocation problem. By effectively using the information derived from tile tree analysis, we can best utilize the data memory while keeping the program small and efficient. This is an important problem that merits further study.
Chapter 5

Code Compression

Object code usually contains some amount of redundancy, in the information-theoretic sense. In fact, it is not uncommon in general-purpose computing to apply high-performance data compression techniques, such as the well-known LZ77 algorithm [Ziv 77], to reduce the size of executable programs. However, in this context, a program to be compressed is treated as sequential data, and a compressed program must be first decompressed and loaded into memory before it can be executed. Hence, savings are achieved only on secondary storage (i.e., disks). These techniques are not directly applicable for code-size minimization in the context of embedded systems, since embedded systems typically do not have secondary storage. Furthermore, programs may have arbitrary control-flow structures. If we were to execute a program from its compressed form, we would need some mechanism of random access, which is impractical, if not impossible, for these compression techniques.

With appropriate models and architectural support, however, it is possible to apply compression techniques to reduce program size and to execute directly from the compressed programs. This is the main theme of the present chapter. We will first review the previous work on code compression. Then we will present our two compression models and the architectural support required for one of the models. In addition, we present an algorithm for our models of code compression based on a set covering formulation (see Appendix A). Not only does the set covering formulation
yield notably better solutions than our initial greedy algorithm, it can also be extended to take into consideration the performance penalty resulting from code compression.

5.1 Previous Work

Fraser et al. presented a compression scheme in [Fraser 84] based on cross-jumping and procedural abstraction. In cross-jumping, common tails of basic blocks that have the same successor are extracted and placed in a new basic block, so that the common code only appears once. Figure 5-1 illustrates this transformation. Blocks L1 and L2 have a common tail sequence Z, and the unique successor of both is L3. We may, without altering the semantics of the program, create a new block L4 consisting of simply the sequence Z and make L4 the new successor of L1 and L2, thereby reducing the number of (static) occurrences of Z. Cross-jumping is performed before and independently of procedural abstraction. Procedural abstraction, on the other hand, is similar to our notion of mini-subroutines. However, their approach is based purely on software and only exploits the suffix relation. (In contrast, one of our methods can exploit the substring relation, which is more flexible.) Moreover, the impact on performance, though reported, was not taken into account in the formulation such that the user may specify trade-offs between code size and performance.

More recently, Wolfe et al. proposed a novel RISC architecture that can execute programs stored in the memory in compressed form [Wolfe 92] [Kozuch 94]. Figure 5-2 (page 144) shows the Wolfe–Chanin architecture. The idea is to decompress, in the cache subsystem, the section of the program immediately to be executed so that the main processor sees the original instructions. To correctly account for branches, a line-address table (LAT) is required to translate the uncompressed address of each basic block to its new address in the compressed code. This table is itself cached in the cache line-address look-aside buffer (CLB).

Although it allows high-performance compression methods to be used, this approach requires a significant redesign of the cache subsystem. Wolfe et al. have not
shown how much additional circuitry is necessary to implement the capability of decompression into the cache subsystem. The area required by the decompression circuitry may offset the gains in code size obtained via compression. Moreover, the use of caches makes it very difficult to predict and estimate the run-time behavior of the system, and in certain real-time applications caches are not utilized precisely for this reason.

5.2 Our Approach

The concept of data compression, however, inspired us to use a scheme that can achieve better results than mere conventional optimization. This scheme is similar to Fraser's procedural abstraction described in the previous section, and is based on a compression model called the external pointer macro (EPM) model [Storer 82]. In this model, compressed data consists of a dictionary and a skeleton. The dictionary contains substrings that occur frequently in the original data. The skeleton contains symbols from the alphabet of the original data, interspersed with pointers to the dictionary.
Figure 5-2 Wolfe–Chanin architecture. In this architecture, the section of the program immediately to be executed is decompressed by the cache subsystem (the cache refill engine) and the main processor sees the original instructions. To correctly account for branches, a line-address table (LAT) is required that translates uncompressed address of a basic block to its new address in the compressed code. This table is itself cached in the cache line-address look-aside buffer (CLB).
This model is particularly suited to our code-minimization approach because the decoding process is simple and can be done in real time, and little or no extra hardware is required to support it. Thus, common sequences of instructions (not just common subexpressions, which may be eliminated by an optimizing compiler) are extracted and stored in a dictionary, and occurrences of these instructions are replaced by pointers (i.e., calls) to the appropriate location in the dictionary. An important characteristic of our approach is that the instruction set of the enhanced machine is a superset of the original machine; hence, all programs that could run on the original machine can also run on the enhanced machine.

In this chapter we will introduce two methods of code size minimization. The first method is purely minimization on the part of the software; no hardware modification is necessary. In this method, common sequences are extracted to form dictionary entries and are replaced by calls to the dictionary. In addition, if a sequence is a suffix of some dictionary entry, it may also be replaced by a call to that entry with the appropriate starting point. We will generalize the suffix relation to a special class of blocks called extended blocks and show that this method applies under this generalization as well.

The second method is more flexible in its use of the dictionary. Unlike the first method, here the dictionary can be viewed as a large entry in itself, not merely a collection of entries. Occurrences of any substring of the dictionary can be replaced by appropriate pointers. Although some extra hardware is required to support this compression model, the total savings in code size are expected to outweigh the cost of the extra hardware.

As we shall see in Section 5.5, these two methods have different restrictions and therefore different strengths. Since the enhanced machine can execute programs written for the original machine, we can combine the two methods together to achieve greater reduction on code size. In the following sections we will present a framework in which such strategies can be utilized for code-size minimization, and a compression algorithm.
5.3 Preliminaries

5.3.1 Data Compression

We briefly review the basic terminology of the macro model of data compression, as defined in [Storer 82]. The source data is treated as a finite string over some alphabet. In the external pointer macro (EPM) model, the compressed form of the source data consists of a dictionary and a skeleton. The dictionary is a string. The skeleton is a sequence of symbols of the alphabet interspersed with pointers to the dictionary. Each pointer represents a substring of the source data that is to be interpreted by the decoding process. A pointer consists of a pair of integers \((addr, len)\) where \(addr\) indicates the position in the dictionary to which the pointer refers, and \(len\) indicates the length of the substring.

As an example, let the alphabet \(\Sigma\) be the set \(\{a, b, c, d, e, f\}\) and consider the source string

\[ x = b b c d a b b e f a b f f b e f \]

with the dictionary

\[ z = a b b e f. \]

One compressed form of \(x\) using dictionary \(z\) is

\[ y = a b b e f \mid (2,2) \text{ cd } (1,5) (1,2) \text{ ff } (3,3) \]

where \(\mid\) serves only as a notational delimiter for the dictionary. Assuming each pointer has a cost of 1 (and \(\mid\) has zero cost), the ratio of the length of \(y\) to that of \(x\) is 13/16. The decoding process is straightforward: we simply scan through the skeleton, replacing each pointer by its reference to the dictionary with the indicated length.

Wagner gave an algorithm based on dynamic programming for optimally parsing the source into a skeleton given a collection of phrases (similar to a dictionary, though less flexible), but did not show how the phrases were best generated [Wagner 73]. A heuristic algorithm for generating a dictionary was presented in [Mayne 75]. Storer
and Szymanski showed that the problem of deciding whether the length of the shortest possible compressed form is less than \( k \) is NP-hard [Storer 82].

### 5.3.2 Definitions, Conventions, and Assumptions

Throughout this chapter we will model our system as a machine with a programmable processor, a program ROM, and some application-specific integrated circuit (ASIC). By *software* we mean the program stored in the program ROM, and by *hardware* we mean the processor and the ASIC. We will consider programs at the level of machine instructions, although the same techniques may be applied to intermediate representations and microinstructions. For intermediate representations this would entail an augmentation of the intermediate language to support the kinds of transformations described in the sequel.

We assume that subroutine linkage is accomplished with a link register or a stack of link registers. Thus the \texttt{CALL} instruction places the address of the next instruction in the link register and transfers control to the destination address, and the \texttt{RET} instruction transfers control back to the address designated in the link register.

Since the underlying compression model is one based on textual substitution, we must define what our alphabet is. To this end, we classify instructions into two types: \textit{control-flow instructions} and \textit{operational instructions}. Conditional branches, unconditional jumps, subroutine calls, and returns from subroutines, and instructions that modify the contents of the link register belong to the former type. All other instructions (i.e., load, store, and arithmetic and logical operations) belong to the latter. Our alphabet \( \Sigma \) consists of the equivalence classes of the set of operational instructions. Note that two instructions with the same operator but different operands are considered different instructions. Two instructions are equivalent if replacing one by the other does not change the semantics of the program. The equivalence may be dependent on the context; for example, \texttt{SUB R1, R1, R1} and \texttt{MOVE R1, 0} are interchangeable if condition codes generated by the former can be ignored (at a particular point in the program).

We can use reaching definition analysis [Aho 86, page 610] on the condition codes
to determine if they are used by subsequent instructions. Henceforth we assume that
this equivalence analysis has already been performed.

Unless otherwise stated, when we speak of a graph we mean a subgraph of the
control-flow graph of the procedure in which each vertex is an instruction (rather than
a basic block), and each edge denotes a possible flow of control between instructions.

**Definition 5.1** A graph is a quadruple \( \langle V, I, E, O \rangle \) where \( V \) is the set of vertices, \( I \) is the
set of internal edges, \( E \) is the set of edges entering \( G \) from the outside, and \( O \) is the set of
edges leaving \( G \).

**Definition 5.2** If \( \langle u, v \rangle \in E \), then \( u \) is called a predecessor of \( G \), and \( v \) is called an entry
point of \( G \). If \( \langle u, v \rangle \in O \), then \( v \) is called a successor of \( G \), and \( u \) is called an exit point
of \( G \).

**Definition 5.3** Two graphs \( G_1 \) and \( G_2 \) are isomorphic if \( \langle V_1, I_1, O_1 \rangle \) is isomorphic to
\( \langle V_2, I_2, O_2 \rangle \), each pair of corresponding vertices denote the same instruction, and each pair
of corresponding edges denote the same condition for control-transfer.

**Definition 5.4** A simple block is a sequence of vertices \( v_1, v_2, ..., v_k \) such that for \( 1 \leq i < k 
\), \( v_i \) is the only predecessor of \( v_{i+1} \). A simple block that is not contained in any other simple
block is called a basic block.

This definition of basic block is equivalent to that in Chapter 3, i.e., if any
instruction of a basic block is executed, so is every other instruction of the basic
block.

### 5.4 Examples

In this section we will first present two examples illustrating the two methods
of compression that we will describe later in the chapter. These examples serve to
expound on the kinds of program transformations that are supported by the proposed
compression framework.
5.4.1 Example Illustrating the First Method

Consider the C program segment shown in Figure 5-3(a) (page 150). In this example, the function max() was defined as a C-preprocessor macro, and is compiled as if it were an in-line function. Consequently, the code of the same function is repeated three times in the object code. On the other hand, suppose we had defined max() as a genuine function, as in Figure 5-3(b). Now the function appears only once in the object code. However, the overhead in the code size associated with a full function call (e.g. saving the contents of registers to the stack and later restoring them) may not be economical in terms of code size after all. A simple solution is given in the following, which we call a mini-subroutine call (in contrast to a full function call generated by compilation).

First of all, assuming that max() is a macro, we note that the two instances of max(a,b) in the function first() are not common subexpressions because a and b have been modified in between, yet the sequences of instructions for these two instances are the same. We can therefore extract this sequence and place it elsewhere. Each occurrence of the sequence in first() is then replaced by a CALL instruction. To the end of the extracted sequence we add a return instruction (RET), completing the mini-subroutine.

Now suppose the same registers or stack-frame-relative addresses are assigned to the variables a and b in both functions first() and second(). Then the same sequence of instructions will be generated for the instance of max(a,b) in second() as well, and the same mini-subroutine is equally applicable even though it is in a different scope. Moreover, during the register allocation and code selection phases of code generation a number of optimizations can be made to improve the likelihood of such matches. To improve the likelihood of matches in code sequences a total ordering can be given to instructions so that whenever there is a freedom in the execution order of the instructions the instructions are issued according to the total order. For example, whenever there is no dependency between an ADD instruction and a LOAD instruction in a straight line sequence then the ADD instruction can be
```c
#define max(x,y)  (((x) > (y)) ? (x) : (y))

int first()
{
    int a, b, f;
    ...
    f = max(a,b);
    ...
    a = ...;
    b = ...;
    f = max(a,b);
    ...
}

int second()
{
    int a, b, h;
    ...
    h = max(a,b);
    ...
}
```

(a)

```c
int max( int x, int y )
{
    return ((x > y) ? x : y);
}
```

(b)

Figure 5-3 Example C program. (a) Program with max() defined as a C-preprocessor macro. (b) Defining max() as a genuine function.
issued first if it is first in the total order. Giving register preferences to particular instructions will also improve the likelihood of matches. For instance, whenever the registers are available the ADD instruction can preferentially receive registers R1, R2, and R3 will improve the likelihood of matches on this instruction. We revisit this approach in Section 5.10.

Mini-subroutine calls are not captured by high-level languages. Compilers are usually conservative and save context information upon function calls. To the best of our knowledge this type of code size optimization is performed manually to this date, though it is conceivable that liveness analysis and dead code elimination can reduce overhead instructions in specific subroutine calls. The first method presented in this chapter will be based on the identification of mini-subroutine calls of this type, without modification of the hardware.

### 5.4.2 Example Illustrating the Second Method

Let $\Sigma$ again be the alphabet as defined in Section 5.3, and let $a$, $b$, etc. be symbols of $\Sigma$. Figure 5-4(a) (page 152) shows three subject strings. From these subject strings we can extract the sequences bdf and eafd as subroutines. The resulting dictionary is shown in Figure 5-4(b), where $r$ denotes the RET instruction. The occurrences of these sequences in the dictionary can be substituted in the original subject strings, and the result is shown to the right of the dictionary, the shaded numbers denoting dictionary accesses (i.e., CALLs). Assuming each instruction has a cost of unity, by extracting these subroutines we only gain a saving of one instruction for these three sequences when the size of the dictionary is also taken into account. (Of course, in another context we may used the same dictionary for other sequences as well, thereby achieving greater savings.)

Now let us exercise a little wishful thinking. Suppose our machine has a new instruction (say CALD, for call-dictionary) which is similar to CALL, but takes an additional argument: a length $len$. When the machine encounters CALD, it transfers the control to the appropriate address in the dictionary, executes $len$ instructions,
Figure 5-4  Two types of dictionaries. (a) Subject strings A, B, and C. (b) Dictionary with explicit RET instructions (denoted by r) as entry delimiters. (c) Dictionary without explicit entry delimiters.
and *implicitly* returns to the instruction following \texttt{CALD}. We can then extract as our dictionary the entire sequence \texttt{A} and replace the subject strings by those shown in Figure 5-4(c). Each pair of numbers \((addr, len)\) appearing in a shaded box represents the instruction \texttt{CALD} \((addr, len)\). By using the implicit returns we gain a saving of six instructions overall.

Evidently this model requires hardware support due to the implicit returns. As we shall see, the hardware modifications required to support this model are quite simple and effective.

### 5.5 Proposed Compression Methods

Our proposed methods are based on the external pointer macro compression model described in Section 5.3. The two methods differ in how the dictionary and pointers are represented, and each has its own strength and limitations.

#### 5.5.1 Method I

The first method is an optimization purely in software. Common sequences are extracted and placed in a dictionary, and instances of these sequences are replaced by *mini-subroutine calls* to the dictionary. By mini-subroutine call we mean using a simple \texttt{CALL} instruction without passing parameters. Determining which sequences to extract and replace is the core of compression algorithms. Unlike Method II, the sequences to be extracted are not restricted to basic blocks; some conditional branches may be allowed.

To characterize the circumstances under which conditional branches are allowed, we define the notion of extended blocks.

**Definition 5.5** An extended block is a graph \(G\) that has a unique successor.

Note that under this definition single entry is not required; there may well be many edges coming into different vertices in the graph.
Definition 5.6 A (generalized) suffix of an extended block $G$ is a subgraph $G' = (V', I', E', O')$ of $G$ such that if $u \in V'$, then all vertices $v \in V$ reachable from $u$ are also in $V'$. The suffix generated by a vertex $u \in G$ consists of all vertices $v \in V$ such that $v$ is reachable from $u$.

Theorem 5.1 allows us to exploit the suffix relation to make better use of dictionary entries, as we shall see in the sequel.

Lemma 5.1 If $G' = (V', I', E', O')$ is a suffix of an extended block $G = (V, I, E, O)$, then $G'$ is also an extended block, and the successor of $G$ is also the successor of $G'$. In other words, $O' = O$.

Proof — Suppose $G'$ is not an extended block. This means there exist edges $(u_1, v_1)$ and $(u_2, v_2)$ where $u_1, u_2 \in G'$ and $v_1, v_2 \notin G'$, $v_1 \neq v_2$. At least one of $v_1$ and $v_2$ must be in $G$, since otherwise $(u_1, v_1)$ and $(u_2, v_2)$ would be in $O$ and $G$ would not be an extended block.

Assume $v_2 \in G$. But since $v_2$ is reachable from a vertex in $G'$ (i.e., $u_2$), by definition $v_2 \in G'$. This is a contradiction; hence, we conclude that $G'$ is an extended block. Furthermore, if the successor of $G'$ were different from that of $G$, this would imply that $G$ has two successors, which again leads to a contradiction. 

Theorem 5.1 Let $H$ be an extended block whose successor is the RET instruction. Suppose $G = (V, I, E, O)$ is isomorphic to a suffix of $H$. We may, without altering the semantics of the program, replace $G$ by $\hat{G} = (\hat{V}, \hat{I}, \hat{E}, \hat{O})$, which is derived as follows:

- For each entry vertex $v$ of $G$, we have a vertex $\hat{v}$ in $\hat{V}$, which is a CALL instruction to $v$ (the corresponding vertex of $v$ in $H$).
- $\hat{I} = \text{the empty set}$.
- $\hat{E} = \{(u, \hat{v}) \mid (u, v) \in E\}$.
- $\hat{O} = \{(\hat{v}, s) \mid \hat{v} \in \hat{V}, s \text{ is the successor of } G'\}$. 
In other words, $H$ is the graph for a dictionary entry, and any extended block that is isomorphic to a suffix of $H$ can be replaced with calls to the appropriate entry points.

**Proof** — Let $v_1$ be an entry point of $G$ and $s$ be the successor of $G$. Let $p = [v_1, ..., v_k]$ be a path (which may contain loops), where $v_k$ is an exit vertex. Note that $v_i \in V$ for each $i$. Now let $w_i$ denote the corresponding vertex of $v_i$ in $H$. By isomorphism, each vertex $w_i$ denotes the same instruction as $v_i$, and each edge $\langle w_i, w_{i+1} \rangle$ denotes the same condition for control transfer as $\langle v_i, v_{i+1} \rangle$. Hence, the sequence $q = [w_1, ..., w_k]$ is semantically equivalent to $p$.

Conversely, for any path $q = [w_1, ..., w_k]$ in $H$ where $w_1$ corresponds to an entry point $v_1$ of $G$, there is a corresponding path $p = [v_1, ..., v_k]$ in $G$ since $G$ is isomorphic to a suffix of $H$ (all vertices reachable from $w_1$ must be in the suffix, by definition).

Hence, by making a call to $w_1$ whenever we would enter $G$ at $v_1$, and then returning to $s$, we preserve the semantics of the program. ■

Figure 5-5 (page 156) shows some examples of extended blocks. In the figure, the circles denote basic blocks, and the squares denote (for the dictionary) entry points and (for the main program) dictionary calls to the corresponding entry points. In particular, □ denotes return from the mini-subroutine. Figure 5-5(d) shows a dictionary entry, and Figure 5-5(e)–(g) show the graphs resulting from replacing the extended blocks in Figure 5-5(a)–(c) by mini-subroutine calls.

With an explicit _RET_ instruction for each mini-subroutine, Method I allows for extended blocks, because an extended block has a unique exit point, and, when extracted as a mini-subroutine, this unique exit point corresponds to the _RET_. Furthermore, Lemma 5.1 allows multiple entry points and suffices to be exploited. For example, the edges entering vertex ◊ can be changed to a mini-subroutine call with entry at □, because the lemma guarantees that the suffix generated by ◊ (which consists of all vertices reachable from ◊) has exactly the same exit point. Similarly, the suffix in Figure 5-5(c) can be replaced by a call with entry at □. Even though the extended block of Figure 5-5(d) has vertices other than ◊ and □, the only path
Figure 5-5  Extended blocks and generalized suffixes. (a) (b) Isomorphic extended blocks. (c) An extended block isomorphic to a suffix thereof. (d) Extracting the extended block to form a mini-subroutine. (e) (f) (g) Extended blocks in the original graphs of (a), (b), and (c), respectively, replaced by mini-subroutine calls.
through this extended block from entry point $\text{a}$ is $\text{b} \rightarrow \text{c}$; hence, this is a legal substitution for the suffix in Figure 5-5(c).

Note that this transformation is not captured in structured high-level languages, because there is no construct to make a subroutine call to the middle of a function. Thus this method can achieve code size reduction not possible by simply writing function calls in the source code.

The present implementation is limited to basic blocks. Under this restriction, extended blocks are just strings and the suffix relation reduces simply to the one in the string-theoretic sense. Even so, we still obtain quite encouraging results (see Section 5.8).

### 5.5.2 Method II

The second method is somewhat more flexible in the use of the dictionary entries. It corresponds directly to a hardware implementation of the EPM model of compression. That is, as a pointer in the model consists of an address and a length, so in the instruction that calls the subroutine (say \texttt{CALD}, for \textit{call-dictionary}) the number of instructions to be executed from the dictionary is specified as well as the address. Hence the return from the dictionary is implicit, without an actual \texttt{RET} instruction. Clearly, in typical architectures an instruction such as \texttt{CALD} does not exist; therefore, we will need to augment the instruction set.

One restriction on this method is that the boundaries of basic blocks have to be observed. This is because the paths of conditional branches may not have equal length, and since the point of return is implied by the length parameter, we cannot in general determine exactly when to return. Consequently, when we apply text compression algorithms, we must be careful that sequences to be extracted consist of operational instructions only.

Extended blocks can be used if all paths from the entry to the exit have equal lengths. \texttt{NOP}s can be inserted so that the extended blocks will have this property. Although this may have an adverse effect on the dictionary size, it can potentially
result in greater compression. The exact trade-offs are dependent on the application itself.

We use the TMS320C25 architecture [TI 93] to exemplify the method. The hardware modifications are shown in Figure 5-6 (for simplicity, much of the data-path is not shown).

An S–R flip-flop, a counter, an AND gate, two OR gates, and a link register have been added to the base processor. The S-R flip-flop, if set, indicates that the processor is in dictionary mode. The counter records the number of instructions in the dictionary that remain to be executed. The link register stack is used to store the return address for CALD as well as the original CALL. Hence, the PUSH signal for the link register file needs to be asserted for both CALL or CALD. Similarly, the POP signal for the link register file needs to be asserted when a RET instruction is encountered or when the counter reaches zero. The OR gates are used for this purpose.

With the hardware support, the steps of the CALD (addr, len) instruction are as follows:

1. Store the return address in the link register.
2. Set the processor in dictionary mode; load the counter with len.
3. Push the return address to the link register stack.
4. Set the PC to addr.

Once the processor is in dictionary mode, the counter begins counting down towards zero. When the counter reaches zero, the path from the link register stack to the PC is selected by the multiplexor, thereby accomplishing the implicit return. At the same time, the dictionary mode bit is reset to normal mode, and the processor continues in the main program. As an aside, we note that the implicit return as implemented works with pipelines as well. There is no complication with delayed branches because the PC is loaded with the return address directly.
Figure 5-6 Architectural support for Method II. When CALD is executed, the *dictionary mode* register is set and the counter is loaded with the number of instructions to execute from the dictionary. The return address is pushed to the link register stack upon the execution of either a normal CALL instruction (PUSH₀ enabled) or the new CALD instruction. Similarly, the return address is popped from the link register stack when either a normal RET instruction is executed (POP₀ enabled) or the counter reaches zero while the machine is in dictionary mode.
Another advantage of the second method over the first is that the number of cycles required for dictionary accesses is reduced. Indeed, a dictionary access uses only one extra instruction: \texttt{CALD}. Provided that the hardware modification does not increase the critical path, this method incurs less performance penalty that the first. Although additional hardware means that off-the-shelf processors cannot be used, we believe that, as code size minimization becomes more important in the context of embedded systems, processors should be designed with architectural support for code compression.

5.6 An Algorithm for Code Compression

We now present an algorithm for code compression based on a set covering formulation. A brief overview of the set covering problem is given in Appendix A.

In this algorithm, the process of code compression consists of two phases: (1) generation of potential dictionary entries, and (2) substitution and dictionary generation. During the generation of potential dictionary entries, substrings that occur several times in the instruction stream are discovered. A set covering problem is derived and solved, with variables corresponding to the selection of substitutions and the selection of dictionary entries. The selected substitutions are then carried out and the dictionary is formed by combining the selected entries.

5.6.1 Generation of Potential Dictionary Entries

The instruction stream is first divided into basic blocks, using the algorithm described in [Aho 86, page 529]. Then each basic block is compared with every other basic block, as well as itself, for common substrings. A threshold on the minimum length \( M \) (for example, 3) of substrings is prescribed, so that only potentially beneficial substrings are extracted.

We use a naive algorithm to identify common substrings. This algorithm is similar to the basic string-matching algorithm of [Cormen 90, page 855]. We may also apply
Figure 5-7  Identifying common substrings. Every basic block is compared with every other basic block, as well as itself, at all possible positions of overlap. Longest common substrings at each position are extracted. In this example, the threshold $M$ is 3.
the modulo techniques of [Karp 81] to improve the average-case complexity. For the purpose of exposition, we will simply use the former.

The operation of the algorithm is illustrated in Figure 5-7 (page 161). The two blocks are placed against each other with every possible region of overlap, beginning with the first $M$ instructions of the first block and the last $M$ instructions of the second. The matching substring or substrings in this overlapping region are identified and stored in a table. The second block is then shifted to the right by one instruction, and the process is repeated until the last $M$ instructions of the first block are reached. If a block is compared against itself, we disregard the case when it is aligned exactly with itself, since the matching substring would be the entire block.

The worst case running time of this algorithm is $O(n^2)$, where $n$ is the total number of instructions in the program. This can be easily shown by the following analysis. It is clear that the process of comparing two basic blocks of lengths $l_1$ and $l_2$ takes at most $l_1 \cdot l_2$ steps. Now assume that there are $m$ basic blocks, of lengths $l_1, l_2, \ldots, l_m$. The total number of steps $S(l_1, l_2, \ldots, l_m)$ is thus bounded by:

\[
S(l_1, l_2, \ldots, l_m) \leq \sum_{i=1}^{m} \sum_{j=1}^{i} l_i \cdot l_j = \frac{1}{2}(l_1 + l_2 + \cdots + l_m)^2 + \frac{1}{2}(l_1^2 + l_2^2 + \cdots + l_m^2) \leq n^2.
\]

(5.1)

5.6.2 Substitution and Dictionary Generation

This step involves replacing occurrences of potential dictionary entries in the instruction stream by appropriate pointers to a generated dictionary. Wagner's dynamic programming method for determining optimal substitutions [Wagner 73], however, is not applicable in our context since it assumes a pre-defined set of strings as the dictionary, whereas at this point we have not yet determined a dictionary. Therefore, we will need to solve the problems of pointer substitution and dictionary generation simultaneously.
In this section, we describe a set covering formulation which benefits from the use of recently developed optimum and heuristic covering algorithms [Coudert 95]. In the set covering formulation, we create a covering matrix in which the columns of the matrix are variables, and the rows are disjunctive clauses over different subsets of the variables. We find an assignment to the variables that satisfies each of the clauses and minimizes a cost function.

The variables in the covering formulation are defined as follows. Let \((i,j)\) denote the substitution of entry \(i\) at position \(j\). For each substitution \((i,j)\), we define \(q_{ij} = 1\) if entry \(i\) is not substituted at position \(j\) and \(q_{ij} = 0\) otherwise. Note the sense of this variable is inverted. This is due to the requirement that the cost function must be nonnegative. The covering procedure is instructed not to set the variable \(q_{ij}\) to 1 (i.e., to enable the substitution) whenever possible. For dictionary entries, we define \(m_i = 1\) if entry \(i\) will appear in the dictionary and \(m_i = 0\) otherwise.

Before we describe the clauses in the covering matrix we will first introduce the notion of subsumption. A dictionary entry \(i\) can be subsumed by another entry \(k\) if it is a suffix (Method I) or a substring (Method II) of entry \(k\). In other words, we do not have to keep entry \(i\) in the dictionary because it is effectively available through entry \(k\), because in our compression model a dictionary call may begin at any location in the entry. In either case, entry \(i\) is said to be subsumable by entry \(k\). Let \(S(i)\) denote the set of entries that subsume \(i\).

Given these definitions, we will prescribe two sets of disjunctive clauses, as follows:

1. \((q_{ij} + q_{kl})\) for each pair of substitutions \((i,j)\) and \((k,l)\) that are mutually exclusive.

Two substitutions \((i,j)\) and \((k,l)\) are mutually exclusive if using either of them precludes the use of the other. This clause says that at least one of \(q_{ij}\) and \(q_{kl}\) has to be 1. In other words, if either entry \(i\) is used at \(j\) or entry \(k\) is used at \(l\), the other cannot be used. Note again that \(q_{ij} = 1\) if \((i,j)\) is not used. The cost of not substituting entry \(i\) at position \(j\) is given by \(\text{cost}(q_{ij}) = (i - P)\)
Figure 5-8 Mutually exclusive substitutions. (a) Subject string and potential dictionary entries. The variable $q_{11}$ corresponds to the substitution of entry 1 at position 1; similarly for $q_{24}$ and $q_{38}$. (b) Clauses for mutual exclusion. Selecting $q_{11}$ (setting it to 0) precludes the selection of $q_{24}$ and vice versa. Likewise, $q_{24}$ and $q_{38}$ are mutually exclusive.
where \( l_i \) is the length of entry \( i \) (without the RET instruction) and \( P \) is the size of pointer. This cost assignment reflects that fact that enabling the substitution \((i, j)\) involves using a pointer of length \( P \) in lieu of a string of length \( l_i \), thereby saving \((l_i - P)\) instruction words.

For example, consider the subject string and potential dictionary entries shown in Figure 5-8(a). The variables \( q_{11}, q_{24}, \) and \( q_{38} \) are the variables for the substitutions \((1, 1), (2, 4), \) and \((3, 8)\). Since selecting \((1, 1)\) precludes the selection of \((2, 4)\) and vice versa, we write the clause \((q_{11} + q_{24})\) to indicate this condition. Likewise, we write \((q_{24} + q_{38})\). The advantage of using clauses is apparent: by doing so, we implicitly enumerate all different combinations of substitutions. In a greedy strategy, one might replace longer substrings first, and then shorter ones. However, in the example we have just seen, if we had first enabled \((2, 4)\), the largest substitution of the three, we would not have been allowed to enable \((1, 1)\) and \((3, 8)\). The combination of the latter two may or may not yield better results, depending on the substitution of other entries at other positions. The context is taken into consideration by the covering formulation.

2. \((q_{ij} + m_i + \sum_{k \in S(i)} m_k)\) for each \((i, j)\).

This clause says that if \( q_{ij} = 0 \) (i.e., substitution \((i, j)\) is enabled), then some entry must be included in the dictionary so that this substitution is implemented. This entry could be \( m_i \), or any other entry \( m_k \) that subsumes \( m_i \), which is given by the set of entries \( S(i) \). The cost of the dictionary entry is given by \( \text{cost}(m_i) = (l_i + 1) \) for Method I and \( \text{cost}(m_i) = l_i \) for Method II.

We find an assignment to the different \( q_{ij} \) variables and the \( m_i \) variables that results in each clause being satisfied. This assignment is to minimize the size cost \( C_{\text{size}} \):

\[
C_{\text{size}} = \sum_{q_{ij}; \ q_{ij} = 1} \text{cost}(q_{ij}) + \sum_{m_i; \ m_i = 1} \text{cost}(m_i),
\]  

(5.2)
which results in a maximal number of entries being substituted and a dictionary of minimal size. We use the covering solver SCHERZO described in [Coudert 95] to find such an assignment. Note that the value of the objective function $C_{\text{size}}$ does not correspond directly to the final code size. It differs from the final code size by a constant due to our definition of $\text{cost}(q_{ij})$, and this constant has no effect on the optimization process. To see this, let us consider the length of the skeleton after all selected substitutions have taken place. Denote by $L_{\text{ske}}$ the length of the skeleton and by $L_{\text{dict}}$ the length of the dictionary. Each substitution $(i,j)$ reduces the size of the skeleton by $(l_i - P)$, which is equal to $\text{cost}(q_{ij})$. Thus, if the length of the original program is $L_{\text{orig}}$, then

$$L_{\text{ske}} = L_{\text{orig}} - \sum_{q_{ij}: q_{ij}=0} \text{cost}(q_{ij}).$$  \hspace{1cm} (5.3)

On the other hand, we have

$$\sum_{q_{ij}: q_{ij}=0} \text{cost}(q_{ij}) + \sum_{q_{ij}: q_{ij}=1} \text{cost}(q_{ij}) = Q,$$  \hspace{1cm} (5.4)

where $Q$ is a constant independent of the assignment of the $q_{ij}$'s (since each $q_{ij}$ must be either 0 or 1). Therefore,

$$L_{\text{ske}} = L_{\text{orig}} - Q + \sum_{q_{ij}: q_{ij}=1} \text{cost}(q_{ij}),$$  \hspace{1cm} (5.5)

and the total length of the compressed program $L_{\text{total}}$ is

$$L_{\text{total}} = L_{\text{ske}} + L_{\text{dict}} = (L_{\text{orig}} - Q) + \sum_{q_{ij}: q_{ij}=1} \text{cost}(q_{ij}) + \sum_{m_i: m_i=1} \text{cost}(m_i).$$  \hspace{1cm} (5.6)

Thus, $C_{\text{size}}$ differs from $L_{\text{total}}$ by the constant $(L_{\text{orig}} - Q)$, and by minimizing $C_{\text{size}}$ we also minimize $L_{\text{total}}$.

After solving the set covering problem, we create the dictionary by concatenating the selected entries, those for which the corresponding $m_i$'s are set to 1 in the chosen assignment. We then replace each selected substitution (the $q_{ij}$'s that are set to 0) by
the appropriate instruction (CALL or CALD) with the required arguments. For Method I the concatenation of entries can be done in any arbitrary order because dictionary entries are explicitly delimited by a RET instruction. For Method II, however, it is possible to exploit inter-entry substrings (substrings in the final dictionary that can cross the boundary of two entries) because there are no explicit delimiters; thus the order in which the entries are concatenated may be of some consequence. In fact, as we will show in Section 5.7, to obtain a smaller dictionary we might need to generate more entries than those substrings that actually appear in the program.

5.7 Refinements to the Proposed Algorithm

Although the algorithm based on the set covering formulation described in the previous section benefits from recent progress in solving set covering problems, two aspects of the algorithm can be improved. When we interpret the solutions found by the set covering solver in the context of code compression, we may discover that they need further refinements, especially when heuristics are used to solve the covering problem. In this section we discuss causes for this need and propose ways to tackle this problem.

5.7.1 Eliminating Inefficiently Used Entries

The first improvement involves the elimination of inefficient uses of entries resulting from direct application of the solutions to the set covering problem to substitution and dictionary generation. By defining \( q_{ij} = 0 \) to mean that entry \( i \) is substituted at position \( j \), we aggressively assume that all substitutions are beneficial: the set covering solver assigns 0 to a variable whenever possible because the cost function is nonnegative. However, by so doing we may be overly aggressive and the solution may result in inefficient uses of entries. In our experiments, we have observed in a few instances that certain entries included in the dictionary are substituted only once in the entire program. In other words, for some \( k, m_k \) is 1 while only one of the
many $q_{kl}$'s (say $q_{kl^*}$) is 0. Is it always profitable to set all $q_{kl}$'s to 1 and $m_k$ to 0? Let us consider two cases separately:

1. Entry $k$ is not used by substitutions of entries that it subsumes (other than itself).

2. Some substitutions make use of entry $k$ by virtue of subsumption.

\[ \diamond \quad \diamond \quad \diamond \]

**Case (1)** First, let us suppose that no substitution needs entry $k$ other than the only $q_{kl^*}$ that is set to 0; i.e., no proper suffixes (Method I) or proper substrings (Method II) of entry $k$ are substituted in the program. Then we may further decrease the size of the compressed program by not including entry $k$ in the dictionary at all. Upon removing this entry and the corresponding substitution we save the cost of the pointer ($P$) and, if Method I is used, the cost of the `RET` instruction. We can be assured that the removal of entry $k$ (setting $q_{kl^*}$ to 1 and $m_k$ to 0) does not invalidate any clause in the cover matrix. The clauses involving $q_{kl}$'s for all $l$ are satisfied because we have set all $q_{kl}$'s to 1. The other clauses in which $m_k$ appear are the ones involving those $m_i$, for $i$ such that entry $i$ is subsumed by entry $k$. Consider a typical clause for $q_{ij}$:

$$q_{ij} + m_i + \cdots + m_k + \cdots.$$  \hfill (5.7)

To determine if entry $k$ is needed for Clause (5.7), we need to simply examine the value of every other variable in this clause. If some variable other than $m_k$ is 1 for every clause containing $m_k$, then indeed entry $k$ is not used by any other substitutions and can therefore be eliminated.

The scenario we have described here arises from using heuristics to obtain an approximate solution to solve the covering problem. However, even when an exact branch-and-bound procedure is used, we may still find inefficient uses of entries. This occurs when an entry is not substituted sufficiently many times to actually decrease
the code size. If particular substitutions do not strictly decrease the code size, the covering solver has no preference for either enabling or disabling the substitutions. In fact, because of our intrinsically aggressive substitution in the definition of $q_{ij}$, it is likely that substitutions will be enabled. For example, if Method I is used and $P = 1$, an entry of three instructions must be substituted strictly more than twice to result in a decrease in code size. There is no advantage in substituting it only twice. Consequently, as we return from set covering to the context of code compression, we would prefer disabling the substitutions when there is a tie, for then we would not incur the unnecessary performance penalty. (Alternatively, we can write additional clauses to take performance costs into account; see Section 5.9.)

Case (2) Now suppose that entry $k$ subsumes other entries and some substitution $q_{ij}$ (with entry $i$ subsumed by entry $k$) needs it; in other words, with respect to Clause (5.7), $m_k$ is the only variable set to 1. Here, Method I and Method II exhibit different properties, and we will address each respectively.

In Method I, eliminating entry $k$ often results in a tie. To see this, we first note that if strings $y$ and $z$ are both suffixes of $x$, then either $y$ is a suffix of $z$ or vice versa. Among all (proper) suffixes of entry $k$ that are substituted, let entry $z$ be the largest one; then every other suffix of entry $x$ is a suffix of $z$. Now let us consider the effect of disabling the substitution of entry $k$ (say at position $l$) and removing it from the dictionary. Because entry $k$ was responsible for its suffixes, we now need another entry to replace it. Entry $z$ is a perfect candidate. Also, entry $z$ can now be substituted at position $(l + |k| - |z|)$. Accounting for the adjustments in code size, we observe that, while we have replaced entry $k$ by a shorter entry $z$ for the dictionary, the substitution at location $(l + |k| - |z|)$ brings less benefit for the skeleton. In fact, the effects cancel each other exactly. The performance penalty does not change, either—we still make as many mini-subroutine calls to entry $z$ as we would call entry $k$ before. It may appear that replacing entry $k$ with entry $z$ will enable other substitutions,
because entry $z$ is shorter and causes fewer conflicts with other substitutions. Our experience, however, indicates that there is rarely any advantage in switching from entry $k$ to entry $z$.

In Method II, it is possible that, even when an entry $k$ is not used by any of the $q_{kl}$'s, including the entry in the dictionary may be advantageous, much more so when one of $q_{kl}$'s is 0. We present the argument in Section 5.7.2. Again, our experience indicates that this situation poses no difficulty for the heuristic set covering solver.

◊ ◊ ◊

The improvements obtained by eliminating inefficiently used entries, in general, account for only a very small percentage in the overall code size reduction (approximately 0.1%–0.2% with respect to the size of the original program). It is relatively easy to discover these opportunities by examining the solution given by the set covering solver and to carry out these improvements. On the other hand, the case in which the entry in question subsumes other entries (Case (2)) rarely causes any difficulties. Therefore, in our implementation, we have included the improvements for Case (1) only.

5.7.2 Generating More Potential Dictionary Entries

The second shortcoming of the algorithm is not directly related to the set covering formulation, but rather the generation of potential dictionary entries. We have assumed that the only possible entries are those sequences that actually appear in the original program. In other words, if a sequence does not occur anywhere in the program (even though its substrings and suffices might), we do not consider it as a potential dictionary entry.

For Method I we have seen in Section 5.7.1 that there is no benefit in including an entry in the dictionary if the entry is used (in its entirety) for only one substitution, much less so if the entry is not even used at all. Therefore, for Method I we do
not need to generate other entries than those found by the procedure described in Section 5.6.1.

This property, however, does not extrapolate to the generalized suffix relation nor the substring relation. A simple counterexample suffices to illustrate the point. Suppose in Section 5.6.1 we find, among others, the following potential dictionary entries: \( e_1 = \text{abcdefgh} \) and \( e_2 = \text{efghijk} \); and suppose the string \( e_3 = \text{abcdefg} \) does not appear anywhere in the program. If both \( e_1 \) and \( e_2 \) are substituted in the program, then we would need to include both entries in the dictionary. However, instead of including \( e_1 \) and \( e_2 \), we could have used \( e_3 \), whereby we would save three instructions in the dictionary.

This counterexample reveals the inadequacy of the procedure for generating potential dictionary entries of Section 5.6.1. Unlike Method I, Method II does not require explicit delimiters for dictionary entries; in fact, the entire dictionary is one large entry from which substrings are substituted in the main program.

Clearly, it is computationally difficult to find the optimal dictionary. A possible approach to this problem is to first solve the covering problem with only the entries generated in Section 5.6.1. We then examine the most frequently used entries and form new entries by combining the original ones wherever possible. A new covering problem is created and solved in which variables corresponding to new entries are added and those corresponding to old entries that were combined to form new ones are deleted.

### 5.8 Experimental Results

We present in this section some experimental results on example programs. We have obtained these results by applying our compression techniques on optimized code generated by Texas Instruments’ TMS320C25 compiler.

The statistics of the examples are summarized in Table 5.1 (page 172). This table shows, for each example, the size of the original code produced by Texas
<table>
<thead>
<tr>
<th>Example</th>
<th>TI Optimized</th>
<th>Matrix Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIPINT</td>
<td>1510</td>
<td>1564 x 608</td>
</tr>
<tr>
<td>BENCH</td>
<td>10963</td>
<td>2418 x 1814</td>
</tr>
<tr>
<td>COMPRESS</td>
<td>2333</td>
<td>238 x 247</td>
</tr>
<tr>
<td>DXFTOHSH</td>
<td>1330</td>
<td>616 x 311</td>
</tr>
<tr>
<td>GNUCRYPT</td>
<td>4026</td>
<td>1896 x 1063</td>
</tr>
<tr>
<td>GZIP</td>
<td>12280</td>
<td>4476 x 2666</td>
</tr>
<tr>
<td>HILL</td>
<td>1030</td>
<td>456 x 238</td>
</tr>
<tr>
<td>JPEG</td>
<td>2376</td>
<td>1808 x 852</td>
</tr>
<tr>
<td>RSAREF</td>
<td>16684</td>
<td>17712 x 5921</td>
</tr>
<tr>
<td>RX</td>
<td>603</td>
<td>45 x 41</td>
</tr>
<tr>
<td>SET</td>
<td>5016</td>
<td>2816 x 1551</td>
</tr>
</tbody>
</table>

Table 5.1 Summary of examples.

Instruments' TMS320C25 optimizing compiler, and the size of the covering matrix that was generated for solving the covering problem.

RX and AIPINT are embedded state machine controller routines. SET is a collection of bit manipulation routines used in a DSP application. BENCH is a controller for a disk cache. DXFTOHSH is a program that converts graphics from one format to another.

JPEG is an implementation of the JPEG image compression algorithm. COMPRESS and GZIP consist of core routines (i.e., without I/O) of the UNIX™ compress(1) and the GNU gzip programs, respectively.

Finally, HILL, GNUCRYPT, and RSAREF are data encryption programs. HILL is an encryption scheme based on matrix multiplication. GNUCRYPT, from the GNU C Library, uses the data encryption standard (DES) [DES 77]. RSAREF is an implementation of the RSA public-key cryptosystem [Rivest 78].

We have performed the experiments varying the size of the pointer, P. Specifically, we used P = 1 and P = 2 for both methods, and we show the results in Tables 5.2 and 5.3 (pages 174–175). The columns “Compr.Size” and “Dict.Size” show the size of
the compressed code (i.e., the skeleton) and the size of the dictionary, respectively. The column "Ratio" gives the ratios of the total size of the compressed code to that of the original code. And the column "CPU" gives the CPU time in seconds (on a SparcStation 10) used by the compression algorithm. A large portion of the CPU time was spent constructing the covering matrix—this was based on a first implementation, and optimizing this implementation will greatly improve the efficiency.

Figure 5-9 compares the two methods with different values of $P$. Not surprisingly, for a fixed value of $P$, Method II outperforms Method I, by approximately 2%–5%. Also, it is clear that the size of the pointer has a great impact on the ratio.

Using the cache-redesign method, Wolfe et al. were able to achieve apparently better compression ratios [Wolfe 92] than we have here, mainly because the instructions are treated as a collection of bits rather than as a collection of many different symbols. However, Wolfe et al. did not show how much hardware was required to implement the cache—the size of extra hardware might offset the savings gained in code size. Using Method I we are able to achieve code size reduction on existing CPUs without requiring any hardware modification. Method II requires a small number of gates to be added to the processor, and possibly some modification of the controller. It is also important to observe the impact of pointer size on the effectiveness of compression. Thus, one must take this factor into account when one designs a processor that has architectural support for Method II.

5.9 Performance Considerations

The code-compression methodology presented in the previous sections trades off speed for size. We now consider the quantitative impact of compression on performance. According to the 90/10 Locality Rule [Patterson 90, pages 11–12], in a typical program 10% of its code accounts for most (about 90%) of the instructions executed at runtime. Thus, we may reasonably expect good compression ratios without incurring much performance penalty by compressing only the "90%" of the program that is
<table>
<thead>
<tr>
<th>Example</th>
<th>Compr.Size</th>
<th>Dict.Size</th>
<th>Total</th>
<th>Ratio</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIPINT</td>
<td>1028</td>
<td>188</td>
<td>1216</td>
<td>0.805</td>
<td>1.40s</td>
</tr>
<tr>
<td>BENCH</td>
<td>8862</td>
<td>952</td>
<td>9814</td>
<td>0.895</td>
<td>25.08s</td>
</tr>
<tr>
<td>COMPRESS</td>
<td>1956</td>
<td>197</td>
<td>2153</td>
<td>0.923</td>
<td>1.11s</td>
</tr>
<tr>
<td>DXFTOHSH</td>
<td>1010</td>
<td>142</td>
<td>1152</td>
<td>0.866</td>
<td>0.61s</td>
</tr>
<tr>
<td>GNUCRYPT</td>
<td>2662</td>
<td>513</td>
<td>3175</td>
<td>0.789</td>
<td>6.09s</td>
</tr>
<tr>
<td>GZIP</td>
<td>9511</td>
<td>1079</td>
<td>10590</td>
<td>0.862</td>
<td>35.95s</td>
</tr>
<tr>
<td>HILL</td>
<td>811</td>
<td>111</td>
<td>922</td>
<td>0.895</td>
<td>0.41s</td>
</tr>
<tr>
<td>JPEG</td>
<td>1756</td>
<td>295</td>
<td>2051</td>
<td>0.863</td>
<td>2.59s</td>
</tr>
<tr>
<td>RSAREF</td>
<td>9663</td>
<td>2567</td>
<td>12230</td>
<td>0.733</td>
<td>93.60s</td>
</tr>
<tr>
<td>RX</td>
<td>525</td>
<td>36</td>
<td>561</td>
<td>0.930</td>
<td>0.09s</td>
</tr>
<tr>
<td>SET</td>
<td>3615</td>
<td>554</td>
<td>4169</td>
<td>0.831</td>
<td>8.28s</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>Example</th>
<th>Compr.Size</th>
<th>Dict.Size</th>
<th>Total</th>
<th>Ratio</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIPINT</td>
<td>1188</td>
<td>162</td>
<td>1350</td>
<td>0.894</td>
<td>1.25s</td>
</tr>
<tr>
<td>BENCH</td>
<td>9825</td>
<td>550</td>
<td>10375</td>
<td>0.946</td>
<td>24.14s</td>
</tr>
<tr>
<td>COMPRESS</td>
<td>2131</td>
<td>124</td>
<td>2255</td>
<td>0.967</td>
<td>1.14s</td>
</tr>
<tr>
<td>DXFTOHSH</td>
<td>1128</td>
<td>101</td>
<td>1229</td>
<td>0.924</td>
<td>0.64s</td>
</tr>
<tr>
<td>GNUCRYPT</td>
<td>3094</td>
<td>383</td>
<td>3477</td>
<td>0.864</td>
<td>5.44s</td>
</tr>
<tr>
<td>GZIP</td>
<td>10665</td>
<td>726</td>
<td>11391</td>
<td>0.928</td>
<td>36.15s</td>
</tr>
<tr>
<td>HILL</td>
<td>930</td>
<td>52</td>
<td>982</td>
<td>0.953</td>
<td>0.41s</td>
</tr>
<tr>
<td>JPEG</td>
<td>2013</td>
<td>198</td>
<td>2211</td>
<td>0.931</td>
<td>3.07s</td>
</tr>
<tr>
<td>RSAREF</td>
<td>11384</td>
<td>2077</td>
<td>13461</td>
<td>0.807</td>
<td>91.94s</td>
</tr>
<tr>
<td>RX</td>
<td>556</td>
<td>31</td>
<td>587</td>
<td>0.973</td>
<td>0.10s</td>
</tr>
<tr>
<td>SET</td>
<td>4199</td>
<td>379</td>
<td>4578</td>
<td>0.913</td>
<td>8.28s</td>
</tr>
</tbody>
</table>

(b)

Table 5.2 Results using Method I. (a) $P = 1$ and (b) $P = 2$. 
### Table 5.3 Results using Method II. (a) $P = 1$ and (b) $P = 2$.}

(a) |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td>Compr.Size</td>
<td>Dict.Size</td>
<td>Total</td>
<td>Ratio</td>
<td>CPU</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------</td>
<td>-----------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>AIPINT</td>
<td>976</td>
<td>156</td>
<td>1132</td>
<td>0.750</td>
<td>1.34s</td>
</tr>
<tr>
<td>BENCH</td>
<td>8608</td>
<td>885</td>
<td>9493</td>
<td>0.866</td>
<td>24.19s</td>
</tr>
<tr>
<td>COMPRESS</td>
<td>1905</td>
<td>194</td>
<td>2099</td>
<td>0.900</td>
<td>1.12s</td>
</tr>
<tr>
<td>DXFTOHSH</td>
<td>1007</td>
<td>100</td>
<td>1107</td>
<td>0.832</td>
<td>0.65s</td>
</tr>
<tr>
<td>GNUCRYPT</td>
<td>2584</td>
<td>427</td>
<td>3011</td>
<td>0.748</td>
<td>5.12s</td>
</tr>
<tr>
<td>GZIP</td>
<td>9244</td>
<td>933</td>
<td>10177</td>
<td>0.829</td>
<td>37.38s</td>
</tr>
<tr>
<td>HILL</td>
<td>791</td>
<td>92</td>
<td>883</td>
<td>0.857</td>
<td>0.44s</td>
</tr>
<tr>
<td>JPEG</td>
<td>1647</td>
<td>297</td>
<td>1944</td>
<td>0.818</td>
<td>2.84s</td>
</tr>
<tr>
<td>RSAREF</td>
<td>9234</td>
<td>2128</td>
<td>11362</td>
<td>0.681</td>
<td>97.97s</td>
</tr>
<tr>
<td>Rx</td>
<td>521</td>
<td>33</td>
<td>554</td>
<td>0.919</td>
<td>0.11s</td>
</tr>
<tr>
<td>SET</td>
<td>3508</td>
<td>441</td>
<td>3949</td>
<td>0.787</td>
<td>8.76s</td>
</tr>
</tbody>
</table>

(b) |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td>Compr.Size</td>
<td>Dict.Size</td>
<td>Total</td>
<td>Ratio</td>
<td>CPU</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------</td>
<td>-----------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>AIPINT</td>
<td>1132</td>
<td>153</td>
<td>1285</td>
<td>0.851</td>
<td>2.23s</td>
</tr>
<tr>
<td>BENCH</td>
<td>9568</td>
<td>616</td>
<td>10184</td>
<td>0.929</td>
<td>24.72s</td>
</tr>
<tr>
<td>COMPRESS</td>
<td>2074</td>
<td>147</td>
<td>2221</td>
<td>0.952</td>
<td>1.08s</td>
</tr>
<tr>
<td>DXFTOHSH</td>
<td>1119</td>
<td>80</td>
<td>1199</td>
<td>0.902</td>
<td>0.88s</td>
</tr>
<tr>
<td>GNUCRYPT</td>
<td>2991</td>
<td>365</td>
<td>3356</td>
<td>0.834</td>
<td>5.08s</td>
</tr>
<tr>
<td>GZIP</td>
<td>10434</td>
<td>676</td>
<td>11110</td>
<td>0.905</td>
<td>37.17s</td>
</tr>
<tr>
<td>HILL</td>
<td>893</td>
<td>62</td>
<td>955</td>
<td>0.927</td>
<td>0.41s</td>
</tr>
<tr>
<td>JPEG</td>
<td>1958</td>
<td>196</td>
<td>2154</td>
<td>0.907</td>
<td>3.62s</td>
</tr>
<tr>
<td>RSAREF</td>
<td>10827</td>
<td>1725</td>
<td>12552</td>
<td>0.752</td>
<td>94.90s</td>
</tr>
<tr>
<td>Rx</td>
<td>556</td>
<td>26</td>
<td>582</td>
<td>0.965</td>
<td>0.10s</td>
</tr>
<tr>
<td>SET</td>
<td>4071</td>
<td>345</td>
<td>4316</td>
<td>0.880</td>
<td>8.86s</td>
</tr>
</tbody>
</table>
Figure 5-9  Comparison of the Methods I and II.
least frequently executed and leave the "10%" as it is. (We use quotes around 90% and 10% because these numbers obviously vary among programs and according to the user's criteria.)

Indeed, for the examples in Section 5.8, we performed a different experiment, in which we first profiled the execution of the programs, and compressed only the noncritical "90%" of the code. We found that the overall compression ratio was degraded by about 2%-3% (with respect to the size of the original program). Although the execution time of the "90%" was increased by 15%-17%, the performance of the entire program was only slightly hurt, by approximately 1%-2%, since the programs spend most of their time in the "10%" that was not compressed.

5.9.1 Extension of Covering Formulation for Performance

As an alternative to compressing the least frequently executed parts (presumed to constitute a major portion) of a program, we may augment the covering formulation of Section 5.6 by taking into account the performance impact of accessing the dictionary. The modified objective function is a linear combination of the size-costs of the substitutions, the entries, and the cycle-penalties of the substitutions.

Given a program, each basic block $n$ in the program has an execution cycle count $E_n$. The program cycle count $T$ is a linear sum of the basic block cycle counts weighted by the expected number of invocations of each basic block. The expected number of invocations of each basic block can be obtained by using profiling tools, or by assigning probabilities to control-flow edges and solving a set of linear equations derived from the structure of the control-flow graph. For every access to the dictionary, we will add $p$ cycles to the execution cycle count $E_n$ of the basic block that accesses the dictionary. The program cycle count $T$ is thus increased. The value of $p$ is 2 for Method I and 1 for Method II when applied to the TMS320C25 processor.

To the covering problem of Section 5.6.2 we will add a third set of variables and a third set of clauses to the covering matrix of Section 5.6.2 to quantify the performance degradation of dictionary accesses.
For each \( q_{ij} \) a corresponding variable \( p_{ij} \) is added. The covering matrix is then augmented with the following clauses:

3. \((q_{ij} + p_{ij})\) for each substitution \((i, j)\).

If \( q_{ij} \) is substituted then \( q_{ij} = 0 \), and this means that \( p_{ij} \) has to be 1 to satisfy the above clause. The performance cost of the dictionary access due to substituting \( q_{ij} \) is \( \text{cost}(p_{ij}) \) which is equal to the increase in total cycle count \( T \) of the program due to this dictionary access. The cost can be computed for each \( q_{ij} \) as described at the beginning of this section, since we know which basic block \( q_{ij} \) belongs to.

We define the performance cost \( C_{\text{perf}} \) as

\[
C_{\text{perf}} = \sum_{p_{ij} \neq 0} \text{cost}(p_{ij}), \quad (5.8)
\]

and total cost \( C_{\text{total}} \) as

\[
C_{\text{total}} = C_{\text{size}} + W \cdot C_{\text{perf}}, \quad (5.9)
\]

where \( W \) is a user-specified parameter that controls the performance of the compacted program. Increasing \( W \) will reduce dictionary accesses that cause a large increase in cycle count. It is also easy, should circumstances so require, to weigh each \( p_{ij} \) differently by multiplying the cost of \( p_{ij} \) with its own multiplier \( w_{ij} \) (in contrast to using a single multiplier \( W \)). For critical sections of the program upon which compression should not be performed, we can either use large \( w_{ij} \)'s or remove from the covering matrix the \( q_{ij} \)'s (which is equivalent to assigning 1 to them) and the clauses that contain them.

### 5.10 Summary and Future Research

In this chapter we have presented a framework for the minimization of code size in VLSI systems containing embedded digital signal processors. Our methods are based
on data compression techniques. They offer different hardware/software design options and have different performance characteristics. The automatic minimization of code size relieves embedded system programmers from worrying about making programs sufficiently small, and thus allows them to enjoy the advantages of programming in high-level languages.

Several aspects of the current compression algorithm can be further improved. First, the current implementation observes boundaries of basic blocks. As we have noted in Section 5.5, Method I can accommodate extended blocks. Extended blocks pose a more difficult problem than basic blocks, because, while the set of basic blocks in a procedure is typically small, the number of extended blocks can be exponential with respect to the number of instructions. Basic blocks are maximal with respect to well-defined boundaries, but a maximal extended block could be the entire procedure. Future work involves developing efficient algorithms to identify and manage potentially useful extended blocks.

At a higher level of compilation, the code generator can assist the code compression process by generating assembly code which is potentially more compressible. For example, by permuting register or frame-relative offset assignment to variables, the code generator may be able to generate more isomorphic extended blocks. This is a combinatorial optimization problem, because while permutation can create new opportunities, at the same time it can cause other opportunities to disappear. This problem can be formulated more precisely and heuristics can be developed for solving it. Information from the source code (such as the use of macros) and the intermediate forms will provide "hints" for arriving at good solutions. It is possible to apply our techniques also to intermediate forms; this will require the intermediate form to support the semantics of out-of-scope jumps.

Interaction with existing optimizations for performance also merits further study (the phase-ordering and phase-coupling problem). For example, if heavy optimizations are performed before extraction of common sequences, opportunities for extracting may disappear because instances of the sequence in different contexts may be changed
differently. On the other hand, if we extract common sequences first, there may be fewer opportunities for other optimizations.
Chapter 6

Conclusion

The ever-increasing complexity of electronic systems has prompted the computer-aided design community to develop new design methodologies in order to cost-effectively exploit the high level of integration provided by deep submicron technology. The trend towards designs containing both hardware and software components is becoming clear—microcontrollers and fixed-point digital signal processors are increasingly being used as embedded core processors in single-chip heterogeneous systems. As a result, high-level language compilers for embedded software have become an essential component in the hardware designer's tool-box.

We have argued for the importance of code size, tantamount to that of performance, in the context of embedded software. It is now essential to produce code of the highest quality that is achievable in a reasonable amount of time, because software is no longer merely code, but will eventually become part of the chip and will be produced in large volumes. To this end, the entire suite of classical compiler optimization techniques, as well as new techniques, is required. Furthermore, the availability of longer compilation time motivates lifting the typical $O(n^2)$ limit on the running time of the algorithms to solve these optimization problems. This often entails problem formulations of higher complexity that consider more factors simultaneously (e.g., the set and binate covering problems with which some of our optimization problems are modeled).
In this thesis we examined several problems in code generation and optimization for fixed-point digital signal processors, which by and large have irregular data-paths and limited addressing capabilities. We have formulated and presented solutions for some practical problems in instruction selection, storage assignment, and code compression. In addition, we describe several other optimizations of a smaller scale in Appendix B.

We formulated the instruction selection problem as a binate covering problem for which effective heuristics and efficient branch-and-bound techniques are known. We first use a DAG-covering formulation, similar to that used in technology mapping, to discover complex patterns with fanout on internal vertices. Once the DAG is covered, a second binate covering problem is created to determine a partial schedule and data transfers. A theory of optimal code generation for noncommutative one-register machines, also based on binate covering, was presented that takes into account the commutativity of individual operators instead of assuming the commutativity, or the absence thereof, in the machine. The binate covering formulation can be extended to handle other scheduling-related optimizations such as the mode optimization problem.

The offset assignment problem finds application in architectures with limited addressing capabilities; most fixed-point DSPs fall in this category. We have shown that for the simple offset assignment problem, which involves the use of a single address register, the decision problem is NP-complete. Nonetheless, by casting the SOA problem into the MWPC problem and proving the equivalence of the two, we have designed a simple greedy heuristic algorithm that performs well in practice. We have also extended our formulation to use multiple address registers and presented an algorithm that utilizes the procedure for the SOA problem. Because of the use of SOA as a subproblem, the improvement in time complexity over previous work proves to be significant.

To further reduce the size of the object code, we apply code compression techniques. We used a dictionary-based compression model that requires little or no hardware support and yields moderate compression ratios. We showed that the sub-
stitution and dictionary generation problems need to be solved simultaneously, and these problems can be naturally modeled as a set covering problem. In addition, it is straightforward to incorporate impacts on performance into the set covering formulation. Thus, even though the primary goal of code compression is to reduce code size, the set-covering formulation allows the user to specify size–performance trade-offs in a uniform manner.

6.1 Future Work

In the summary sections of Chapters 3–5 we have discussed avenues for future research pertinent to each code generation or optimization problem. In this final chapter, therefore, we will discuss the general goal towards which our compiler work will continue to evolve.

One of the greatest challenges of the present research work is to achieve a higher degree of retargetability while retaining the effectiveness of the optimization techniques. As we have noted in Chapter 2, we approach the problem of retargetable code generation from the standpoint of optimizations, and we believe that retargetability should by no means be gained at the expense of code quality. Therefore, in our future work we will continue to focus on the following issues:

1. **Automatically generating optimizers.** Some optimizations need to be tailored for specific machines or specific contexts, but the underlying formulation and algorithms remain similar. For example, Tjiang developed a tool for the automatic generation of data-flow analyzers [Tjiang 92]. Various data-flow problems can be described with the language SHARLI, and an underlying engine, with the same set of algorithms, is used to solve the problems. We may apply the same principles to the problems we have described previously: storage assignment, static allocation (Appendix B.2), and global mode optimization (Appendix B.4). Also, the approaches cited in Section 2.2 for the derivation of machine-specific optimizations are attractive.
2. **Program representation.** In a logic synthesis environment there is typically a unified representation that is used throughout optimization stages, e.g., a net-list of gates. This allows for the use of scripts with which the user may experiment with different orderings of transformations. Similarly, an intermediate representation for programs that allows for more-flexible cooperation of program optimizations is valuable for a compiler. As our studies on optimizations progress, we will be better equipped to make judgments on the requirements for designing such a representation.

3. **Impact of compiler optimizations on ASIP design.** By studying optimization problems we gain insight into what architectural features are amenable to compilers. This insight will, in turn, serve as a guide for the design of ASIPs, which should take compiler support into consideration when making architectural decisions. There has been some work along this direction, e.g., that of Chow et al. [P Chow 94].
Appendix A

Covering Problems

This appendix describes the set (unate) and binate covering problems that we encountered in Chapter 5 for code compression and Chapter 3 for code generation. Covering problems have also been extensively used in other areas of computer-aided design of digital circuits. For example, they appear in several stages of logic synthesis, including two-level logic minimization and DAG covering for technology mapping [Rudell 89].

Covering problems are well-known intractable problems and have received considerable attention from researchers. Exact solutions have been given in [Grasselli 65] and [Brayton 89] using branch-and-bound search. Heuristic methods have also been proposed, e.g., [Grasselli 65], [Gimpel 67], and [Rudell 89]. Recently, Coudert and Madre discovered new pruning conditions that have substantially improved the efficiency of the search without compromising optimality [Coudert 95]. The resulting solver, called SCHERZO, is able to find exact solutions to difficult instances 10–100 times faster than earlier methods.

We shall only concisely introduce the formulation of the two variants of covering problems, and describe the approaches to solving these problems, so that the reader may gain an intuitive understanding of the objectives of these problems and how they are applied to code generation and code compression. For full treatment of covering problems, the reader is referred to the above-cited references.
A.1 Set Covering

Let $X$ be a set of variables and $Y$ be a subset of $2^X$. An element $y$ of $Y$ is said to cover an element $x$ of $X$ if $x \in y$. With each element $y$ of $Y$ is associated a nonnegative cost $\text{cost}(y)$. The set covering (or unate covering) problem is to select a subset $Z$ of $Y$ with the smallest cost such that every element of $X$ is covered by at least one element of $Z$.

Consider the following example. Let

$$X = \{x_1, x_2, x_3, x_4, x_5\}$$
$$Y = \{y_1, y_2, y_3, y_4\}$$

where

$$y_1 = \{x_1, x_2, x_5\}$$
$$y_2 = \{x_1, x_3, x_4\}$$
$$y_3 = \{x_2, x_4, x_5\}$$
$$y_4 = \{x_2, x_3, x_4\}$$

with

$$\text{cost}(y_1) = 2$$
$$\text{cost}(y_2) = 3$$
$$\text{cost}(y_3) = 1$$
$$\text{cost}(y_4) = 3.$$ 

Two solutions (among others) for the covering problem are $\{y_1, y_4\}$ and $\{y_2, y_3\}$. The cost of the first solution is 5, whereas that of the second is 4. Thus the second solution is preferred.

An alternative way to describe the covering problem is to write a set of Boolean expressions that indicate how each $x$ might be covered. Let $y_i$ also denote the Boolean variable if $y_i$ is selected in a solution. In order to cover $x_1$, we must select at least one of $y_1$ and $y_2$; thus we write $(y_1 + y_2)$. Similarly, to cover the other elements of $X$ we write the following expressions, each of which must be satisfied, in addition to $(y_1 + y_2)$:

$$y_1 + y_3 + y_4$$
$$y_2 + y_4$$
$$y_2 + y_3 + y_4$$
$$y_1 + y_3.$$
A.1 SET COVERING

![Table]

<table>
<thead>
<tr>
<th></th>
<th>$y_1$</th>
<th>$y_2$</th>
<th>$y_3$</th>
<th>$y_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_1 + y_2$</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y_1 + y_3 + y_4$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$y_2 + y_4$</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$y_2 + y_3 + y_4$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$y_1 + y_3$</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**Figure A-1** A set covering matrix. An entry $(i,j)$ is marked 1 if column $j$ covers row $i$. The goal of the covering problem is to select a set of columns of minimum cost such that there is at least a 1 in every row.

The set of Boolean expressions comprises a conjunction of disjunctions. Moreover, each variable in a disjunction appears in the true form only, hence the name *unate*. The set covering problem finds application whenever we need to satisfy, with minimum cost, a set of clauses in which all variables appear in the true form.

For the purpose of implementation a covering problem is usually represented as a covering matrix. Figure A-1 shows the covering matrix for the above example. An entry $(i,j)$ in the matrix is marked 1 if column $j$ covers row $i$. Thus, in terms of the covering matrix, the goal of the covering problem is to select a set of columns of minimum cost such that there is at least a 1 in every row.

We can apply several reduction techniques to simplify a covering matrix. We will briefly describe the notions of essentiality and dominance. The reader is referred to [Gimpel 67] and [Rudell 89] for additional reduction techniques.

A column $j$ is *essential* if there exists some row $i$ such that column $j$ is the only column that covers row $i$. Clearly, the corresponding variable $y_j$ must be set to 1, and we can remove all rows that are covered by column $j$. A row $i$ is said to be *dominated* by another row $i'$ if covering row $i'$ necessarily results in covering row $i$. In the above example, the row $(y_2 + y_3 + y_4)$ is dominated by the row $(y_2 + y_4)$. A column $j$ is said to be *dominated* by another column $j'$ if column $j$ covers a subset of the rows covered by column $j'$ and $\text{cost}(y'_j) \leq \text{cost}(y_j)$. We can repeatedly remove
essential columns, dominated rows, and dominated columns to arrive at a smaller matrix, called the cyclic core of the covering matrix [Quine 59].

A.2 Binate Covering

As in unate covering, the binate covering problem is one of finding a minimum cost set of variables that satisfy a set of clauses. However, in contrast to unate covering in which all variables appear in the true form, in the binate covering problem clauses are allowed to consist of variables appearing in both true and complemented forms.

For example, consider the following set of clauses:

\[
egin{align*}
s_1 &= y_1 + y_2 + y_3 + \overline{y_4} \\
s_2 &= y_1 + y_3 + y_4 + y_5 \\
s_3 &= y_2 + \overline{y_3} + y_5 + y_6 \\
s_4 &= y_2 + y_3 + y_4 + \overline{y_6}
\end{align*}
\]

and let the cost function \text{cost} be:

\[
\begin{align*}
cost(y_1) &= 4 \\
cost(y_2) &= 2 \\
cost(y_3) &= 1 \\
cost(y_4) &= 1 \\
cost(y_5) &= 3 \\
cost(y_6) &= 1.
\end{align*}
\]

The minimum-cost assignment is to set the variables \(y_3\) and \(y_6\) to 1, and every other variable to 0. The cost of this assignment is 2. Another possible solution is to set only \(y_5\) to 1. However, due to the cost function, this solution has a cost of 3. Thus the first solution is preferred.

A binate covering problem can be described as a covering matrix. Entry \((i,j)\) of the matrix is set to 1 if \(y_j \Rightarrow s_i\), to 0 if \(\overline{y_j} \Rightarrow s_i\), and to 2 (don't-care) otherwise. The covering matrix for the above example is shown in Figure A-2.

A binate covering problem can be easily transformed into a unate covering problem as follows. For each variable \(y_i\) we create an additional column labeled \(\overline{y_i}\), and an additional row labeled \(r_i\). We also create a row labeled \(d\). The purpose of rows \(r_i\) is to ensure that either \(y_i\) or \(\overline{y_i}\) is selected, and the purpose of column \(d\) is to ensure
that exactly one of $y_j$ or $\overline{y}_j$ is selected. The modified matrix will be a \textit{unate} covering matrix. The cost of each $\overline{y}_j$ and of $d$ is zero. For the example above, we show the modified matrix in Figure A-3 (page 190).

The modified matrix is derived as follows. If entry $(i,j)$ in the original matrix is 1, then we insert a 1 in the entry $(i,y_j)$ in the new matrix. If entry $(i,j)$ in the original matrix is 0, then we insert a 1 in the entry $(i,\overline{y}_j)$ in the new matrix. For each row $i$ in the original matrix, we also insert a 1 in entry $(i,d)$ of the new matrix. Each additional rows $r_j$ have 1s at columns $y_j$ and $\overline{y}_j$ as well.

Note that each row $r_j$ can be covered by $y_j$ and $\overline{y}_j$ only. This means that either $y_j$ or $\overline{y}_j$ must be selected. Now suppose we select $y_j$ thereby covering row $r_j$. Now column $\overline{y}_j$ is dominated by column $d$, and therefore can be eliminated. Similarly, if we had selected column $\overline{y}_j$, column $y_j$ can be eliminated because it is dominated by column $d$. Thus column $d$ ensures that (by virtue of the cost function) at most one of $y_j$ or $\overline{y}_j$ will be selected.

**A.3 Solving Covering Problems**

Set covering problems can be solved exactly using branch-and-bound procedures, or approximately using heuristic algorithms. The latter are usually non-backtracking or limited-backtracking versions of the former. In a typical branch-and-bound procedure, a column is selected at each iteration, and this column, along with the rows it covers,
<table>
<thead>
<tr>
<th></th>
<th>$y_1$</th>
<th>$y_1$</th>
<th>$y_2$</th>
<th>$y_3$</th>
<th>$y_4$</th>
<th>$y_4$</th>
<th>$y_5$</th>
<th>$y_5$</th>
<th>$y_6$</th>
<th>$y_6$</th>
<th>$d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_1 + y_2 + y_3 + y_4$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>$y_1 + y_3 + y_4 + y_5$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>$y_2 + y_3 + y_5 + y_6$</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y_2 + y_3 + y_4 + y_6$</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$r_1$</td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$r_2$</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$r_3$</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$r_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$r_5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$r_6$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure A-3 Transforming a binate covering problem into a set covering problem. New variables and new rows are created. We use the variables $r_j$ to require the selection of either $y_j$ or $\overline{y}_j$. The new variable $d$, along with the cost function, ensures that $y_j$ and $\overline{y}_j$ are not both selected.

is removed. Reductions such as essentiality and dominance are again applied to the matrix. A lower bound is estimated on the resulting matrix, and a decision to further proceed on this branch is made based on the comparison of the lower bound against the cost of the best solution found so far.

The choice of the column to remove at each iteration and the tightness of the lower-bound computation have great impact on the amount time for the branch-and-bound procedure to complete. Coudert and Madre give an excellent description of effective algorithms for selecting columns and lower-bound computation in [Coudert 95].
Appendix B

Other Optimizations

In addition to the traditional compiler optimizations and the optimizations described in Chapters 3–5 that are targeted to DSP architectures, several other optimizations can be used in conjunction therewith. These optimizations can be easily incorporated into our compiler framework. Although each of these optimizations, when applied separately, may improve the code quality by only a relatively small margin, they may cooperate to contribute more substantially and, therefore, should not be neglected. In Sections B.1–B.4, we will first briefly review interprocedural analysis, and then describe some optimizations that make use of the information gathered by the analysis.

B.1 Interprocedural Analysis and Optimizations

It has been widely recognized as good programming practice to use abstractions (both procedural and data) [Abelson 85]; hence, most software-development environments allow for the division of a large program into a number of files and for the separate compilation of each file. To achieve this, a distinct stage of compilation, linking, is necessary to resolve function calls and global variables.

When compiling a procedure independently of other procedures, however, the compiler must make pessimistic assumptions about its surroundings, thereby potentially introducing inefficiency in the compiled code. The main benefit of interprocedural
analysis is that it allows the compiler to gather more context information than would be available if procedures were compiled separately. The context information enables, among others, the following optimizations (in descending order of aggressiveness):

1. **Inline substitution.** The body of the callee is substituted at each call site. Although inlining eliminates overhead associated with the procedure call, it should be carefully controlled in order to prevent explosion in code size.

2. **Procedure cloning.** Copies of the callee are optimized for groups of call sites. For example, if a procedure is called with a constant \(a\) at some sites and another constant \(b\) at some other call sites, we may create two clones of the procedure, one optimized for constant \(a\) (e.g., via constant propagation [F Allen 72]) and the other optimized for constant \(b\).

3. **Global optimization enhanced by interprocedural data-flow information.** This has proven to be effective in removing spurious dependencies for parallel compilation. For scalar compilation, mixed results ranging from marginal to moderate improvements have been reported [Hall 91]. Global mode optimization to be described in Section B.4 can also naturally benefit from the information gathered by this analysis.

Hall presented a comprehensive investigation and experimentation of these interprocedural optimizations in [Hall 91]. The first two of these optimizations, however, often trade off size for performance. In the context of embedded software, especially, we will need to take special caution in applying these optimizations to prevent the program size from growing too large.

In interprocedural analysis and optimizations, an entire program is usually expressed by means of a call multigraph (or simply call graph). Each vertex in the call graph corresponds to a procedure in the program, and there is a directed edge from vertex \(u\) to vertex \(v\) for every site at which \(u\) may call \(v\). There may be more than one edge between two vertices since a procedure may call another procedure more
than once. Constructing a call graph in the absence of procedure-valued parameters is
trivial: only a single pass over the procedures and call sites is required. An efficient
algorithm for constructing the call multigraph in the general case is presented in
[Hall 92].

In addition to the aforesaid optimizations, the call graph provides information
for other types of optimizations as well. Three optimization problems relevant to the
context of our work are: static allocation of automatic variables, efficient use of the
hardware link register stack, and global mode optimization. We will discuss these relatively
simple yet useful optimizations in the sequel.

B.2 Static Allocation

Although it is rare for a DSP- or control-oriented program to contain recursive calls,
the C language does allow for recursion. Consequently, without directives from the
user the compiler must either assume that all procedures could potentially be called
recursively, or examine the entire program for further details. If we can determine
that a procedure is never called recursively (and therefore has at most one activation
at any point in the course of program execution), we may opt to statically allocate
the automatic variables of the procedure.

One of the advantages of static allocation is that there is no need for a run-time
stack; thus the overhead associated with setting up the run-time stack dissipates. The
greater benefit of static allocation, however, manifests itself in architectures whose
instruction sets are such that immediate addressing mode is encoded in page-offset
style. For example, in the TMS320C25 architecture, an effective 16-bit address in
the immediate addressing mode is constructed by concatenating the 9-bit DP (data
page number) register with the least significant seven bits from the instruction word.
When it is known to the compiler that the physical location of a variable name
does not change during program execution, as is the case when the variable is
statically allocated, the compiler may use immediate addressing mode, with symbolic
addresses that will be resolved during the final code generation phase, instead of address register indirect mode. If using immediate addressing does not incur the cost of an extra instruction word, as is the case with the TMS320C25 architecture, then static allocation effectively subsumes the storage assignment problem of Chapter 4 and uses fewer resources—the address registers may be used for striding through arrays, for example.

With such a style of effective address formation, the only costs in accessing the memory are due to the instructions that set the page number. If the size of a memory page is sufficiently large, then page switches are unlikely to occur frequently. Also, during the final code generation phase we may resolve symbolic addresses in such a way that the number of page switches is minimized. Note that the page number is also a mode variable, and the same principles of Section B.4 apply. The experiments with our prototype TMS320C25 code generator indicate that with static allocation we may reach within a few instructions (for each procedure) of the lower bounds exhibited in Table 4.1 (page 133). These results, however, do not make the offset assignment problems dispensable. In many other architectures in which immediate addressing requires an additional instruction word and is therefore more expensive than register-indirect mode, it is still beneficial to use address registers with general offset assignment, even if static allocation is used to reduce procedure call overhead.

To determine which procedures are eligible for static allocation, our compiler finds all strongly connected components in the call graph. A strongly connected component (SCC) of a directed graph is an equivalence class of vertices of the graph under the relation of mutual reachability. An SCC is said to be trivial if it contains only one vertex without a self-loop. If a vertex belongs to a nontrivial SCC, then the corresponding procedure may have more than one activation during the course of program execution. Because each distinct activation must have its own frame, the variables of the procedure cannot be statically allocated. Conversely, if a vertex belongs to a trivial SCC, then it may have at most one activation at any time. Therefore, all activations of the procedure may share the same data space, and
Figure B-1  Strongly connected components in a call graph. Procedures $F_6$ and $F_{10}$ form a nontrivial strongly connected component; they are mutually recursive. Every other procedure is a trivial SCC by itself and is therefore nonrecursive; their automatic variables may be statically allocated.
we can allocate storage for variables during compilation time (static allocation). For instance, consider the call graph shown in Figure B-1. Procedures \( F_6 \) and \( F_{10} \) form a nontrivial strongly connected component; they are mutually recursive. Every other procedure is nonrecursive, and their automatic variables may be statically allocated.

### B.3 Efficient Use of the Hardware Stack

A number of architectures provide a hardware link register stack, to which the return address is automatically pushed when a subroutine call is initiated, and from which the return address is popped upon subroutine exit. For example, the TMS320C25 provides an 8-word stack, and the DSP56000 provides a 15-word stack.

If the stack contains sufficient space for procedure calls, it is perfectly acceptable to leave return addresses on the stack. However, if the call graph is too deep or if it contains recursive procedures, care must be taken to pop return addresses from the stacks into the memory after entry to a procedure and push them back to the stack before exit, in order to prevent stack overflow. The placement of these pushes and pops affects the number of instructions at run-time, and can be optimized according to the execution frequencies of the procedures in the program. Like static allocation, this optimization requires program-wide analysis.

We can express the optimization problem in terms of a very simple integer linear program (ILP) as follows. Let \( G \) be the call graph. For each vertex \( n \) of \( G \) we associate two integer variables \( p_n \) and \( q_n \). The variable \( p_n \) is the maximum number of return addresses that are allowed on the link register stack when procedure \( n \) is first entered, including the address freshly pushed onto the stack by the caller of \( n \). The variable \( q_n \) is the number of return addresses that remain on the link register stack after possibly popping one or more return addresses from the stack and storing them in the activation frame (in the main memory). Thus, \( q_n = p_n \) if procedure \( n \) does not save any return address into the main memory, and \( q_n < p_n \) otherwise. The difference
between \( q_n \) and \( p_n \) is the number of return addresses to be saved and restored by procedure \( n \), usually but not always one.

Our goal is to determine \( p_n \) and \( q_n \) such that the total number of operations involved in saving and restoring return addresses is minimized. Let \( \text{freq}(n) \) be the expected execution frequency of procedure \( n \), and let \( H \) denote the depth of the hardware link register stack. The optimization problem is therefore:

\[
\min \sum_{n \in G} \text{freq}(n) \cdot (p_n - q_n) \tag{B.1}
\]

subject to the following constraints, for each vertex \( n \in G \):

\[
p_m - q_n \geq 1, \text{ for each successor } m \text{ of } n \tag{B.2}
\]

\[
p_n \geq 1 \tag{B.3}
\]

\[
p_n \leq H \tag{B.4}
\]

\[
p_n - q_n = 1, \text{ if } n \text{ belongs to a nontrivial SCC} \tag{B.5}
\]

\[
p_n - q_n \geq 0, \text{ if } n \text{ belongs to a trivial SCC} \tag{B.6}
\]

Constraint (B.2) expresses the requirement that immediately after entry into \( m \) from \( n \), the number of return addresses (which is equal to \((q_n+1)\)) must not be greater than that allowed in procedure \( m \). This has to hold for all successors \( m \) of \( n \). Constraints (B.3) and (B.4) require that when a procedure call is initiated, the hardware stack must contain at least one address (which is automatically pushed to the stack by the instruction), and the stack must not overflow. Constraint (B.5) requires the procedure to save a return address if the procedure belongs to a mutually recursive set. Otherwise, it is for the ILP to decide which procedures have to save. Clearly, \( q_n \) must not be greater than \( p_n \) (Constraint (B.6)).

Figure B-2 shows the same example call graph, assuming the depth of the hardware stack is four. The number shown in italics next to each vertex \( F_n \) indicates the number of times the corresponding procedure is called, and the pair of numbers gives the values of the optimal \( p_n \) and \( q_n \) for this call graph. Note that since \( F_6 \) and \( F_{10} \) are mutually recursive, they must save the return address of their caller. Also
Figure B-2 Using the hardware stack efficiently. The number shown in italics next to each vertex denotes the frequency the corresponding procedure is invoked. Optimal values of the $p_n$ and $q_n$ for each procedure are shown in parentheses.
note that, due to its low execution frequency, \( F_2 \) is requested to save two return addresses to reserve more space on the stack for other procedures deeper in the call graph.

It is trivial to show that the constraint matrix of this ILP is totally unimodular (Theorem 13.3 of [Papadimitriou 82]). A matrix whose entries are \(-1, 0, \) or \(1\), is said to be totally unimodular if the determinant of every nonsingular square submatrix is \(-1\) or \(1\). If the constraint matrix of an ILP is totally unimodular, then the ILP may be solved as a linear program without the integer constraints, and the solution will still be integral. Hence, this optimization may be solved very efficiently even for large programs.

## B.4 Global Mode Optimization

The mode optimization technique described in Section 3.5.1 tackles the problem of local mode optimization, i.e., optimization within a basic block. With this approach, however, we will need to make pessimistic assumptions about the state of the machine at entries of basic blocks, since on different control-flow edges entering a basic block a mode variable may assume different values. It is possible to examine the program from a global (intraprocedurally or interprocedurally) perspective and eliminate redundant mode-setting instructions.

Mode variables, though certainly machine-dependent, behave very much like user-defined variables: If we treat mode-setting instructions as definitions of a mode variable and instructions affected by a mode variable as uses, we may then use the standard reaching-definition and liveness analyses (see [Hecht 77] and [Aho 86]) to determine the definition–use characteristics of each mode variable. Partial redundancy elimination (see [Morel 79] and [Knoop 95]), which makes use of this information, can be readily applied to the mode variables. Interprocedural information may also be used for moving mode-setting instructions across procedural boundaries to less frequently executed procedures or removing them if permitted.
Figure B-3  Partial redundancy elimination applied to mode variables. The symbols out and req denote the value of mode variable upon exit of a basic block and the value the first use of the mode variable requires. (a) The mode signed is totally redundant on entry to $n_2$. There is no need to set the mode. (b) The mode signed is partially redundant on entry to $n_2$. We only need to set the variable to signed at the end of $n_5$, which is executed less frequently than $n_2$. 
Figure B-3 shows examples of the application of partial redundancy elimination to the sign-extension mode variable. In part (a), upon exit of both basic blocks $n_1$ and $n_5$, the sign-extension mode variable is found to have the value $signed$. Since the first use of this mode variable in $n_2$ requires the value $signed$, the sign-extension mode is totally redundant on entry to $n_2$, and there is no need to set the mode at the beginning of $n_2$. In part (b), the signed-extension mode variable has the value $unsigned$ upon exit of $n_5$, and is partially redundant with respect to $n_2$. Hence, we only need to set the mode variable to $signed$ at the end of $n_5$ instead of the beginning of $n_2$. Since $n_5$ is executed less frequently than $n_2$, by so doing we improve the performance of the program.

This optimization can be very easily parameterized—the underlying formulation and algorithm can be shared across a range of architectures that use mode variables. It can be driven by a description of the mode variables and of the instructions that define or use these variables.
Bibliography


