Register Allocation Issues in Embedded Code Generation

David J. Kolson

Technical Report UCI-ICS 98-24
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425

June 12, 1998

UNIVERSITY OF CALIFORNIA
Irvine

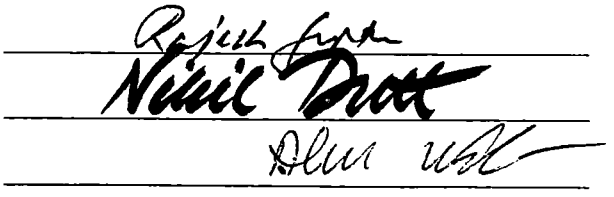
Register Allocation Issues in Embedded Code Generation

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in
Information and Computer Science
by
David J. Kolson

Committee in charge:
Professor Alexandru Nicolau, Chair
Professor Nikil Dutt
Professor Rajesh Gupta

1998

The dissertation of David J. Kolson is approved,
and is acceptable in quality and form for
publication on microfilm:



Committee Chair

University of California, Irvine

1998

Dedication

To my family and friends
who supported me through a long education,
and believed in me, even when I did not,
my deepest and most sincere gratitude.

Contents

List of Illustrations vii

List of Tables ix

Acknowledgement x

Curriculum Vitae xi

Abstract xiii

Chapter 1 Introduction **1**
1.1 Code Generation and the Role of Register Allocation 3
1.1.1 Optimizing Compilers 4
1.1.2 Parallelizing Compilers 7
1.1.3 Embedded Compilers 8
1.2 Motivation for Embedded Compiler Complexity 10
1.3 Scope of Thesis 11

Chapter 2 Related Work **13**
2.1 Register Allocation and Assignment 13
2.1.1 Graph Coloring 14
2.1.2 Interval Graphs 16
2.1.3 Optimal Register Allocation 16
2.1.4 Register Allocation and Program Transformation 16
2.2 Memory-Related Transformations 18
2.3 Memory Access Models 20
2.4 Memory Structure Synthesis 20

Chapter 3 Phases of Compilation **22**
3.1 A Parallelizing Compiler 22
3.2 Program Model 23
3.3 Scheduling 23

Chapter 4 Eliminating Redundant Memory Accesses 28
4.1 Introductory Example 30
4.2 Detecting Redundancy 31
4.2.1 Symbolic Expressions 32
4.2.2 Memory Disambiguation 35
4.3 Eliminating Redundancy 36
4.3.1 Removing Invariants 38
4.3.2 Load-After-Load Optimization 40
4.3.3 Load-After-Store Optimization 41
4.3.4 A Note on Move Instructions 42
4.3.5 An Example 44
4.4 Effects of Redundancy Elimination on Register Allocation 45
4.4.1 Examining the Effects of Optimization 45
4.4.2 An Enhanced Redundant Elimination Algorithm 49

Chapter 5 Integrating Program Transformations 54
5.1 Introductory Example 55
5.2 Overview of Incremental Tree Height Reduction 56
5.2.1 Previous Work 56
5.2.2 Definitions 58
5.2.3 An Incremental THR Algorithm 58
5.3 Integrating Transformations 60
5.3.1 The META-Transformation 60
5.3.2 Heuristic META-Transformation 62
5.3.3 An Example 67
5.4 Effects of META-Transformation on Register Allocation 69
5.4.1 Examining the Effects of Optimization 69
5.4.2 An Enhanced META-Transformation 75

Chapter 6 Generalizing Copy Elimination 77
6.1 Introductory Example 79
6.2 Eliminating Copy Instructions 81
6.2.1 Definitions 83
6.2.2 An Algorithm for Copy Elimination 83
6.2.3 Determining Minimal Number of Unrollings 90
6.2.4 Heuristic Copy Elimination 91
6.2.5 An Example 92
6.3 Effects of Copy Elimination on Register Allocation 95
6.3.1 Examining the Effects of Optimization 96
6.3.2 An Enhanced Copy Elimination Algorithm 99

Chapter 7 Allocating Registers to Loops 101
7.1 Optimal Allocation in Basic Blocks 102
7.2 Extending the Basic Block Algorithm to Loops 106
7.2.1 Terminology 107
7.2.2 An Algorithm for Loop Register Allocation 107
7.2.3 Heuristic Pruning 108
7.3 Convergence and Optimality of the Loop Algorithm 110
7.3.1 Convergence 112
7.3.2 Optimality 112
7.4 Extending the Loop Algorithm to Distributed Memories 114
7.4.1 Adding Register Classes to the Model 114
7.4.2 Extension to Special-Purpose Registers 116
7.4.3 Extension to Multiple Register Files 118
7.5 Comparing Loop Register Allocation with Other Approaches 120
7.5.1 Comparison with Graph Coloring 120
7.5.2 Comparison with Cyclic Interval Graphs 122
7.5.3 Comparison with the Basic Block Strategy 125

Chapter 8 Experimentation 129
8.1 Measuring Improvement 130
8.2 Experimental Set-up 131
8.3 Eliminating Redundancy 133
8.3.1 Observed Results 134
8.3.2 Analysis 139
8.4 Integrating Program Transformations 140
8.4.1 Observed Results 141
8.4.2 Analysis 143
8.5 Copy Elimination 143
8.5.1 Observed Results 143
8.5.2 Analysis 145
8.6 Combining Techniques 147
8.6.1 Observed Results 148
8.6.2 Analysis 149
8.7 Allocating Registers to Loops 149
8.7.1 Experimentation with a Consolidated Model 149
8.7.2 Experimentation with Distributed Register Files 159
8.8 Summary of Results 165

Chapter 9 Conclusions 166

Illustrations

1.1 Typical embedded system architecture. 2
1.2 Phases of an optimizing compiler. 6
1.3 Phases of a parallelizing compiler. 9
1.4 Available register organization. 10

3.1 Code Generation process. 23
3.2 Datapath for microcode macro example. 25

4.1 Removing redundant memory traffic. 29
4.2 An algorithm to build symbolic expressions. 33
4.3 Symbolic expression example. 34
4.4 An algorithm for redundant elimination. 37
4.5 An algorithm for loop invariant removal. 38
4.6 An algorithm for the load-after-load optimization. 40
4.7 An algorithm for the load-after-store optimization. 43
4.8 Redundancy elimination example. 44
4.9 Invariant load removal. 47
4.10 Invariant store removal. 48
4.11 Invariant load and store removal. 48
4.12 Multiple effects of invariant removal. 49
4.13 An algorithm for the load-after-store heuristic. 50
4.14 An algorithm for the loop invariant removal heuristic. 52

5.1 Example image convolution code. 55
5.2 Schedules for tree height reduced graphs. 57
5.3 Main procedure for incremental THR algorithm. 59
5.4 Methods of approach. 61
5.5 An algorithm for determining removal candidates. 65
5.6 Algorithm for propagating redundancy information. 66
5.7 The META-Transformation 68
5.8 Application of META-Transformation on low-pass filter. 70
5.9 Pseudo code for an incremental scheduler. 71
5.10 Traditional scheduling approach. 72
5.11 META-Transformation approach. 73

6.1 Introducing copy instructions into the code. 79

6.2 Loop code with copy instructions. 80
6.3 Unrolling loop code to eliminate copies. 82
6.4 An algorithm for copy elimination. 84
6.5 An algorithm for computing register mappings. 87
6.6 An algorithm for removing copies and updating register usages. . . 89
6.7 Loop template for example. 92
6.8 Unrolling loop code to eliminate copies. 93
6.9 Example code and register mappings during copy elimination. . . . 98
6.10 Example code and register mappings with annotations. 100

7.1 An optimal register assignment algorithm for basic blocks. 104
7.2 Building an assignment tree. 105
7.3 A loop register assignment algorithm. 109
7.4 Building a configuration graph from the assignment trees. 111
7.5 An algorithm to derive register classes. 115
7.6 Extending BB-Opt to special-purpose registers. 117
7.7 Extending BB-Opt to multiple register files. 119
7.8 Example for graph coloring comparison. 121
7.9 Solution for graph coloring comparison. 123
7.10 Example for cyclic interval graph coloring comparison. 124
7.11 A loop basic block allocated with BB-Opt. 127
7.12 A Loop-Opt allocated loop basic block with cost nine. 128

8.1 Architecture model for experimentation. 132
8.2 Average performance improvement with unlimited functional units. 138
8.3 Average performance improvement with functional unit constraints. 138
8.4 A simplified view of the TMX320C44. 161

Tables

6.1 Register mappings for the code of Figure 6.2. 88

8.1 A description of the redundancy elimination benchmark suite. . . . 133
8.2 Statistics for the redundancy elimination benchmark suite. 134
8.3 Results of experimentation with redundant elimination. 135
8.4 Results of experimentation with redundant elimination (con't). . . . 136
8.5 A description of the META-Transformation benchmark set. 140
8.6 Results of experimentation with META-Transformation. 142
8.7 A description of the copy elimination benchmark suite. 144
8.8 Results of experimentation with copy elimination. 144
8.9 Results of experimentation with heuristic copy elimination. 146
8.10 A description of the benchmark suite for all transformations. 147
8.11 Remaining number of copy instructions after parallelization. 147
8.12 Results of experimentation with all transformations. 148
8.13 A description of the benchmark suite for register allocation. 150
8.14 Spill costs for the two methods of matching register maps. 152
8.15 Results of loop register allocation algorithm. 154
8.16 Results of heuristic width restriction only. 156
8.17 Results of heuristic depth of two. 157
8.18 Results of heuristic depth of three. 158
8.19 Comparison of results between GCC and heuristic Loop-Opt. 160
8.20 Execution times of the various methods. 162
8.21 Basic block optimal vs. Loop optimal for the TMX320C44. 163
8.22 Comparison of loop assignments for the TMX320C44. 163
8.23 Comparison of loop code sizes for the TMX320C44. 164
8.24 Comparison of running times for the TMX320C44. 165

Acknowledgement

In looking back over the past years of graduate study, a number of people have played an influential role on my development as a person and researcher. Here I wish to express my thanks, but these words alone cannot express the gratitude, nor the admiration that I hold for them.

I would like to thank my advisor, Professor Alexandru Nicolau, who spent countless hours with me in enlightening and thought-provoking discussion. His insight and direction provided the focus and impetus for me to conduct high-quality research. His input and constructive criticism on written works and presentations greatly improved my communication skills.

I would like to thank Professor Nikil Dutt who has served as a co-advisor over the past years. He provided a different point-of-view that greatly improved the content of many manuscripts and research goals. His input and critique were also instrumental in the development of my presentation and communication skills.

I would like to thank Professor Rajesh Gupta for serving on my doctoral committee and providing valuable comments on earlier revisions of this manuscript.

I would like to thank Professor Daniel Gajski for serving on my doctoral candidacy committee and providing input over the years on my research.

I would like to thank Professor Tatsuya Suda for encouraging me to pursue a graduate career.

And, especially, I would like to thank my Mother and Father, my sister Kristin, my high-school friends, Albert, David, Ed and Steve, my lab colleagues, Dr. Joseph Hummel and Dr. Steven Novack, and the countless friends who were always there for me and whose support and encouragement helped me through the darkest times.

Curriculum Vitae

1990	B.S. in Information and Computer Science, University of California, Irvine, Magna Cum Laude, Phi Beta Kappa
1990–1992	Teaching Assistant, Department of Information and Computer Science, University of California, Irvine
1993	M.S. in Information and Computer Science, University of California, Irvine
1990–1998	Graduate Student Researcher, Computer Systems Design (CSD) Group, UCI-VLIW Compiler project, Department of Information and Computer Science, University of California, Irvine
1998	Ph.D. in Information and Computer Science, University of California, Irvine Dissertation: *Register Allocation Issues in Embedded Code Generation*

Publications

Kolson, D. J., Nicolau, A. and Dutt, N., Elimination of Redundant Memory Traffic in High-Level Synthesis, *Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, November, 1996.

Kolson, D. J., Nicolau, A., Dutt, N. and Kennedy, K., Optimal Register Assignment to Loops for Embedded Code Generation, *Transactions on Design Automation of Electronic Systems (TODAES)*, April, 1996.

Kolson, D. J., Nicolau, A., Dutt, N. and Kennedy, K., A Method for Register Allocation to Loops in Multiple Register File Architectures, *IEEE 10th International Parallel Processing Symposium (IPPS)*, April, 1996.

Kolson, D. J., Nicolau, A., Dutt, N. and Kennedy, K., Optimal Register Assignment to Loops for Embedded Code Generation, *IEEE 8th International Symposium on System Synthesis (ISSS)*, September, 1995.

Kolson, D. J., Nicolau, A. and Dutt, N., Integrating Program Transformations in the Memory-Based Synthesis of Image and Video Algorithms, *IEEE International Conference on Computer-Aided Design (ICCAD)*, November, 1994.

Kolson, D. J., Nicolau, A. and Dutt, N., Minimization of Memory Traffic in High-Level Synthesis, *ACM/IEEE 30th Design Automation Conference (DAC)*, June, 1994.

Kolson, D. J., Nicolau, A. and Dutt, N., Ultra Fine-Grain Template-Driven Synthesis, *IEEE 7th International Conference on VLSI Design*, January, 1994.

Abstract of the Dissertation

Register Allocation Issues in Embedded Code Generation

by

David J. Kolson

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1998

Professor Alexandru Nicolau, Chair

In conventional compilation, register allocation—the mapping of program variables to the registers of the target architecture—plays an important role in the performance of application code. In particular, for load/store architectures, good register allocation is exceedingly important as all operands to instructions, in this type of architecture, must be contained within the register set.

Typical processors selected as the core unit or core processor for an embedded system closely resemble the load/store or RISC-type of architecture, and, thus, conventional register allocation techniques are applicable in the generation of code for an embedded processor. However, architectural features of the core processor, features designed to reduce core size/cost and/or are specific to the target application area for improved performance—such as disjoint register files and/or requirements that operands to particular instructions reside in specialized registers—complicate the register allocation process. This, coupled with the time-sensitive nature of typical embedded applications necessitates high-quality register allocation.

This thesis demonstrates that beyond the specific task of register allocation, there are subtle issues related to register allocation that must be addressed in order to generate high quality code for an embedded application. Issues

investigated relate to promotion of data items from secondary memory to primary memory, global guiding of transformation interaction, integrating register allocation and instruction scheduling and optimal allocation to loops.

This thesis presents a technique which eliminates redundancies found in array accessing over iterations of a loop. Essentially this technique allocates a register to an array data item that is used frequently over a loop or within a window of iterations of a loop, thus promoting it from the secondary memory to the primary memory.

Transformation interaction within a parallelizing compiler has been studied relatively little, however, it remains an important issue in generating high quality code. This thesis presents a paradigm for integrating transformations so that transformations are applied based upon global knowledge instead of local knowledge, leading to better resource/register allocation and the development of better scheduling/allocation heuristics.

One strategy which integrates register allocation and instruction scheduling performs register allocation "on-the-fly" by a technique called renaming. This has the disadvantage of adding many copy instructions to the code which adversely affects performance and this thesis presents a post-scheduling technique to unroll loop code and re-allocate registers to eliminate these copy instructions.

Finally, the issue of optimal register allocation to loops is addressed. Register allocation has been extensively studied with proposed solutions being heuristic in nature. However, for embedded applications which contain time-critical loops and/or loop kernels, an optimal allocation is necessary. This thesis presents a technique for optimal allocation of registers to loop code.

Chapter 1

Introduction

One of the goals in the field of Design Automation is the synthesis of a hardware solution from a problem specification. Typically, the problem is formulated as a behavioral specification in which the desired actions of the hardware are detailed from an algorithmic standpoint. Various high-level languages, such as VHDL and HardwareC, exist for expressing behaviors. Through the process of *High-Level Synthesis* a behavior is translated into a hardware solution in some format suitable for implementation [22, 35, 81].

Because it can be too costly and time consuming to completely re-synthesize hardware when slight behavioral changes are made, the sub-area of Hardware/Software Co-Design seeks to partition a given behavior into hardware and software components. The hope is that, for a given application domain, common characteristics or time-critical aspects of the applications can be extracted and implemented in hardware while the software component implements the various differences in the applications and allows for future refinement.

This type of architecture is depicted in Figure 1.1. This architecture integrates a programmable component, one or more specialized hardware components, a ROM (to store the sequence of instructions executed by the programmable component) and a memory (for storage, as necessary), has been termed an *embedded system.* Depending upon the required complexity of the system, various solutions exist for the programmable component of an embedded system. Solutions ranging from:

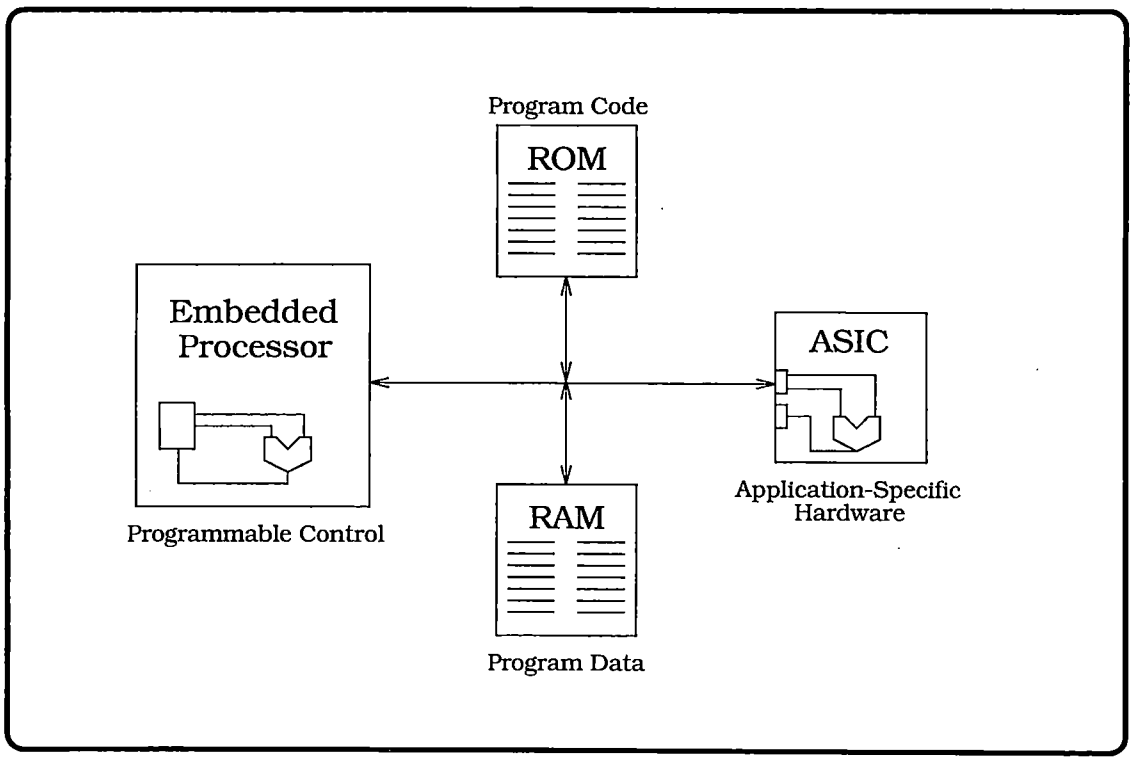


Figure 1.1: Typical embedded system architecture.

- micro-processors
- processor cores (a "stripped down" version of a given processor in which a subset of the processor's instruction set is implemented and/or the datapath bit-width is reduced).
- digital signal processors (DSPs) (processors with explicit instructions for supporting digital signal processing)

are all available. Typically, these processor's architecture's closely resemble the load/store or RISC-type of architecture where operands to instructions must be contained within the register set for execution of that instruction. Transfers of values between memory and the register set are explicitly performed by load and store instructions as necessary.

The process of "synthesizing" the instruction sequence corresponding to the software partition of the behavior (that which is to be executed by the embedded processor) is referred to as *code generation* and is closely related to the conventional task of compilation. One integral phase in the compilation process is that of *register allocation*—the mapping of program variables to the available registers of the target architecture.

Although good register allocation is exceedingly important as, in a load/store-type of architecture all operands to instructions must be in registers, this thesis demonstrates that there are subtle issues related to register allocation that must be addressed in order to generate high-quality code for a given application.

1.1 Code Generation and the Role of Register Allocation

Generating code for the type of system discussed previously involves the translation of application code written in a high-level language into the native instruction set (the machine language) of the programmable component.

However, because the target programmable architecture is flexible in its ability to contain some or all of the available application-specific datapaths, traditional compiler techniques must be refined to maximally utilize that provided flexibility and achieve high code performance.

Previously, little investigation has been made into standard compilation techniques which focus on the implications of a partitioned registers set, as the traditional viewpoint is one of a consolidated register file with equal (all registers in the register set may be accessed at any time) and uniform (any register can be used as any operand of an instruction) access to the contained registers.

In contrast, an embedded processor may have the available registers partitioned into multiple memories and/or scattered singly throughout the architecture with some registers having restricted or specialized uses. The placement of data values then becomes an issue as certain instructions may require data in specific registers. Further, this may be complicated by data necessary in one register or register set for one instruction and then necessary in a different register or register set for some later instruction.

Below, the process of code generation is briefly reviewed in optimizing compilers, parallelizing compilers and embedded compilers and the role played by register allocation is noted.

1.1.1 Optimizing Compilers

Classically, an optimizing compiler is divided into two main modules: the *front-end* and the *back-end* and is pictured in Figure 1.2. Traditionally, the front-end handles the language-specific details of the compiler; it parses the input (source) language (lexical analysis), determines if any syntax errors are present (syntax analysis), determines if variables, expressions, function calls, etc., are used correctly (semantic analysis), and finally, translates the input code into an intermediate form (intermediate code generation). The back-end, on the other hand, handles the machine-dependent aspects of the compiler; it performs

optimization[1] on the intermediate code (code optimization) and then generates instructions native to the target machine from the intermediate representation (code generation).

The influential view proposed by Backus [4] in the context of allocating index registers and Cocke [26] in the development of the PL.8 compiler, is to separate the tasks of code optimization and register allocation as this allows code optimization to proceed in a relatively simple manner and removes the complications arising from dealing with limited resources (registers). Also, optimizations could then be formulated and implemented in a general manner without the instruction set and register usages found in a particular architecture influencing development.

Thus, the tasks of instruction selection and register allocation are performed within the code generation phase. Instruction selection generates native instructions for the target machine from the internal intermediate representation. Register allocation performs mapping of the variables and temporaries to the registers of the target machine. When more live values than registers are present, *spill code*, explicit transfers of values between memory and the register set is necessary. Traditionally, the goal of the register allocator is to minimize the amount of spill code generated.

Typically, the target machine of an optimizing compiler is sequential in nature. That is, there is single flow-of-control and only one instruction may be issued per machine cycle. The register allocation produced by an optimizing compiler for a sequential architecture often exhibits heavy register re-use as during the instruction selection task, the instructions mapped to the various operators in the expression trees seek to evaluate the expressions generating as few temporaries as possible. This high re-use is a result of mapping those temporaries, whose lifetimes are short, to the available registers. This serves to reduce the amount of spill code produced for a sequential architecture.

[1]Although not all code optimizations are necessarily specific to the particular architecture, typically all optimizations are grouped together into one module as optimizations which are machine-specific create opportunities for other optimizations to apply.

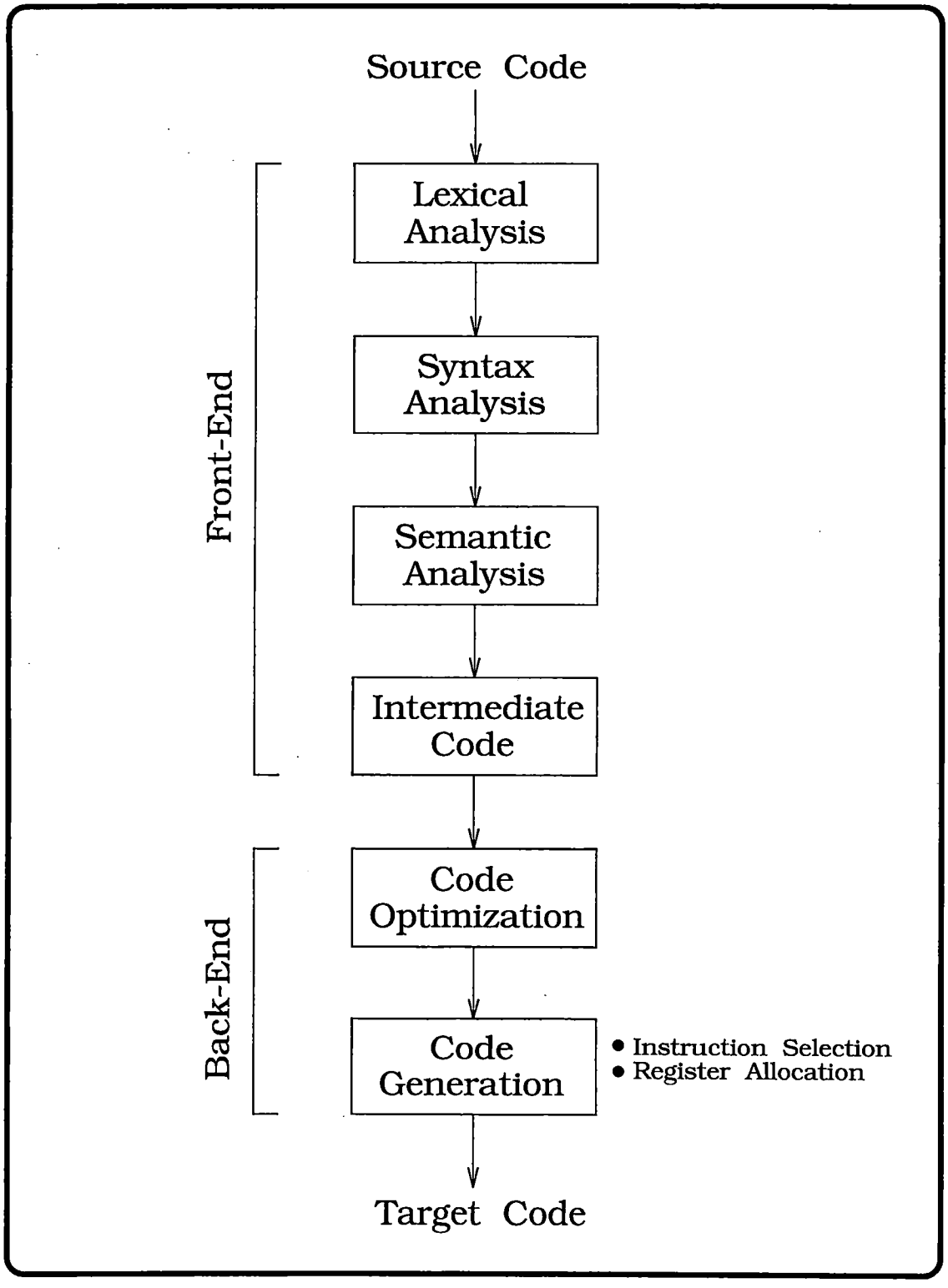


Figure 1.2: Phases of an optimizing compiler.

1.1.2 Parallelizing Compilers

To increase the throughput of a sequential processor, two methods have been investigated and implemented: *pipelining* and *parallelism.* Pipelining divides a functional unit into a series of sub-units, while parallelism increases the number of functional units present in the architecture. In order to exploit the peak performance of architectures with functional pipelining and/or parallelism, a compiler must expose and exploit the parallelism inherent in an application as instructions independent of one another are necessary for issue bandwidth.

Exposing and exploiting parallelism are the responsibilities of the *code scheduler.* In a parallelizing compiler, optimizations and transformations are typically applied during scheduling so that their effects on the code are exploited. In fact, many transformations are designed to increase the available parallelism.

However, in order to schedule code, to map instructions in the native instruction set to the functional resources of the target architecture, the phase of instruction selection is required prior to scheduling. This also implies that a register allocation phase is necessary prior to instruction scheduling as knowledge of the location of instruction operands and the resources (functional as well as register) required by an instruction is essential.

However, high register re-use leads to many false (anti-) dependencies that can potentially limit the extracted parallelism. As depicted in Figure 1.3, two strategies proposed by various researchers [13, 28, 38, 64] for overcoming this limiting factor are:

- Post-allocation - "Pre-allocate" registers then perform scheduling with a "loose" number of register (more than is present in the architecture) and then apply final register allocation.
- Integrated scheduling and register allocation - Perform code motion re-allocating registers "on-the-fly" as necessary to increase the exposed parallelism.

The first strategy has the advantage of simplicity, but has the disadvantage that spill code decisions and effects made after the scheduling process potentially

degrades the code's performance and possibly requires another scheduling phase. The second strategy has the advantage that spill decisions are factored into the code and spill instructions can be scheduled effectively, but these approaches tend to be more complex in terms of implementation and run-time.

Architectures targeted by a parallel compiler include VLIW, superscalar and heavily pipelined processors which may include mechanisms for multiple instruction issue and multi-way branching.

1.1.3 Embedded Compilers

An embedded compiler encompasses the same features as a parallelizing compiler, as, parallelism in the architecture must necessarily be exploited for high performance, but also has other requirements due to the characteristics of the application domain and target architecture. Characteristics such as:

- timing constraints (time-critical code segments)
- code size
- application-specific datapaths
- available registers and their usages

all serve to place constraints on a compiler and thus affect the quality of the code generated.

Embedded applications are often time-critical, so timing constraints must be factored into the scheduling process, as well as the timing and interaction (if any) of the embedded processor with other system components.

The amount of code that is generated is also an issue as the code must be embeddable within the system. That is, the application code partitioned to the programmable component resides in ROM. Since ROM size directly affects the size of the embedded system, constraints are typically placed on ROM size, thus limiting the amount of memory available for application code.

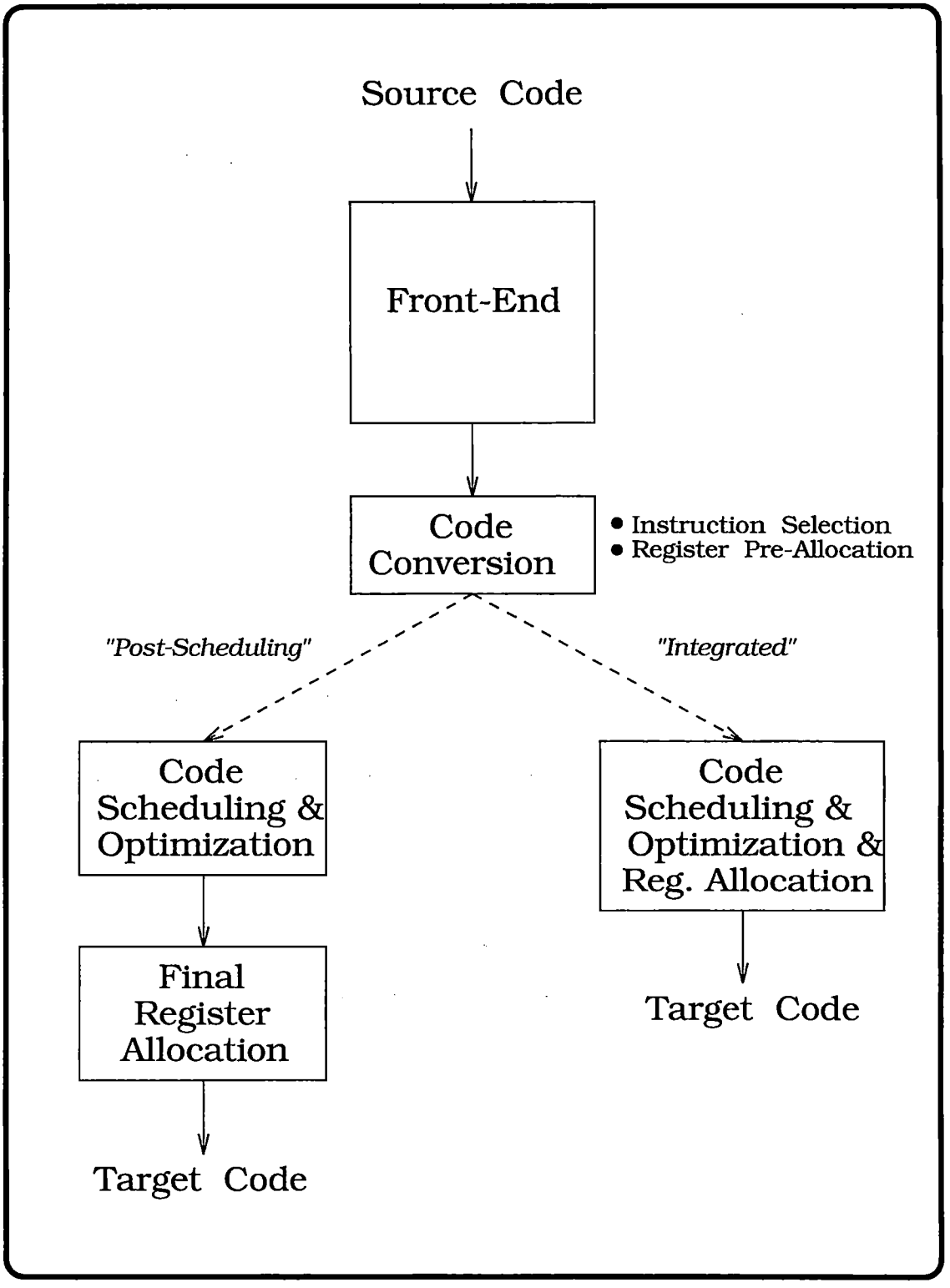


Figure 1.3: Phases of a parallelizing compiler.

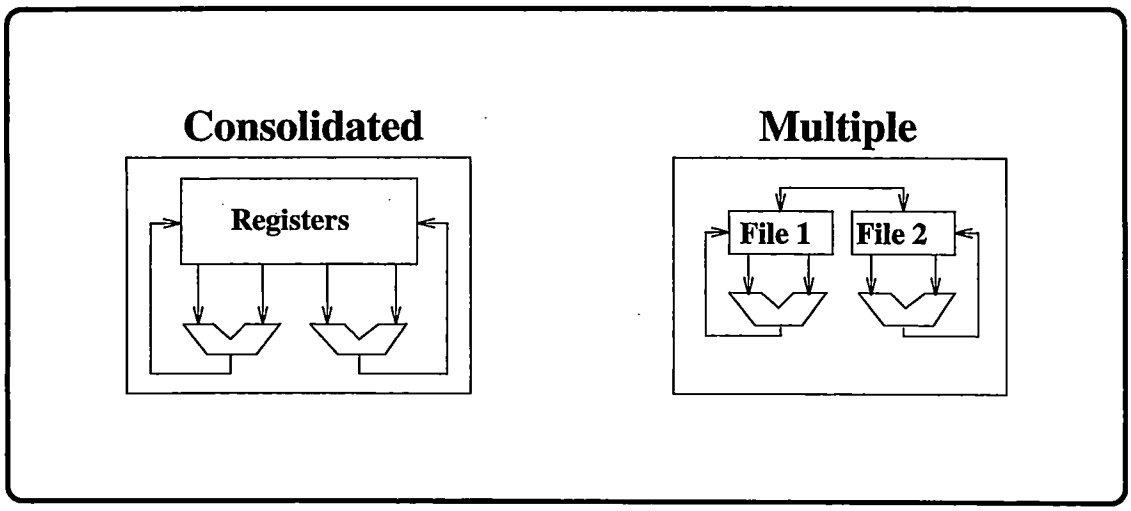


Figure 1.4: Available register organization.

Also, non-conventional (non-uniform) datapaths place stricter requirements on instruction selection and scheduling as some forms of computations may not be supported and alternative forms must be considered.

Lastly, the fewer number of available registers in the architecture and the presence of partitioned register sets increases the complexity of register allocation. Thus, effective register allocation plays a critical role in the generation of high-quality and high-performance code. Improving a register allocator to accomodate partitioned register sets and augmenting the heuristics which guide code optimization and transformation to make better use of registers allows this goal to be fulfilled.

1.2 Motivation for Embedded Compiler Complexity

Traditionally, one demand placed on a compiler is that of fast translation of source code to target code. Many compilers offer "switches," parameters to the compiler, that allow the user to specify the "level" of optimization, with higher levels requiring more compile time. Thus, most optimizations that are

implemented with heuristics, utilize more complex heuristics as the level of optimization increases.

One reason for heuristic implementation of the optimizations is, for most application areas, absolutely "perfect" code is not necessary, highly optimized code is sufficient. However, for embedded systems, the best code possible is usually highly desirable as the application code is compiled once, but is resident and executed for the lifetime of the product being developed. Further, many embedded systems contain fewer numbers of registers than that found in traditional processors, so more complex program optimization is plausible.

Also, there is a lack of research, both in conventional compilation and embedded code generation focusing on the effects of optimizations on register allocation as well as developing heuristics for optimization which have the underlying goal of improving register allocation.

1.3 Scope of Thesis

This thesis demonstrates that beyond the specific *task* of register allocation, there are subtle issues related to register allocation that must be addressed in order to generate high quality code for an embedded application. This thesis investigates:

- A technique which eliminates redundancies found in array accessing over iterations of a loop is presented in Chapter 4. This technique allocates a register to an array data item that is used frequently over a loop or within a window of iterations of a loop, thus promoting it from the secondary memory to the primary memory, thereby reducing the amount of memory traffic and, thus, the memory bandwidth requirements.
- Transformation interaction within a parallelizing compiler has been studied relatively little; however, it remains an important issue in generating high quality code. Chapter 5 presents a paradigm for integrating transformations so that transformations are applied based upon global knowledge (e.g., *across* basic blocks) instead of local knowledge (e.g., *within* basic blocks) ,

leading to better resource/register allocation and the development of better scheduling/allocation heuristics.

- One strategy which integrates register allocation and instruction scheduling performs register allocation "on-the-fly" by a technique called renaming. This approach has the disadvantage of adding overhead to the code in the form of many copy instructions which adversely affects performance. Chapter 6 presents a post-scheduling technique to unroll loop code and re-allocate registers to instructions so that copy instructions may be removed from the code.
- Register allocation has been extensively studied with proposed solutions largely being heuristic in nature. However, for embedded applications which contain time-critical loops and/or loop kernels, an optimal allocation is necessary. Chapter 7 presents a technique for optimal allocation of registers to loop code.

Chapter 8 presents the results of experimentation with these techniques and Chapter 9 presents some final remarks.

Chapter 2

Related Work

Register allocation has been the subject of much research in the past, both in the areas of conventional compiler design and design synthesis. In conventional compiler design, the term *register allocation* refers to the task of determining which program constants and variables should be contained within a register while the term *register assignment* refers to the task of determining within which register a value should reside. Conventionally, in design synthesis, the term register allocation refers to the task of determining the number of registers that are necessary to store values across machine states, while the term register assignment refers to assigning variables to the available registers. In the sub-area of Hardware/Software Co-Design, the term register assignment refers to the same mapping problem as compilation, but, as the target architecture may be specialized for a given application domain, the available registers may be partitioned into multiple files with some registers having restricted or specialized usages.

This chapter provides a survey of register allocation and assignment techniques as well as other memory-related program transformations.

2.1 Register Allocation and Assignment

The general problem in conventional compilation of register allocation is to assign the variables and temporary values of a program to the registers of the

target architecture, usually with uniform access to all available registers which are consolidated into one file. When more values exist than registers, some values will have to reside in main memory and spill code—loads and stores from main memory to the available registers—is generated when those values are referenced. The goal of a register allocator is to minimize the amount of spill code that is generated. This task is inherently NP-Hard [2, 36] and, as such, heuristic algorithms have been commonly utilized to determine some "sub-optimal, but good" solution.

2.1.1 Graph Coloring

One of the most popular heuristic approaches to allocating variables to registers is graph coloring [16, 23, 24]. In this strategy, the live ranges of program variables are examined. When two variable's lifetimes overlap they are said to *interfere*. An *interference graph* is then constructed wherein the nodes represent the variables and the edges joining nodes represent the interference of the two particular nodes being joined. The task is then to "color" the graph nodes with the same number of colors as registers. If no coloring of the graph is found, some variable is heuristically selected and spilled; all references to that variable then refer to that memory location. Once the original code has been updated with the spill code, a new graph is then constructed to reflect the new interferences and this process repeats until some colorable graph is found.

As the key to good register allocation in this scheme is the selection of a particular variable to spill, heuristics for selection have received attention [17] along with methods of coloring the graph [14]. Freiburghouse [34] suggests prioritizing coloring based upon the number of uses of a variable. Proebsting and Fischer [73] propose allocating registers to variables based upon the probabilities that a variable is "profitably" held in a register.

Bernstein *et. al.* [10] propose a strategy for reducing the amount of spill code generated by selecting between *multiple* heuristics to color the interference graph. Their results show that a best-of-three strategy results in less code and adds a

nominal amount of overhead to the graph coloring phase. Also, the authors propose a technique called *cleaning* to reduce the amount of spills generated for a particular variable. Rather than generating spill code each time that a spill candidate is referenced, the lifetime of the spill candidate is examined to determine if intervening stores and loads can be eliminated.

Chow and Hennessy [25] propose a method to split variable lifetimes when the interference graph is not colorable. After the lifetimes are split, a new interference graph results with the degree of some nodes (corresponding to those split variables) reduced and further coloring can become possible.

Leuh, Gross and Adl-Tabatabai [54] develop a global register allocation strategy similar to [25] which incorporates live range splitting and spilling. However, this strategy differs as execution probabilities are given primary importance when spill decisions are considered.

For large programs with many variables, the interference graph can become quite complex, necessitating large amounts of storage. Gupta *et. al.* [41, 42] propose a method to reduce the space requirements of a graph coloring register allocator by partitioning the program code, separately constructing and coloring the interference graphs for each partition and combining the colorings for a final program solution. Their results show significant improvements in space savings without sacrificing the quality of a graph coloring strategy.

Callahan, Carr and Kennedy [18] extend the graph coloring technique to handle arrays by *scalar replacement* which replaces repeated references to an array location by references to a (newly created) scalar variable.

Liem, May and Paulin [55] extend the left-edge register allocation scheme to allocate registers for architectures with registers which are specialized or have restricted uses. The proposed extension incorporates register classes into the model to specify a subset of the architecture's registers which are available for a given instruction.

2.1.2 Interval Graphs

Another paradigm for the register allocation problem is that of modeling the interferences of variable lifetimes by *interval graphs.* The interference graphs of basic blocks belong to the class of graphs known as interval graphs and can be optimally colored in polynomial time [37]. However, by definition, interval graphs are not applicable to a global context for allocating registers over conditional branches and loops. Some work [43] has been done, however, to extend the interval graph approach to loops, but in order to reduce complexity, arbitrarily breaks the lifetimes of cyclic variables at loop boundaries. A consequence of this is that, since register usages must match at the top and bottom of the loop, register-to-register move instructions (which consume functional unit resources) are necessary to correctly map values contained within the registers while load/store operations are necessary to load values necessary at loop top that are not contained within the registers and to store live values that

2.1.3 Optimal Register Allocation

Many researchers have felt that for particularly critical code segments, such as the innermost loops of time-sensitive applications, an optimal allocation is necessary. Horwitz *et. al.* [45] present a method for obtaining an optimal register allocation to *index* registers which minimizes the number of loads and stores due to added spill code. Further work either improves upon the efficiency of the Horwitz algorithm [60] or extends the basic algorithm to deal with simple loops [48], but in doing so loses optimality and degrades performance. More recent research has extended the basic idea in Horwitz's algorithm to include register allocation for *general purpose* registers [46].

2.1.4 Register Allocation and Program Transformation

Research on integrating transformations has addressed the issue at both the fine-grain (register-transfer) and coarse-grain (source-code) levels. In fine-grain

compilation, the thrust is to integrate register allocation with instruction scheduling while in coarse-grain compilation, the thrust is to study the effects of ordering of coarse-level loop re-structuring transformations.

Fine-grain

Bradlee, Eggers and Henry [13] study the problem of integrating instruction scheduling and register allocation. Three strategies are examined: 1) register allocation preceeding scheduling; 2) register allocation after scheduling; and 3) an integrated approach which does pre-scheduling and global register allocation before final scheduling. Their results indicate that, for the i860 target architecture, the integrated approach generally produces better results than register allocation preceeding scheduling, while there were no significant differences between the schedules produced by the integrated approach and register allocation following scheduling.

Goodman and Hsu [38] investigate the integration of code scheduling and register allocation in the context of pipelined processors. Their scheduling algorithm alternates between two sub-algorithms: one which is devoted to filling pipeline latencies and avoiding pipeline interlocking and stalls, and another which re-organizes the code to reduce register pressure—the ratio of the number of allocated register to the total number of registers. Scheduling first starts with code re-organization to reducing register pressure, and, when the register pressure has been reduced to a threshold level, code motion to fill pipeline slots is done until the register pressure becomes too high when the process repeats. From analyzing performance results, the authors note that if a processor is heavily pipelined, with a possible interlock being very expensive, spills might be more profitable (i.e., reducing register pressure is less critical).

In [64] resource trade-offs are made dynamically ("on-the-fly") during scheduling to better utilize free resources which, in some cases, results in the modification of the code. For example, the computation $X = 2 * X$ can be performed by shift ($X = SHL(X)$), addition ($X = X + X$) or by multiplication ($X = 2 * X$) depending upon which functional resources are currently free *at that point* in the schedule. This work, however, does not address the interaction of transformations

during scheduling or the effects that a transformation has on the resultant code.

Proposed techniques [67] typically make allocation decisions based on the parallelism automatically detected.

Rau *et. al.* [77] investigate register allocation heuristics for the strategy of register allocation following scheduling, specifically for DO loops that have been software pipelined via modulo scheduling. The issue in deriving a suitable heuristic for a register allocator centers around the expanded lifetimes created by modulo scheduling as well as lifetimes which have now become loop-carried. Two architectural types are considered: those with explicit support for modulo scheduling (i.e., a rotating register file) and those without (i.e., uniform, random access to all registers). Their results indicate that schedules for architectures which support modulo scheduling require more registers but have a reduced code size compared to those for other architectures.

Berson *et. al.* [11] integrate classic compiler optimizations such as constant propagation, loop invariant code motion, dead code elimination, etc., into the instruction scheduling process, as application of these techniques prior to scheduling can adversely affect the quality of the code produced. Their heuristics for application of the techniques to the code during scheduling focus on measures determine that the technique will be beneficial.

Coarse-grain

In [83] a tool for studying the ordering of coarse-grain transformations is discussed. However, this approach does not allow for assessing and making trade-offs between transformations.

2.2 Memory-Related Transformations

Callahan, Cocke and Kennedy [19] present a technique which improves the balance of a loop—the ratio of memory words accessed to the number of operations performed. Their algorithm restructures the loop at the input level based upon estimated functional unit pipeline hazards and interlocking

mechanisms, but deals only with single-dimension arrays and single flow-of-control within a loop body.

Callahan, Kennedy and Porterfield [20] investigate the idea of *software prefetching*—a technique to explicitly fetch data into a cache. Although the goal of the technique is to improve performance, memory traffic can actually increase in situations where a prefetched item is bumped from the cache before its use.

Davidson and Jinturkar [29] reduce memory traffic in the context of instruction set processors without a cache by combining (or "coalescing") multiple memory accesses into a single memory access. Their technique combines small bit-length accesses of adjacent data into one larger bit-length access. Two limitations to this strategy are: 1) architectural support (a wide bus) is necessary to be able to combine small bit-length adjacent accesses into a single larger access; and, 2) combining multiple memory accesses scattered throughout the program can adversely affect scheduling due to the new dependencies created.

Pöchmüller, Glesner and Longsen [68] discuss the notion of background memory management—the allocation of memory modules to arrays at the behavioral level. The authors note that background memory management is an important subtask in High Level Synthesis as it is necessary to remove unnecessary (redundant) array references due to the hardware that they generate.

Data blocking [53, 69] is another technique aimed at improving cache effectiveness. This approach partitions a large data-space that is not containable within a cache into smaller data *blocks* which are containable within a cache and then the program is restructured to improve data reuse within those blocks.

Work in scheduler transformations includes fast prototyping of architectures [75], the shortening critical path lengths in pipelined datapath synthesis [59], minimizing path explosion in path/trace-based systems [65], redundant operator creation to reduce execution time [58], level compression of expressions [80] and incremental tree height reduction [62].

2.3 Memory Access Models

Duesterwald, Gupta and Soffa [30] extend traditional dataflow analysis for scalar variables to include subscripted (array) variables through the use of *iteration distance values.* These values denote the number of iterations between the production of an array value and its final use and is utilized in analyzing dependencies between memory operations.

Verhaegh *et. al.* at Philips [82] utilize a *stream* concept to model periodicity in array references. Dependencies between streams are revealed by *stream graphs* which incorporate start times and periodic information of array references into *iteration vectors.* This information is then used to assess memory requirements.

Franssen *et. al.* at IMEC [33] use a complex *polyhedral dependency graph* to model multi-dimensional array references. A polyhedral dependency graph is a dataflow graph which contains sets of points defined by the ranges of variables used in an array's indexing expression. This graph then represents the dependencies between memory operations.

2.4 Memory Structure Synthesis

Another formulation of the memory allocation problem is to determine the number of registers necessary to preserve values produced in one control step and used in subsequent control steps. Kurdahi and Parker [52] present an algorithm which accomplishes this by analyzing variable lifetimes in the dataflow graph. Some work [39, 66, 79] has been done to improve register usage for loops which breaks a variable's lifetime at the loop boundary, creating two "coupled" variables which the assignment process tries to assign to the same register. Register sharing—the use of one register by multiple variables when the lifetimes of those variables do not overlap—is employed to reduce the necessary number of registers.

Goossens [39] uses a heuristic to fill the gaps between coupled variables and then

applies the left-edge algorithm. Stok [79] iteratively tries to improve an initial allocation produced by the left-edge algorithm by "permuting" variables in registers at loop end to match the variable assignment at loop beginning. Park, Kim and Liu [66] extend the left-edge algorithm to deal with behaviors having conditional and looping constructs.

A natural extension to this formulation is the grouping of those allocated registers into memory modules. Two issues arise in doing so: 1) the exact grouping of variables so as to minimize the number of modules used; and, 2) the interconnections now necessary to connect modules to functional units. Balakrisnan *et. al.* [6] provide a solution which places primary importance on variable grouping. Ahmad and Chen [1] extend Balakrishnan's approach to take into account the commutative property of operations, thus allowing the connection of a memory module to either input of a functional unit. Kim and Liu [49] note that the first issue heavily influences the second, and, thus, in their approach, they place primary importance on interconnection cost rather than variable grouping.

Ramachandran, Gajski and Chaiyakul [76] present an algorithm to allocate storage for array variables. Their algorithm performs *array-variable clustering* which allows more than one array to be allocated to the same memory module subject to user performance and cost criteria.

Chapter 3

Phases of Compilation

In this chapter, the phases of the parallelizing compiler used for the development of the techniques studied in this thesis are outlined and the underlying execution model is presented.

3.1 A Parallelizing Compiler

As illustrated in Figure 3.1, in the parallelizing compiler developed at UCI [70], the compilation process of application code written in "C" starts with a sequential version of that code produced by a version of the GNU "C" Compiler. This version has been modified to output code in a three-address instruction format for a load/store-type architecture model. This code is then input into the parallelizing compiler. After initial analysis of the code is performed, the bulk of the compiler's implementation performs code parallelization. It is during the loop pipelining phase, when iterations of a loop are overlapped to reveal a pattern in execution, that the transformations and optimizations implemented in the compiler are applied to the code. After loop pipelining, the code is maximally parallelized (subject only to data dependencies). Resource constraints are input and a scheduling phase applies resource constraints to the code. Once the code has been resource constrained, final code is generated.

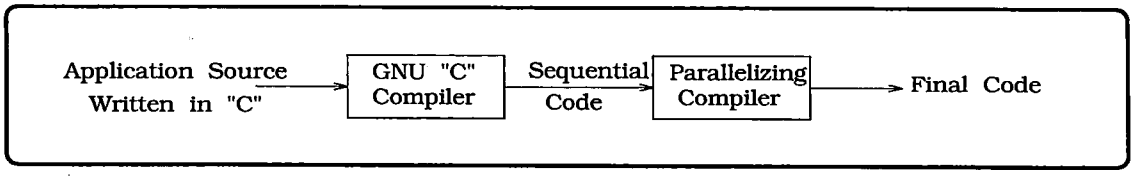


Figure 3.1: Code Generation process.

3.2 Program Model

The compiler's internal representation of the input code is in the form of a control/data-flow graph. In this graph, nodes correspond to machine cycles and all instructions contained within a respective node are performed (issued) in parallel, while edges between nodes correspond to flow-of-control and determine the next set of instructions to be issued. Thus, nodes with multiple successor nodes must contain conditional branching instructions, the correct successor is chosen at run-time subject to the values of the condition codes being tested.

For all instructions contained within a node, the following three-phase execution model is used:

Phase 1: All instruction operands are read from registers.
Phase 2: All instructions are executed.
Phase 3: All results are written to the appropriate registers.

With this model, performing the instructions $R1 = R2$ and $R2 = R1$ in parallel will correctly swap two values as the operands will be read before the results are written.

3.3 Scheduling

Although the program model presented above is effective for scheduling uni-cycle instructions, modeling pipelining with this model is more complex. Especially in cases where hardware conflicts or pipeline hazards between different functional

units occur. For example, consider a bus that is shared by two functional units, but used by those units in different stages of execution. If the first unit uses the bus in its second pipeline stage and the second unit uses it in its third pipeline stage, no conflicts occur if instructions are issued to both units in the same cycle. However, if an instruction is issued to the second unit in the current cycle and an instruction is issued to the first unit in the next, a conflict will occur.

Further, in some architectures with specialized hardware, each of the phases described above may not be atomic, but may encompass several lower level operations. This leads to a disparity between the relative performance of the generated code and the performance attained when the code is executed by the hardware.

These sorts of "lower-level" details are lost or are difficult to track at the fine-grain level as they are not made explicit to the scheduler. To remedy this, the sematics of an "atomic unit" are modified to allow these lower-level details to be made explicit. This lower-level of granularity has been termed the ultra fine-grain level and the extension of this compiler to the ultra fine-grain level is termed Ultra Fine-Grain Percolation Scheduling (UFG-PS).

The lower-level or ultra fine-grain level characteristics of the architecture are detailed to the compiler via a set of *microcode macros.* The compiler uses these macros to translate fine-grain instructions into ultra fine-grain instructions as these macros detail explicitly the data transfers associated with each of the fine-grain instructions. For instance, if the target architecture has the floating-point multiply-accumulate unit as shown in Figure 3.2 with a three-cycle latency multiplier and a two-cycle latency adder (for a total latency of five cycles for the fine-grain level multiply-accumulate instruction), the following macro details this:

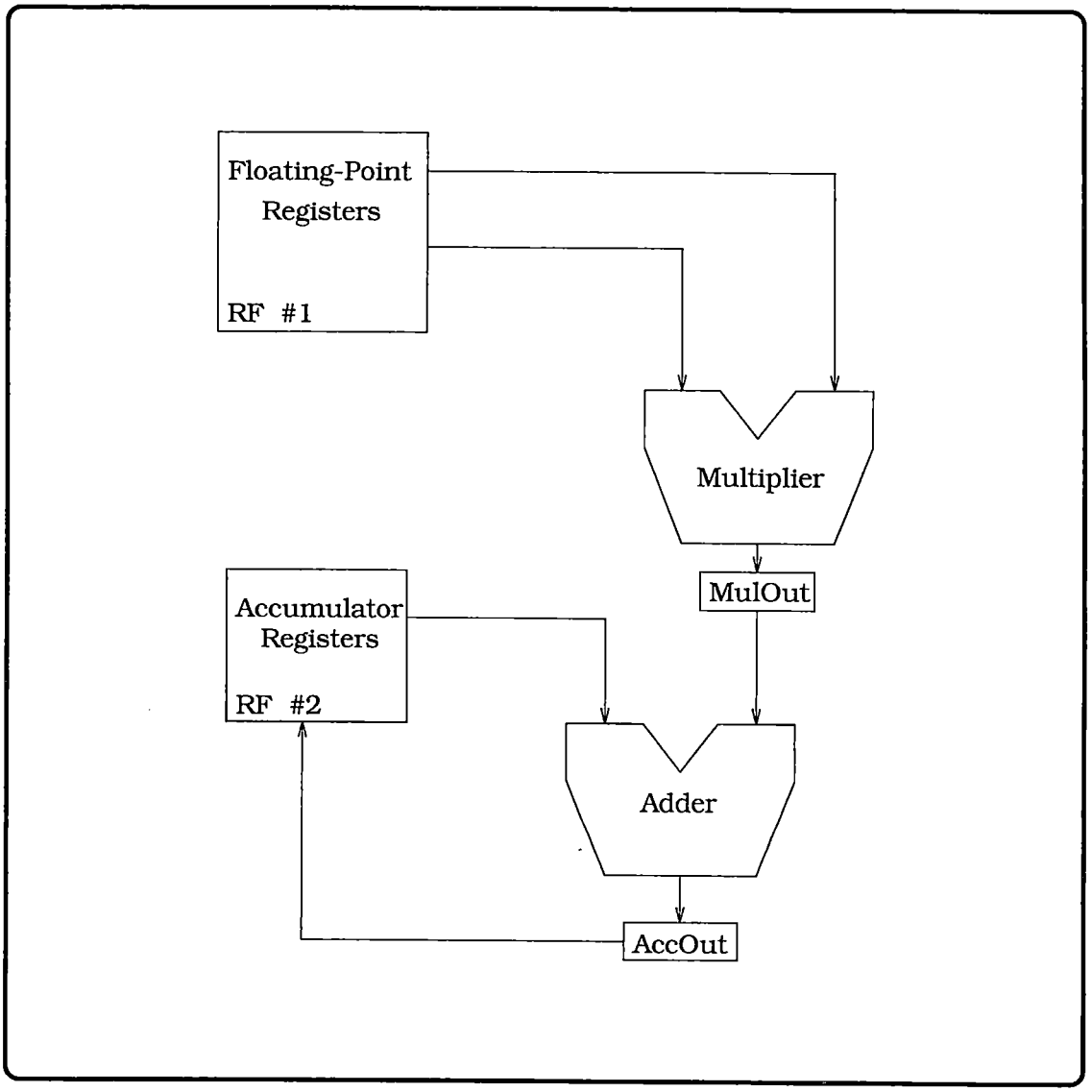


Figure 3.2: Datapath for microcode macro example.

Fmac @f1, @f2, @f3 :: *f1 = f1 + (f2 * f3)*

- **Fmulst1** (**RFread** #1, @f2), (**RFread** #1, @f3)
  perform 1^{st} *stage of f2*f3*
- **Fmulst2** fmulst1latch, fmulst2latch 2^{nd} *stage of multiply*
- **Fmulst3** fmulst2latch, MulOut 3^{rd} *stage of multiply*
- **Faddst1** (**RFread** #2, @f1), MulOut 1^{st} *stage of f1 + multiply_result*
- **Faddst2** faddst2latch, (**RFwrite** AccOut, #2, @f1)
  2^{nd} *stage of add, writeback into f1*

END

The first instruction details the first stage of the multiplication when the reading of the operands from the floating-point register set are input into the multiplier unit. The next two instructions show the transfer of the calculation through the intermediate (internal) latches and the final writing of the multiplier output into the latch labelled "MulOut" in Figure 3.2. The next instruction, the first of the addition sequence, inputs the operands to the adder from the accumulator register set and the output of the multiplier. Finally, the last instruction performs the second stage of the addition and the writeback of the result into the accumulator register file.

During the initial analysis phase, the fine-grain instructions are converted into their equivalent microcode sequences, deriving an ultra fine-grain level graph which is then subject to transformations and parallelization techniques implemented within the compiler.

Thus, in this representation, lower-level conflicts can be efficiently handled and avoided and, as this allows a closer modelling of the underlying target architecture, better quality schedules can be generated. Also, the notion of a dependency is extended from data dependencies to include *architecture* dependencies. Pipeline conflicts between various functional units can then be

detailed as dependencies between the respective UFG instructions. During scheduling, the compiler honors these dependencies, thereby generating code without those conflicts.

Chapter 4

Eliminating Redundant Memory Accesses

Typically, embedded systems which run applications that contain arrays employ a secondary store (e.g., a memory system) as a primary store (e.g., register storage) sufficiently large enough to completely contain all arrays would be impractical. References to arrays in the application code are accomplished by load and store instructions to the appropriate memories. Thus, array references in the application code translate into memory traffic, which can affect both the performance and memory bandwidth requirements of the resulting system.

In order to improve the performance and/or reduce the bandwidth requirements of an application, this chapter presents a technique to remove memory accessing instructions when those accesses are to locations that have been previously referenced. That is, this techniques eliminates memory instructions which *redundantly* load/store values from/to memory by first exposing any redundancy found in array (memory) accesses. Next, registers are selectively allocated to those values and the redundant memory accessing instructions are eliminated from the code. Future array references then become references to a register.

This transformation has many significant benefits. By eliminating redundant load instructions occurring on the critical dependency path through the code, the performance of the resulting schedule can increase dramatically as the length of the critical path can be shortened, thus generating more compact schedules and

```
for i = 1 to N
    for j = 2 to N
        a[i] := a[i] + 1/2(b[i][j − 1] + b[i][j − 2])
        b[i][j] := fcn(b[i][j])
    end
end
```

before optimization

```
for i = 1 to N
    t1 := a[i];   t2 := b[i][1];   t3 := b[i][0]
    for j = 2 to N
        t1 := t1 + 1/2(t2 + t3)
        b[i][j] := fcn(b[i][j])
        t2 := t3;   t3 := b[i][j]
    end
    a[i] := t1
end
```

after optimization

Figure 4.1: Removing redundant memory traffic.

reducing code size. In cases where array accesses are present in loops, elimination reduces the generated bandwidth over loop execution as well as the number of instructions that are executed as all of the instructions that are involved in an address calculation for a memory-accessing instruction are removed. Also, due to the transformation's local nature, it integrates easily into other parallelizing transformations [21, 65]. Finally, utilizing the transformation provides potential for savings in hardware due to the decrease in memory bandwidth requirements and/or the exploration of more cost-effective implementations.

4.1 Introductory Example

As an introductory example, consider the code in Figure 4.1. Before optimization, the inner loop requires four load and two store instructions per iteration:

load	$a[i]$	
load	$b[i][j]$	store $a[i]$
load	$b[i][j-1]$	store $b[i][j]$
load	$b[i][j-2]$	

However, the value computed in the current j iteration with $b[i][j]$ is the value that will be used by $b[i][j-1]$ in the next iteration and $b[i][j-2]$ in the subsequent iteration. At the end of the current iteration that value is stored to memory (necessary to preserve program semantics), and then re-loaded next iteration. The subsequent loads of that value may be circumvented by keeping the values in local storage—then only $b[i][j]$ is necessarily loaded each iteration while previous values are available and passed to future iterations via local (register) storage.

Also, with respect to the inner loop, the address calculation of $a[i]$ is invariant. Further, $a[i]$ is being used only as temporary storage (it is continually overwritten), and, unless $a[i]$ represents a memory-mapped input/output location, the load and store each iteration are unnecessary—the value loaded and stored can be kept in local memory.

Thus, as depicted in the Figure 4.1 after optimization code, only the load and store of $b[i][j]$ are necessary each iteration, resulting in the elimination of three load instructions and one store instruction per iteration[1].

[1]Note that the assignment to $t3$ is not an actual memory fetching instruction—the value stored to $b[i][j]$ is copied to $t3$.

4.2 Detecting Redundancy

The strategy employed in this technique for optimizing memory access is to eliminate redundancy in memory interaction as it is found during scheduling. Such redundancies can be found *within* loop iterations, possibly over multiple paths, when compacting code. Also, redundancy is exhibited *across* loop iterations during loop pipelining [65, 71] when multiple iterations of the loop are overlapped (subject to data dependencies and resource constraints) to find a repeating pattern in loop execution. As each iteration is integrated and overlapped with the current loop schedule, memory instructions from the new iteration are exposed to the memory instructions from previous iterations. With the aid of memory disambiguation—the ability to discern whether two memory instructions refer to the same location—the recurrence patterns in array accessing become apparent from loop iteration to iteration as the new memory instructions move into cycles with loads and stores from previous iterations.

However, as many array mappings are possible—multiple arrays to the same module, separate modules for each array, splitting a (large) array among several (smaller) memories or any combination thereof—it is necessary with this approach to have a preliminary allocation and binding of array variables to storage (secondary memory). This information is then given to the compiler along with the system resource constraints (e.g., number of memory ports on secondary memories, size and number (if partitioned) of primary memories, number and type of functional units, functional and memory instruction latencies, etc.). This is in contrast to [7, 57], for instance, which both perform memory transformations before allocation.

The method used to support memory disambiguation is to derive *symbolic expressions* which formulate the address calculation without the program variables. The symbolic expressions are then used by the memory disambiguator to determine dependencies between memory instructions.

First, the method of deriving symbolic expressions for memory instructions is outlined and then memory disambiguation is briefly discussed.

4.2.1 Symbolic Expressions

Each memory instruction contains an indexing function which is composed of the variables used in indexing each dimension of an array access, as well as either a *destination* for load instructions or a *value* for store instructions. The semantics of a load instruction are that issuing a load reserves the *destination* (local register storage) at issuance time (i.e., *destination* is unavailable during the load's latency) while the register storing the *value* argument of a store instruction is assumed free after the issuance of the store.

For the purpose of dependency analysis on memory instructions (memory disambiguation), every memory instruction contains a *symbolic expression* which is a string of symbols that formulates the memory address calculation without the source level variables [50]. The purpose of the symbolic expression is to be able to efficiently compare memory instructions for dependency analysis. In this approach, the variables used in the memory address calculation are "normalized" to a unique symbol for each loop thereby re-formulating the expression in as reduced a form as possible.

The *build_symbolic_exprs* algorithm in Figure 4.2 creates symbolic expressions for each memory instruction in the program by taking the induction variable (iv) definitions that define the current instruction's indexing function and deriving an expression for each. Next, the base of the memory structure is added to each expression. An instruction is then annotated with its expression, combining multiple expressions into a disjunctive form "((*expr1*) *or* ... *or* (*exprN*))."

The function *derive_expr* constructs the expression "(LoopId * Const)" if iv is self-referencing (e.g., $i = i + const$) where LoopId is the identifier of the loop over which iv inducts and Const is a constant derived from the constant in the iv instruction multiplied by a data size and possibly other variables and constants. If iv is defined in terms of another iv (e.g. $i = j + 1$, where j is an iv) then recursive calls are made on all definitions of that other iv. In this case, marking of iv's is necessary to detect cyclic dependencies which are handled by a technique called *variable folding*. Essentially variable folding determines an initial

```
 1:  Procedure build_symbolic_exprs(program)
 2:  begin
 3:    foreach mem_op in program
 4:      /* Set defs to all var defs */
 5:      foreach multiple def in defs
 6:        /* Select subset of iv_defs - one def for each var. */
 7:        sym_expr = φ
 8:        foreach variable in the indexing fcns
 9:          new_expr = derive_expr(variable)
10:          /* Add the expressions sym_expr and new_expr. */
11:        end
12:        /* Add the base offset to sym_expr. */
13:        if (/* mem_op's sym_expr is φ */)
14:          /* Set mem_op's sym_expr to sym_expr. */
15:        else
16:          /* Combine sym_expr with mem_op's sym_expr with "or" */
17:        end
18:      end
19:    end
20:  end build_symbolic_exprs
```

```
 1:  function derive_expr(term)
 2:  begin
 3:    Case (term):
 4:      CONSTANT: return "term * A.D. * D.S"
 5:      VARIABLE: return "(term * AD * DS)"
 6:      IND_VARIABLE:
 7:        if (/* term is self-referencing */) then
 8:          return "(term * AD * DS * Ind_Const )"
 9:        else if (/* term is marked */)
10:          /* do variable folding */
11:        else
12:          /* mark term */
13:          /* recursively derive term that defines this term */
14:        end if
15:      end
16:    end derive_expr
```

Figure 4.2: An algorithm to build symbolic expressions.

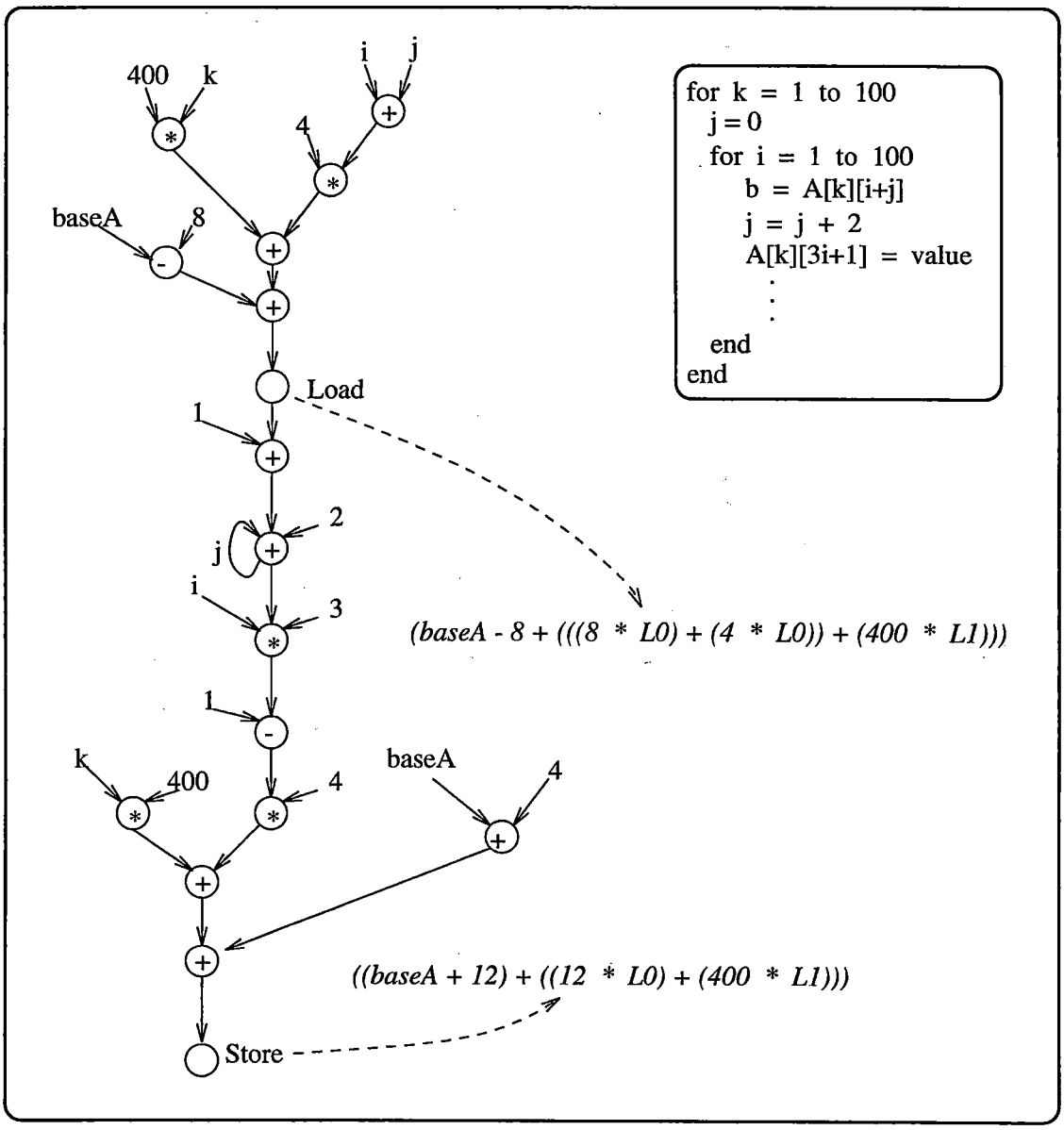


Figure 4.3: Symbolic expression example.

value of a variable on input to the loop or resulting from the first iteration (i.e., loop-carried values are not considered) from the reverse-flow of the graph. The result can be either a constant or another variable (which is recursively folded, until the beginning of the loop is reached).

Figure 4.3 shows example code and its CDFG annotated with symbolic expressions. The load from A builds the expression "((8 * L0) + (4 * L0))" which is the addition of 2 (the const for iv j) times 4 (the element size) and 1 (the const for iv i) times 4. The second loop over k adds the expression "(400 * L1)." Finally the base address of A is added. For the store instruction, the expression "(12 * L0)" is created which is 1 times 4 (the element size) times 3 (the constant in the behavior). Due to the +1 in the index expression, the constant 4 is added to the base address of A.

4.2.2 Memory Disambiguation

Memory disambiguation is the ability to determine if two memory access instructions are *aliases* for the same location [3, 9]. In the context of redundant load elimination, the interest is in static memory disambiguation, or the ability to disambiguate memory references during scheduling. In the general case, memory indexing functions can be arbitrarily complex due to explicit and implicit induction variables and loop index increments. Therefore, a simplistic pattern matching approach (one which bases equivalency of memory instructions by source level text) to matching loads and stores over loop iterations cannot provide the power that memory aliasing analysis does. For instance, in the following behavior if arrays a and b are aliases:

for $i = 1$ to N
    $a[i] := \frac{1}{2}a[i-1] + \frac{1}{3}b[i-2]$
    $Coef[i] := b[i] + 1$
end

pattern matching will fail to find the redundancy.

In this compiler, memory disambiguation is based on the well-known *greatest common divisor*, or GCD test [8]. Performing memory disambiguation on two instructions, *op*1 and *op*2, involves determining if the difference equation: *(op1's symbolic expression) - (op2's symbolic expression) = 0* has any integer solution. If a solution does exist, then the two instructions refer to the same memory location. If a solution cannot be found, then the conservative estimate that the two memory instructions may not be parallelized is used.

The importance of memory disambiguation to programs which index a secondary store can be seen in the ability to move load instructions beyond store instructions during parallelization. Consider an arbitrary data-flow graph where the critical path begins with a load and ends with a store. If the ability to discern whether the load and store access the same location is not present, a conservative approach must be taken. This would disallow the motion of the load instruction beyond the store instruction because of the possibility that they refer to the same location and in doing so, realizes no overlapping of loop iterations. Thus, the parallelism is unnecessarily limited.

Other methods for performing memory dependence testing exist including the Omega test [74], PIP [31] and inexact methods such as Fuzzy Array Analysis [27].

4.3 Eliminating Redundancy

This technique is employed during scheduling rather than as a pre-pass or post-pass scheduling phase; a pre-pass phase may not remove all the redundancies since other optimizations can create opportunities for redundancy removal that may not have otherwise existed, while a post-pass phase cannot derive as compact a schedule since instructions eliminated on the critical path allow for further refinement.

Figure 4.4 shows the main algorithm for removing unnecessary memory instructions. This function is invoked in this Percolation-based scheduler [71] by the *move_op* transform (or any suitable local code motion routine in other

```
 1:  function redundant_elimination(op, from_step)
 2:    begin
 3:      if (INVARIANT(op)) then
 4:        status = remove_inv_mem_op(op, from_step)
 5:        /* if op was removed, return REMOVED */
 6:      foreach memory operation, mem_op, in to
 7:        if (disambiguate(op, mem_op) == EQUAL) then
 8:          switch op-mem_op
 9:            case load-load:
10:              return do_load_load_opt(op, mem_op)
11:            case load-store:
12:              status = try_load_store_opt(op, mem_op)
13:              /* if op was removed, return REMOVED */
14:            case store-store:
15:              /* If op's arg and mem_op's arg have */
16:              /* the same reaching defs, delete op, */
17:              /* and update necessary information. */
18:              return REMOVED
19:            case store-load:
20:              return ANTI-DEPENDENCE
21:          end
22:        end if
23:      end
24:    end redundant_elimination
```

Figure 4.4: An algorithm for redundant elimination.

systems) when moving a memory instruction from one cycle into a previous cycle that contains other memory instructions.

The function *redundant_elimination* checks to see if the memory instruction is invariant. If so, then the function *remove_inv_mem_op* tries to remove it. If it is not invariant or could not be removed, then *op* is checked against each memory instruction in the previous cycle for possible optimization. If two instructions refer to the same location then the appropriate action is taken depending upon their types. The load-after-load and load-after-store cases will be discussed shortly. In the case of a store-after-store, the first store instruction is dead [3] and can be removed if it stores the same argument as the second instruction and

```
 1:  function remove_inv_mem_op(op, from_step)
 2:  begin
 3:    /* Conditions necessary for hoisting: */
 4:    /* 1. from_step must dominate all exit nodes. */
 5:    /* 2. Only one definition exists. */
 6:    /* 3. No other defs reach users. */
 7:    /* 4. If other invariant memory operations */
 8:    /* load/store to the same location, all */
 9:    /* must be "hoistable." */
10:    /* 5. (stores) Defs of argument are same at loop exits. */
11:    if (/* conditions met */) then
12:      /* Move op to pre-loop steps if it's a load */
13:      /* or to all post-loop steps if it's a store. */
14:      return REMOVED
15:    end if
16:    return NO_OPT
17:  end remove_inv_mem_op
```

Figure 4.5: An algorithm for loop invariant removal.

the argument has the same reaching definitions[2]. In this case, *op* is simply removed rather than removing *mem_op* and moving *op* into its place. For the store-after-load, nothing is done as this is a false (anti-) dependency on a memory location that should be preserved to retain code correctness. Status reflecting the outcome of the optimization is returned, allowing instructions to continue to move if no redundancy was discovered.

4.3.1 Removing Invariants

Removing invariant memory instructions is slightly different from general loop invariant removal. Traditional loop invariant removal moves an invariant into a pre-loop cycle. For load instructions this is correct; for store instructions it is not. Conceptually, invariant loads are "inputs" to the loop, while invariant stores

[2]Care must be taken if the store address is memory-mapped I/O, it may not necessarily be safely removed.

are "outputs." Therefore, loads must be placed into pre-loop cycles and stores must be placed into loop exit cycles.

To illustrate this, consider the following example:

```
for i = 1 to N
    for j = 1 to N
        b[j] := a[i − 1] + b[j]
    end
end
```

In the j loop, the load of $a[i-1]$ is invariant. Prior to this loop the value of $a[i-1]$ can be loaded into a temporary variable and then the temporary is referenced within the j loop, thereby considerably reducing the amount of generated memory traffic.

A more complex code example illustrating both invariant loads and stores is:

```
for i = 1 to N
    for j = 1 to N
        a[i] := a[i] + b[j]
    end
end
```

In this example, the memory location $a[i]$ is used from iteration to iteration as a temporary accumulator. To maintain correct semantics, both memory instructions—load and store—must be removed. If only the load instruction is replaced with a temporary then incorrect values are stored; if only the store instruction is replaced with a temporary then incorrect values are loaded. The solution to this situation (or, in general, when a loop contains scattered invariant loads and stores to the same location) is to remove the invariant if and only if all other invariants to the same location can be removed (i.e., hoisted out of the loop).

An algorithm to perform invariant removal appears in Figure 4.5. The conditions necessary for loop invariant removal (adapted from [3]) are: 1) the cycle that *op*

```
 1:  function do_load_load_opt(op, mem_op)
 2:  begin
 3:    /* set field to the nodes at the latency of mem_op */
 4:    foreach node in field
 5:      if (/* node is reachable by op */) then
 6:        /* Create move from mem_op's arg to the */
 7:        /* arg of op. Add this move to node. */
 8:      end if
 9:    /* Delete op and update necessary information. */
10:    Return REMOVED
11:  end do_load_load_opt
```

Figure 4.6: An algorithm for the load-after-load optimization.

is in must dominate all loop exits (i.e., op *must* be executed every iteration), 2) only one definition of the variable (for loads) or memory location (for stores) occurs in the loop and 3) no other definition of the variable or memory location reaches their users. Additionally, store instructions require that the definition of its argument be the same at the loop exits so that correctness is preserved. If these conditions are met, then the instruction can be hoisted out of the loop. If condition 2 fails and the instruction is a load, it still might be possible to hoist the instruction if a register can be allocated to the loaded value for the duration of the loop.

4.3.2 Load-After-Load Optimization

The load-after-load optimization is applied in situations where a load instruction accesses a memory value that has been previously loaded and no intervening modification has occurred to that location's value (i.e., there is no intermittent store). In Figure 4.6 the load-after-load optimization is detailed. The method employed with this optimization is to insert copy instructions (more on this in subsection 4.3.4) which will transfer the value without re-loading it (i.e., transfer the value from the destination of the first load to the destination of the second load thereby obviating the second load).

The algorithm is invoked with the initial load, *mem_op*, and the redundant load, *op* and commences by gathering all of the nodes in the graph where the latency of *mem_op* expires. Then, for each of those nodes, if that node is on a path in common with *op*, a copy instruction from the destination of *mem_op* to the destination of *op* is inserted into that node. Finally, *op* is deleted from the graph and any necessary local information is updated.

4.3.3 Load-After-Store Optimization

The load-after-store optimization is used to remove a load instruction which accesses a value that a store instruction previously wrote to the memory. Again, the method employed is to insert a copy instruction into the schedule to transfer the value. Due to limited resources it is possible that this optimization cannot be applied in some cases. Consider the partial code fragment:

Cycle 1: $a[i] := b \quad b := \alpha$
Cycle 2: $c := a[i]$

To eliminate the load $c := a[i]$, and replace it with the copy instruction $c := b$ in cycle 2 would violate code semantics because it introduces a *read-wrong* conflict—it would incorrectly read the value α rather than the value written to memory. The copy instruction must be placed in cycle 1 to guarantee correct results. However, in this code fragment:

Cycle 1: $a[i] := b \quad c := \gamma$
Cycle 2: $c := a[i] \quad d := f(c)$

placing a copy instruction in cycle 1 will violate code semantics because it introduces a *write-live* conflict—it would incorrectly overwrite the value γ; the copy must be inserted into cycle 2. Notice that in both cases, the transformation is still possible; analysis is required to determine which cycle is applicable.

This optimization might not be feasible in the following situation:

Cycle 1: $a[i] := b \quad b := \alpha \quad c := \gamma$
Cycle 2: $c := a[i] \quad d := f(c)$

Semantics are violated by placing $c := b$ into either cycle. However, if a free register exists, then the optimization can be done:

Cycle 1: $a[i] := b \quad b := \alpha \quad c := \gamma \quad e := b$
Cycle 2: $c := e \quad d := f(c)$

Therefore, the precise case when the load-after-store optimization fails to remove a redundant load is composed of three conditions:

1. A copy in this cycle results in a *read-wrong.*
2. A copy in the previous cycle results in a *write-live.*
3. No free register exists in the previous cycle.

In practice, this situation occurs very infrequently.

The load-after-store optimization algorithm is found in Figure 4.7. This algorithm determines within which cycle to place a copy instruction. Initially, the cycle that *op* is in is tried. If a *read-wrong* conflict occurs, the previous cycle is tried. If a *write-live* conflict arises, a free register is necessary to transfer the value. In this case, two copy instructions are added to the schedule. If a free register is not available, no optimization is done. If no conflicts occur (or they can be alleviated by switching cycles) then a copy instruction is inserted. Finally, the load instruction is deleted and necessary information updated.

4.3.4 A Note on Move Instructions

The copy instructions inserted into the schedule by this transformation, may or may not be found in the final schedule depending upon the context in which the copies appear. If the primary memory is consolidated, then the semantics of a copy keep the value within the same memory, but in a different register. As it is not necessary to actually perform a copy in this case, traditional copy propagation [3] and copy elimination [63] techniques (which will be discussed further in Chapter 6), implemented in this compiler, can delete copy instructions from the schedule.

```
 1:  function try_load_store_opt(op, from_step, mem_op, to_step)
 2:  begin
 3:    node = from_step
 4:    if (/* there is a read-wrong conflict */) then
 5:      node = to_step
 6:    end if
 7:    if (/* there is a write-live conflict */) then
 8:      if (/* free cell exists*/) then
 9:        /* Create move of mem_op's arg to free cell. */
10:        /* Add move op to to_step. */
11:        /* Create move of free cell to op's arg. */
12:        /* Add move op to from_step */
13:      else
14:        return NO_OPT
15:    else
16:      /* Create move of mem_op's arg to op's arg. */
17:      /* Add move op to node */
18:    end if
19:    /* Delete op and update necessary information. */
20:    return REMOVED
21:  end try_load_store_opt
```

Figure 4.7: An algorithm for the load-after-store optimization.

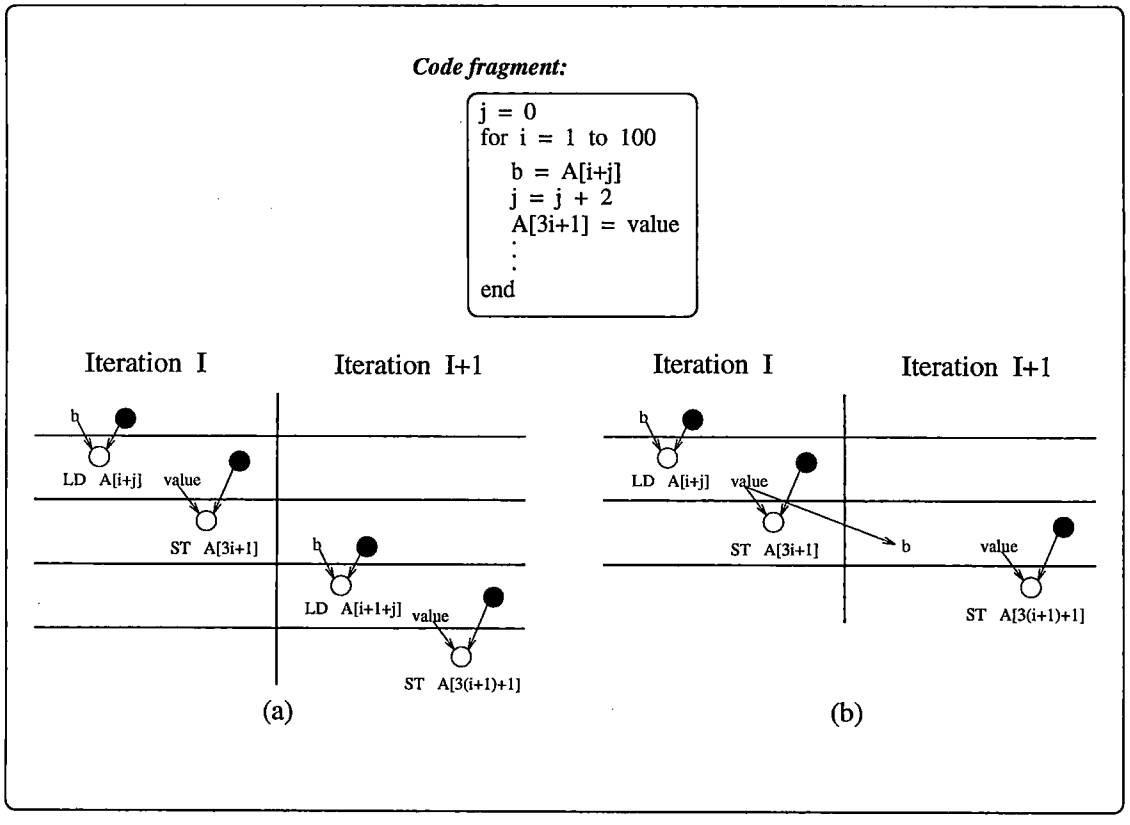


Figure 4.8: Redundancy elimination example.

However, the primary memory may be partitioned into multiple memories. If this is the case, then care in applying removal techniques is necessary. The copy instructions may be deleted if the source and destination registers are contained within the same memory module, otherwise the copy instructions are necessary to transfer values from one memory module to another.

4.3.5 An Example

Figure 4.8 illustrates the application of this transformation to an example code fragment. For simplicity, all of the instructions constituting the address calculations for the memory instructions have been grouped into one node which has been darkened in the illustration. In practice, the memory instructions will have symbolic expressions; however, the memory instructions are shown

annotated by the source-level counterpart for illustration.

During loop pipelining when a future iteration is overlapped with the current loop schedule, the schedule in Figure 4.8(a) results. When percolating the load instruction from iteration $I+1$ into the previous cycle, *redundant_elimination* is invoked as the previous cycle contains another memory instruction. The disambiguator discovers that the symbolic expressions for the load and store instructions are the same, so the load-after-store optimization is applied and the schedule in Figure 4.8(b) results.

4.4 Effects of Redundancy Elimination on Register Allocation

In the previous two sections, techniques for discovering and eliminating redundancy are discussed. As presented, the elimination techniques are applied whenever possible during scheduling, the premise being that, by removing memory instructions, less memory traffic will result, thereby reducing memory bandwidth requirements and possibly increasing performance. However, in some cases, removing memory instructions can have a negative effect on register allocation, causing memory traffic due to spill code.

In this section, the effects of redundancy elimination on register allocation are examined. In cases where negative effects are observed, heuristic guidelines are outlined and a modified elimination algorithm is presented.

4.4.1 Examining the Effects of Optimization

From the previous section, there are three cases when memory instructions are removed—by the load-after-load, load-after-store and store-after-store optimizations—and one where memory instructions are hoisted out of the loop—by loop invariant removal. As the store-after-store elimination optimization does not affect register usage, it will not be further discussed; this

section will focus on the load-after-load and load-after-store optimizations and loop invariant removal.

Load-After-Load Optimization

Recall that in the load-after-load optimization copy instructions are introduced into the code. These instructions are necessary to transfer the value loaded into a register by some load instruction into the destination register of another load instruction as the two destinations are not guaranteed to be the same. However, due to the machine model, the register count remains exactly the same and is not affected by load-after-load optimization.

Theorem 1: The load-after-load optimization does not increase the number of registers used.

Proof: Let Op_t denote a load instruction which defines variable a and is scheduled in cycle t and Op_{t+1} denote a redundant load which defines variable b and is scheduled in cycle $t+1$. Let load instructions have latency l. Before the transformation, a is available for computation in cycle $t+l$ and b is available in cycle $t+l+1$. Concerning only a and b, the register counts in each cycle are: cycle t, one and cycle $t+1$ to $t+l+1$, two. After the optimization is applied, a copy instruction $b := a$ appears in cycle $t+l$ when a is available. Now the register counts are: cycle t to $t+l-1$, one and cycle $t+l$ to $t+l+1$, two. Therefore the total register count in each cycle has not increased. In fact, in the period $t+1$ to $t+l-1$ the register usage has *decreased* leaving a free register which other transformations can possibly exploit.□

Load-After-Store Optimization

As discussed previously, there are cases where the load-after-store optimization may require an additional register to correctly transfer a value written to memory by a store instruction to the destination register of a load instruction. Here is the example code again (the additional register "e" is necessary to transfer the stored value b to c):

Before:

Cycle 1: $a[i] := b \qquad b := \alpha \qquad c := \gamma$

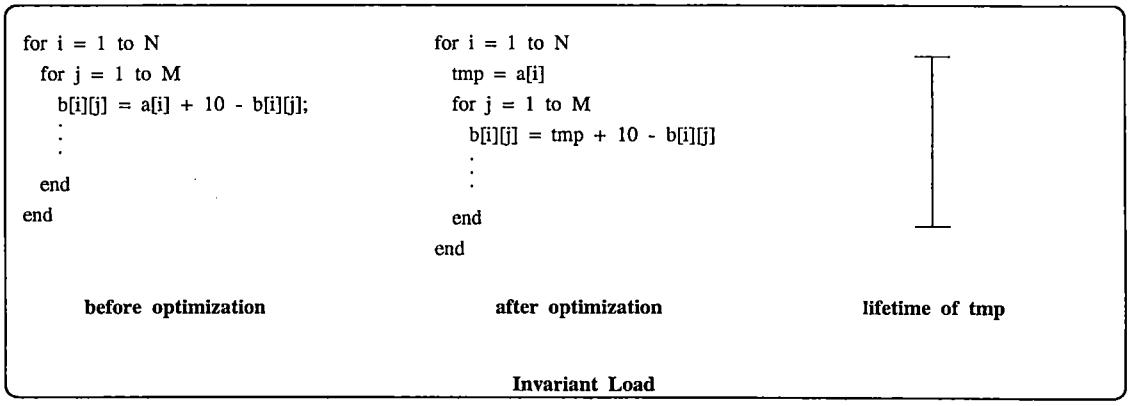


Figure 4.9: Invariant load removal.

Cycle 2: $c := a[i]$ $d := f(c)$

After:

Cycle 1: $a[i] := b$ $b := \alpha$ $c := \gamma$ $e := b$

Cycle 2: $c := e$ $d := f(c)$

In this case, this optimization could only be performed if a free register exists over cycles one and two (i.e., the free register's lifetime can extend from the store to the load). In general, if the latency of a load instruction is l and the store instruction is in cycle t, then the newly allocated register will extend from t to $t+l$, a total of $l+1$ cycles. Thus, the register pressure increases over those cycles.

Loop Invariant Removal

In the invariant case, there are three cases, with respect to a memory location, where memory instructions are hoisted from a loop, if it is a load, a store or both. Figures 4.9-4.11 contains examples of each case along with the optimized versions and the lifetimes of the resulting allocations. In order for a performance increase to be realized (i.e., the latency of memory instructions removed from the loop), in each case of Figures 4.9-4.11, **tmp** (and **tmp2**) must represent a register[3]. However, this implies that a register has been allocated globally over the loop (or loops) from which the memory instruction was hoisted. This will

[3]Arguably **tmp** may be a *specific* memory location (e.g., frame relative) where the value may be kept, which would, at the very least, not reduce the amount of memory traffic but would eliminate the instructions associated with the address calculation.

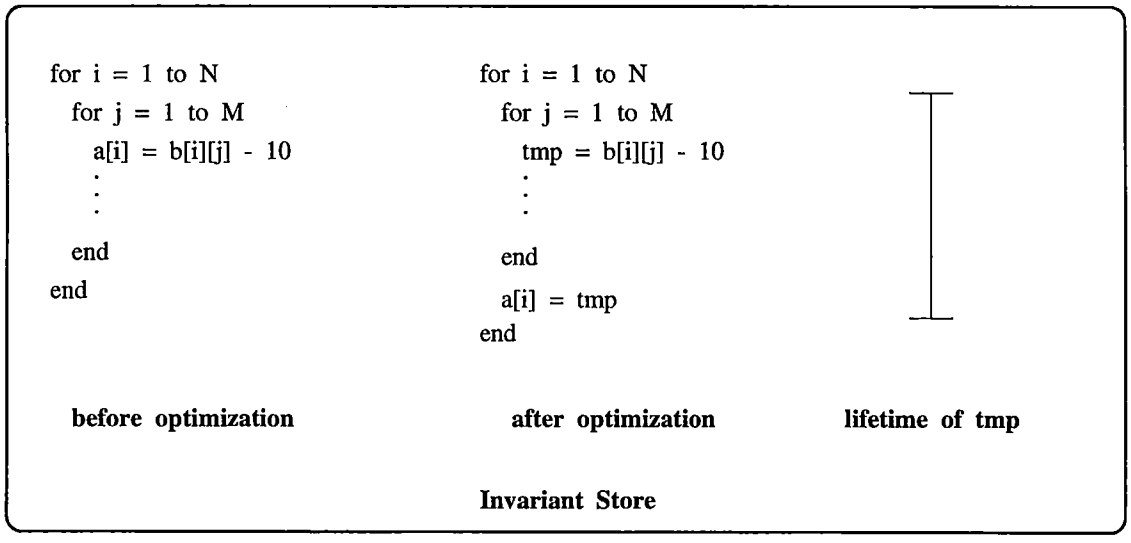


Figure 4.10: Invariant store removal.

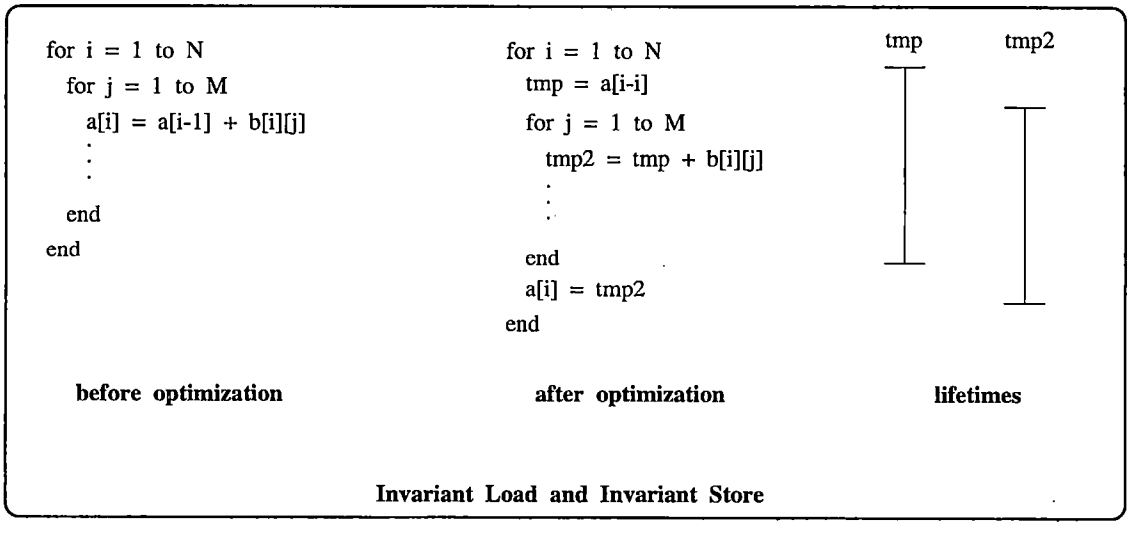


Figure 4.11: Invariant load and store removal.

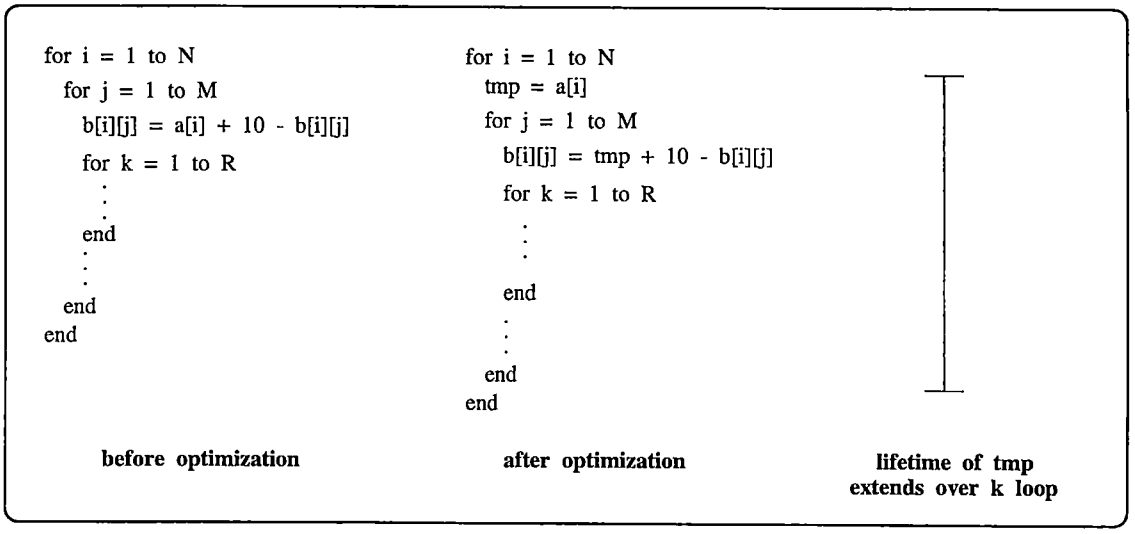


Figure 4.12: Multiple effects of invariant removal.

increase the register pressure within the loop as fewer registers are available and may result in more spill code than if hoisting had not been performed.

Also, Figure 4.12 depicts a case where invariant removal further complicates register allocation. Before optimization, the invariant code (the reference to $a[i]$) does not affect the performance of the innermost k loop. However, after the memory instruction is hoisted out of the middle j loop and a register is allocated to that value, the k loop is affected as the live range of the newly allocated register now extends over the k loop, reducing the number of registers available for allocation within the k loop.

4.4.2 An Enhanced Redundant Elimination Algorithm

There are two cases when the register pressure is affected by eliminating redundant memory accesses: by the load-after-store optimization and by loop invariant removal. In this section, heuristic modifications are discussed and modified elimination algorithms are proposed.

Load-After-Store Optimization

For the load-after-store optimization, one enhancement is to disallow the

```
 1:  function load_store_heuristic(load) returns boolean
 2:  begin
 3:    if (/* number of free registers < 2 */)
 4:      Return FALSE
 5:    Dep_Instrs = /* collect instructions dependent upon load */
 6:    can_move = TRUE
 7:    foreach instruction I in Dep_Instrs
 8:      can_move &= instr_could_move (I)
 9:    Return can_move
10:  end load_store_heuristic
```

Figure 4.13: An algorithm for the load-after-store heuristic.

optimization in the case when a free register is necessary, if this will cause all of the registers to be allocated, thus avoiding *resource barriers*. Resource barriers occur in incremental parallelizing compilers when all available resources of a given type have been allocated in a particular cycle t. This limits the motion of instructions in cycle $t+1$ into cycle t, which require the allocated resources, as well as the application of some transformations in t and $t+1$, even though motion of instructions in $t+1$ could continue through cycle t to cycle $t-1$. Thus, the allocation of all resources of a given type at some point in the code can form a barrier to further parallelization and by not allowing all registers to be allocated, this situation can be avoided as far as registers are concerned.

Also, another enhancement is to examine the instructions which are dependent upon the load instruction which is to be removed, to determine if performing the optimization will allow the motion of those instructions to progress. Recall that the lifetime of the newly allocated register will span $l+1$ time cycles. If the dependent instructions can be moved into earlier cycles, then the lifetime of the newly allocated register is shortened. Depending upon the latency of loads (the value of l), various metrics can be used (e.g., Will the lifetime of the new register shorten to $l-1$? $l-2$? $\frac{l}{2}$? etc.) to determine transformation application.

Figure 4.13 contains an algorithm to perform the heuristic decision of whether to perform the load-after-store optimization or not. First, if there would be no more

free registers if the current free register is allocated (i.e., there are not two or more free registers), then the heuristic inhibits the optimization by returning false. If there are free registers, then all of the instructions dependent upon the load are collected to determine if they can move into earlier cycles by *instr_could_move*. If the instructions can move, then true is returned, otherwise false. To use this heuristic, line #8 of the load-after-store algorithm in Figure 4.7 must be modified to:

8: if (/* a free cell exists */ && *load_store_heuristic*(load))

Loop Invariant Removal

For loop invariant removal, there are two cases to consider: the effects of loop invariant removal 1) from an innermost loop (as depicted in Figures 4.9-4.11) and 2) from an outer loop (as depicted in Figure 4.12).

For the first case, several possible heuristics exist:

1. The number of uses of the invariant value.
2. The number of invariants within the loop.
3. The number of instructions associated with the address calculation of an invariant instruction.

The first heuristic trades-off the allocation of a register to the number of uses of the invariant value within the loop. If the number of uses is above some threshold value (which can be determined *a priori* according to the characteristics of the applications), then the optimization is performed, thus avoiding high memory traffic when an invariant is heavily accessed. The second heuristic seeks to reduce the code size within the loop in the case that there are several loop invariants. Associated with each invariant is a series of address calculation instructions. When many invariants exist, many instructions (those involved with the address calculations) will be removed from the loop as well, once loop invariant removal is performed. Similarly, the last heuristic seeks to eliminate the instructions associated with address calculation when that number is large, which occurs with multi-dimensional array accesses. In this case, both

```
 1:  function loop_invariant_heuristic(invariant) returns boolean
 2:  begin
 3:    usage_cnt = get_num_uses (invariant)
 4:    if (usage_cnt <¡ usage_threshold)
 5:      Return FALSE
 6:    num_invars = get_num_invars (loop)
 7:    if (num_invars < num_threshold)
 8:      Return FALSE
 9:    Instrs = get_definitions (invariant)
10:    if (num_invars < num_threshold)
11:      Return FALSE
12:    Return TRUE
13:  end loop_invariant_heuristic
```

Figure 4.14: An algorithm for the loop invariant removal heuristic.

the number of instructions as well as the latency of that calculation removed from the loop can improve performance.

For the second case, invariant removal from an outer loop, the same heuristics that have been outlined above are applicable. However, the loop invariant removal optimization should be inhibited from outer loops until all contained inner loops have been scheduled. This will keep the loop invariant removal optimization from allocating registers, which may be useful to the optimization of the inner loops, until it is known that those registers are not necessary.

Figure 4.14 contains an algorithm to perform the loop invariant removal heuristic decision of whether to apply the optimization or not. This algorithm is broken into three tests, one for each of the heuristics outlined above. Depending upon which heuristic is selected, true/false values are returned to indicate: 1) if the invariant is heavily used; 2) there are a large number of invariants in the code; or 3) there are a large number of address computation instructions. Also, all heuristics may be selected and used causing invariants to be removed only when an invariant is heavily used, has several instructions involved in the address calculation and the number of invariants in the loop is high. Of course, the respective thresholds may be lowered when all heuristics are used together. Line

#11 of the loop invariant removal algorithm in Figure 4.5 changes to:

11: if (/* conditions met */ && *loop_invariant_heuristic* (invariant))

to use this heuristic.

Chapter 5

Integrating Program Transformations

In Chapter 4, a technique is presented to eliminate redundant memory accesses. This technique is especially suited to memory-intensive application areas such as image and video algorithms where, during the manipulation of images, temporary working storage for the new (output) image as well as the old (input) image is required. Accessing the images is done explicitly in the application code by array references and translates into many load/store instructions in the final embedded code. In some cases, "windows" of data (a neighborhood of values) from the input are necessary to compute single values in the output. This suggests a large unbalanced expression tree with the neighborhood as leaves and the single output value as the root—an expression tree particularly suited to optimization by redundant elimination and tree height reduction.

In the process of implementation and integration of the redundant elimination algorithm into the UCI VLIW parallelizing compiler, it was observed that the optimization of typical image application codes was less than satisfactory. The cause: the "interference" of redundant elimination and tree height reduction. Although both optimizations share the common goal of reducing the critical path length and do not compete in terms of the resource utilization that they optimize, the expression trees were not being compacted as much as possible due to the local nature of the optimizations.

In this chapter, a methodology for integrating program transformations is proposed. Transformations are integrated through the use of a "meta-transformation" which *globally* guides the application of transformations so as to make better decisions concerning the utilization of resources and to allow trade-offs between transformations rather than allowing transformations to optimize based on local factors as opportunities arise.

```
For J = 1 to N
  For I = 1 to M
    B(i,j) = (A(i-1, j-1) + /* A */
             A(i, j-1) +    /* B */
             A(i+1, j-1) + /* C */
             A(i-1, j) +    /* D */
             A(i, j) +      /* E */
             A(i+1, j) +    /* F */
             A(i-1, j+1) + /* G */
             A(i, j+1) +    /* H */
             A(i+1, j+1))/9  /* I */
```

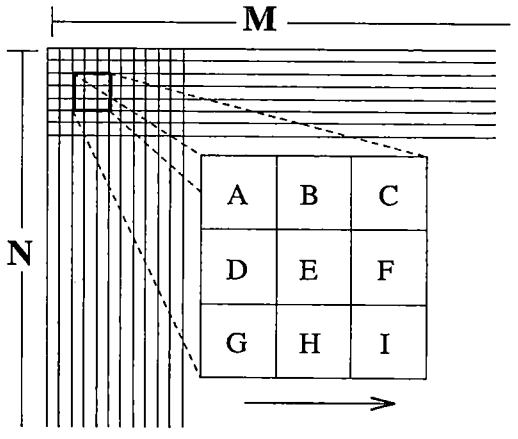


Figure 5.1: Example image convolution code.

5.1 Introductory Example

As an illustration, consider the (simple) image convolution code in Figure 5.1 which assigns a new value to an image based upon the average of values from an old image. As the algorithm progresses, a window "sweeps" across the image from left-to-right. From iteration-to-iteration of the inner loop, the window shifts one position to the right so that, for instance, "B" from the current iteration becomes "A" in the next iteration. Thus, due to this redundancy in array accessing, only three new values are loaded each iteration ("C," "F" and "I").

In Figure 5.2(a), a dataflow graph for the inner loop appears and in (b) and (c) two distinct versions of that graph appear, scheduled with the resource constraints of one pipelined two-cycle latency adder and one non-pipelined

memory port with two-cycle latency[1]. Both versions consist of eight instructions and the height of the tree (i.e., the length of the critical dependency chain) in (b) is four, while in (c) it is five. Although the tree in (c) has greater height, the performance of the resulting schedule is faster than in (b). In (b) instructions add redundant[2] and non-redundant memory values (C and D, for instance) whereas in (c) instructions add memory values with the same redundancy types (E and D, for instance) resulting in different schedule lengths *given the available resources.*

5.2 Overview of Incremental Tree Height Reduction

The tree height reduction (THR) transformation reduces the height of an expression tree by balancing sub-trees of computation. In [62], an incremental algorithm for THR is presented. This algorithm is invoked by the Percolation scheduling *move_op* transformation (or other suitable local compaction routines) when an instruction cannot move into earlier time steps due to data-dependency. THR analyzes the chains of instructions that the current instruction is dependent upon to determine if re-structuring (i.e., balancing) that chain of instructions results in the earlier execution of the current instruction.

5.2.1 Previous Work

Tree height reduction was first studied [5, 61] as a method for reducing critical dependency chains to increase parallelism and was later extended in [15]. Various synthesis systems [47, 80] include support for THR, but do not specifically factor resource availability into the process, or do so in an exhaustive manner [58]. In contrast, an *incremental* reduction technique is presented in [62] which globally

[1]For clarity, no restriction on the number of registers (i.e., temporaries) is imposed and the division of the sum by nine is omitted.

[2]Throughout this chapter, the term *redundant* refers to computation on memory values loaded and/or stored redundantly in loop execution.

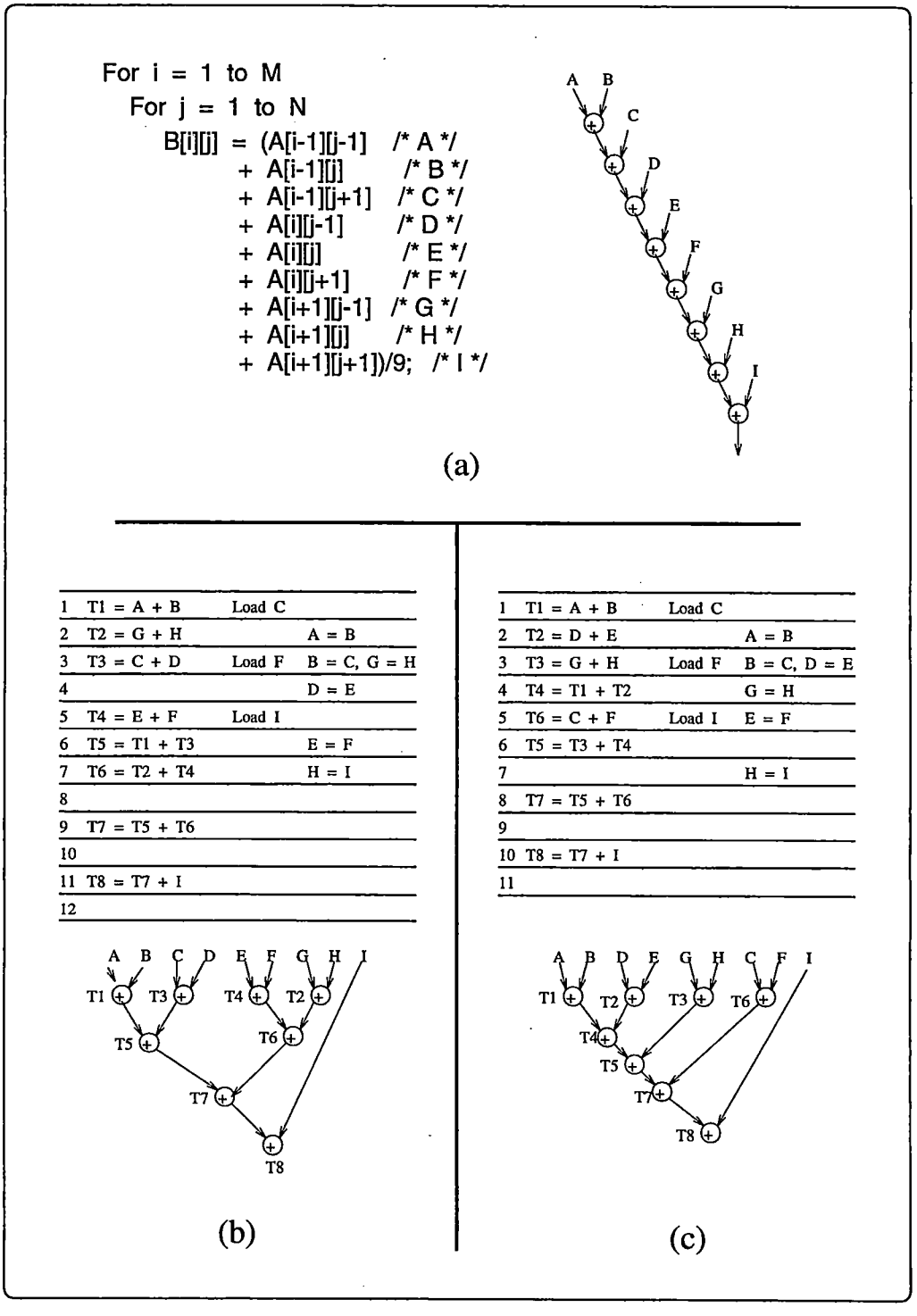


Figure 5.2: Schedules for tree height reduced graphs.

(i.e., over multiple expressions in the program) exploits unused resources by factoring resource availability into the THR conditions.

5.2.2 Definitions

These terms are used in the exposition of the THR algorithm:

Operation: Each instruction is composed of an operator (the instruction's type) and one or more *use* variables to which the operator is applied resulting in the *definition* of a single variable, which may or may not be one of those referenced by this instruction. In the instruction: $a := b + c$, addition is the operator, the used variables are b and c and the defined variable is a.

Early_op and **late_op**: Denote the relation between the instructions which define the variables used by the current instruction in terms of the "time" when those arguments are available for computation. In the following:

cycle (i) $a := b + c \quad d := e + f$
cycle (i+1) $g := a + 7$
cycle (i+2) $h := g * d$

the instruction $d := e + f$ is the *early_op* and $g := a + 7$ is the *late_op* of instruction $h := g * d$. In the case where instructions which define use variables are scheduled in the same cycle, the choice is arbitrary.

Brother: Two instructions are said to be "brothers" if the variables that each define are used as arguments to some other instruction. In the example above, instructions $g := a + 7$ and $d := e + f$ are *brothers*.

5.2.3 An Incremental THR Algorithm

In [62] the algorithm presented continually applies THR along a selected path as long as resources and opportunities for transformation exist. Here a variant of that algorithm is presented which only applies THR to a chain of instructions

```
 1:  function perform_THR(op, from_step)
 2:  begin
 3:    /* get early_op and late_op */
 4:    if (/* conditions satisfied */) then
 5:      switch:
 6:        case /* associativity */ :
 7:          Associativity_Analysis(op)
 8:        case /* distributivity */ :
 9:          Distributivity_Analysis(op)
10:      end switch
11:      perform_THR(/* new sub-expression */)
12:    endif
13:  end perform_THR
```

Figure 5.3: Main procedure for incremental THR algorithm.

(i.e., one particular expression) if THR application causes the current instruction to be executed in an earlier time step, continuing down (i.e., towards leaf instructions) the expression if THR was applied to further compact newly created sub-expressions. Figure 5.3 contains an incremental THR algorithm.

This incremental THR algorithm first finds *early_op* and *late_op* of the current instruction and *early_op* of the current instruction's *late_op* (which we will call *late_early_op*). These instructions are then examined under the following rules to determine if application of THR is beneficial:

1. The current instruction's *early_op* must be available two cycles[3] earlier than the current cycle.
2. If the current instruction is an ADD or SUB, then the instruction it is dependent upon must also be ADD or SUB. If the current instruction is a MUL, then the dependent instruction may be any one of ADD or SUB or MUL.
3. There exist free resources for the (to be) newly created instructions in the relevent nodes.

If the conditions for THR are satisfied then the appropriate routine is invoked depending upon the relationship between the current instruction and *late_op*. For

[3]This is presented assuming one cycle instructions. The extension to multi-cycle instructions is straight-forward.

the ADD/ADD, ADD/SUB, SUB/ADD, SUB/SUB, and MUL/MUL combinations the associative arithmetic property is applicable. For MUL/ADD and MUL/SUB the distributive law is used. The respective analysis routines create new instructions according to the respective mathematical properties, thereby re-structuring the code into a more compact form. Finally, recursive calls are made to further compact the newly created sub-expression.

5.3 Integrating Transformations

As noted earlier, tree height reduction (THR) and redundant memory-access elimination (RE) share the common goal of reducing the critical path length and do not compete in terms of the resource utilization that they optimize. However, if the THR transformation does not exploit redundancy information, then the schedule produced may not be as compact as possible. Likewise, if the RE transformation does not exploit the knowledge that THR has "paired" two redundant memory instructions, then, in the presence of scarce resources, registers may not be allocated to redundant values properly so as to allow the best compaction possible with those resources. In general, an approach is necessary which guides the application of transformations so that trade-offs may be assessed and made between various transformations as possibilities arise.

5.3.1 The META-Transformation

In the traditional approach, a suite of program transformations is available to the scheduler and these transformations are applied whenever opportunity arises. Typically, there are two heuristic decisions that are implemented within a transformation: *when* to apply the transformation (e.g., what conditions are necessary) and *how* the transformation is performed (e.g., implementation details of a transformation). In Figure 5.4(a), this approach is depicted with the heuristic decisions noted in the figure using "w" and "h" symbols, respectively.

In cases when multiple transformations are ready (i.e., may be applied), one is

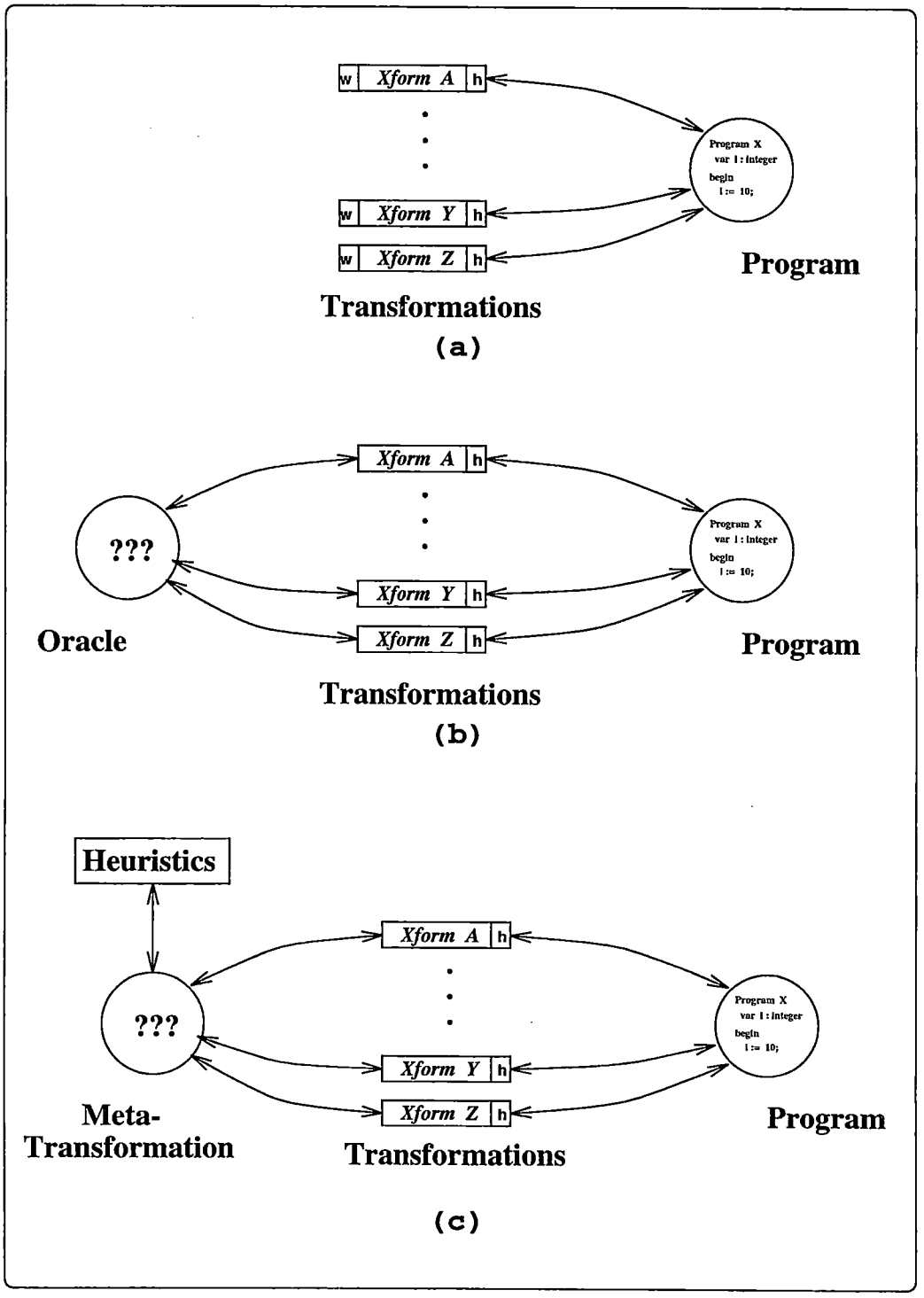


Figure 5.4: Methods of approach.

chosen, applied and then those transformations which become ready as a result and allow for further refinement are added to the "pool" of transformations that can be applied. Conventionally, an ordering of those transformations is used which is determined *a priori* by some investigation into "common" situations [3, 8, 83] (e.g., transformation X is always applied before transformation Y, etc.). Thus, in this approach, transformations are applied without consideration to their effects on the current application code and the applicability of other tranformations on the resultant code.

In order to improve the quality of the schedules generated, consolidatation of all of the "when" decisions for each transformation into one entity is proposed. As depicted in Figure 5.4(b), with perfect information, this decision entity becomes an oracle. During scheduling, the oracle is queried to determine which transformation should be applied at any given time thereby obtaining an optimal solution.

However, practically, a heuristic decision maker is necessary for this task. In Figure 5.4(c), the oracle is replaced with the META-Transformation and a set of heuristics. The META-Transformation is queried by the scheduler during optimization to determine if a ready transformation should be applied or not. This approach differs from the traditional approach as, dependent on the heuristics, transformation X may be applied before transformation Y, but under different conditions, transformation Y may be applied before transformation X when both transformations can be applied to the current code. Using this approach, the global effects of all transformations are localized into one region and their combined effects can be studied and trade-offs assessed, possibly resulting in the refinement of the heuristics as more interactions are discovered.

5.3.2 Heuristic META-Transformation

The heuristic that is selected for integrating THR and RE is based upon the observations that: 1) instructions which become paired as a result of THR can move farthest if their operands are redundant loads or intermediate computations

on those values; and 2) preferential allocation of registers should be given to redundant values so that variable lifetimes are shortest. Thus, the heuristic is a greedy scheme that only allows THR to progress when the operands of a new THR instruction are either all redundant or all non-redundant sub-trees and allows RE to eliminate redundant memory instructions once they become part of a redundant sub-tree.

In order to determine redundancy a memory analysis phase is conducted prior to scheduling and is composed of two parts: determining memory instruction candidates for removal and the propagation of the redundant value information. Algorithms for both phases are given and then the heuristic META-Transformation is presented.

Determining Candidates

Determining candidates is done through comparison of the symbolic expressions for each pair of memory instructions that access the same array. Since an instruction's symbolic expression encodes the referenced address for the current iteration, subtracting one expression from that of another memory instruction will reveal the difference in addresses between the two instructions under consideration. If the value resulting from normalization of this difference by the appropriate element-size is an integer (if not, then it is not possible that the two instructions exhibit redundancy), this normalized value denotes the distance in terms of loop iterations. For example, in the behavior (numbered from left to right):

```
for i = 1 to N
   A[i] = A[i-1] + A[i-2] + A[i-3] + B[i]
end
```

the symbolic expressions for memory instructions one ($A[i]$) and two ($A[i-1]$) differ by the element size of four. When normalized by that size, a distance of one results. Thus, instruction one stores a value in the current iteration that instruction two loads in the next iteration, exposing instruction two as a redundant load over loop execution and a candidate for removal.

If the distance between two instructions is large, either the instructions refer to disjoint arrays (e.g., *A[i]* and *B[i]*) or they reference array locations that are separated by a number of iterations which makes redundancy removal during loop pipelining[4] not practical (e.g., *A[i]* and *A[i+50]*). In the later case, the criteria for candidacy can be restricted to the case where the distance between two instructions is less than some threshold value[5].

Figure 5.5 contains an algorithm to determine memory instructions which are candidates for removal and is essentially built on top of the techniques presented in Chapter 4. The symbolic expression of each memory instruction in the program is compared with all other memory instructions' symbolic expressions that reference the same array to determine the distance between two references. When this distance is normalized by the array element size, the number of iterations over which redundancy spans is known. If this distance is less then the selected threshold, the instruction is tagged as redundant.

Because of the order in which instructions are compared, the distance value can be either negative or positive. A negative value corresponds to the situation where the first instruction will use a value in the future that the second instruction currently loads or stores. Therefore, for a negative distance, the first instruction is redundant. A positive distance value denotes the case where the first instruction currently loads or stores a value that will be used in the future by the second instruction. In this case the second instruction is the redundant instruction.

Although it is not necessary to determine the actual iteration distances between all memory instructions (it is sufficient to know only that an instruction is redundant—that the iteration distance is an integer), this information can be useful when assessing trade-offs between candidates for removal. For instance, if redundancy spans more than five iterations, then it might be likely that the

[4]Most loop pipelining algorithms converge with a new loop schedule within a small (i.e., four or five) number of iterations. Therefore, determining redundancy which spans more than that number of iterations is most likely not useful.

[5]Picking this value is not an issue. Our experience has shown that a value of five is enough to expose a sufficient amount of redundancy which can be exploited during loop pipelining.

```
1:  Procedure memory_analysis(program)
2:  begin
3:    foreach node in program
4:      if (/* node contains memory operations */) then
5:        foreach mem_op in node
6:          /* Add mem_op to op_list */
7:          /* Convert mem_op's sym. expr. into disjunctive form. */
8:        end
9:      endif
10:   end
11:
12:   while (op_list ≠ φ) do
13:     /* remove curr_op from op_list */
14:     foreach mem_op in op_list
15:       if (/* m1 and m2 access the same array */) then
16:         foreach ex1 in m1's symbolic expressions
17:           foreach ex2 in m2's symbolic expressions
18:             set expr to ex1 - ex2
19:             set distance to expr/data element size
20:             if (/* distance is integer and distance < threshold */) then
21:               if (distance < 0) then
22:                 /* tag m1 as a candidate for removal */
23:               else
24:                 /* tag m2 as a candidate for removal */
25:               endif
26:             endif
27:           end
28:         end
29:       endif
30:     end
31:   end
32: end memory_analysis
```

Figure 5.5: An algorithm for determining removal candidates.

```
 1:  Procedure propagate_redundancy_info(program)
 2:  begin
 3:    changes = false
 4:    while (changes) do
 5:      foreach node in program
 6:        foreach op in node
 7:          def_ops = /* ops which define vars that op reads */
 8:          tmp_tags = /* union of red. tags for def_ops */
 9:          changes = changes or tmp_tags ≠ op's tags
10:          /* tag op with tmp_tags */
11:        end
12:      end
13:    end
14:  end propagate_redundancy_info
```

Figure 5.6: Algorithm for propagating redundancy information.

register allocated for that value can be more profitably utilized elsewhere if that value is not accessed often.

Propagating Redundant Memory Value Information

Once the candidates have been identified, that redundancy information is propagated up (or down!) the expression tree. Figure 5.6 contains an algorithm that marks all instructions in the program with information about redundancy in their sub-trees and is patterned after standard flow analysis routines.

Initially, each instruction is tagged with its candidacy status if it is a memory instruction, otherwise it is tagged with null (denoted as ϕ). Then the algorithm iterates until no further changes occur to the tagging status. For each instruction in the program, the instructions which define the variables used by the current instruction are found and their tagging information is unioned to derive the local redundancy information on that sub-expression.

As an example, consider the following code segment:

cycle (k): $b = A[i]$ $c = A[i-1]$
cycle (k+1): $d = c + 7$
cycle (k+2): $e = d + b$

The instruction which defines c is redundant and tagged with $\{r\}$ while the instruction which defines b is non-redundant and tagged with $\{n\}$. Operation $d = c + 7$ is tagged with $\{r\}$ since the instructions (only one, in this case) that define the variables used are tagged with $\{r\}$. Finally, instruction $e = d + b$ is tagged with $\{n, r\}$ denoted as $\{both\}$, which is the union of the tag information for the instructions defining those variables used.

Heuristic META-Transformation

The META-Transformation appears in Figure 5.7. Because the goal is to pair values with the same redundancy tags, the THR transformation is given the "go ahead" when the tags match. In this case the sub-trees are recursively descended to tag redundant memory instructions as "ready" for elimination. If the redundancy tags mismatch, the redundant sub-trees are rotated. This has the effect of moving computation on redundant values towards the leaves and non-redundant computation towards the root allowing the schedule to better tolerate the latency of memory instructions which will not be removed.

The RE transformation is inhibited from removing redundant memory instructions until they become part of redundant sub-trees (i.e., tagged as "ready"). This has the effect of minimizing the live ranges of redundant memory values stored in the register file.

5.3.3 An Example

As an example of the workings of the META-transformation, the simple low-pass filter from section 5.1 (from which an excerpt appears) is used with the same constraints of one two-cycle pipelined adder and one five-cycle pipelined memory port. The initial schedule is depicted in Figure 5.8(a).

During the memory analysis phase, instructions A, B, D, and E are tagged as redundant and instructions C and F as non-redundant. This information is propagated up the expression tree.

```
 1:  function META-Transformation(xform, its arguments)
 2:  begin
 3:    case xform is:
 4:      Tree-height reduction:
 5:        if ( /* paired args redundancy tags match */) then
 6:          /* recursively descend tree tagging redundant */
 7:          /* memory operations "ready" */
 8:          Return Go_Ahead
 9:        else
10:          /* "rotate-down" any redundant subtrees */
11:          Return Inhibit
12:      Redundant Memory Elimination:
13:        if ( /* memory op is tagged "ready" */) then
14:          Return Go_Ahead
15:        else
16:          Return Inhibit
17:    end case
18:  end META-Transformation
```

Figure 5.7: The META-Transformation

The META-transformation tags A and B as "ready" when it assesses the possibility of applying THR to the instruction $T1 = A + B$. Once they are tagged ready, those load instructions are removed. And, after instructions are compacted, the schedule in Figure 5.8(b) results.

The THR algorithm is inhibited by the META-Transformation from application on the instructions which define T2 and T3 in (b) as this would result in the pairing of redundant and non-redundant memory instructions D and C, respectively. However, THR is applied to the instructions which define T3 and T4 as the operands (D and E) for the new instruction would have the same redundancy type. Once D and E become paired, they are marked "ready" and those loads are eliminated. After compaction the schedule of Figure 5.8(c) results.

Next, THR is applied to the instruction $T4 = T2 + T3$. At this point, subtrees at $T2$ and $T4$ are "rotated-down" so that both operands ($T1$ and $T3$) of a new instruction will have same redundancy types. After compaction, the schedule in Figure 5.8(d) results, thus producing the sub-tree rooted at $T7$ in Figure 5.2(b).

5.4 Effects of META-Transformation on Register Allocation

In this section, the introductory example is used to illustrate the effects of code transformation on register allocation. As the code is scheduled, both with the META-Transformation and the traditional approach, the resulting code is examined in terms of the lifetimes of the values.

5.4.1 Examining the Effects of Optimization

In Figure 5.9, pseudo-code for an incremental code motion routine in a scheduler appears. In the traditional approach, the ordering of transformation application is "hard-coded" while, in contrast, with the META-Transformation, they may be

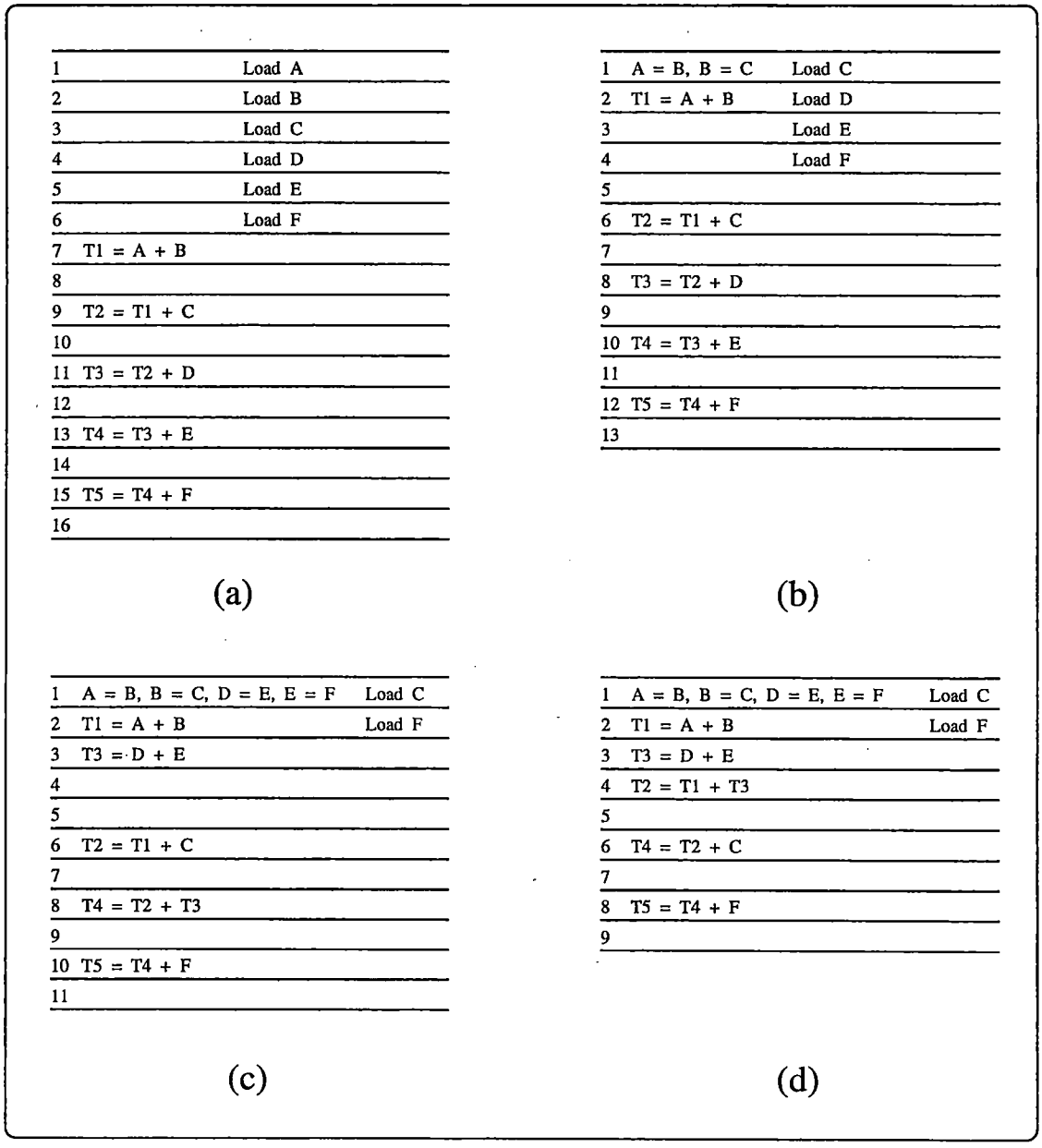


Figure 5.8: Application of META-Transformation on low-pass filter.

For all instructions, I, in cycle
MoveInstr(I)

Traditional

```
if (THR_applied(I))
    return THR;
else if (RME_applied(I))
    return RME;
else ...
end
```

META-Transformation

```
if (META-Xform(THR, I) == go_ahead)
    THR(I)
else if (META-Xform(RME, I) == go_ahead)
    RME(I)
else ...
end
```

Figure 5.9: Pseudo code for an incremental scheduler.

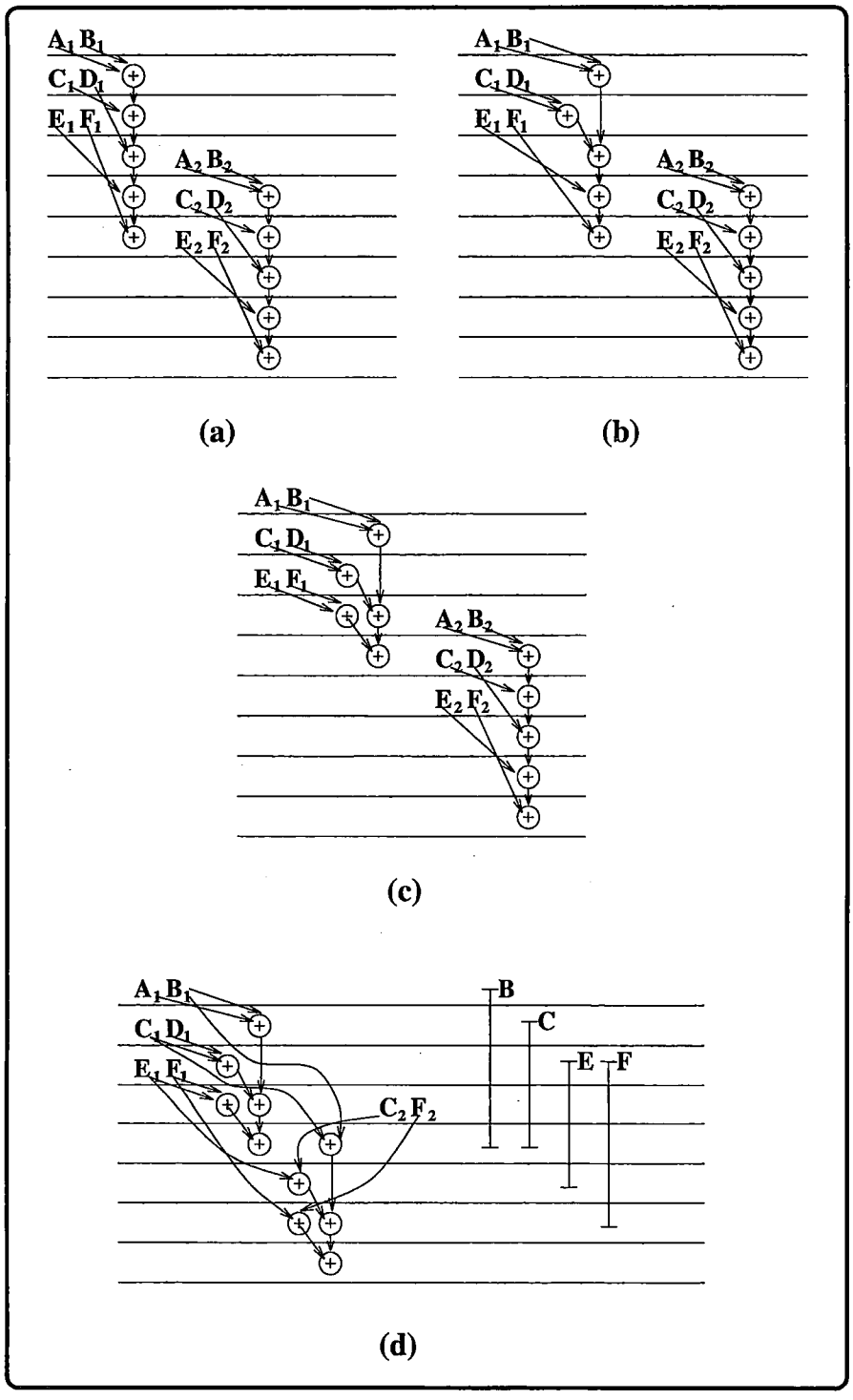


Figure 5.10: Traditional scheduling approach.

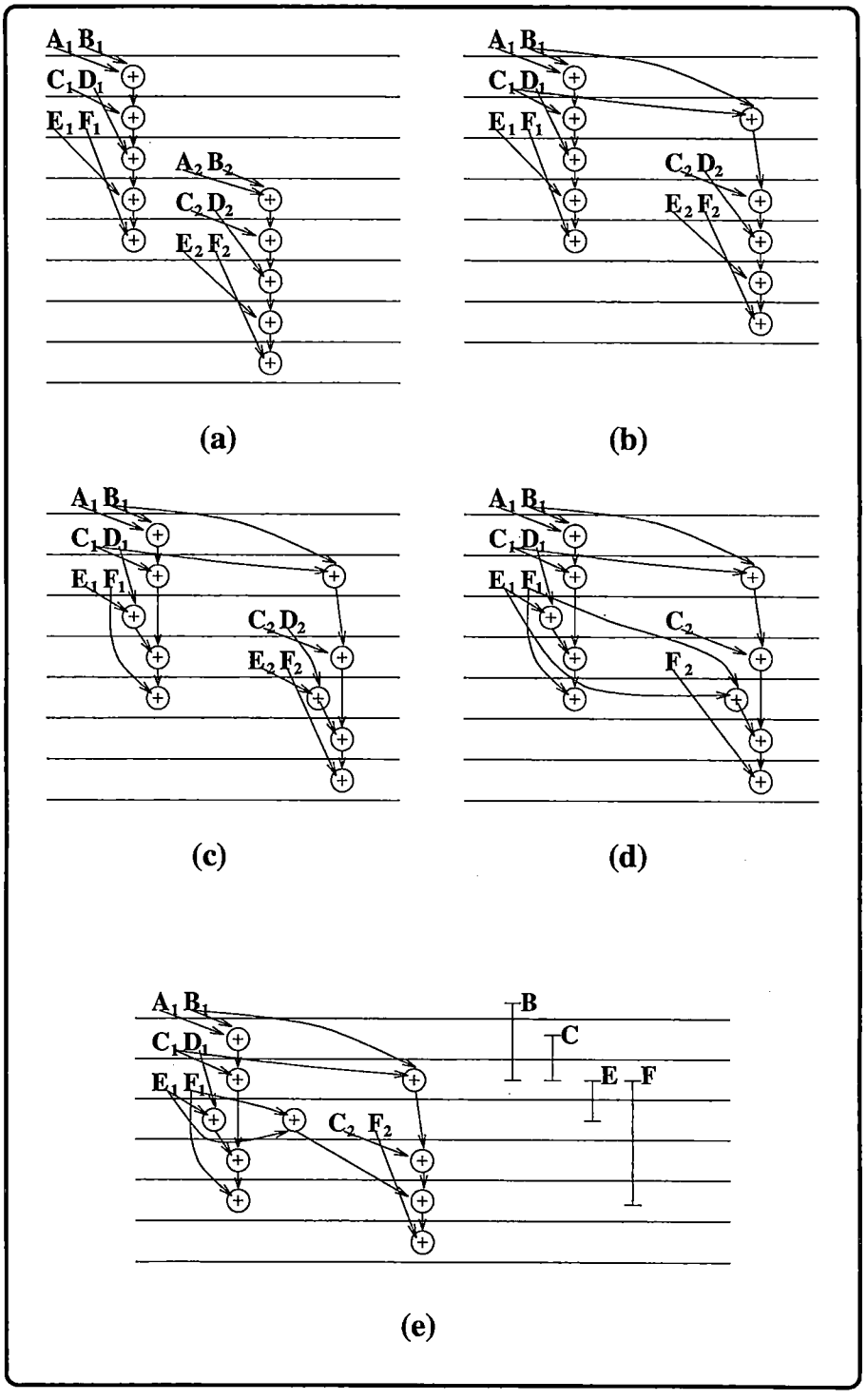


Figure 5.11: META-Transformation approach.

accomplished in any order[6].

This routine is used to examine the effects of register lifetimes when scheduling the example code with the resources of two memory ports and two adders and all instructions are uni-cycle.

In Figure 5.10, a partial tree for the introductory example to this chapter appears. In (a) two iterations of the graph appear which have been scheduled with the given resource constraints. As the traditional scheduler passes through the code in (a) there are three instructions in cycle four which may be optimized, an add and two redundant memory instructions. Because the decision for THR is made first, THR is applied to form the subtree which adds C_1 and D_1. This forms the tree and schedule found in (b). As the scheduler continues, THR is again applied to form the subtree which adds E_1 and F_1, deriving the schedule found in (c). The graph for iteration two is likewise re-structured. After re-structuring, the redundant loads of iteration two are removed, leaving the schedule of (d).

Also depicted in (d) are the lifetimes of the values which were redundantly loaded: B_1 extends five cycles, C_1 extends four cycles, E_1 extends four cycles and F_1 extends five cycles. Note that this long extension of the lifetimes (particularly B_1) is due to the re-structuring of the first iteration graph before the redundant loads were removed. The instruction which added A_2 and B_2 is blocked (due to a resource barrier) serving to extend those lifetimes over many cycles.

In Figure 5.11, the partial tree for the introductory example again appears. Initially, the loads of A_2, B_2, D_2 and E_2 are tagged as redundant. As the META-Transformation passes through the code in (a) it discovers opportunities for THR and RME in cycle four. The THR optimization is not applied as it does not pair subtrees with the same redundancy tags. The loads of A_2 and B_2 are removed as they are both redundant and have become "paired" by the add instruction in cycle five. After the add instruction moves up to cycle three the graph in (b) results.

[6]Although it appears ordering is fixed, the META-Transformation mechanism actually applies transformations in differing orders based upon global decisions as previously discussed.

As the META-Transformation continues, THR is enabled to pair the loads of E_1 and D_1 and E_2 and D_2 which derives the graph of (c). Because the redundant loads D_2 and E_2 are now paired and are part of a redundant subtree, they are removed resulting in the graph of (d). As scheduling continues on the graph of (d), the operation which adds the redundant values D_2 and E_2 is "rotated down" and the schedule in (e) results.

Also depicted in (e) are the lifetimes of the redundantly loaded values: B_1 extends three cycles, C_1 extends two cycles, E_1 extends two cycles and F_1 extends four cycles. This is an improvement over the lifetime lengths produced by the traditional approach and serves to reduce register pressure as lifetimes are now shorter and less contention for those resources results.

5.4.2 An Enhanced META-Transformation

The presented version of the META-Transformation is able to utilize resources more effectively than the traditional approach. This is due to the global decisions that the heuristics provide, rather than relying solely on local factors. Although the presented heuristics achieve the goal of reducing value lifetimes to reduce register demands, it is possible that the scheme may be slightly improved.

In the previous example, when redundant loads became "paired" they were removed. Potentially, there may be multiple memory instructions that become paired before any are removed. When there are scarce resources, only two registers available, for instance, and there are many memory instructions that can be paired and then removed, the heuristic must be refined so that loads from the same pairings are removed before other loads. In the previous case, this would be pairing A_2, B_2 and D_2, E_2 and then removing A_2 and D_2, for instance. Clearly the benefit of load removal is lessened in this case as the compaction of the code is not fully realized.

A simple solution to this problem is to generate unique *keys* for memory operations and tag those instructions with them so that when considering optimizing memory operations a particular key is selected and then *both*

associated memory instructions are removed before memory instructions with a different key are removed. This will keep the above case, where there were four memory instructions ready for removal, but only two available registers, from hindering the compaction process.

Chapter 6

Generalizing Copy Elimination

Coupling register allocation with instruction scheduling or instruction parallelization is a challenging task. Traditionally, register allocation is accomplished as a separate phase during compilation with the most pervasive paradigm being graph coloring. Because of its complexity, it is impractical to apply a register allocator, such as the graph allocator, to the code repeatedly during parallelization. As discussed in Chapter 2, strategies for performing register allocation in a parallelizing compiler include pre-scheduling allocation, post-scheduling allocation and integrated scheduling and register allocation. Post-scheduling allocation strategies suffer from introduced spill code degrading the schedule while the integrated approaches require substantially more compilation time.

In the UCI VLIW parallelizing compiler project [70], the strategy adopted is to start with heavily optimized sequential code which has been register allocated. Because the sequential code has been heavily optimized, registers are re-used as much as possible. This presents a problem during parallelization as it introduces many "anti"-dependencies. To remedy this, registers are re-allocated "on-the-fly" during scheduling by a technique called register or variable *renaming* [28]. This technique allocates a currently free register to the destination of an anti-dependent instruction to remove the dependency and then inserts a copy instruction (a register-to-register move instruction) into the code to copy the

value in the newly allocated register to the original destination register.

Although the largest source of copy instructions in this compiler arises from renaming, other sources of copy generation exist. Aggressive optimization and parallelization techniques, including "classic" optimizations such as common sub-expression elimination [3] and induction variable elimination [3] as well as more sophisticated techniques such as variable lifetime splitting [25] and redundant load elimination [12, 51] (see Chapter 4), add copy instructions to the code in order to control compiler complexity as global search-and-replace of register names or variables, each time that an optimization is performed, is too costly.

Even though the introduced copy instructions perform no significant computation (they preserve program correctness), they require the use of a functional unit[1] or specialized bussing structure to accomplish the data transfer. Thus, as copy instructions are essentially overhead instructions, their presence in a schedule represents a negative impact to performance. While aggressive program optimization and parallelization techniques improve the performance of application code, their improvement potential is possibly limited by the presence of those copy instructions that remain after optimization. Removing these copy instructions becomes crucial to achieving improved performance and/or eliminating specialized hardware for supporting those data transfers.

In this chapter, a generalized technique for copy removal is presented. This technique removes copies that keep values live over loop iterations by unrolling the loop code and re-allocating registers to instructions. As a result, copies which preserve values within an iteration—"traditional" copies—are also removed. Thus, this technique subsumes conventional copy propagation techniques while providing a method for removing more advanced forms of copy instruction chains in the generation of high-performance code.

[1]Practically, a copy $R1 = R2$ would be performed by executing some instruction such as $R1 = R2 + 0$ or $R1 = R2 << 0$.

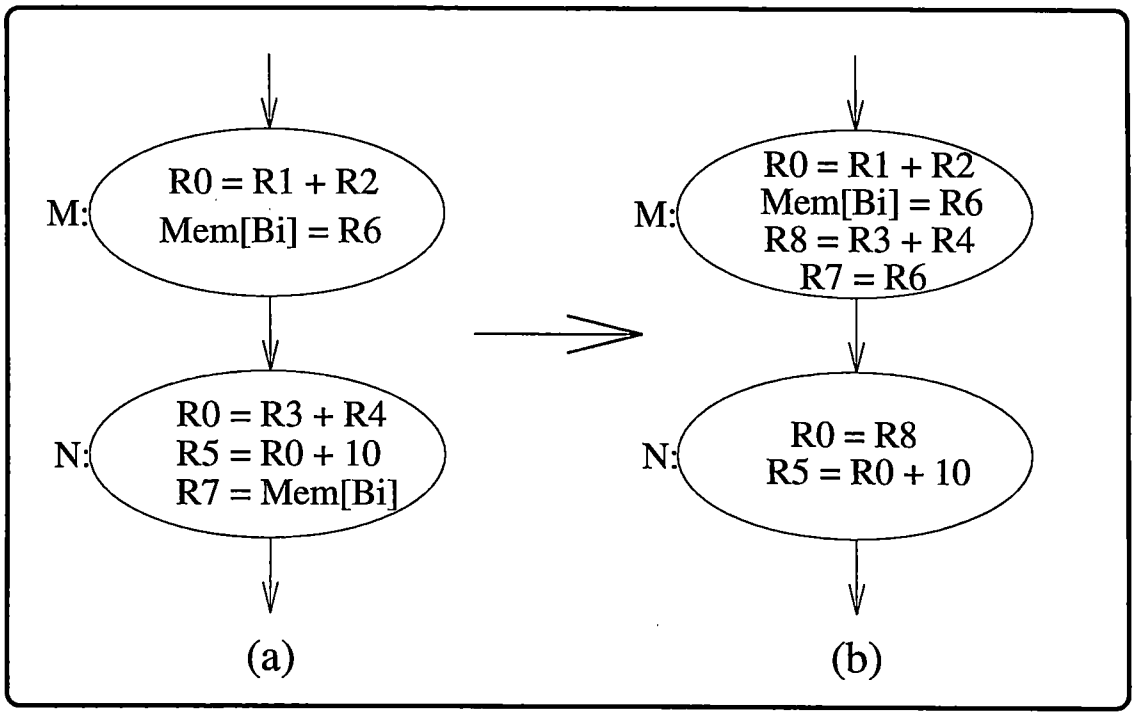


Figure 6.1: Introducing copy instructions into the code.

6.1 Introductory Example

As an introductory example, consider the code in Figure 6.1(a) which will serve to demonstrate how parallelization inserts copy instructions into code. In this example, the value written into $R0$ in node M is used by the instruction $R5 = R0 + 10$ in the next node[2]. During parallelization, the instruction $R0 = R3 + R4$ from node N may not be moved into (i.e., executed in parallel with the instructions of) node M as it redefines $R0$ and would cause incorrect values to be computed by the instruction $R5 = R0 + 10$ in node N. If a free register exists ($R8$ in this example), then the destination register of instruction $R0 = R3 + R4$ is renamed and the new instruction, $R8 = R3 + R4$, is moved into node M. To maintain correctness, a copy instruction $R0 = R8$ is necessary in node N to correctly move the value from $R8$ into $R0$, thereby compacting the

[2]Note that the value read by the instruction $R5 = R0 + 10$ is that produced by the instruction $R0 = R1 + R2$ in node M rather than the instruction $R0 = R3 + R4$ in node N as, due to the machine model, all operands are read before any results are written.

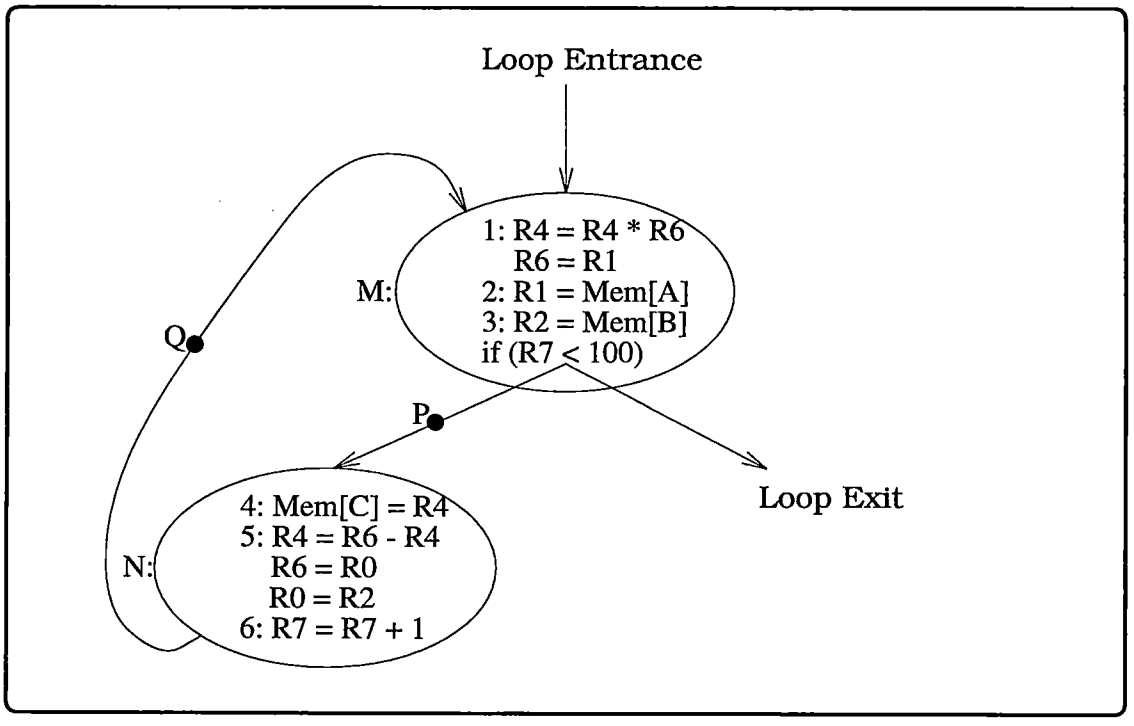


Figure 6.2: Loop code with copy instructions.

code of (a) to that of (b).

Also, in node M, a value is stored[3] to the memory location B_i. In node N, this value is loaded from the memory by the load instruction $R7 = Mem[B_i]$. Rather than re-loading the value from memory, the value stored to memory by instruction $Mem[Bi] = R6$ can be directly copied. Thus, the load is removed, resulting in the earlier availability of the value (i.e., the latency of the load is removed), and the copy instruction $R7 = R6$ is added to node M.

In the previous example, simple copy propagation may be used to eliminate the copies as the code is straight-line. However, during software pipelining when multiple iterations of a loop are overlapped and optimizations are performed, the uses of copied values can span across iteration boundaries. Thus, conventional copy propagation techniques [3] are not powerful enough to remove many of the

[3]For simplicity, loads and stores are shown here symbolically. Typically, the address is calculated into a register and analysis [12, 51] is required to determine equivalency in memory references. (See Chapter 4.)

copies introduced during loop parallelization.

As an example, in Figure 6.2, several optimizations were applied during the parallelization of a loop. As a result, several copy instructions are found in this code[4]. Conventional copy propagation cannot remove any of these copy instructions as those copies serve to keep values live over multiple iterations of the loop. For instance, the copy $R6 = R1$ in node M cannot be removed as it preserves a value loaded by $R1 = Mem[A]$ from the previous iteration (the previous execution of node M) and propagating $R1$ in place of $R6$ into the following node (into the instruction $R4 = R6 - R4$) would result in the use of an incorrect value (the newly loaded value from instruction #2 would be incorrectly used rather than the previously loaded value). Thus, conventional copy propagation is not adequate to remove these copies.

6.2 Eliminating Copy Instructions

Copies generated during parallelization do not *produce* new values but, rather, *preserve* already computed values for future uses. In other words, multiple values produced in various iterations by an instruction are simultaneously live and transferred from definition to last use by chains of copy instructions. Copies related to a specific copy chain cannot be removed without affecting the 'queueing' of values for use. As depicted in the code of Figure 6.3, which has been compacted into one node, by unrolling the loop body sufficiently, the definition of a value and its last use become explicit thereby eliminating the need for the copy chain and, thus, enabling copy elimination[5]. In this example, the solution loop spans three iterations of the original loop.

In this section, an algorithm for copy instruction removal is presented. This

[4]Note that four ALUs and two memory units are necessary to execute this parallel code, while, if copies were removed, only two ALUs and two memory units are necessary.

[5]Once again, recall that, due to the machine model, all operands are read before any results are written. Therefore, in the first node, for instance, the value "X" used by A = X + B is that generated by the previous execution of that node.

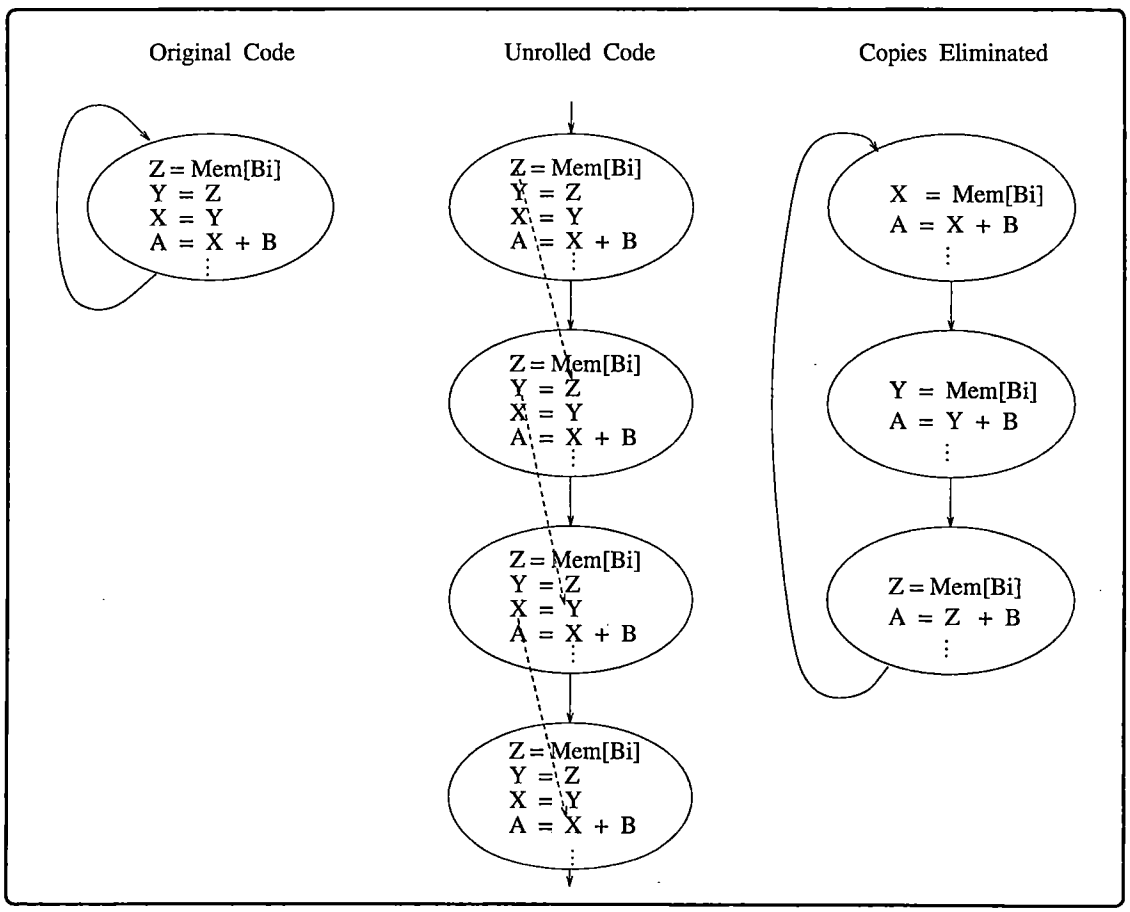


Figure 6.3: Unrolling loop code to eliminate copies.

technique focuses on parallelized loops and is based upon incremental loop unrolling and re-allocation of registers to values. First, some terms are defined and then the algorithm is presented.

6.2.1 Definitions

- A *register mapping* is a set of tuples of the form $(register, op.age)$, where *register* denotes the register name, *op* denotes the unique instruction identifier that produced the value and *age* corresponds to the number of times that a value has been copied (i.e., the length of the chain thus far). The *op.age* field will be collectively referred to as a *tag*. A register mapping may have multiple tuples for a given register due to multiple paths through the loop.
- A *loop* is a set of nodes in the program graph that form a cycle such that there is a path from each node in the loop to any other node in the loop.
- A *header* of a loop is the loop entry point, the node that corresponds to the top of a loop.
- A *backedge* is an edge connecting two nodes in a loop such that the destination node has a depth first search (DFS) number less than the source node; a loop may have multiple backedges.
- An *unrolling* L' of loop L along some backedge is an exact code duplication of the loop body L.

6.2.2 An Algorithm for Copy Elimination

Figure 6.4 contains an algorithm which performs copy elimination on a loop. As input, this algorithm takes a loop body or loop graph where each node contains instructions which are executed in parallel and produces a new loop without copy instructions which may span multiple iterations of the original (input) loop.

The first step of the algorithm is to compute the register mappings for each node in the graph. The register mapping for a node represents the contents of the

```
1:  Procedure copy_elimination(L : loop)
2:  begin
3:    /* compute initial register maps for L */
4:    /* split nodes having multiple defs, recalc register maps */
5:    /* make loop template */
6:    /* add the header of L to the headers_list */
7:    /* add all backedges found in L to backedges_list */
8:    scan_and_reallocate (header_of(L));
9:    while (not empty backedges_list) {
10:     /* unroll along backedge b from backedges_list */
11:     /* add new iteration headers and backedges to respective lists */
12:     scan_and_reallocate (header);
13:     /* match backedges to header nodes */
14:   }
15: end copy_elimination
```

Figure 6.4: An algorithm for copy elimination.

registers *immediately preceding* the execution of that node and are derived similar to program data-flow analysis. (An algorithm for this is presented shortly.)

In the next step, the derived register mappings are scanned and any node which has multiple annotations for a register mapping is split[6] as multiple definitions for a register are present. For example, for the code segment,

```
if (condition) then
    R0 = R1
else
    R0 = R2
endif
R3 = R0 + 3
```

the node containing the instruction $R3 = R0 + 3$ will have two predecessors, one for the true path and one for the false path which each uniquely define $R0$. Since the copies will be later eliminated, if the paths are not split then it is necessary to allocate the same register to the destination registers of both instructions which define $R1$ and $R2$ so that the instruction $R3 = R0 + 3$ will compute correct

[6]Multiple mappings are due to multiple paths or conditionals in the loop.

results. However, this restriction considerably complicates the (straight-forward) allocation process and the proposed solution is to split the node which contains the instruction $R3 = R0 + 3$ so that each path contains a unique copy of that instruction different allocations (to $R0$ in this case) are possible on each path. If any nodes are split by this step, the register mappings are recalculated.

At this point, a loop template, or copy of the loop with its register mappings, is made and used for future reference. During copy elimination, the source registers of each instruction are looked up in this template to determine the tags for the values they use in the original loop. As values may be re-allocated to registers during copy elimination, it may become necessary to update or to change the source operands of some instructions. For instance, suppose the following is the register mapping for a node in the loop template:

$$(R0,\ 10.1)\ \ (R1,\ 9.0)\quad (R2,\ 7.0)\quad (R3,\ 10.0)$$

and the current instruction from that respective node in the current unrolling is $R2 = R0 - R3$ with a current register mapping of:

$$(R0,\ 9.0)\quad (R1,\ 10.0)\ \ (R2,\ 7.0)\quad (R3,\ 10.1)$$

In the original loop, this instruction uses values 10.1 for the first argument and 10.0 for the second. During copy elimination when considering the instruction $R2 = R0 - R3$, these values (10.1 and 10.0) are looked up in the current register mapping to find the registers which contain them. If the registers currently containing those values are different from the source operands, the source operands are updated as appropriate, as is the case here where the first argument must be changed to $R2$ and the second argument becomes $R1$, thus, $R2 = R3 - R1$ is the instruction for the copy eliminated code.

Once the register mappings are calculated and the loop template is made, the header of the loop is added to a *headers_list* and all of the backedges of that loop are added to the *backedges_list*. The *headers_list* is used to keep track of all loop entry points to determine, after unrolling and copy elimination, if a

backedge may be directed to any previous header that has an identical register mapping. The *backedges_list* is the list of loop iterative points along which an iteration of the loop is to be unrolled.

While there are backedges to unroll along, the algorithm iterates over the following steps: unroll the loop along that backedge; add the new header and backedges to the respective lists; *scan_and_reallocate*() (discussed further) and, finally, once copy elimination has been performed on the current unrolling, the backedges of this loop iteration are checked against all headers in the *headers_list*. Those backedges with register mappings that match the register mappings of an iteration header are directed to the respective matching header while those with no match are added to the *backedges_list*. The algorithm terminates once there are no more backedges left.

Computing Register Mappings

Figure 6.5 contains an algorithm for computing the register mappings of a loop. The algorithm first initializes all register maps to ϕ and then iterates until no changes occur to any node's register map. The algorithm scans the nodes of the loop in the forward-flow direction adding nodes to the consideration list, l. For each node, the computed "output" register map is the input register map with the additional mappings that occur as a result of executing this node. Thus, the output map reflects the values contained in registers after execution of that node.

To derive the new mappings of a node, each instruction in the node is considered. If the instruction is a copy, then a new entry is added to the output mapping annotated with the destination register of the copy and the lookup in the loop template of the source register to determine what value the source register contains (i.e., what is the tag being copied).

If the instruction is not a copy, then it defines a new value. An entry is added to the register mapping annotated with the destination register of the instruction and a tag of instruction identifier and 0 (zero signifies the birth of a value). Because copies serve to keep values live over multiple iterations, any annotation in the mapping currently being derived with the same instruction identifier will have its *age* field incremented. This signifies that the value has become "older"

```
 1:  Procedure compute_register_maps (L : loop)
 2:    /* initialize all register maps to φ */
 3:    changes = true
 4:    while (changes) {
 5:      changes = false
 6:      add loop header to l
 7:      while (not empty l) {
 8:        remove node, N, from l
 9:        Rmaps = reg_map_of(N)
10:        new_maps = copy(Rmaps)
11:        Foreach operation, op, in N {
12:          if (op is a copy)
13:            (value.age) = lookup src1_reg_of(op) in Rmaps
14:            Add (dest_reg_of(op), value.age) to new_maps
15:          else
16:            Foreach map in new_maps with same id_of(op)
17:              increment age
18:            Delete all maps from new_maps with dest_reg_of(op)
19:            Add (dest_reg_of(op), op_id_of(op).0) to new_maps
20:          endif
21:        }
22:        For all successors, S, of N {
23:          if (new_maps != reg_map_of(S))
24:            changes = true
25:            reg_map_of(S) = copy(new_maps)
26:          endif
27:          Add S to l
28:        }
29:      }
30:    }
31:  end compute_register_maps
```

Figure 6.5: An algorithm for computing register mappings.

	Iteration of Algorithm				
	Initially	1^{st}	2^{nd}	3^{rd}	no changes
Maps At Point P	ϕ		($R0$, 3.1)	($R0$, 3.1)	($R0$, 3.1)
		($R1$, 2.0)	($R1$, 2.0)	($R1$, 2.0)	($R1$, 2.0)
		($R2$, 3.0)	($R2$, 3.0)	($R2$, 3.0)	($R2$, 3.0)
		($R4$, 1.0)	($R4$, 1.0)	($R4$, 1.0)	($R4$, 1.0)
				($R6$, 2.1)	($R6$, 2.1)
Maps At Point Q	ϕ	($R0$, 3.0)	($R0$, 3.0)	($R0$, 3.0)	($R0$, 3.0)
			($R1$, 2.0)	($R1$, 2.0)	($R1$, 2.0)
		($R4$, 5.0)	($R4$, 5.0)	($R4$, 5.0)	($R4$, 5.0)
			($R6$, 3.1)	($R6$, 3.1)	($R6$, 3.1)
		($R7$, 6.0)	($R7$, 6.0)	($R7$, 6.0)	($R7$, 6.0)

Table 6.1: Register mappings for the code of Figure 6.2.

by the generation of a new value in the current instruction. Also, as the current instruction generates a value into its destination register, all maps in the output mapping with that destination register are deleted as those values become killed by the generation of a new value.

As an example of deriving the register mappings of a loop, the register mappings for the second introductory example in Table 6.1 are derived. Initially, the register mappings at points P and Q are set to empty. Using the previous mapping (that at point Q), the register mapping for point P is calculated. For each instruction in node M, the source registers are looked up in the starting mapping. For the copy operation, there is no value for $R1$ yet, so no annotation is added for that instruction. However, three instructions define new values and annotations are added for $R1$, $R2$ and $R4$. Using the new map derived for point P, the mapping for point Q is derived.

Scan-and-Reallocate

Figure 6.6 contains an algorithm for scanning a loop. This algorithm is similar to the algorithm for computing register mappings as it is necessary when re-allocating registers to instructions to keep track of the values in the registers.

Initially, the register maps are initialized to ϕ and the entry register mapping to the loop is the register mapping found at the end of the previous iteration, or that

```
1:  Procedure Scan_and_Reallocate(L : loop)
2:    /* initialize all register maps to φ */
3:    reg_map_of(header) = /* output map of backedge */
4:    add loop header to l
5:    while (not empty l) {
6:      remove node, N, from l
7:      Rmaps = reg_map_of(N)
8:      new_maps = copy(Rmaps)
9:      Foreach operation, op, in N
10:       if (op is a copy)
11:         Remove op
12:       else
13:         Update_Args(op)
14:         if (dest_reg_of(op) ∈ Rmap and live)
15:           dest_reg_of(op) = get free register
16:         Delete all maps from new_maps with dest_reg_of(op)
17:         Add (dest_reg_of(op), (op_id_of(op), 0)) to new_maps
18:       endif
19:     For all successors, S, of N {
20:       reg_map_of(S) = copy(new_maps)
21:       Add S to l
22:     }
23:   }
24: end scan_and_reallocate
```

Figure 6.6: An algorithm for removing copies and updating register usages.

found along the backedge unrolled upon. The loop is scanned in the forward-flow direction, and for each node in the loop all of its contained instructions are examined. If the instruction is a copy, it is removed from the node. If not, the arguments to the instruction are updated. This entails looking up the source arguments of the current instruction in the loop template to determine the values that the operand references. Then, those values are looked up in the current register mapping to obtain the register that currently contains the appropriate value(s). If this register and the particular source operand differ, the source operand is updated to use that correct register. Next, it might be necessary to re-allocate a register to the destination of this instruction if the register used by the instruction currently contains a value that is live beyond this point (i.e., that register was reallocated at some previous point when it was free). Finally, values killed by this instruction are removed from the current mapping and annotations are made for the new value generated by this instruction in the current mapping.

6.2.3 Determining Minimal Number of Unrollings

Because the elimination algorithm unrolls the loop to make the definitions of a value and the uses of that value explicit, thereby facilitating copy removal, a natural question that arises is: "How many iterations of the original loop must the solution loop span in order to remove all copies?" Recall from Figure 6.3 that the copies serve to preserve or to queue values from three iterations in the past. That is, the definition of a value is three iterations ahead of its last use. Thus, three iterations of unrolling are sufficient in order to eliminate the intervening copies.

In general, to remove a copy chain, the number of iterations is equal to the number of transfers of the value from definition to last use through copy instructions. This number is exactly the maximal *age* of a copy *tag*. Therefore, prior to applying the *scan_and_realloc*() procedure to the loop, the loop template can be analyzed to determine the maximal age values as the necessary number of iterations.

When multiple copy chains are present in the code, calculating the number of necessary iterations is slightly more complex. Consider two copy chains, one with maximal age two and the other, three. Every two iterations the first copy chain is eliminated, but three iterations of the loop are required for removal of the second copy chain. Thus, only after six iterations in this case, will both chains be "synchronized" so that both chains can be eliminated. In general, the necessary number of iterations becomes the least common denominator over all of the maximal age values. Again, this number can be directly calculated from the loop template prior to the application of copy elimination to determine how worthwhile application of copy elimination to the code is.

6.2.4 Heuristic Copy Elimination

Possibly the most noticeable feature of the copy elimination algorithm is that the final loop solution spans multiple iterations of the original loop in order to make value definitions and uses explicit. In some cases, it may not be desirable to unroll the loop for the necessary number of iterations. In this case, the algorithm may be parameterized with the maximal number of iterations to unroll. However, when this threshold value is reached, it is not guaranteed that the backedges for that unrolling depth will match any of the previous iteration headers.

For each backedge that remains after the threshold value has been reached, the number of mismatches (i.e., differences in the tags associated with a register in the backedge map and the respective tag in the header map) is calculated. As this is the number of differences, it is also the number of copy instructions necessary to be able to direct the backedge to the header under consideration. The backedge is then directed to the header (via a *copy node*) with which there are the fewest number of mismatches. A copy node—a node containing only copy instructions—is inserted along this path to re-arrange the values in the registers so that program correctness is preserved.

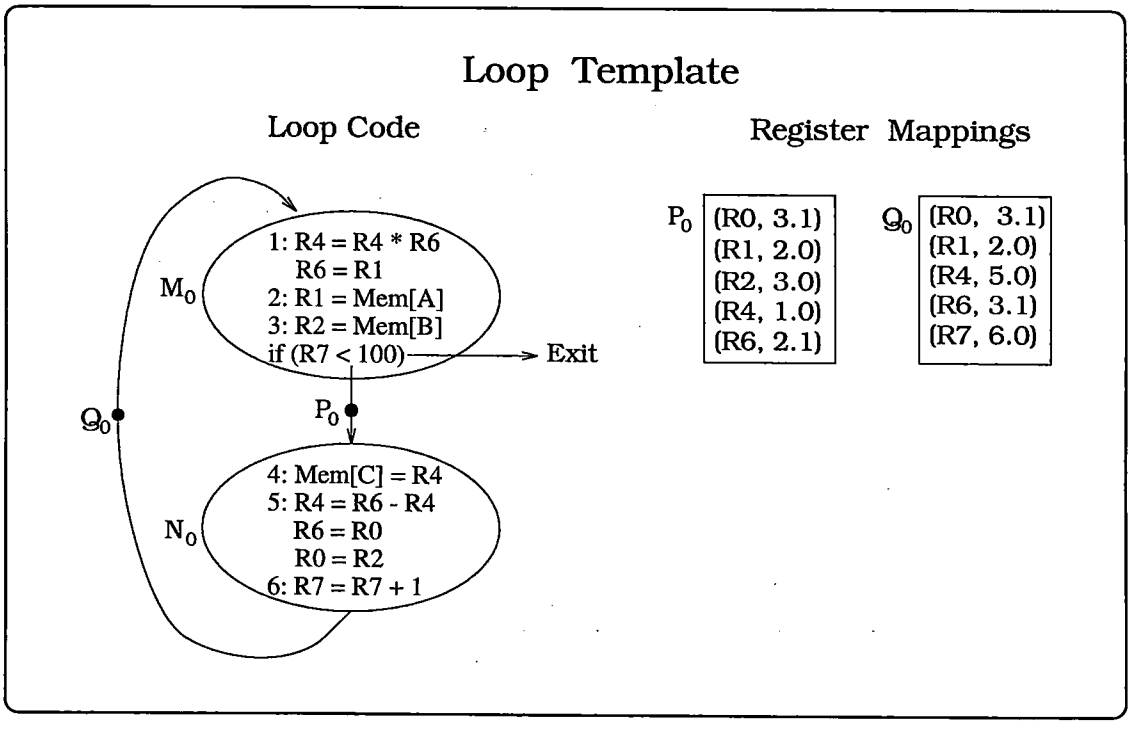


Figure 6.7: Loop template for example.

6.2.5 An Example

As an example, copy elimination is performed on the loop code of Figure 6.2. The initial register mappings for this loop were previously computed and are found in Table 6.1. At this point, a copy of the loop code and the initial register mappings are made to form the loop template of Figure 6.7.

The copy elimination algorithm applies the *scan_and_realloc*() procedure to the first iteration of the loop using the register mappings found at the loop entrance as illustrated in Figure 6.8.

As node M_0 is scanned, the first instruction to be considered is $R4 = R4 - R6$. The procedure *update_args*() looks up this instruction in the loop template (Figure 6.7) and determines that argument one is the value (5.0) and argument two is the value (3.1). These values, (5.0) and (3.1), are looked up in the current register mapping (the register mapping found at point I in Figure 6.8) and are currently found in the registers $R4$ and $R6$, respectively. Since these values are

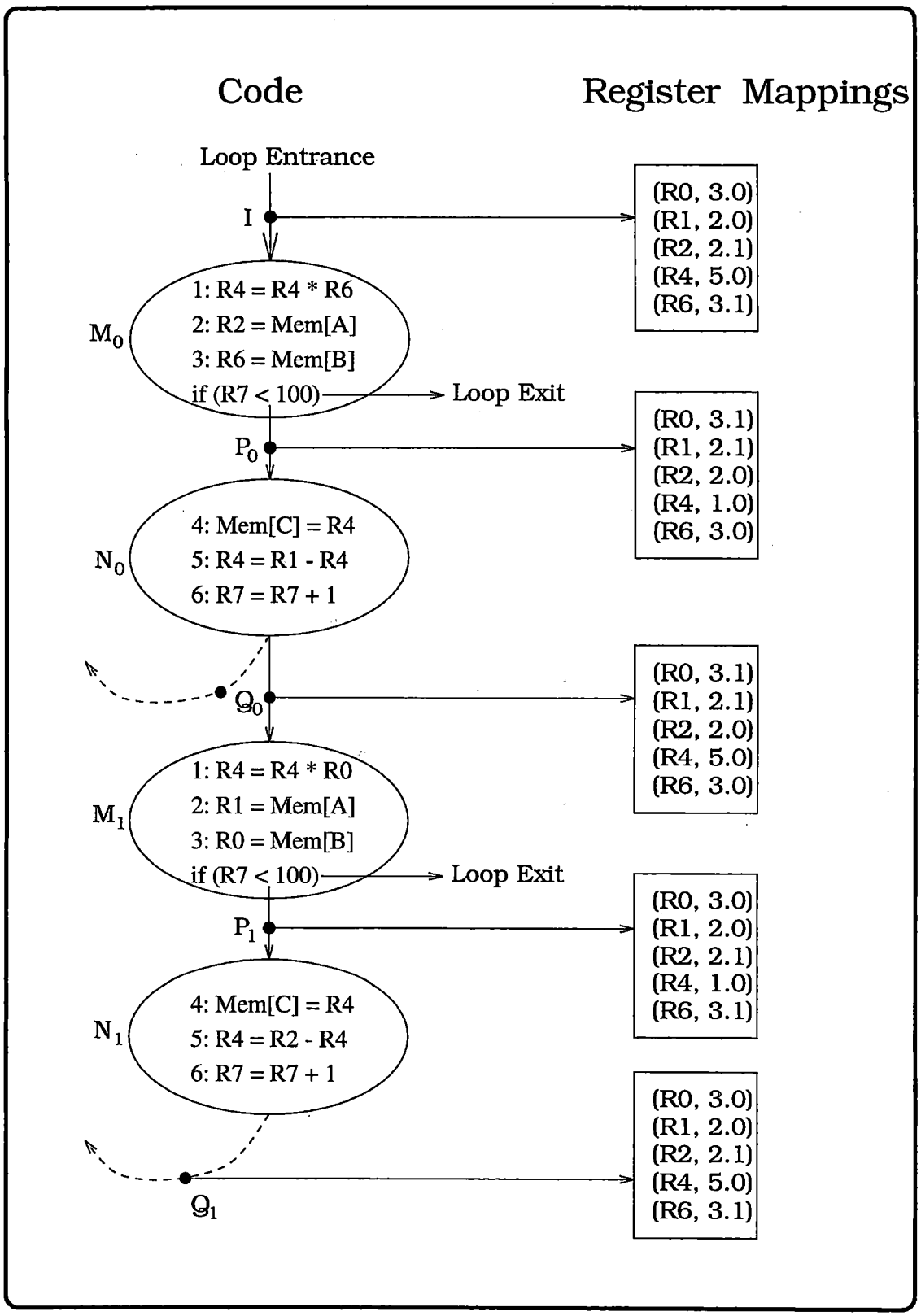


Figure 6.8: Unrolling loop code to eliminate copies.

already referenced in the appropriate registers, no changes to the instruction's operands are made. Since this is the last use of the value (5.0), contained in register $R4$, a new register is not needed for the destination register of the instruction. Lastly, an entry is made in the current register mapping of $(R4, (1.0))$.

The next instruction is the copy instruction $R6 = R1$ and is removed from the code. The next instruction, $R1 = Mem[A]$, is examined. Since there are no source operands to update, the destination register is examined and found to contain a currently live value (i.e., a value that is used beyond this node) in the register mappings at point I in Figure 6.8. Thus, a call to $get\_free\_register()$ is necessary to re-allocate a register to the destination of this instruction. In this case, the function call returns the register $R2$ as the value it contains, (2.1) becomes dead in this node[7] Thus, as depicted in node M_0 of Figure 6.8, the destination register of instruction $R1 = Mem[A]$ is changed to $R2 = Mem[A]$.

Next, the node N_0 is scanned. The argument to the instruction $Mem[C] = R4$ is checked in the loop template. From Figure 6.7, this instruction uses $R4$ which contains the value (1.0) at point P. In the current register mapping, $R4$ contains this value, so no updating takes place. The next instruction is $R4 = R6 - R4$ and uses the values (2.1) and (1.0), as source values, respectively, in the loop template. In the current register mapping, those values are contained in $R1$ and $R4$, respectively. As the value (1.0) dies, the destination register of this instruction does not require updating. The next two instructions, $R6 = R0$ and $R0 = R2$, are copies and are removed. Finally, the last instruction requires no updating, leaving the code found in node N_0 of Figure 6.8.

As the register mapping found at point Q_0 does not match that found at point I, the loop is unrolled for an iteration and copy elimination is applied to the new unrolling.

[7]When the instruction $R1 = Mem[A]$ generates a new value, it will cause all other values in the register mapping with the same identifier to become "older" (i.e., the age value increases). Thus, the value in $R1$, (2.0), will become (2.1) and the value in $R2$, (2.1), will become (2.2). As the value (2.2) is never used, the register containing that value is free for re-allocation in this node.

Again, as node M_1 is scanned, the operands for the instruction $R4 = R4 - R6$ are found in the loop template at (5.0) and (3.1), respectively. In the current register mapping (found at point Q_0), these values are in $R4$ and $R0$, respectively, so the source operands are updated to reflect this. The value (1.0) dies in this node, so there is no updating of the destination register. The instruction $R1 = Mem[A]$ requires no updating as the destination register contains a dead value and can be re-used. The instruction $R2 = Mem[B]$, however, writes to a register with a live value, so a new register is needed as the destination. Since the value (3.1) dies, the register containing this value is re-allocated to the destination of this instruction, resulting in the code found in node M_1 in Figure 6.8.

Next, the node N_1 is scanned. The value stored to memory by the instruction $Mem[C] = R4$ is in $R4$, so no operand updating is done. From the template, the source operands to the instruction $R4 = R4 - R6$ are (2.1) and (1.0) and currently contained in $R2$ and $R4$, respectively. The source operands are updated and after the copies $R6 = R0$ and $R0 = R2$ are removed from the code, node N_1 of Figure 6.8 results.

Since the values in the register mappings found at points Q_1 and I are equivalent, the backedge at point Q_1 is directed to point I resulting in the new loop code of Figure 6.8. The copy elimination algorithm then converges as there are no more backedges to unroll upon.

6.3 Effects of Copy Elimination on Register Allocation

Copy elimination is applied to the code in a post-scheduling phase. Therefore, it is important that the effects of copy elimination do not affect the code in an adverse manner. For instance, if copy elimination required extra registers that were not available, then spill code would be necessary which could either affect the performance of the code or require another scheduling phase for the spill instructions. In this section, the effects of copy elimination on register allocation

are studied.

6.3.1 Examining the Effects of Optimization

Because performing register re-allocation is an integral part of the copy elimination algorithm, one concern is that more registers are necessary for the optimized code than the number used prior to copy elimination. This is not the case as the parallelization techniques are only performed if a free register exists at the point in the code where optimization is considered. Thus, the number of registers used within a copy chain is exactly the number necessary for allocation to the unrolled code as the value lifetimes are now static and explicit and the number of copy instructions is the minimum number of registers necessary to keep simultaneous values alive.

Since the loop code is not changed by unrolling (i.e., definitions and last uses of values are not changed as no code re-organization takes place), a given register becomes free at the same respective point in the unrolled code as in the original code. Thus, there are exactly the same number of registers as values whose lifetimes overlap. When the loop is unrolled and copies eliminated, one register is used to hold a value from its definition to its last use rather than passing the value through the copy chain. Since the maximum number of overlapping lifetimes is no more than the number of registers involved in a copy chain, the number of registers used in a copy chain is sufficient to preserve values from explicit generation to last use.

Proof: A value's lifetime cannot be longer than the length of the copy chain—a value is produced and transfers through the copy chain where it eventually has its last use. Further, a single copy instruction may only preserve a value for a maximum of one iteration as the copy instruction will be executed again next iteration and receive a new value. Therefore, the value's lifetime may span no more than k iterations, where k denotes the length of the value's copy chain. If the loop is unrolled for k iterations[8] then the first definition of a value and all

[8]After the kth iteration, the register allocated to the first iteration of the unrolling becomes

subsequent uses of that value will refer to the same register in the unrolled code. This leaves $k-1$ duplications of the defining instruction that will require registers for the saving of values. However, as there were k registers involved in the original copy chain, and one has since been re-allocated, the remaining $k-1$ registers may be delegated to the remaining $k-1$ definitions. $\therefore$

A more subtle issue is involved with the *get_free_register*() function called by *scan_and_realloc*(), and can affect the number of iterations spanned by the final loop solution.

The copy elimination algorithm iterates through the unroll and re-allocation phases until all backedges match previous headers, essentially converging when a pattern in register usage emerges. As discussed earlier, there is a minimum number of iterations of unrolling to completely eliminate all copy instructions. However, if a pattern in register usage is not forthcoming within that number, it is due to the function *get_free_register*(). When a free register is requested care must be taken in choosing which register to (re-)allocate.

Consider Figure 6.9 where node N is the last node of the loop body (i.e., the edge out of this node is a backedge) and the copy of that node from iteration i has already undergone register re-allocation. As node N_{i+1} (the same node, one iteration later) is scanned, the instructions $R3 = Mem[Bi]$ and $R4 = R7 + R8$ are reached and found to write into registers which contain live values, so new registers must be allocated to the destinations of those instructions[9]. At this point, both $R1$ and $R2$ are free. If $R1$ is allocated to the first instruction (instruction #7) and $R2$ is allocated to the second instruction (instruction #10), then it will not be possible for the backedge out of node N_{i+1} to be directed to the header node M_{i+1} at point P as the register mappings will not match.

For this to converge, $R1$ must be allocated to instruction #7 and $R2$ must be allocated to instruction #10. In general, when there are n free registers and n values requiring new destination registers, the number of possible mappings becomes the number of permutations of the values in those registers or $n!$.

dead, and hence free for re-allocation.

[9]In practice, the two instructions are processed separately.

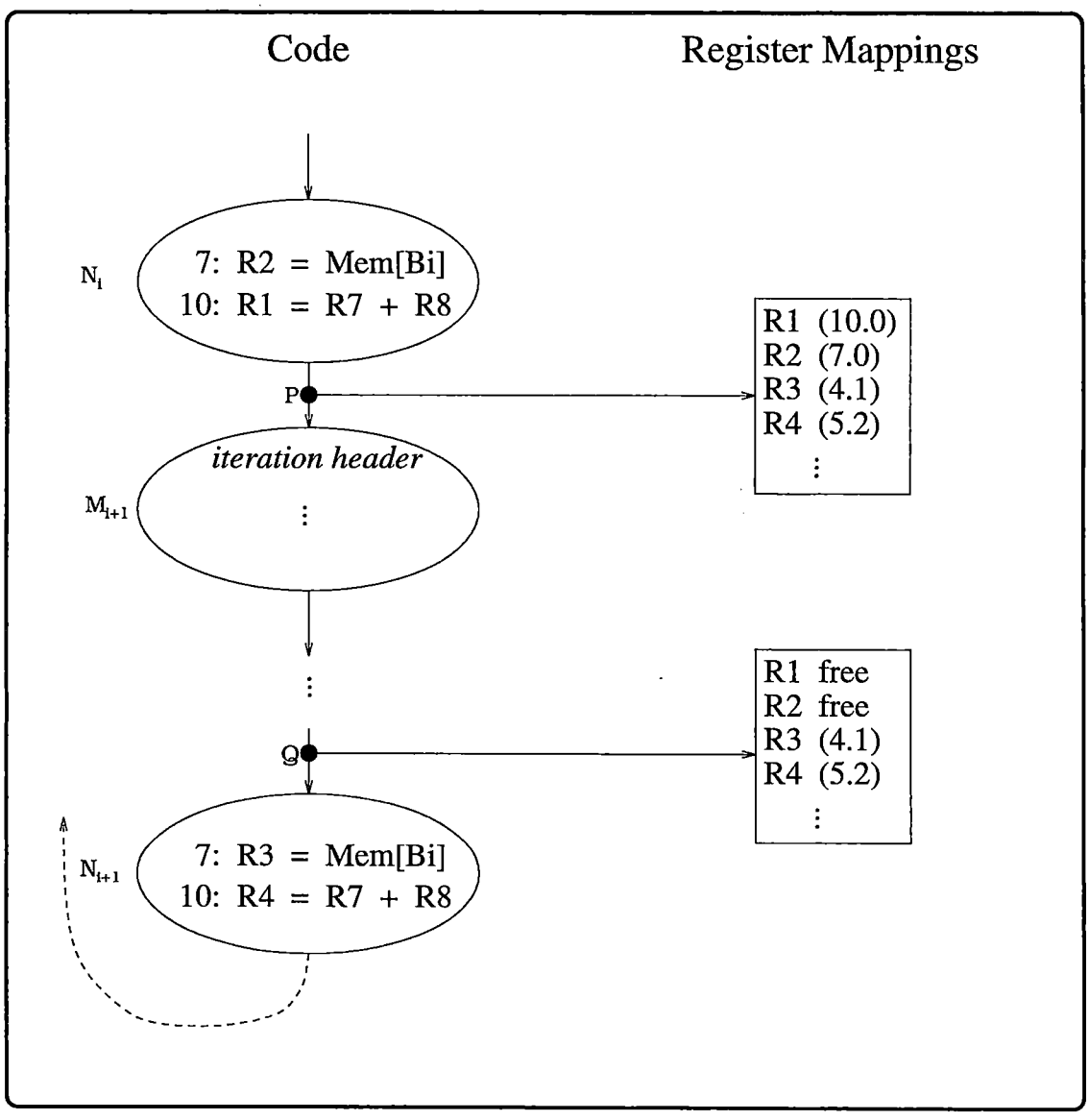


Figure 6.9: Example code and register mappings during copy elimination.

Although this number can be quite large in principle, in practice, this is rarely the case as registers are reallocated deterministically (i.e., not randomly) and a simple heuristic, presented shortly, eases this considerably.

6.3.2 An Enhanced Copy Elimination Algorithm

In order to remedy the (potential) re-allocation problem previously discussed an improved version of the *get_free_register*() function is presented. A simple, but effective heuristic that has been implemented is to keep track of the last register that has been allocated to an instruction. This entails building a table which pairs the unique instruction identifier and last register allocated and passing the unique instruction identifier to the *get_free_register*() function, so line #15 of *scan_and_realloc*() procedure (Figure 6.6) becomes:

$$dest\_reg\_of(op) = get\_free\_register(id\_of(instr))$$

When a request for a new register is made, the instruction identifier is looked up in the table and if the register entry found there is currently free, it is returned as the "prefered" register. If that particular register is not free, then a register is selected from the free set and the table is updated.

This heuristic is designed to aid the convergence process. However, in cases where the final solution has the value for some instruction oscillating between two registers or between some set of registers, this heuristic's merits lessen. A simple extension, however, can easily remedy this: rather than tracking the last register allocated, all registers assigned to that instruction are tracked. In this scheme, each time that a register is allocated, it is added to the set of allocated registers for the given instruction. If it has been previously allocated, then a counter is incremented which indicates the number of times that the particular register has been allocated to that instruction. When determining a register to allocated, preferential treatment is given to the register with highest frequency. If that register is unavailable, the registers in the set are tried in order of frequency.

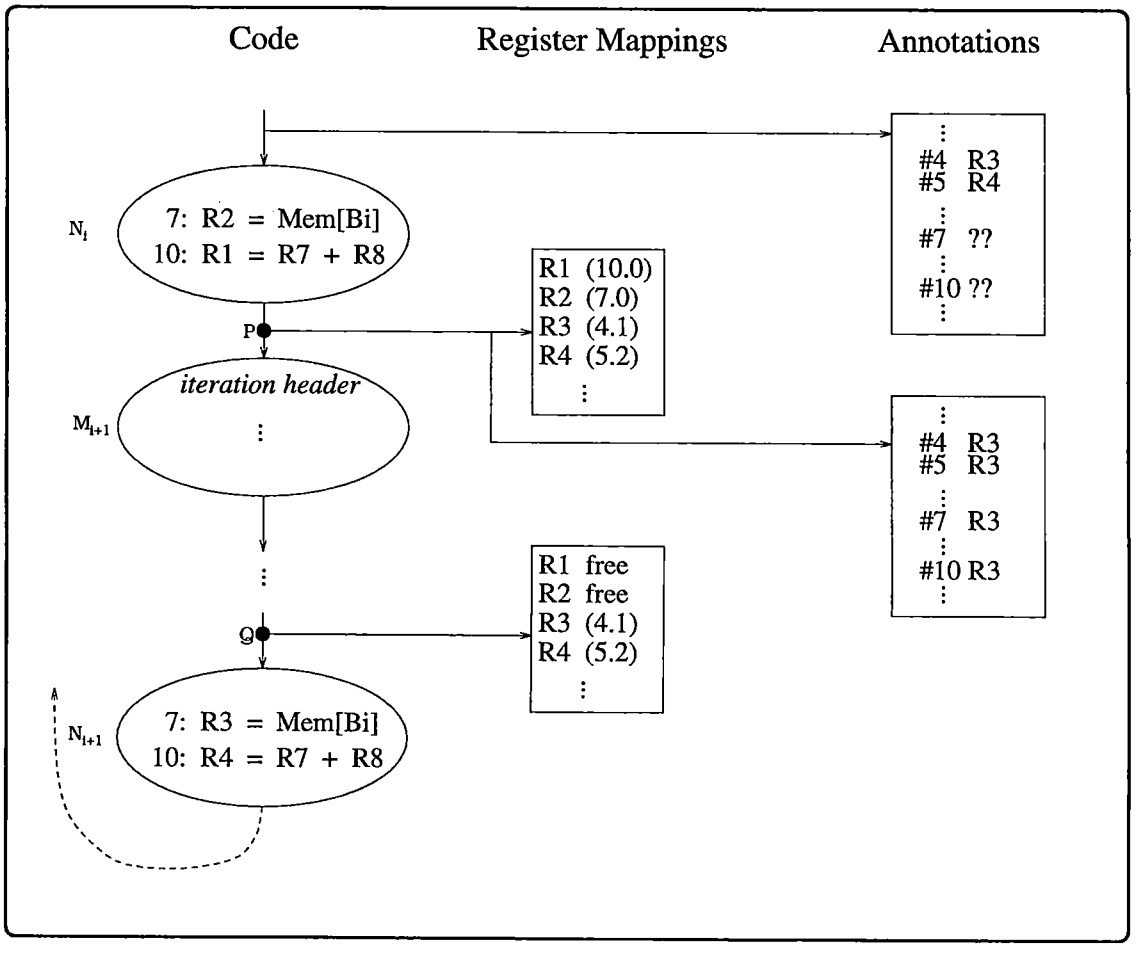


Figure 6.10: Example code and register mappings with annotations.

Thus, this extension aids convergence in the cases where a value generated by some instruction is allocated to multiple registers over multiple iterations.

Figure 6.10 shows the previous example annotated with the last register allocated information. When $get\_free\_reg()$ is called at node N_{i+1}, the last register allocated function consults the table to find which register was last allocated to instruction #7. Register $R2$ was the last register allocated and is also found to be free so it is allocated to instruction #7 in node N_{i+1}. Note that if $R1$ and $R2$ were occupied and $R3$ and $R4$ were free (vice versa of the depicted allocation in the figure), when the last register allocated table was consulted to find $R2$ as the last register, but not currently free, then $R3$ would be allocated and the table entry updated.

Chapter 7

Allocating Registers to Loops

The task of register allocation is to assign program variables or intermediate, "virtual" registers to the real registers of a target architecture. When more variables than real registers are present, some variables will inevitably be placed in main memory and require memory accesses when they are needed. Due to the latency incurred when accessing these "spilled" variables, the quality of the allocation generated by a register allocation algorithm directly affects the running time of the resulting code.

Due to the importance of register allocation, *optimal* solutions have been extensively studied [24, 45, 46, 48] and advocated as potentially practical for time-critical innermost loops. Because register allocation is an NP-Hard problem in the general case [2], attempts at optimal solutions have either simplified the problem to index registers [45, 48] and/or have used a brute-force exponential method with heuristics to prune the search space [24, 25, 46].

These approaches yield an effective (if expensive) strategy for straight-line code, but the only place such extreme techniques may be applied in practice is to increase performance in innermost loop bodies. Thus, the goal is to minimize the number of loads and stores, due to spill code, that will be repeatedly executed within a loop. In order to extend the methods in [45, 46] to handle loops, a fundamental difficulty that these previous techniques did not address

satisfactorily—that of matching the register usages at the beginning and end of a loop iteration—must be overcome. That is, for loop code to be correct, the mapping of variables to registers found at the beginning of an iteration as well as that found at the end of that iteration must be equivalent so that it is correct to iterate over that loop code. While a previous paper [48] provided a technique for handling simple loops, optimality (and significant performance) was lost in the process.

Previously, it was not known whether optimal register allocation for a loop could be accomplished, regardless of efficiency of the algorithm. The difficulty stems from the need to match register usages at the top and bottom of a loop body. In order to ensure optimality for a loop, additional spills might be required at loop bottom to match usages at top. To optimally minimize these spills, loop unwinding with different register allocations in each unwound iteration may be necessary.

This chapter demonstrates the viability of extending the techniques in [45, 46, 48] to deal with loops by incorporating loop unrolling techniques into the algorithm. Thus, a distinguishing characteristic of this approach is that the allocation derived may span multiple iterations of the original loop. Heuristic modifications to this algorithm are considered that, in practice, seem to perform as well as their exponential counterpart (as demonstrated in the results of experimentation in Chapter 8).

7.1 Optimal Allocation in Basic Blocks

Many researchers have felt that for particularly critical code segments, such as the innermost loops of time-sensitive applications, an optimal allocation is necessary. Horwitz *et al.* [45] present a method for obtaining an optimal register allocation to *index* registers which minimizes the number of loads and stores due to added spill code. Further work either improves upon the efficiency of the Horwitz algorithm [60] or extends the basic algorithm to deal with simple loops [48], but in doing so loses optimality and degrades performance. More recent

research has extended the basic idea in Horwitz's algorithm to include register allocation for *general purpose* registers [46].

An Optimal Algorithm for Basic Blocks:

As the loop algorithm is based, in part, on the ideas in [45, 46, 48], a variant of a basic block optimal algorithm is presented here. This algorithm explores every possible register allocation at each virtual register or variable access in a basic block, and produces an allocation which contains the minimal memory traffic. That is, the allocation produced is optimal with respect to the *cost*[1] of memory loads and stores due to spill code. This algorithm, referred to as BB-Opt, is found in Figure 7.1.

The algorithm BB-Opt is given a *register access pattern* which corresponds to the accessing of the virtual registers or variables ie the input code segment. For instance, for the sequential code segment: **VR1 = VR2 + VR3; VR4 = VR5 - VR6**, the register access pattern is **VR2, VR3, VR1*, VR5, VR6, VR4*** (reads before writes) where writes are distinguished by a '*'. BB-Opt also takes as input the *mapping* (or *configuration*) of virtual registers or variables to real registers that immediately precedes the code segment. For instance, if two real registers exist and **VR1** is mapped to **R1** and **VR5*** is mapped to **R2**, the configuration would be: {**VR1, VR5***}. This leads to the following definitions:

Definition 7.1.1 *A* **register access pattern** *is a sequence of virtual register reads and writes found in some code segment. A virtual register read is denoted by the virtual register name and a virtual register write is denoted by the virtual register name concatenated with '*'.*

Definition 7.1.2 *A* **register configuration** *or* **register mapping** *is a binding of virtual to real registers and represents the contents of the real registers at some point in computation.*

BB-Opt then builds an allocation tree where the root is the given initial configuration. Each successive level in the tree is derived by taking the next virtual register access from the register access pattern and examining each node

[1]Cost can be based on any appropriate measure: load/store count, latency, etc.

```
    Function BB-OPT (REGS : Initial register configuration;
 1:                  VA : Variable access pattern)
 2: Begin
 3:    Set curr_states set to REGS
 4:    Foreach varible access V in VA do
 5:       Foreach config. N in curr_state set do
 6:          If V ∈ N then
 7:             Copy N to new_states set
 8:          Otherwise
 9:             Forall registers R do
10:                N' = copy_state(N)
11:                Replace variable, V', currently in R with V
12:                Cost(N') = Load-Cost(V) + Store-Cost(V') + Cost(N)
13:                Add N' to children of N
14:                Add N' to new_states set
15:             Enddo
16:          Endif
17:       Enddo
18:       Set curr_states set to new_states set
19:    Enddo
20:    Return new_states set
21: End BB-OPT
22:
```

Figure 7.1: An optimal register assignment algorithm for basic blocks.

in the previous level to determine if that configuration contains the virtual register being accessed. If the virtual register is contained in the configuration, that node is duplicated at the next level and a zero-cost edge connects the two. If the virtual register is not contained in a node, a virtual register access miss occurs and spill code becomes necessary. For any configuration causing an access miss, each virtual register in that configuration is replaced in turn by an access to the faulting virtual register. An edge joining the two represents the cost, in spill code, of going from the first mapping to the second. This cost is composed of the cost of (possibly) storing the replaced register if it is dirty[2] and/or the cost of

[2]Every virtual register is assumed to have a unique memory location. Dirty refers to the case where the value in a register is inconsistent with the value stored in the respective memory

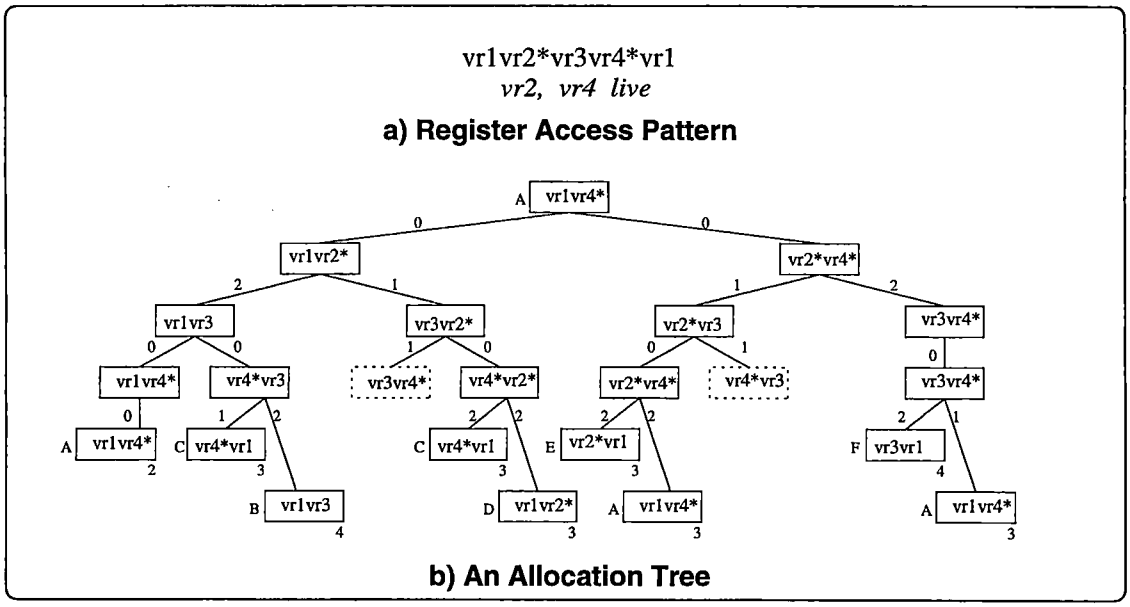


Figure 7.2: Building an assignment tree.

(possibly) loading the faulting register. Thus, if there are r real registers, a faulting configuration in the current level will generate r configurations in the next level, resulting in an optimal, but exponential method. Heuristics can (and have been) used to prune the search space [45, 46, 48].

Once the last virtual register access is considered, all the leaves of the allocation tree are examined for the lowest cost node. Tracing the path from the root to this lowest cost node will yield an allocation of real registers to virtual registers that results in the minimal cost in terms of memory loads and stores due to spill code because it has exhaustively generated every possible allocation.

As an example, consider the register access pattern in Figure 7.2(a). In this example there are two registers which have been initially assigned **vr1** to **R1** and **vr4*** to **R2**. The assignment tree in Figure 7.2(b) is constructed by applying BB-Opt to the register access pattern with the initial mapping.

The first access to **vr1** is to a virtual register currently in the configuration, therefore, no spill code is necessary. However, the next access to **vr2*** causes an

location.

access miss and spill code might become necessary. Two configurations are generated at the next level, corresponding to assigning **vr2*** to R1 or to R2. The left child assigns **vr2*** to R2, displacing **vr4***. Since **vr4*** does not need to be stored (its value is dead), there is no spill code generated as **vr2*** is a write. The other possibility, assigning **vr2*** to R2, is represented by the right child and displaces the variable **vr1**. Since **vr1** is consistent with the value in memory, it does not need to be stored.

This process, examining the next variable access(-es) and checking whether they are contained within the current configurations, continues for the remainder of the variable access stream and the full tree in Figure 7.2(b) is generated. In Figure 7.2(b), there are some nodes which have dashed outlines. These nodes can be pruned from the tree as there are identical nodes at the same level (which will generate identical sub-trees). Within a group of identical nodes, only the one with lowest cost need be kept, breaking ties arbitrarily.

7.2 Extending the Basic Block Algorithm to Loops

By applying the BB-Opt algorithm to the body of a loop, an optimal assignment *for a single execution* of that code is obtained. Since this code is contained within a looping construct, it is necessary for the register mappings at the beginning and end of the code segment to match in order to correctly iterate over that segment. In general, the assignment produced by BB-Opt will not satisfy this criteria (i.e., the lowest cost configuration at a leaf of the assignment tree does not necessarily match the root). Thus, this basic algorithm is not adequate to optimally assign registers to loop code.

To remedy this, simply adding register-to-register move instructions and/or spill code (loads and/or stores) to enforce a match could be tried. However, since the cost of this additional spill code can vary greatly from each conceivable mapping to another, and would vary further by unrolling the loop some number of times,

BB-Opt's results, which ignore this effect, cannot be optimal. Another sub-optimal approach is to 'force' a match between loop top and bottom, i.e., to choose from the exponential tree derived by BB-Opt the least cost leaf node which is identical to the initial configuration (leaf configurations which match the root configuration are not necessarily guaranteed to be those with lowest cost).

7.2.1 Terminology

The following terms are used in conjunction with the discussion of the Loop-Opt algorithm.

Definition 7.2.1 *An* **allocation tree** *is the tree produced by the application of BB-Opt to some register configuration and register access pattern. The root of the tree is the starting (given) register configuration and the leaves of the tree are register configurations which represent the virtual-to-real register bindings at the completion of the code segment. The allocation tree may contain many iterations of the original loop.*

Definition 7.2.2 *An* **exit configuration** *is a particular register configuration that is a leaf in some allocation tree.*

Definition 7.2.3 *An* **allocation path** *is a path through some allocation tree from the root to some leaf. This path defines a (unique) virtual-to-real register binding for the code segment.*

Definition 7.2.4 *An* **iteration ancestor** *of register configuration X is a register configuration Y which lies on the allocation path from the initial register configuration to X and the parent of Y and Y belong to iterations i and $i+1$ for some i, respectively.*

7.2.2 An Algorithm for Loop Register Allocation

It is not immediately obvious how many iterations suffice to produce an assignment which results in the minimal amount of spill code. In fact, this is why this problem has been an open issue. If the process of unwinding a loop and applying BB-Opt is continued, the cost may be decreased. By iteratively unrolling one loop iteration and applying BB-Opt to the resulting code, a new

loop body is produced which potentially spans several iterations of the original loop with the characteristics that: a) the cost of spills per iteration in the loop body is minimal; and b) the entry and exit configurations of the new loop match.

The algorithm for assigning registers to loop code, referred to as Loop-Opt, is found in Figure 7.3. The general structure of this algorithm is to iteratively unroll the loop one iteration and then to apply BB-Opt to the new iteration once for each possible previous iteration exit mapping. Then the algorithm analyzes each resulting exit mappings of that new iteration to determine if matches between those nodes and iteration ancestors (i.e., a node in the assignment tree that lies on the path from the root to this node and also lies on an iteration boundary). If so, a legal register assignment to the unrolled loop has been found. If not, then that exit mapping becomes one of the mappings which will be used as an initial configuration to the next iteration.

Each time that a match is found, this algorithm computes the average cost per iteration for that assignment (since the assignment may span multiple iterations). If the loop were fully unrolled, the assignment with the lowest average cost per iteration would be the optimal assignment for the loop. Since full unrolling of the loop is not necessarily practical, this algorithm is parameterized with K, the number of unrollings of the loop body to perform. The lowest cost mapping found with this "cut-off" scheme is a local minimum, but is "global" over the number of iterations unrolled so far (K). Note that this algorithm must always get an average cost less than or equal to what BB-Opt would get because the algorithm deals strictly with the costs calculated by BB-Opt and adds nothing more—beyond unrolling.

7.2.3 Heuristic Pruning

The heuristic modifications to the loop algorithm consist of two simplistic pruning strategies. The first is **width** restriction where only the m best configurations are kept for future expansion once all mappings at a particular level are generated. That is, for each node in the current level, when an access

```
1:  Function Loop-Opt (REGS : Initial mapping;
2:                         VA : Variable access pattern;
3:                         K : number of iterations)
4:  begin
5:    set MIN to an empty configuration with ∞ average cost
6:    set i to 0
7:    set curr_states set to REGS
8:    loop
9:      set save_state_set to null
10:     foreach state S in the curr_state set do
11:       new state set = BB-Opt(S, VA)
12:       foreach state N in new state set do
13:         if N matches an ancestor A then
14:           Direct N to A
15:           Delete N from new_states set
16:           AveCost(N) = (Cost(N)-Cost(A)) / (Iter(N)-Iter(A))
17:           if AveCost(MIN) > AveCost(N) then
18:             MIN = N
19:           endif
20:         endif
21:       enddo
22:       set save_state_set to save_set_state ∪ new_set_state
23:     enddo
24:     set i to i + 1
25:     set current register state set to save_state_set
26:   until i = K
27:   Return MIN
28: end Loop-Opt
```

Figure 7.3: A loop register assignment algorithm.

miss occurs, all possibilities for spills are considered. Then, of those newly generated nodes, the m lowest cost nodes are retained for consideration. The second is **depth** restriction and refers to the number of levels in the allocation tree that are generated before pruning (i.e., width restriction is applied).

7.3 Convergence and Optimality of the Loop Algorithm

Previously it was not known whether optimal register assignment for a loop could be accomplished, regardless of the efficiency of the algorithm. The difficulty was due to the fact that in order to ensure optimality for the overall loop, matching of registers at the top and bottom of the loop body may require additional spills. To optimally minimize these spills, loop unwinding with different register assignments in each unwound iteration may be needed. Furthermore, it was not known whether any finite unwinding can be guaranteed to converge and result in an optimal assignment.

To answer these questions, the notion of a *configuration graph* is introduced. A node in the configuration graph corresponds to a specific mapping of virtual-to-real registers found at an iteration boundary and a directed edge in the configuration graph corresponds to the cost in spill code of using the source node as the initial mapping to an iteration, applying the loop algorithm and having the sink node as one of the resultant nodes. Thus, the edge represents the cost of spill code with the source node as the initial register assignment to and the sink node resulting from an iteration of the loop.

Figure 7.4 illustrates the method of building a configuration graph. This example uses the allocation tree from Figure 7.2 and has the leaf nodes labelled. A partial configuration graph, shown in (b), can be constructed from the assignment tree in (a). Traversing a path from the root configuration, which has been labelled A, to each leaf configuration gives a directed edge in the configuration graph from A to that respective node with a weight equal to the cost of the path. For instance,

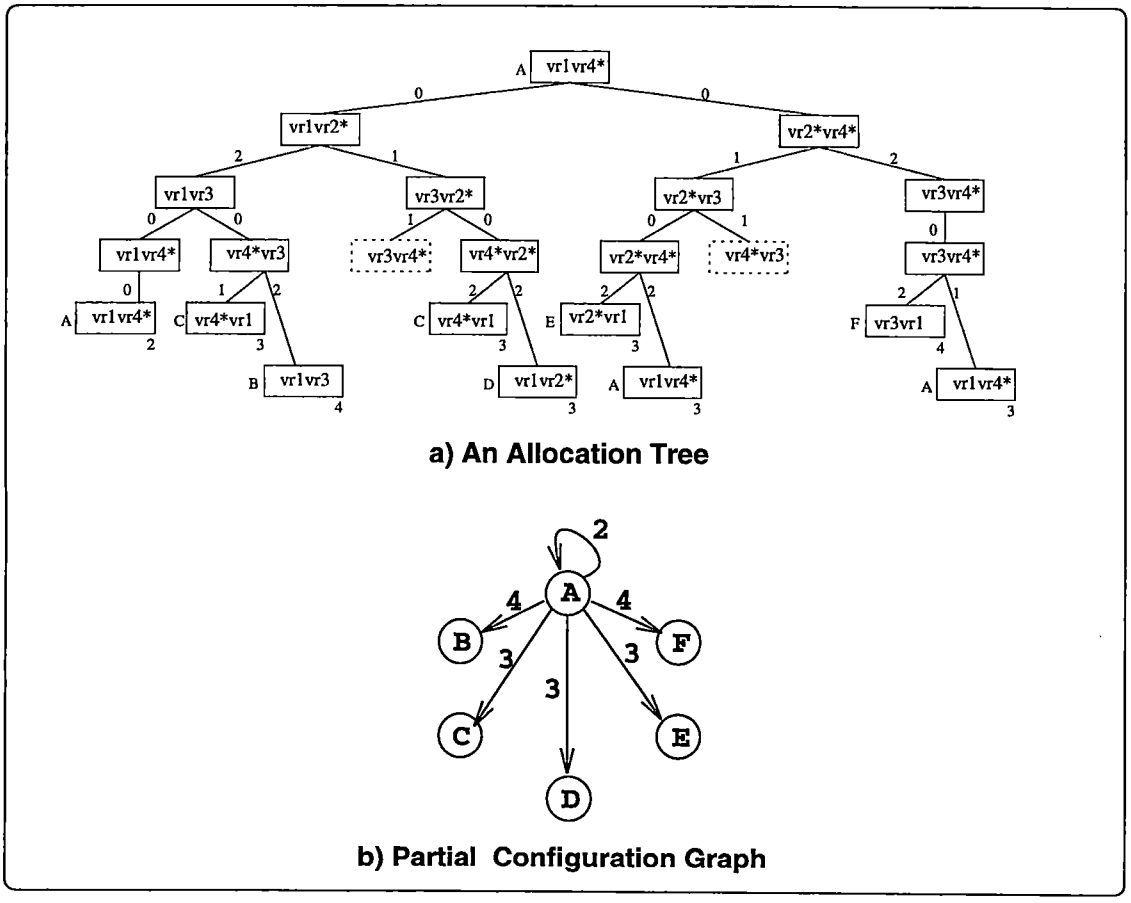


Figure 7.4: Building a configuration graph from the assignment trees.

the path from the root to the first leaf node on the left, labelled A, has a cost of two. Thus, an edge in the configuration graph from A to A is added with that edge having weight two. Similarly, other edges are added to the configuration graph by traversing the various paths. The partial configuration graph in (b) results. To construct the complete graph requires that the assignment trees for each possible exit configuration be built.

7.3.1 Convergence

In order to guarantee that this algorithm converges, it must shown that by unrolling, new exit configurations (i.e., mappings of variables to registers) that previously did not exist are not generated. Because this algorithm exhaustively replaces registers each time a variable access miss occurs, all conceivable mappings are generated. Stated another way, when an unrolling of the loop body and assignment to that iteration is performed, the costs associated with going from the initial to the derived exit mappings become known. Thus, the edges in the configuration graph which connect the initial configuration with all of the possible exit configurations are generated. If the assignment algorithm is again applied to each of these nodes (e.g. unroll the loop body for another iteration), directed edges from each of those exit configurations to one another are obtained. Convergence of this algorithm, therefore, is equivalent to finding a cycle in the configuration graph. Thus, this algorithm converges because the number of variables and the number of registers is finite and, therefore, the number of permutations of the variables in the registers is finite, although exponential.

7.3.2 Optimality

The question of optimality can also be addressed with the notion of the configuration graph. An optimal allocation is one in which the memory traffic is minimized. When the loop body is unrolled, an optimal allocation becomes the allocation which has minimal memory traffic or spill cost *over the iterations* that are contained within the unrolled loop. Thus, in the optimal allocation, the ratio

of the spill cost for the new unrolled loop body to the number of iterations it contains, is minimized. In the configuration graph this corresponds to the ratio of the total cost of a cycle to the number of nodes in that cycle.

Definition 7.3.1 *The* **minimal average spill cost** *for a loop is*

$$\min \ \langle 1 \leq i \leq n \rangle \quad \frac{\sum_{j=1}^{i} Cost_j}{i}$$

where $Cost_j$ is the cost associated with edge j in some cycle of length i.

That is, the optimal allocation is found by examining the average cost of all possible cycles in the configuration graph and taking the minimum.

Note that this does not simply correspond to the minimal cycle of length one in the graph. A cycle of length one would imply that some allocation to the loop body is minimal *and* its initial configuration naturally (i.e. without spills or moves) matches its exit configuration. In the worst-case it is possible that the optimal cycle must make a complete tour of the graph.

While the above can become quite expensive, Loop-Opt can be invoked with some K significantly lower than the (expected) loop bound. In this case, the allocation returned by Loop-Opt will be minimal for the given initial configuration and number of unrollings (K), since the algorithm will explore all possible allocations with the given initial configuration. In the graph this is equivalent to finding the minimal average cycle with a bound (K) on the length of the cycle, and the start node equal to the node in the graph which corresponds to the initial configuration. If a lower average cost cycle exists, it must be farther away from the given initial configuration than K, or it may be reachable within length K from some other configuration. Hence, the allocation returned must be minimal for the given parameters. In addition, all the pruning optimizations proposed in [45, 48] would still apply.

7.4 Extending the Loop Algorithm to Distributed Memories

The algorithm presented earlier for register assignment in loops has the underlying assumption that access to all available registers is equivalent, as is found, for instance, in general-purpose processors and some embedded processors. That is, all registers are consolidated into one register file and any variable mapped to a register is uniformly available to any operation using that variable. However, in the case of an architecture where the available registers have been partitioned into disjoint register files or some of the available registers have specialized purposes, this assumption must be modified to generate feasible register assignments. Previously there was the assumption that enough ports on the register file exist to support the reading and writing of all variables accessed in a particular step. However, it is possible that restrictions are present on the number of registers that are concurrently accessible (i.e., the number of read/write ports on a register file constrains the number of reads/writes to that register file).

7.4.1 Adding Register Classes to the Model

To extend the algorithms, the notion of *register classes* is introduced. Register classes have been used in compilers [3, 78] and in microcode synthesis [32, 55] to denote functional equivalences between registers. However, combining all registers having the same (potential) usages into one class is not precise enough for register assignment to the target architecture class. To see this, consider a simple case where two register files are composed of "general-purpose" registers, each register file connected to a different ALU. Clearly, any operation scheduled on either ALU must have its operands present in the respective register file. However, if the collective registers are grouped into one register class (called "general-purpose"), it is possible that the necessary operands have been assigned in such a way as to honor the register classes, but be invalid for execution, thus,

```
 1: Procedure Derive-Register-Classes()
 2: Begin
 3:     Foreach FU, f in the architecture do
 4:         Foreach Input, i of f do
 5:             Set RegClass(f,i) to ∪ all regs connected to i
 6:         Enddo
 7:         Foreach Output, o of f do
 8:             Set RegClass(f,o) to ∪ all regs that f may write
 9:         Enddo
10:         Foreach operation, op, that f can execute do
11:             Set RegClass(f,op_input) to ∪ all regs that op may read
12:             Set RegClass(f,op_output) to ∪ all regs that op may write
13:         Enddo
14:     Enddo
15: End Derive-Register-Classes
```

Figure 7.5: An algorithm to derive register classes.

making that register assignment invalid. The main cause of this problem is not due to registers being grouped by their equivalency, but, rather, how the equivalency is established.

In this approach, two types of register classes are defined: *connectivity register classes* and *operation register classes.* The connectivity register class (conn_RC) defines the equivalency between registers as a function of the architecture's connectivity, while the operation register class (oper_RC) defines the equivalency between registers as a function of an operation's semantics. The motivation for deriving both of these classes is that the connectivity of the architecture defines which registers may be read from or written into by some functional unit, while the semantics of a particular operation executing on a particular functional unit may preclude the use some of the connected registers (a load operation, for instance, may require that the memory address reside in a specific register, while the functional unit that executes that load operation may be connected to many registers which do not serve the same purpose).

Figure 7.5 contains an algorithm to derive the register classes for a given

architecture. A connectivity register class is derived for each of the inputs and outputs of each functional unit in the architecture based upon which registers may be accessed by that input or output. Operation register classes are derived by examining which operations a functional unit can execute and selecting all of the readable (and writable) registers imposed by an operation's semantics. In a large number of cases, the conn_RC and oper_RC will be equivalent.

7.4.2 Extension to Special-Purpose Registers

The algorithm BB-Opt exhaustively generates variable mappings by placing a variable in each register either when a read miss occurs (requiring a load of the variable) or when a variable is written. When access to all registers is uniform, this strategy is correct. However, when some registers have specialized usages, this strategy generates some mappings which are invalid as variables have been assigned to registers which cannot perform the required specialized function. Thus, it is necessary to restrict the placement of variables into registers so that variables only reside in registers which can perform the necessary functionality.

With the notion of register classes, the BB-Opt algorithm can be extended to handle registers which have specialized usages. When a variable causes an access miss, only those registers which perform the necessary functionality are considered. These are found by intersecting the operation register class for the accessing operation and the connectivity register class for the functional unit that is executing that operation. Recall that register assignment is performed on a scheduled dataflow graph. Thus, when performing register assignment the operations (and their types) which access variables, as well as the functional units that those operations execute on, are known—retrieving this information is a simple matter.

Figure 7.6 contains an extended version of the BB-Opt algorithm for register assignment with specialized register usages. The function *op_of* returns the operation which currently accesses the variable V. From this, the type of operation and the functional unit that executes the operation are found via calls

```
1: Function OPT-Assign (REGS : Initial register configuration;
2:                      VA : Variable access pattern)
3: Begin
4:     Set curr_states set to REGS
5:     Foreach variable access V in VA do
6:         Set op_type to OperationType(op_of(V))
7:         Set fu to FunctionalUnit(op_of(V))
8:         Set RC_intersect to RegisterClass(op_type) ∩ RegisterClass(fu)
9:         Foreach config. N in curr_state set do
10:            If V ∈ RC_intersect then
11:                Copy N to new_states set
12:            Otherwise
13:                Forall registers R ∈ RC_intersect do
14:                    N' = copy_state(N)
15:                    Replace variable, V', currently in R with V
16:                    Cost(N') = Load-Cost(V) + Store-Cost(V') + Cost(N)
17:                    Add N' to children of N
18:                    Add N' to new_states set
19:                Enddo
20:            Endif
21:        Enddo
22:        Set curr_states set to new_states set
23:    Enddo
24:    Return new_states set
25: End OPT-Assign
```

Figure 7.6: Extending BB-Opt to special-purpose registers.

to functions *OperationType* and *FunctionalUnit*, respectively. Then, the appropriate operation register class and connectivity register class are found and intersected. RC_intersect, the intersection of these classes, defines the feasible registers in which a variable V may reside. If the variable is in one of those registers, then no spill code is necessary. If not, then all of the registers contained in RC_intersect are candidates for replacement and spill code is generated.

7.4.3 Extension to Multiple Register Files

To extend the algorithms to assign registers to multiple register files requires that the notion of a node in the assignment tree be altered. In assigning registers to an architecture with a consolidated register file, the semantics of a node are that all registers are uniformly available. For instance, if there are eight registers filled with the variables **a–h**, a mapping of variables to registers is represented as **{a,b,c,d,e,f,g,h}**, signifying that **a** is mapped to register one, **b** is mapped to register two, etc.

To model multiple register files, the information contained in a node is augmented to reflect the grouping of registers into a register file. Each node in the assignment tree is then composed of a number of *register sets* equal to the number of register files.

Assigning Variables

Figure 7.7 contains an extended version of the BB-Opt algorithm which assigns registers to multiple register files. The main modification required when multiple register files exist is, only the registers in the respective register file are examined to determine if a variable is resident. If a variable is not contained within the necessary register file, rather than loading it from memory, a check is first made to see if the variable is contained within one of the other register files. If so, then a move operation is used to transfer the value into the necessary register file if the necessary connections exist as this transfer is likely to have a lower latency than a load from (slower) memory. Otherwise, the variable is loaded from memory. Once

```
 1: Function OPT-Assign (REGS : Initial register configuration;
 2:                      VA : Variable access pattern)
 3: Begin
 4:     Set curr_states set to REGS
 5:     Foreach variable access V in VA do
 6:         Foreach config. N in curr_state set do
 7:             If /* V is already in proper RF */ then
 8:                 /* Copy the current state to the next level */
 9:             Else Forall register files, RF, do
10:                 If /* V is contained in another RF */ then
11:                     /* Generate a move of V to this RF */
12:                     /* Check access restrictions */
13:                 Endif
14:             Otherwise /* V must be loaded from memory */
15:                 /* Generate all possible spills of */
16:                 /* variables contained in this RF */
17:                 /* Check access restrictions */
18:             Endif
19:         Enddo
20:         Set curr_states set to new_states set
21:     Enddo
22:     Return new_states set
23: End OPT-Assign
```

Figure 7.7: Extending BB-Opt to multiple register files.

a spill is considered[3], any access restrictions present on the register files, such as the number of registers which can be simultaneously accessed, are considered. If the restrictions are satisfied, then the assignment is valid and is maintained for future assignment, otherwise the mapping represents an assignment which causes an access conflict to exist and the node is removed from future consideration.

A Note on Optimality

With the addition of register classes and extension to special purpose registers, the algorithm derives optimal (i.e., spill minimizing) results. However, with multiple register files and the presented version of this algorithm, it may be

[3]Different cost can be assigned for spilling to memory, fetching from memory and fetching from another register file.

possible that sub-optimal results are obtained. Previously, when a variable was assigned to a register, all registers were viewed as candidates for replacement. Extending this to cases where some registers have special purposes merely removes some number of registers as candidates (and, thus, serves to restrict the growth of the assignment tree). However, in the case of multiple register files, the algorithm may no longer derive an optimal solution. When one variable is displaced by another in the same register file, that displaced variable may need to be stored into data memory, requiring a load when it is needed in the future. However, if a free register exists in some other register file, then it might be possible to "store" the displaced value there temporarily until its future use. Further, even if there is no free register in the remote file, it still possible that some remote variable can be spilled without loss of performance, thus freeing a register. In general, this effect can have cascading effect and become quite complex, with a variable "hopping" from register file to register file until its future use. Extending the algorithms to handle this would be straight-forward, but impractical.

7.5 Comparing Loop Register Allocation with Other Approaches

In this section the deficiency of previous approaches in obtaining an allocation of registers to loops which contains a minimal amount of memory traffic is discussed. Comparison is made with the approaches previously mention in Chapter 2: graph coloring, interval graphs and optimal (brute-force) allocation in basic blocks.

7.5.1 Comparison with Graph Coloring

In the traditional graph coloring approach, once a variable has been assigned to a register, it is contained within that register for its entire lifetime. Equivalently, if it is spilled to memory, it is always accessed from memory. This can lead to situations in which a variable that is currently in a register is not being referenced

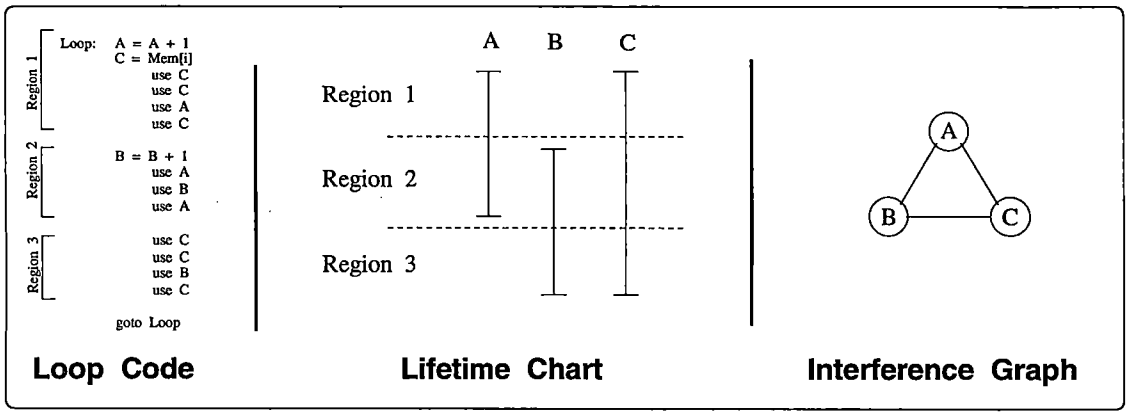


Figure 7.8: Example for graph coloring comparison.

and a variable that is in memory is currently accessed heavily. Thus, the traditional graph coloring approach does not necessarily derive an allocation that is optimal with respect to the minimization of loads and stores due to spill code.

As an illustration of this, consider the loop code found in Figure 7.8 with two registers for allocation. In this example, there are three code regions: in Region 1, two variables (**A** and **C**) are defined and used; in Region 2, another variable **B** is defined, and, at the end of this region, the variable **A** has its last use; in Region 3, the variables **B** and **C** are used and **B** has its last use. Also pictured is the lifetime chart and interference graph for the variables **A**, **B**, and **C**.

In this case, the interference graph is not 2-colorable, so one of the variables **A**, **B** or **C** must be spilled. Because, in this example, **C** has a long lifetime and is heavily accessed in Regions 1 and 3, most coloring heuristics will assign a register to this variable and will pick one of **A** or **B** to spill. However, once either **A** or **B** is selected and spilled, more spill code is generated than is necessary: if **A** is spilled, then unnecessary spill code for **A** is inserted in Region 1 (as the register allocated to **B** is not currently used); similarly if **B** is selected and spilled, more spill code than necessary is inserted in Region 3 (as the register allocated to **A** is not currently used).

Some research [25] has investigated this problem and has proposed a solution based on splitting variable lifetimes. In this approach, rather that deciding

whether to spill **A** or **B**, the lifetime of one of those variables will be split. If the lifetime of **A** is selected for splitting, then it will be split at the definition of **B**, the point where the register pressure becomes too high, forming two lifetimes **A1** (the lifetime of A in Region 1) and **A2** (the lifetime of A in Region 2). Then, in the subsequent allocation phase, **A1** and **B** can be allocated to a register while **A2** will be referenced from memory. The case for **B** is analogous.

This strategy will find a solution with less spill code as compared to the traditional method, but still suffers from allocating a variable for the duration of a lifetime (whether it be split or not)—**C** in this example. That is, once a variable is allocated, it is not subject to further lifetime splitting. For instance, to minimize memory traffic, the register allocated to **C** could be used by **A2** (or **B1**) in Region 2 as **C** is not accessed. However, since **C** has already been allocated to a register, its lifetime will not be split, and thus, this solution will not be found by lifetime splitting.

Figure 7.9 contains spill code produced by graph coloring and the loop allocation technique. In the graph coloring solution there are nine loads and stores that are executed *each* iteration, while in the loop solution, there are twelve loads and stores executed every *two* iterations (an average of six per iteration), yielding a significant improvement in performance when the loop executes for many iterations.

7.5.2 Comparison with Cyclic Interval Graphs

As mentioned previously, the interval graph for basic blocks can be optimally colored in polynomial time [37]. However, in a global context, this is an NP-Hard problem [36]. Cyclic interval graphs are an attempt to extend interval graph coloring to loops. This approach arbitrarily breaks the lifetime of cyclic variables at loop boundaries and then colors the lifetime intervals. When the intervals for a given cyclic variable do not have the same color, register-to-register move instructions and (possibly) load and store instructions are necessary to match the register usages at the top and bottom of a loop.

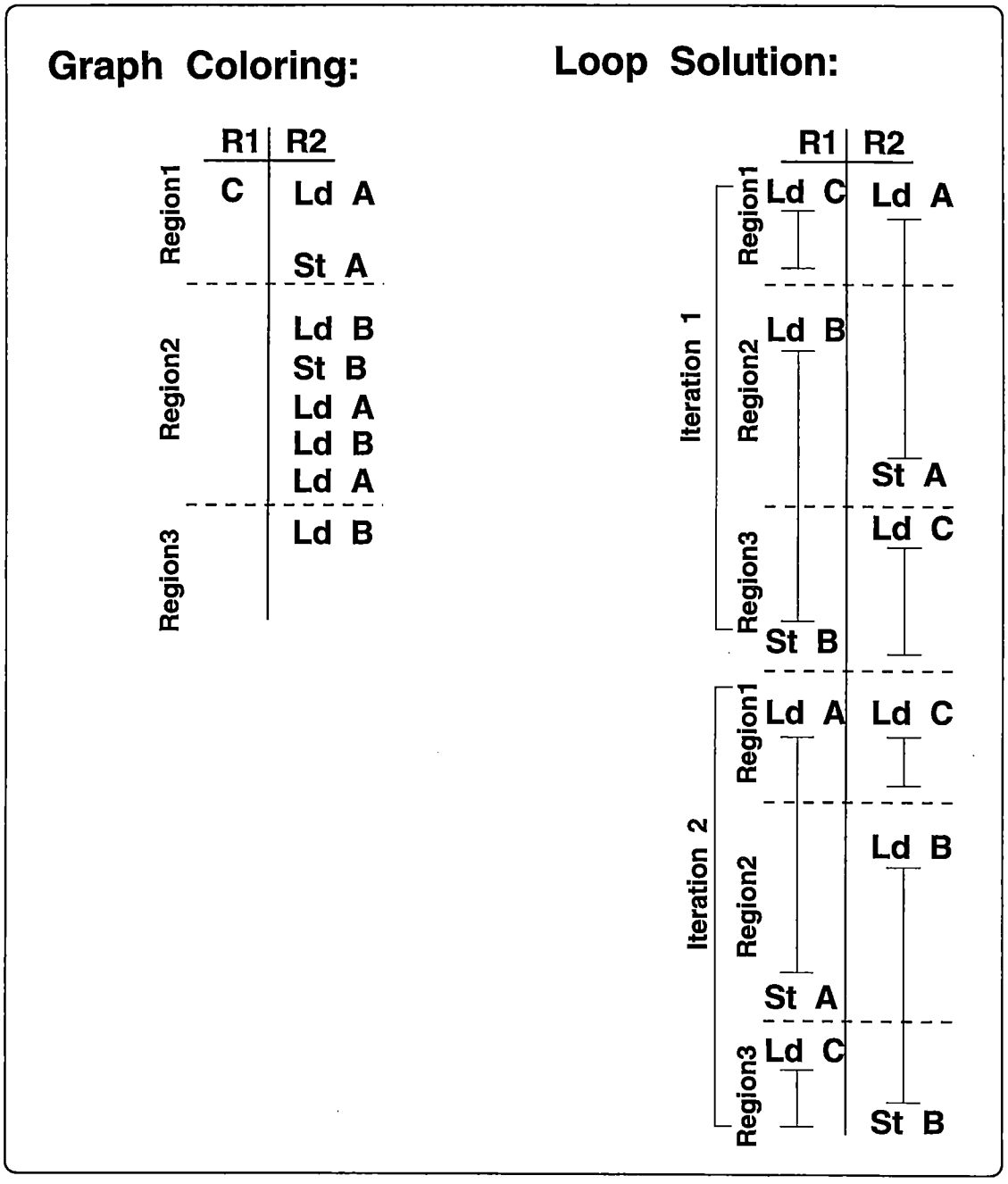


Figure 7.9: Solution for graph coloring comparison.

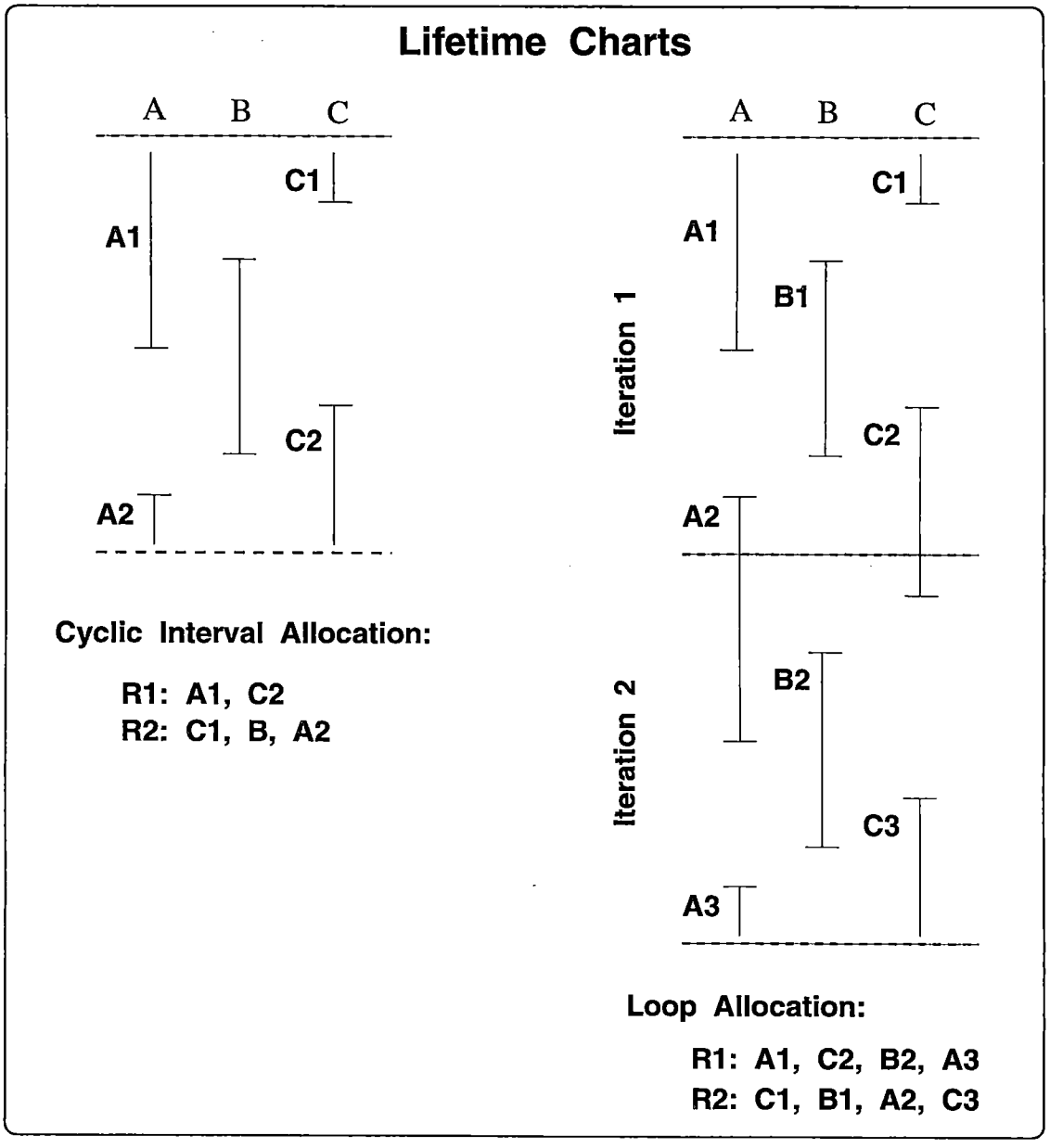


Figure 7.10: Example for cyclic interval graph coloring comparison.

As an example, consider the lifetime chart in Figure 7.10. In this example there are two cyclic variables **A** and **C** which, when split at the loop boundary, have two intervals each (labelled with a 1 or 2). Since there are at most two overlapping lifetimes, two registers are sufficient for allocation. In the allocation derived by the cyclic interval approach, the variable-to-register mapping at the loop end is { **A2** → **R1**, **C2** → **R2** }. Because, at loop top, **A** is expected in **R1** and **C** in **R2**, move instructions are necessary to swap **A** and **C**. However, for a sequential processor or a parallel processor which cannot realize all the moves in parallel (i.e., the moves must all be executed concurrently), this requires temporary storage. If another register is not available, then load and store instructions as well as move instructions are necessary to properly swap the values.

Also depicted in Figure 7.10 is the allocation produced by the loop algorithm presented here for the example. In the loop solution, rather than forcing the values into registers (which requires spill code) the loop is unrolled for another iteration and the next iteration is allocated with the variable-to-register mapping found at the end of the previous iteration. Thus, an allocation is found where the variable-to-register mappings at the iteration boundaries *naturally* match (i.e., no spill code is necessary to enforce the match) and a solution with minimal spill code—none in this case—is found.

7.5.3 Comparison with the Basic Block Strategy

The algorithm presented earlier finds a register allocation for a basic block with minimal spill code cost as it exhaustively explores every possible register allocation at each point in the code. However, this basic algorithm is not adequate to allocate registers to loop bodies[4] as the virtual-to-real register configurations at the end of the loop body must match the initial register assignments at the top of the next iteration body. The difficulty lies in the loop body having two preceding basic blocks—the loop entry and "backedge." This

[4]For simplicity of exposition the loop body does not contain conditionals. Although extensions exist to handle this, they are beyond the scope of this thesis.

considerably complicates matters as it is now necessary, when allocating registers to a loop body, to have the register configurations at the end of a loop body match those at the beginning. If the register configurations, resulting from the optimal basic block algorithm in [45, 46, 48], are not the same at these two points, the computation performed by the resultant code is not correct unless register moves and/or spills are generated to enforce a match. However, since the cost of this additional spill code may vary greatly from configuration to configuration and would vary further by unrolling the loop some number of times, BB-Opt's results, which ignore this effect, cannot be optimal (i.e., cannot produce minimal spill code) for loops.

As an illustration, consider Figure 7.11 with two registers for allocation. In this example, VR1 denotes virtual register one and R1 denotes register one. In (a), the original code with virtual register assignment is found. In (b) the spill code generated by applying BB-Opt to (a) appears using {VR3,VR5} as the initial configuration (the exit configuration from the previous basic block).

Since the values present in the registers at loop body entry are VR3 and VR5, it becomes necessary to re-load these values at the end of the loop body so that the loop code remains correct in (b). Notice that the spill cost for the loop body in (b) is eight (resulting from BB-Opt) for a single sequential execution, and a cost of three was added in order to make it possible to iterate over this loop body (i.e., so that the "backedge" values match the loop entry values).

Flow information can be used to further improve the code. The code segment in (c) has been adjusted accordingly and results in a cost of ten. Note that even though the register-to-register move does not read or write main memory, it is still counted as it is a necessary consequence of the generated spill code and requires functional resources to execute. This is another extension necessary in adapting the BB-Opt algorithm to the handling of loops. Unfortunately, even with this optimization, this method does not yield the minimal spill cost per iteration.

Figure 7.12 illustrates an allocation where the cost per iteration is lower. This

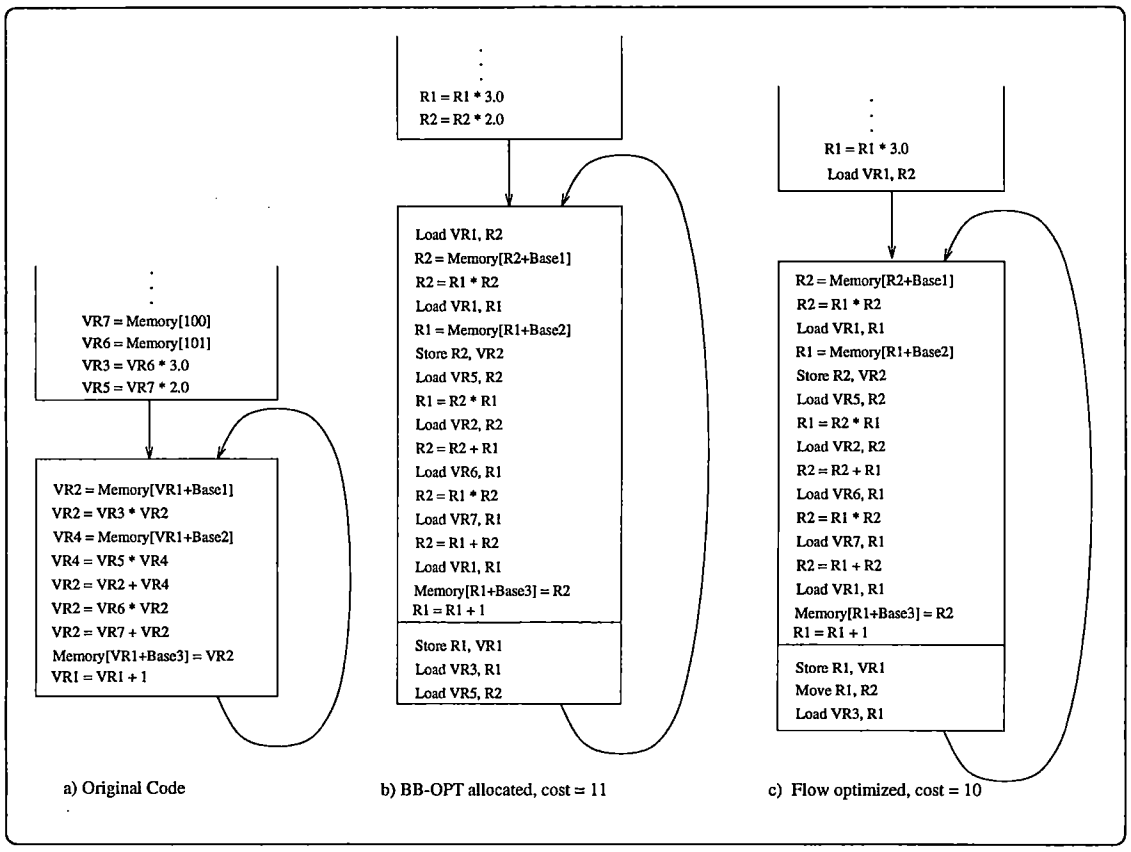


Figure 7.11: A loop basic block allocated with BB-Opt.

lower cost of nine was found as a result of unrolling[5] the loop one iteration. When a match in register usages was found between loop top and bottom, flow of control is directed back to the loop entry. The resultant code is minimal and correct as the allocation for the body of the second iteration resulted in an exit configuration that is minimal *and* matches the entry configuration of the first iteration as well. Thus correctness is preserved and a lower spill cost per iteration is found at the expense of a larger object code size. Note that, in general, the allocation produced may span multiple iterations.

[5]When using this technique, the intermediate exit tests of the loop can be removed if the number of unwindings is a multiple of the number of executed iterations. If this is not the case, the exit test can simply be left. In general, an adjustment to the code speculatively executed before the exit may be required.

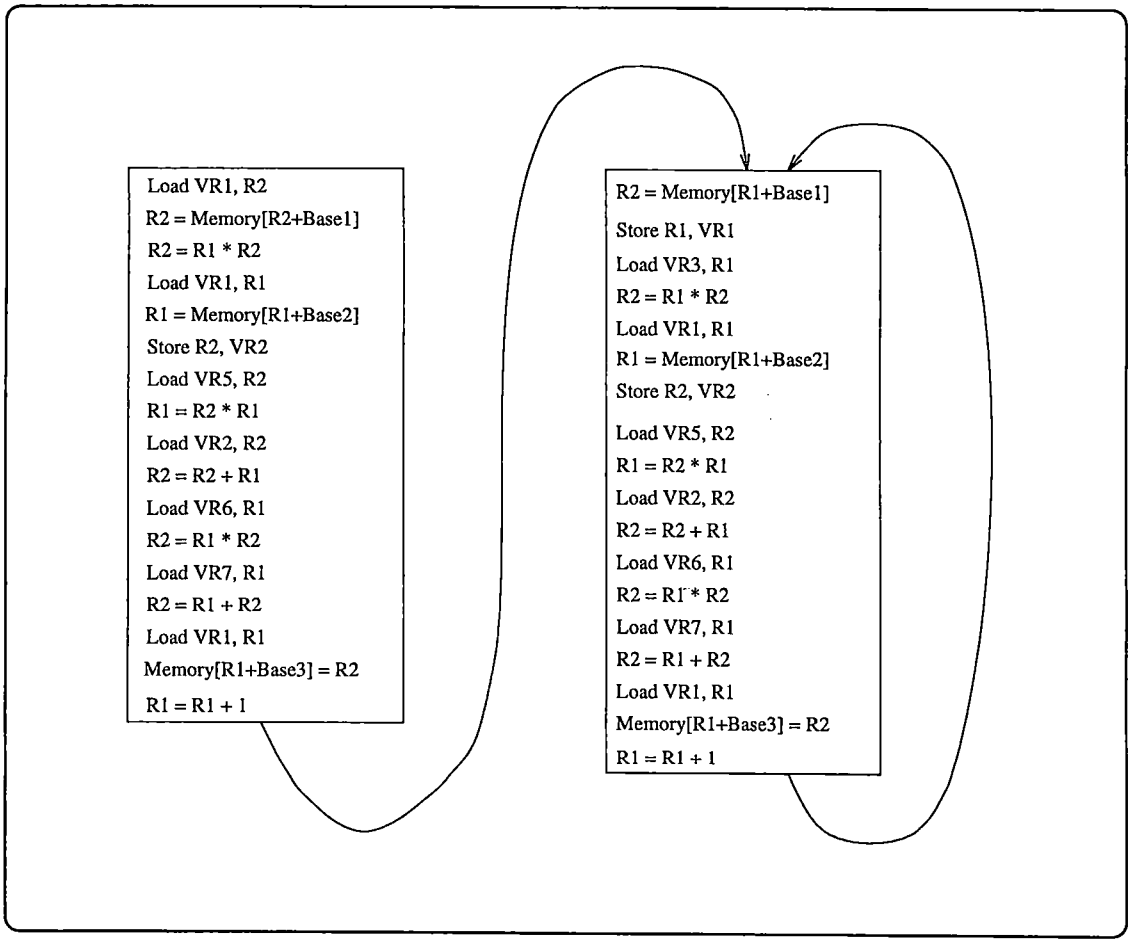


Figure 7.12: A Loop-Opt allocated loop basic block with cost nine.

Chapter 8

Experimentation

In this chapter, experimentation with the techniques discussed in Chapter 4 through Chapter 7 is presented. Experiments were conducted with two separate software entities: the first being a parallelizing compiler with an underlying VLIW model in which the techniques for Redundancy Elimination (Chapter 4), Transformation Integration (Chapter 5) and Copy Elimination (Chapter 6) are implemented; the second, implemented separately due to complexity issues, is a register allocator which performs the allocation scheme presented in Chapter 7.

These optimization techniques are applied to benchmark codes to study their benefits. Code *performance* is chosen as the metric for studying merit as a measure which specifically enumerates the improvement in register allocation (e.g., the decrease in register pressure at various program points) does not demonstrate the practical (attainable) benefits of the optimizations. Further, reducing the register pressure reduces the resource demands at the respective points in the code, allowing a code compactor greater freedom and flexibility in scheduling. Finally, as the code is ultimately executed in a system, performance improvement directly reflects the quality of the generated code.

In the first set of experiments with the parallelizing compiler, each technique was studied in isolation. That is, only one of Redundancy Elimination, META-Transformation or Copy Elimination was applied to the code with the benchmark suite selected for experimentation exhibiting characteristics appropriate for optimization. The results of this experimentation are found in

Section 8.3 through Section 8.5. Results for experimentation combining the techniques, that is, applying all of the presented techniques to the code, are found in Section 8.6. Results for experimentation with the register allocator on a suite of benchmarks are presented in Section 8.7. Finally, results are summarized in Section 8.8.

8.1 Measuring Improvement

For all experiments, the measure of performance is based upon *Amdahl's Law* which measures *Speed-Up* as:

$$Speed\ Up = \frac{cycles_{unoptimized}}{cycles_{optimized}},$$

where $cycles_{unoptimized}$ is the number of cycles of execution for the unoptimized or untransformed code and $cycles_{optimized}$ is the number of cycles of execution for the code to which the optimization or transformation has been applied [44]. With this measure, optimized code which executes twice as fast as unoptimized code will have a Speed-Up of 2.00, for instance.

Typically, the unoptimized code is a sequential version of the input code while the optimized code is a parallelized version. However, because the enhancements discussed in this thesis are applicable during the parallelization process, "unoptimized" code refers to parallelized code *without* application of the particular transformation under consideration (but with all other optimizations and transformations used to parallelized the code), while "optimized" code refers to the application of that transformation during parallelization.

In order to assess the benefits of the transformations presented in this thesis, a modification of the Speed-Up measure is proposed. The value of one is subtracted from the above measure of Speed-Up to derive the *net* improvement and that result is multiplied by 100 to indicate a percentage.

Thus, *percentage improvement* is measured as:

$$\begin{aligned} \%Improvement &= (Speed\ Up - 1) * 100 \\ &= \left(\frac{cycles_{unoptimized}}{cycles_{optimized}} - 1 \right) * 100 \\ &= \left(\frac{cycles_{unoptimized}}{cycles_{optimized}} - \frac{cycles_{optimized}}{cycles_{optimized}} \right) * 100 \\ &= \left(\frac{cycles_{unoptimized} - cycles_{optimized}}{cycles_{optimized}} \right) * 100 \end{aligned}$$

With this measure, optimized code which executes twice as fast as unoptimized code will have a percentage improvement of 100%.

Throughout this chapter, the phrase *percentage improvement of X over Y* refers to the above method of measure and reflects the net improvement or net gain as a percentage of X over Y. The above equations then demonstrate percentage improvement of optimized code over unoptimized code.

8.2 Experimental Set-up

For the purposes of experimentation, an architectural model was selected and is found in Figure 8.1. As the information for this particular organization is detailed to the compiler through a set of microcode macros, not "hard-coded" into the compiler, other organizations—architectures with partitioned register files and/or specialized registers, for example—are possible. The results presented in this chapter reflect experimentation with this architecture, but may differ from experiment to experiment in the instruction latency assumptions.

In this architecture, all functional units are connected to a single register file and are able to equally access all contained registers. Values read from the register file are latched at the inputs to the functional unit while the result computed by a functional unit is latched at the output before being written into the register file. Each functional unit can execute only one instruction per cycle, but may be pipelined for some number of execution cycles (the *latency* of the type of

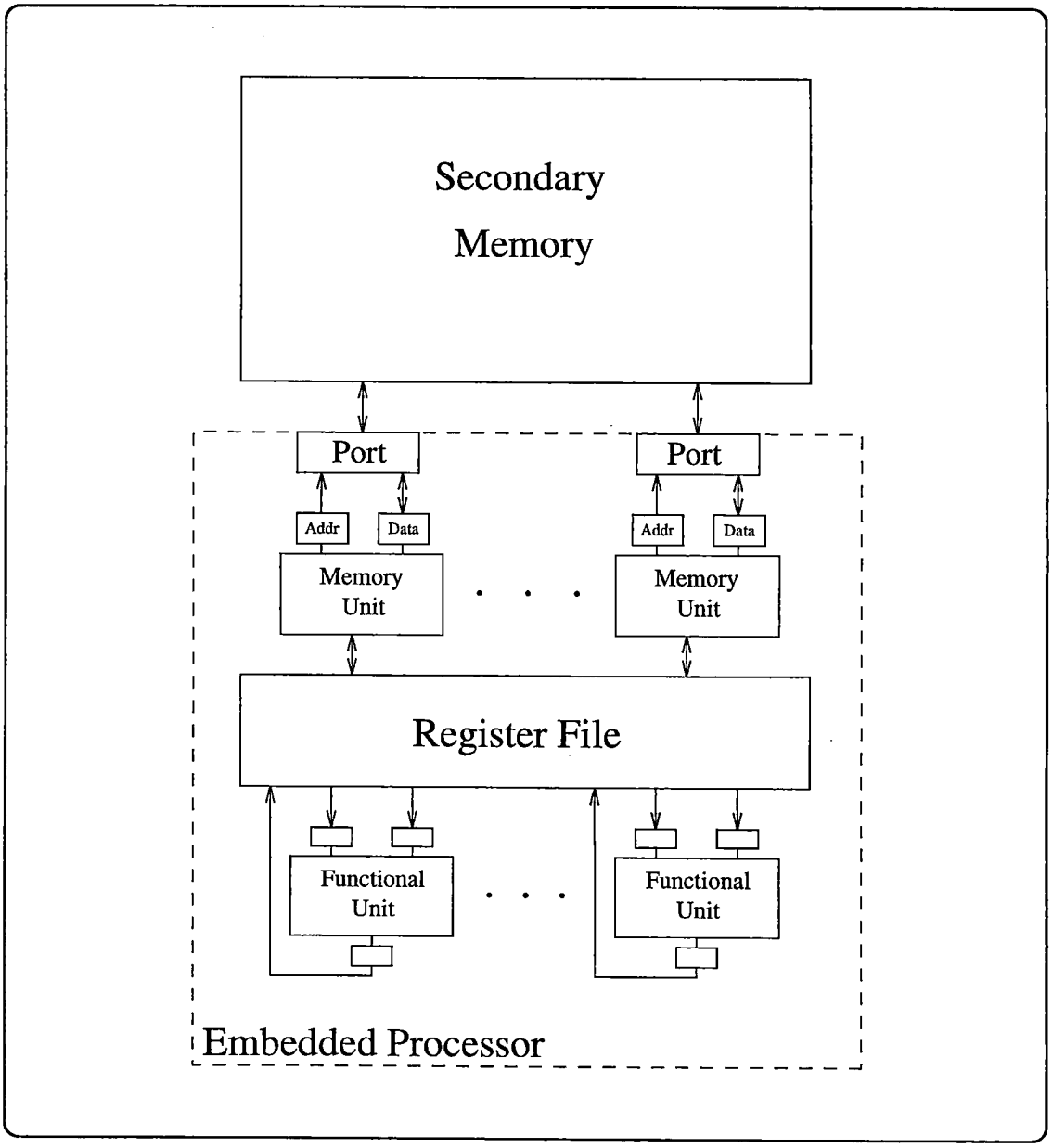


Figure 8.1: Architecture model for experimentation.

Table 8.1: A description of the redundancy elimination benchmark suite.

Benchmark	Abbr.	Description
Numeric		
General linear recurrence equations	GLR	Solves series of linear recurrences.
Prefix Sums	Prefix	Calculation of partial sums (scan).
Tri-diagonal Eliminiation	Tri-diag	Performs tri-diagonal elimination below the diagonal of a matrix.
Scientific		
2D-Hydrodynamics implicit calculation	2D-Hydro	Hydrodynamics particle simulation exerpt.
Successive Over-Relaxation	SOR	Partial differential equation solver.
Image		
Laplace edge enhancement	Laplace	Laplace method for enhancing all edges.
Low-pass filter (arb. coeff.)	Low-pass1	Accentuates low frequencies (parameterized for arbitrary enhancement).
Low-pass filter	Low-pass2	Accentuates low frequencies.
North-edge gradient enhancement	North-edge	Enhances northerly direction edges.
Wavelet Transform	Wavelet	Image compression algorithm.

instruction). Instructions are mapped to functional units based upon their type, with the exception of memory instructions.

Memory accessing instructions are mapped to specialized memory units. Depending upon the type of port (read or write) the unit performs the necessary address computation and provides an address to a load port for load instructions, later writing the fetched data into the specified register in the register file, or an address and datum fetched from the specified register for store instructions.

8.3 Eliminating Redundancy

The benchmark suite chosen to study the effects of redundancy elimination is composed of codes from several application areas. Table 8.1 lists the benchmarks

Table 8.2: Statistics for the redundancy elimination benchmark suite.

Benchmark	Total # mem ops	Redundancy		Removed	
		#	%	#	%
GLR ⋆	8	2	25%	2	100%
Prefix	3	1	33%	1	100%
Tri-diag	4	1	25%	1	100%
2D-Hydro	12	4	33%	4	100%
SOR	14	3	21%	3	100%
Laplace	9	6	66%	6	100%
Low-pass1	9	6	66%	6	100%
Low-pass2	9	6	66%	6	100%
North-edge	9	6	66%	6	100%
Wavelet	20	14	70%	14	100%

along with a brief description of each. For each benchmark, Table 8.2 contains the number of memory instructions that are contained within the respective innermost loops[1], the number and percentage of redundant memory instructions and the number and percentage of redundant memory instructions that were removed from the code. In all cases, the transformation was able to remove all of the redundant memory instructions found in the benchmark suite.

The instruction latencies given to the compiler for this experimentation are: two cycles for ALU instructions, three cycles for multiply instructions, and five cycles for load/store instructions.

8.3.1 Observed Results

Two experiments were conducted: for each benchmark, two schedules were generated with the sole difference between them being the application of the redundancy elimination transformation. In the first experiment, schedules were generated with the number of memory ports constrained between one and four and *no* functional unit constraints to isolate the difference in transformed schedules without the bias of functional unit constraints. In the second

[1]The GLR benchmark (marked with ⋆) has two loops at the same nesting level. Throughout this section, measurements of this benchmark relate to the sum of both loops.

Table 8.3: Results of experimentation with redundant elimination.

Benchmark	Number of Ports	∞ Resources			2 Adders, 1 Multiplier		
		without	with	% Impr.	without	with	% Imp r.
GLR⋆	1	25	14	79%	28	17	65%
	2	24	12	100%	28	16	75%
	3	24	12	100%	28	16	75%
	4	24	12	100%	28	16	75%
Prefix	1	8	6	33%	9	6	50%
	2	8	5	60%	9	5	80%
	3	8	5	60%	9	5	80%
	4	8	5	60%	9	5	80%
Tri-diag	1	10	7	43%	10	7	43%
	2	9	6	50%	9	6	50%
	3	9	5	80%	9	5	80%
	4	9	5	80%	9	5	80%
2D-Hydro	1	32	26	23%	38	26	46%
	2	28	24	17%	32	25	28%
	3	26	24	8%	28	24	17%
	4	25	22	14%	26	23	13%
SOR	1	24	22	9%	28	24	17%
	2	22	21	5%	26	23	13%
	3	21	20	5%	23	21	10%
	4	20	20	–	20	19	5%

experiment, schedules were generated with one to four memory ports and the functional unit constraints of two adders and one multiplier to study the effects on performance in the presence of realistic functional unit resources. For each experiment, the number of cycles in the schedule of the innermost loop is counted.

The results for the numeric and scientific codes are found in Table 8.3 while the results for the image codes are found in Table 8.4. The results are divided into two columns, one for the cycle counts of schedules generated with infinite resources and another for those generated with the resources of two adders and one multiplier. In each resource column, the results are organized by the schedules without redundancy removed, schedules with redundancy removed and then the percentage improvement (% Impr) of the transformed schedules over the untransformed schedules for the various memory port resources.

Table 8.4: Results of experimentation with redundant elimination (con't).

Benchmark	Number of Ports	∞ Resources			2 Adders, 1 Multiplier		
		without	with	% Impr.	without	with	% Impr.
Laplace	1	21	16	31%	29	21	38%
	2	19	15	27%	28	20	40%
	3	19	15	27%	26	20	30%
	4	18	14	29%	24	18	33%
Low-pass1	1	22	20	10%	30	28	7%
	2	18	16	13%	28	26	8%
	3	17	16	7%	27	24	16%
	4	16	16	–	25	21	19%
Low-pass2	1	17	15	13%	27	24	17%
	2	16	14	14%	25	23	9%
	3	14	13	8%	24	22	9%
	4	13	12	8%	22	22	–
North-edge	1	19	17	12%	25	22	14%
	2	18	17	6%	22	21	5%
	3	18	16	13%	22	20	10%
	4	17	16	6%	21	20	5%
Wavelet	1	12	6	100%	12	6	100%
	2	8	6	33%	8	6	33%
	3	7	6	17%	7	6	17%
	4	6	6	–	6	6	–

Results by Benchmark Type

The results of experimentation with the numerical benchmarks are found in Table 8.3. From Table 8.2, it is noted that these benchmarks have a moderate degree (21% to 33%) of redundancy and therefore a moderate increase in performance is expected. The results for these benchmarks are better than expected—the performance increases observed are between 33% and 100%, with a majority of them above 50%.

Results for the scientific benchmarks are found in Table 8.3. These benchmarks have a moderate degree (between 25% and 33%) of redundancy, so moderate performance increases could be expected. The 2D-Hydro benchmark does demonstrate a moderate increase (13% to 46%) in performance. However, observed performance increases for SOR range between 5% and 17%. Although applying our transformation is worthwhile, the lower performance increase stems from the large number of memory instructions that remain after redundancy removal. Since there are a large number of memory instructions which contend for memory port resources, the schedules still remain bottlenecked by the memory.

The results for the image benchmarks are found in Table 8.4. These benchmarks have a high degree (66% to 70%) of redundancy, and therefore we expect our transformation to make a significant impact on performance. For the Laplace and Wavelet codes this is indeed the case—we observed 27% to 40% increase in performance for Laplace and 17% to 100% for Wavelet. For the other benchmarks, the range of performance increase is lower, ranging between 5% and 19%. With so much redundancy, why are the performance improvements less than expected? These benchmarks have long chains of computations which effectively "hide" the latency of the memory instructions. When memory port resources are scarce, delaying memory instructions has little effect on the critical path and, hence, little effect on performance. However, when combining our technique with other transformations that reduce the critical path length (tree height reduction, for instance) the importance of removing redundant memory instructions increases as those instructions now affect the critical path and thus affect performance. (This is discussed further in the next section.)

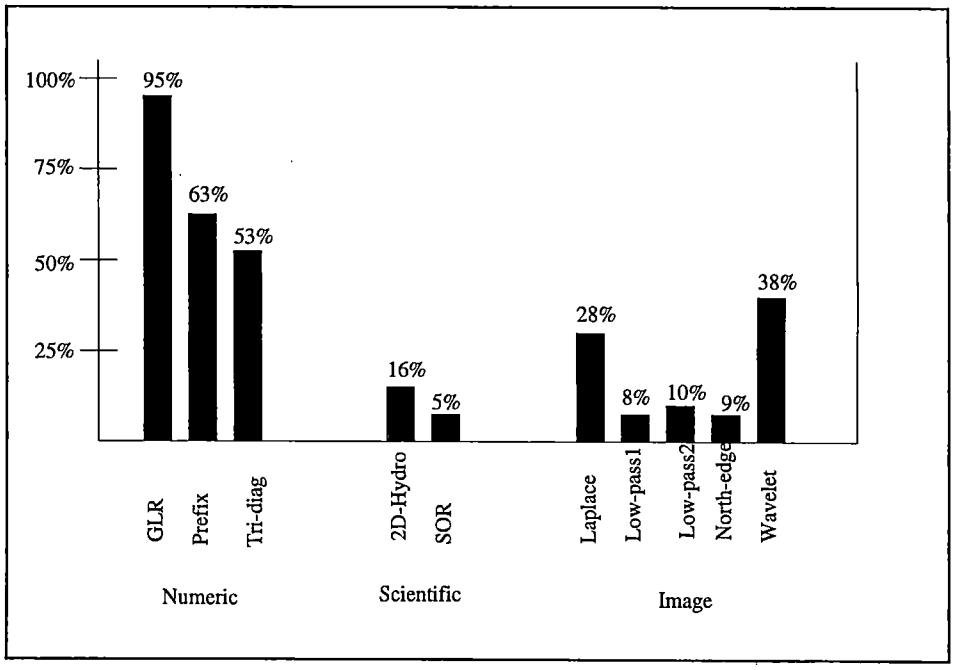


Figure 8.2: Average performance improvement with unlimited functional units.

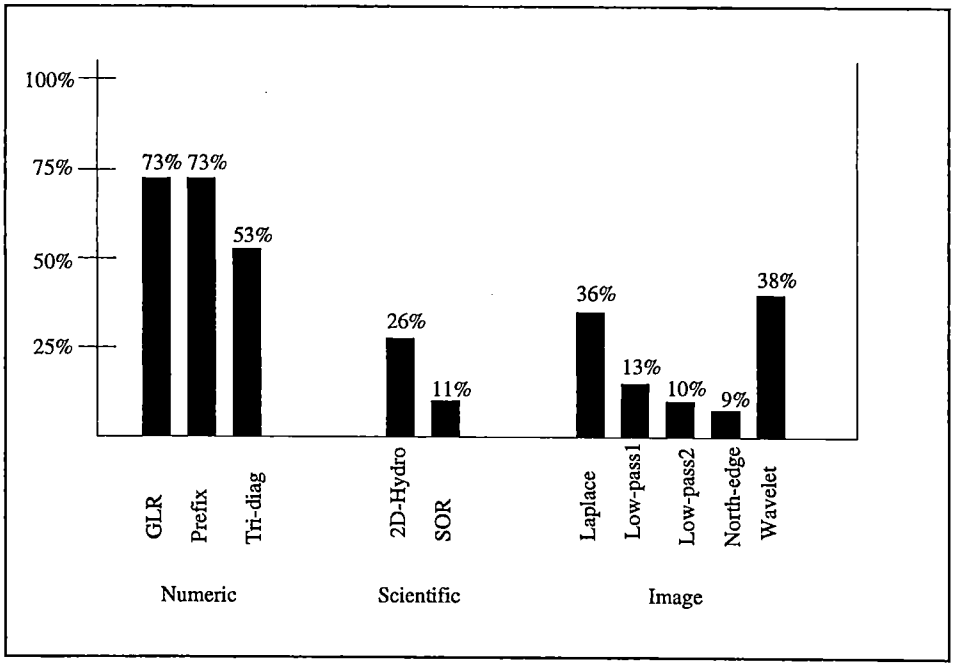


Figure 8.3: Average performance improvement with functional unit constraints.

Results by Resource Constraints

Figure 8.2 graphically shows the average increase in performance improvement for the schedules produced with no functional unit constraints. Because the bias of functional unit constraints has been removed from these measurements, bandwidth limitations have effectively become the bottleneck to performance. Clearly the numeric benchmarks benefit the most (54% to 95%), but the scientific and image benchmarks show considerable performance improvements too, 5% to 16% and 8% to 38%, respectively.

Figure 8.3 graphically shows the average increase in performance improvement for the schedules produced with the functional unit constraints of two adders and one multiplier. Except for the GLR benchmark (which is affected to a greater extent by the particular resource constraints used), the average performance improvement between the untransformed and transformed codes increase or stay the same as compared with the results in Figure 8.2. These results indicate that redundant elimination plays an important role in the presence of realistic resource constraints, yielding 64% to 73% performance improvement for the numerical benchmarks, 12% to 26% for the scientific benchmarks and 9% to 38% for the image benchmarks.

8.3.2 Analysis

These improvements are largely due to the increased flexibility in scheduling that redundancy removal offers. When redundant memory instructions are removed from the code, this flexibility results from less contention for memory port resources as compared to the untransformed code. Higher instruction mobility, which also contributes to scheduling flexibility, is exhibited by instructions which were originally dependent upon (now removed) memory instructions. These instructions can now be scheduled into time cycles corresponding to the latency cycles of the removed instruction, possibly utilizing free resources in those cycles. Finally, critical path length can be reduced when memory instructions on the critical dependency path in the loop are removed, again providing more flexibility to the scheduler.

Table 8.5: A description of the META-Transformation benchmark set.

Benchmark	Abbr.	Description
Spatial Filters		
Generalized Filter	General	Applies spatial filter having arbitrary co-efficients.
Low-pass filter	Low-Pass1	Applies median low-pass filtering.
Low-pass filter	Low-Pass2	Applies low-pass filtering (all co-efficients powers of two).
High-pass filter	High-Pass1	Applies median High-pass filtering.
High-pass filter	High-Pass2	Applies High-pass filtering (all co-efficients powers of two).
Edge Enhancement		
Laplace Transform	Laplace1	Applies Laplace method for edge enhancement.
Laplace Transform	Laplace2	Applies Laplace method for edge enhancement (all co-efficients powers of two).
North-gradient	North-Grad	Accentuates northerly (upward) edges.
Matched-edge Enhancement	Match	Accentuates matching edges.
Blurring		
Blurring	Blur	Performs blurring of an image.
Image Compression		
Wavelet Transform	Wavelet	Image compression algorithm.
Predictor/Corrector	Pred-Corr	Image compression algorithm.

All of these factors—less resource contention, higher instruction mobility and critical path length reduction—combine to increase scheduler flexibility and result in the generation of higher quality schedules.

8.4 Integrating Program Transformations

The benchmark suite chosen to study the META-Transformation comes from image codes as these codes exhibit both accessing of large amounts of data and long chains of computation, making them particularly suited to redundancy

elimination and tree height reduction. Table 8.5 lists the benchmarks along with a brief description of each. The spatial filters, edge enhancements and blurring benchmarks are obtained from [56], while the image compression benchmarks are obtained from [72].

Latencies given to the compiler are: two cycles for add/subtract instructions, three cycles for multiply instructions, and two cycles for load/store instructions. Schedules were generated with the functional resources of two adders and one multiplier[2] and a two-port memory.

8.4.1 Observed Results

Two schedules were produced for each benchmark: The first schedule used the traditional approach of applying transformations as they become possible; the second schedule utilized the META-Transformation approach with the heuristic as previously outlined (Section 5.3.2). The goal of this experiment is to determine the added freedom and flexibility gained by the scheduler and the ability of the heuristic META-Transformation to produce more compact schedules and, thus, generate higher performance code.

Table 8.6 presents the observed results on the benchmark suite. The column labelled "FUs" indicates the functional unit resources used in generating schedules for the respective benchmark (e.g., 2+,1* uses two adders and one multiplier). The columns labelled "Trad" and "Meta" contain the number of cycles of execution for the inner loop schedules produced with the traditional approach and with the META-Transformation, respectively. The last column contains the percentage performance improvement (% Impr) of the META-Transformation over the traditional approach.

The percentage improvements over the benchmark suite ranged from 9% to 23%. This improvement is significant as it is derived solely from the *ordering* of transformation application. That is, for both approaches, the same set of program

[2]In some cases the multiplier was not necessary as some co-efficients were powers of two and resulted in shifts in the generated code.

Table 8.6: Results of experimentation with META-Transformation.

Benchmark	Resources		META-Transformation	Traditional Approach	%Impr.
	2 Adders	1 Mult.			
General	•	•	18	20	11%
Low-pass1	•	•	11	12	9%
Low-pass2	•		12	14	17%
High-pass1	•	•	15	17	13%
High-pass2	•		13	16	23%
Laplace1	•	•	10	12	20%
Laplace2	•		16	19	19%
North-Grad	•		15	18	20%
Match	•		21	25	19%
Blur	•		28	31	11%
Wavelet	•	•	15	18	20%
Pred-Corr	•		9	11	22%

transformations was available to the code scheduler: in one case, transformations were applied whenever possible, and, in the other, were sometimes inhibited by the META-Transformation with purpose of improving register usage.

For the spatial filters, an average of 15% improvement was observed, while for the edge enhancements, an average of 20% improvement was observed. Both of these types of benchmarks exhibit the "neighborhood of values" characteristic found in the introductory example to Chapter 5 where a new value is based upon a window of adjacent array items. Thus, these benchmarks exhibit a high degree of redundancy as well as long computation chains, offering many opportunities for the META-Transformation to influence the optimization of the code and produce significant improvement.

For the blur benchmark, which has a moderate amount of redundancy and moderate computation chains, an 11% improvement was observed, commensurate with the opportunity for META-Transformation to improve the code.

Finally, a 21% improvement was observed for the image compression benchmarks which have a small amount of redundancy, but very long computation chains, demonstrating that the META-Transformation was able to derive significant

improvement when the application code displays lower degrees of redundancy.

8.4.2 Analysis

These results demonstrate that integration of the transformations results in better performance due to better resource allocation. With the integrated approach, array values and intermediate values are kept in registers for shorter amounts of time (versus the traditional approach) as instructions on redundant values are executed during the latency of those memory instructions that remain. This allows the registers containing values with short lifetimes to be available sooner for allocation to other instructions, resulting in greater scheduling freedom when registers are a critical resource.

8.5 Copy Elimination

The benchmark suite chosen to study the effects of copy elimination is composed of codes from several application areas. Table 8.7 lists the benchmarks along with a brief description of each. The latencies used for scheduling are: one cycle for copy instructions, two cycles for add/subtract instructions, three cycles for multiply instructions and three cycles for memory accessing instructions. For functional unit constraints, the architecture contains two add/subtract units, one multiply and a single port to memory.

8.5.1 Observed Results

Three schedules were generated: the first schedule contained copy instructions, the second schedule applied the copy elimination algorithm and the third used a heuristic version of copy elimination with the maximum number of iterations of unrolling on any path bounded by three.

Table 8.8 provides a comparison of the observed results on the benchmark suite

Table 8.7: A description of the copy elimination benchmark suite.

Benchmark	Abbr.	Description
Numeric		
Cholesky Conjugate Gradient	Cholesky	Computes (incomplete) Cholesky conjugate gradient.
Tri-diagonal Eliminiation	Tri-diag	Performs tri-diagonal elimination below the diagonal of a matrix.
General linear recurrence equations	GLR	Solves series of linear recurrences.
State Equations	State Eqs.	Solves state equations.
Partial Differential Equations Solver	PDS	Partial differential equation solver. exerpt.
Integrator Predictor	Integrator	Performs integration prediction.
Difference Predictor	Difference	Performs difference prediction.
Prefix Sums	Prefix	Calculation of partial sums (scan).
Difference Sums	Diff. Sums	Calculation of difference sums.
Scientific		
2D-Hydrodynamics implicit calculation	2D-Hydro	Hydrodynamics particle simulation exerpt.
2D-Particle in Cell	2D-Particle	2D particle in cell simulation.

Table 8.8: Results of experimentation with copy elimination.

Benchmark	Code with Copies	Copies Eliminated	% Impr
Cholesky	13	11	18%
Tri-diag.	16	14	14%
GLR	10	9	11%
State Eqs	23	19	21%
PDS	16	12	34%
Integrator	18	16	13%
Difference	18	15	34%
Prefix	10	8	25%
Difference sum	22	17	29%
2D-Hydro	35	31	13%
2D Particle	24	20	20%

for the schedules generated with copy elimination and without copy elimination. The first two columns contain cycle counts for one iteration of the respective schedules while the last column contains the percentage improvement of copy eliminated code over code scheduled with copies.

The performance improvement over the benchmark suite ranged from 11% to 34%. For the scientific benchmarks a 13% performance improvement was observed for the 2D-Hydro while a 20% performance improvement was observed for 2D-Particle. The performance improvement for the 2D-Hydro is due to the particular computation being performed. In this case, several variables are defined and used throughout the computation. During loop pipelining, many copies are generated. When those copies are removed several resources (those that the copy instructions occupied) are free, providing greater scheduling freedom.

For the numerical benchmarks, an average performance improvement of 21% was observed. Most of the numerical codes contain recurrences which span few iterations. During loop pipelining, copy chains are generated with the length of the copy chain equal to the number of iterations spanned by the recurrences. When copies are eliminated, some resources become free over those iterations allowing better code compaction.

Table 8.9 contains the observed results on the benchmark suite for copy elimination and heuristic copy elimination with an unrolling bound of three iterations. Noted in the table is the cycle counts for one iteration of the respective schedules as well as the number of iterations spanned by the unbounded copy elimination schedules. In some cases, the heuristic version was able to derive the same performance solution as the unbounded version (3 of 11) and in other cases derived solutions with results which are close to the unbounded (8 of 11).

8.5.2 Analysis

As noted in Chapter 6, the copy elimination algorithm does not affect the register requirements of the code—the copy eliminated code uses the same number of

Table 8.9: Results of experimentation with heuristic copy elimination.

Benchmark	All Copies Eliminated	Heuristic Version	Number of Iterations	Number of Copies Remaining
Cholesky	11	11	3	0
Tri-diag	14	15	5	2
GLR	9	9	3	0
State Equations	19	20	3	1
PDS	12	13	5	2
Integrator	16	17	3	1
Difference	15	16	3	2
Prefix	8	8	2	0
Difference sum	17	20	3	3
2D-Hydro	23	29	4	5
2D Particle	20	22	4	3

registers as the original code: Without copy elimination, values are passed in the code from definition to last use through copy chains. In code where copies have been eliminated, registers have been re-allocated so that the definition and all uses of a value are made explicit (i.e., they refer to the same register).

The performance improvements resulting from the use of this technique arise from the freeing of resources that copy instructions require for execution, allowing the scheduler to better compact the code (perform code motion into the free resources). In some cases, this compaction may result in reducing the register requirements as values' lifetime's may actually *decrease* with respect to the lifetimes of the code scheduled with copy instructions.

Therefore, code which benefits the most from this technique will be that in which there is a significant ratio of number of copy instructions to total number of instructions and the necessary number of iterations to unroll to eliminate those copies is few. The resulting code will then span a few iterations of the original code, but will have many free resources to provide significant compaction.

Another consideration in embedded systems is the size of the code that is generated. Examining the results of Table 8.9 demonstrates that using a heuristic version of copy elimination provides enough flexibility to gain substantial

Table 8.10: A description of the benchmark suite for all transformations.

Benchmark	Abbr.	Description
Numeric		
Prefix Sums	Prefix	Calculation of partial sums (scan).
Scientific		
2D-Hydrodynamics implicit calculation	2D-Hydro	Hydrodynamics particle simulation exerpt.
Successive Over-Relaxation	SOR	Partial differential equation solver.
Image		
Low-pass filter	Low-pass	Accentuates low frequencies.
North-edge gradient enhancement	North-edge	Enhances northerly direction edges.

Table 8.11: Remaining number of copy instructions after parallelization.

Benchmark	Number of Copy Instructions	
	Renaming	Redundancy Elim.
Prefix	3	1
2D-Hydro	6	4
SOR	5	3
Low-pass	5	6
North-edge	4	6

performance improvement while keeping the code size manageable. For example, for the Tri-diag benchmark, completely removing all copies results in a schedule of 14 cycles per iteration and spans 5 iterations. For the heuristic version, which spans 3 iterations, a schedule of 15 cycles per iteration results. Thus, by exploring different heuristic unrolling bounds, various performance versus code size measures can be gained.

8.6 Combining Techniques

In this section, the observed results for experimentation combining the techniques examined thus far, Redundant Elimination, META-Transformation and Copy Elimination, are presented. The benchmark suite used for this

Table 8.12: Results of experimentation with all transformations.

Benchmark	All Transformations	No Optimizations	% Impr.
Prefix	8	10	25%
2D-Hydro	23	31	35%
SOR	22	27	23%
Low-pass	10	14	40%
North-edge	15	19	27%

experimentation is composed of numerical, scientific and image codes. Table 8.10 lists the benchmarks along with a brief description of each. For each benchmark, Table 8.11 lists the number of copy instructions that remain in the code after parallelization. These numbers are broken down into two categories: copies generated by the redundant elimination optimization and copies generated by renaming. In all cases, the copies were eliminated.

The functional unit resources used for scheduling are two ALUs, one multiplier and one port to memory. The instruction latencies used are one cycle for ALU instructions, two cycles for multiply and two cycles for loads/stores.

8.6.1 Observed Results

For each benchmark, two schedules were generated: the first schedule did not have any of the techniques applied to it while the second had all techniques applied. From each schedule cycle counts were taken.

Table 8.12 provides a comparison of the observed results between the schedules generated with the techniques and those generated without the optimizations. The first two columns present the cycle counts for the respective schedules while the last column contains the percentage improvement of the transformed schedules over those without optimization. The performance improvement over the benchmark suite ranged from 23% to 40%.

8.6.2 Analysis

One of the drawbacks with both the Redundancy Elimination and Renaming techniques is the generation of copy instructions which can serve to reduce the potential performance enhancement of the optimizations. From Table 8.11, both techniques generate a significant number of copies. The results presented in this section demonstrate that these techniques can be utilized to exploit parallelism in the code, as, once all transformations are integrated into one parallelization entity, as this experimentation elucidates, the copy instructions (overhead instructions) can be eliminated. However, eliminating the copy instructions requires an increase in the code size due to loop unrolling and must be constrained accordingly.

8.7 Allocating Registers to Loops

This section presents the observed results of experimentation with the register allocator described in Chapter 7. Two series of experiments were conducted. The first utilized a single file register model, while the second utilized a partitioned register file model.

Table 8.13 lists the benchmark suite used for experimentation which consists of several numerical codes which are compiled into assembly code. To obtain larger code segments and live ranges, the assembly source is unrolled three times and then the register access patterns are derived.

8.7.1 Experimentation with a Consolidated Model

Using the register access patterns and a consolidated register file model, several experiments were conducted with the BB-Opt algorithm, the Loop-Opt algorithm and heuristic versions and then a comparison of a heuristic version of Loop-Opt with a graph coloring algorithm was done.

Table 8.13: A description of the benchmark suite for register allocation.

Benchmark	Abbr.	Description
Numeric		
Hydrodynamics fragment	Hydro	Calculates (partial) hydrodynamics equations.
Cholesky Conjugate Gradient	Cholesky	Computes (incomplete) Cholesky conjugate gradient.
Inner Product	Inner	Computes inner product of two matrices.
Banded Linear Equations	BLE	(Partial) Banded linear. equations solver.
Tri-diagonal Elimination (below diagonal)	Tri-diag	Performs diagonal elimination on a matrix (below diagonal).
General Linear Recurrence Equations	GLR	Solves systems of linear recurrences.
State Equations	State Eqs	Simulates state equations.
ADI Integration	ADI	Performs ADI Integration.
Integrator Predictor	Integ	Performs (partial) integrator. prediction.
Difference Predictor	Diff	Performs (partial) difference. prediction.
Prefix sums	Prefix	Calculation of partial sums (scan).
Difference sums	Diff Sums	Calculation of partial differences.

BB-Opt

Forcing Register Usages. One idea presented in [48] as an extension to Horwitz's algorithm to deal with simple loops is to "force" the register mappings to be equal at the loop entry and exit (referred to as the "Force Method"). In this scheme only one iteration of the loop body is examined. Some set of values is chosen as members in both the initial (loop top) and final (loop bottom) mappings and allocation proceeds.

With this strategy, although all the selected values are present in the registers at loop end, it may still be possible that "spill" code is necessary as the values are not in the correct registers (e.g., two registers may need to be swapped). Mismatches can be dealt with by introducing register-to-register copy instructions at this point[3].

Note that, since this technique only examines one iteration, it may not necessarily find an allocation with minimal spill code over loop execution as the minimal allocation may span multiple iterations.

Patching Register Usages. Another approach in adapting BB-Opt to loops is to find the minimal leaf from the allocation tree and introduce spill code to match it with the initial configuration (referred to as the "Patch Method"). This method is not guaranteed to produce minimal spill code as the costs of patching each conceivable exit mapping with each conceivable entry mapping can vary greatly. Further, even if all possibilities were examined, a solution with a minimal amount of spill code may still not be found as the minimal solution may span multiple iterations.

Results of Forcing and Patching Methods. Table 8.14 shows the results for both of the Force and Patch methods for two and four registers. For each method, a column labelled "min" indicates whether this method produced less overall spill code. In some cases, results are unavailable as they were

[3]Fixing the selected values in registers *a priori* severely restricts this approach. Consider the selected values of **a**, **b** and **c** and three free registers. Fixing **a**, **b** and **c** positionally in registers precludes the solutions {**b a c**} and {**a c b**}, for instance, that would normally be found by the "force" method.

Table 8.14: Spill costs for the two methods of matching register maps.

	BB-Opt		
Benchmark	Number of Registers	Force Method	Patch Method
Hydro	2	13,734	13,334
	4	6,668	5,868
Cholesky	2	4,736	4,670
	4	1,934	1,467
Inner	2	11,334	10,334
	4	5,334	4,000
BLE	2	3,920	3,752
	4	2,296	1,904
Tri-diag	2	17,316	17,427
	4	5,772	5,994
GLR	2	17,649	17,760
	4	7,326	6,438
State Eqs	2	6,060	5,760
	4	–	–
ADI	2	–	–
	4	–	–
Integ	2	5,338	5,202
	4	1,360	1,224
Diff	2	6,494	6,732
	4	–	–
Prefix	2	12,987	13,653
	4	6,660	5,661
Diff Sums	2	12,987	13,653
	4	6,660	5,661

computationally infeasible to calculate.

In many cases, it is more cost-effective to introduce spill code at the loop exit than to force the register mappings to be equal. That is, the Patch method produced the best results in 14 of 20 cases. These results are largely due to the upper bound on "Patching" spill code that results with this technique. In the worst case, all values in registers need to be spilled (stores to memory) and must be replaced by new values (loads from memory). With two registers, for instance, the total patch spill code is bounded by four, as potentially two values need to be stored and two values need to be loaded.

Although the Force method out-produced the Patch method in 6 of 20 cases, the main drawback of this technique lies with the need to keep certain values in registers. Because values are necessary at loop end (those values that were selected), keeping them in registers introduces an artificial increase in the register pressure. This, in turn, results in a uniform (i.e., all solutions in the allocation tree are affected) increase in spill code.

Loop-Opt

Table 8.15 contains the observed results when Loop-Opt was applied to the benchmark suite. The column labelled BB-Opt contains the observed results from Table 8.14 which provided the least spill cost while the Loop-Opt column contains the observed results for the loop algorithm. The spills per iteration columns shows the number of spills in a single iteration. Note that these are not necessarily whole numbers for the BB-Opt column as the spill code was allocated for a trace of three iterations, so this cost represents an amortized cost. The last column contains the performance improvement of Loop-Opt over BB-Opt.

These results demonstrate that the savings in spill cost uniformly increases as the number of available registers increases. When only two registers are present, the maximal difference between the two methods is bounded by four (in the worst case both registers have to be spilled and then loaded at the end of the loop). As the number of registers increases, this bound also increases. Thus, Loop-Opt will have an increased performance advantage over BB-Opt as the number of registers increases (especially if the code has high register pressure). However, due to limited computing resources, some of the longer loops and higher number of registers were too time consuming to compute.

Heuristic Loop-Opt

In Table 8.16, the observed results for the heuristic version of Loop-Opt are found. Because the heuristic bounds the width of the search tree, the first column, labelled "Width = 1", represents the case where the minimum configuration is expanded at each step when deriving the allocation tree. This strategy yields results very similar to those obtained by the BB-Opt algorithm with Patching. As the width value is increased, closer approximation of the

Table 8.15: Results of loop register allocation algorithm.

Benchmark	Number of Registers	BB-Opt Dynamic Spill Cost	BB-Opt Spills per Iter.	Loop-Opt Dynamic Spill Cost	Loop-Opt Spills per Iter.	% Impr.
Hydro	2	13,334	33.3	12,800	32	4%
	4	5,868	14.7	4,800	12	23%
Cholesky	2	10,000	50	9,000	45	11%
	4	1,467	7.3	1,400	7	4%
Inner	2	10,334	10.3	9,000	9	14%
	4	4,000	4	2,001	2	100%
BLE	2	3,752	22.3	3,528	21	6%
	4	1,904	11.3	1,514	9	26%
Tri-diag	2	17,264	52	17,264	52	–
	4	5,772	17.4	5,312	16	9%
GLR	2	17,649	53	17,649	53	–
	4	6,438	19.3	5,661	17	14%
State Eqs	2	5,760	48	5,760	48	–
	4	–	–	–	–	–
ADI	2	–	–	–	–	–
	4	–	–	–	–	–
Integ	2	5,102	51	5,000	50	2%
	4	1,224	12	1,100	11	9%
Diff	2	6,494	65	6,200	62	5%
	4	–	–	–	–	–
Prefix	2	12,974	13	12,974	13	–
	4	5,661	5.7	4,991	4	43%
Diff Sums	2	12,974	13	12,974	13	–
	4	5,661	5.7	4,991	4	43%

optimal allocation by the heuristic occurs. With very few exceptions, increasing the width of the allocation tree yields improved results. In the cases where the results did not continue to improve, the heuristic expanded a node which generated children that locally appeared to be better choices (i.e. they had lower costs than their siblings) and served to "knock" other nodes out of the expansion set. As the width of the tree is expanded, this phenomena diminishes rapidly because, at each stage in the allocation tree, there are only a few nodes which are good candidates and as the width of the tree expands, it becomes evident which of them will eventually lead to the minimum.

Table 8.16 demonstrates some cases with eight registers where no spill code is necessary. In these cases, the number of available registers has increased beyond the number of overlapping lifetimes, therefore no spill code is required.

Tables 8.17 and 8.18 show observed results for the algorithm with the depth heuristic and the width heuristic working in conjunction. In both cases, heuristic widths of one, two, and five were used. In Table 8.17 a depth of two was used; while in Table 8.18 a depth of three was used. These results show that the spill cost per original iteration decreases as the width increases with very few exceptions, particularly as the depth increases.

Comparison of Heuristic Loop-Opt with Graph Coloring

The register allocation algorithm currently most widely used is based upon the graph coloring paradigm. The Gnu Standard Distribution C Compiler (GCC) implements a graph coloring scheme in allocating registers. Also, the code produced by this compiler is generally accepted to be of high quality [40]. This compiler is therefore used as a metric of code produced by a graph coloring algorithm for the benchmarks.

Gcc was configured to produce code for the SPARC architecture and the register allocation module was modified so that GCC would produce code for four and eight registers[4]. For the heuristic version of Loop-Opt a width of two is used. Table 8.19 summarizes the results of the code produced by GCC as well as the

[4]Gcc produced an internal compiler error when the register count was set to two.

Table 8.16: Results of heuristic width restriction only.

Benchmark	Number of Registers	Width = 1			Width = 5		
		Cost	Iters	Cost / Iter	Cost	Iters	Cost / Iter
Hydro	2	13,200	3	33	12,800	3	32
	4	5,600	3	14	5,600	3	14
	8	1,200	3	3	1,200	3	3
Cholesky	2	10,000	3	50	9,000	3	45
	4	1,600	4	8	1,600	4	8
	8	0	2	0	0	2	0
Inner	2	10,000	3	10	9,000	3	9
	4	2,000	4	2	2,000	4	2
	8	0	2	0	0	2	0
BLE	2	4,032	3	24	3,528	3	21
	4	1,680	5	10	1,680	4	10
	8	168	3	1	168	3	1
Tri-diag	2	19,256	3	58	17,264	3	52
	4	5,312	4	16	5,312	4	16
	8	0	3	0	0	3	0
GLR	2	19,647	3	59	17,649	3	53
	4	5,661	4	17	5,661	4	17
	8	0	2	0	0	2	0
State Eqs	2	5,760	3	48	5,760	3	48
	4	2,040	4	17	2,160	4	18
	8	360	4	3	360	4	3
ADI	2	5,624	3	296	4,788	3	252
	4	2,128	4	112	2,109	4	111
	8	798	3	42	855	3	45
Integ	2	5,100	3	51	5,000	3	50
	4	1,300	4	13	1,200	4	12
	8	100	3	1	100	3	1
Diff	2	6,200	3	62	6,200	3	62
	4	2,200	4	22	2,100	4	21
	8	600	3	6	500	3	5
Prefix	2	14,970	3	15	12,974	3	13
	4	3,992	4	4	3,992	4	4
	8	0	2	0	0	2	0
Diff Sums	2	14,970	3	15	12,974	3	13
	4	3,992	4	4	3,992	4	4
	8	0	2	0	0	2	0

Table 8.17: Results of heuristic depth of two.

Benchmark	Number of Registers	Depth = 2					
		Width = 1			Width = 2		
		Cost	Iters	Cost / Iter	Cost	Iters	Cost / Iter
Hydro	2	13,200	3	33	12,800	3	32
	4	7,200	3	18	5,200	3	13
	8	1,200	2	3	1,200	2	3
Cholesky	2	10,400	3	52	9,200	3	46
	4	1,600	3	8	1,400	3	7
	8	0	2	0	0	2	0
Inner	2	10,000	3	10	9,000	3	9
	4	2,000	2	2	2,000	2	2
	8	0	2	0	0	2	0
BLE	2	4,032	3	24	3,528	3	21
	4	1,680	4	10	1,680	3	10
	8	168	2	1	168	2	1
Tri-diag	2	19,256	3	58	17,264	3	52
	4	6,640	3	20	5,312	3	16
	8	0	2	0	0	2	0
GLR	2	19,647	3	59	17,649	3	53
	4	5,661	4	17	5,661	4	17
	8	0	2	0	0	2	0
State Eqs	2	5,760	3	48	5,760	3	48
	4	2,040	4	17	2,160	4	18
	8	360	2	3	360	2	3
ADI	2	5,605	3	295	4,731	3	249
	4	2,090	3	110	1,995	3	105
	8	836	3	44	836	3	44
Integ	2	5,100	3	51	5,100	3	51
	4	1,600	3	16	1,200	3	12
	8	100	2	1	100	2	1
Diff	2	6,200	3	62	6,200	3	62
	4	3,700	3	37	2,400	3	24
	8	600	3	6	500	3	5
Prefix	2	13,972	3	14	12,974	3	13
	4	3,992	3	4	3,992	3	4
	8	0	2	0	0	2	0
Diff Sums	2	13,972	3	14	12,974	3	13
	4	3,992	3	4	3,992	3	4
	8	0	2	0	0	2	0

Table 8.18: Results of heuristic depth of three.

Benchmark	Number of Registers	Depth = 3					
		Width = 1			Width = 2		
		Cost	Iters	Cost / Iter	Cost	Iters	Cost / Iter
Hydro	2	13,200	3	33	12,800	3	32
	4	6,000	3	15	4,800	3	12
	8	1,200	2	3	1,200	2	3
Cholesky	2	10,000	3	50	9,000	3	45
	4	2,200	3	11	1,600	3	8
	8	0	2	0	0	2	0
Inner	2	10,000	3	10	9,000	3	9
	4	2,000	2	2	2,000	2	2
	8	0	2	0	0	2	0
BLE	2	3,528	3	21	3,696	3	22
	4	1,680	4	10	1,680	3	10
	8	168	2	1	168	2	1
Tri-diag	2	19,256	3	58	17,264	3	52
	4	5,312	4	16	5,312	4	16
	8	0	2	0	0	2	0
GLR	2	18,981	3	57	17,649	3	53
	4	5,994	3	18	5,661	3	17
	8	0	2	0	0	2	0
State Eqs	2	5,760	3	48	5,760	3	48
	4	2,040	3	17	2,040	3	17
	8	360	2	3	360	2	3
ADI	2	4,655	3	245	4,636	3	244
	4	2,033	3	107	1,957		103
	8	817	3	43	760	3	40
Integ	2	5,000	3	50	5,000	3	50
	4	1,300	3	13	1,200	3	12
	8	100	2	1	100	2	1
Diff	2	6,200	3	62	6,200	3	62
	4	2,200	4	22	2,100	4	21
	8	500	3	5	500	3	5
Prefix	2	12,974	3	13	12,974	3	13
	4	6,986	3	7	3,992	3	4
	8	0	2	0	0	2	0
Diff Sums	2	12,974	3	13	12,974	3	13
	4	6,986	3	7	3,992	3	4
	8	0	2	0	0	2	0

heuristic algorithm with the last column containing performance improvement of heuristic Loop-Opt over GCC.

In all cases, the heuristic produced allocations that were superior to GCC. Furthermore, the heuristic allocations perform better than BB-Opt by an average of 8% (via comparison of Tables 8.15 and 8.19), while the heuristic run-time is comparable to production compilers (see Table 8.20). Also, there are a number of cases with eight registers where the heuristic generated allocations where no spill code was necessary while GCC did not find such allocations.

8.7.2 Experimentation with Distributed Register Files

The TMX320C44 is used as an example of an architecture with distributed register files. Figure 8.4 shows a simplified view of the TMX320C44. In this architecture, there are three register files: Extended Precision Registers which are 40-bits wide and used for floating-point and long integer arithmetic; Auxiliary Registers which are 32-bits wide and used as address pointers with dedicated address generation hardware to auto-increment and auto-decrement address values; and General-Purpose Registers which are 32-bits wide. All register files are connected to the Reg1 and Reg2 busses and available to the Multiplier and ALU. The Multiplier and ALU may both write to the Extended Precision Registers or one of them may write to either the Auxiliary Registers or the General Purpose Registers. Additionally, an operand may be supplied to the Multiplier or ALU by the memory.

Comparison of BB-Opt and Loop-Opt

Table 8.21 contains the observed results for the number of spills per iteration for BB-Opt and Loop-Opt, as well as the percentage improvement of Loop-Opt over BB-Opt measured. Similar to the observed results of experimentation with a consolidated model, there is a general trend for the percentage improvement to increase as the number of registers increases due to the ability of the loop algorithm to naturally match the register usages at loop top and bottom. Upon inspection of the assignments produced it was noted in some cases that the

Table 8.19: Comparison of results between GCC and heuristic Loop-Opt.

Benchmark	Number of Registers	Gnu gcc		Heuristic Loop-Opt Depth = 1, Width = 2		% Impr.
		Dynamic Spill Cost	Spills per Iter.	Dynamic Spill Cost	Spills per Iteration	
Hydro	4	7,600	19	5,600	14	36%
	8	4,800	12	1,200	3	300%
Cholesky	4	4000	20	1,600	8	150%
	8	2,600	13	0	0	∞
Inner	4	8,000	8	2,000	2	300%
	8	8,000	8	0	0	∞
BLE	4	2016	12	1,680	10	20%
	8	1,344	8	168	1	700%
Tri-diag	4	9,628	29	5,312	16	81%
	8	5,644	17	0	0	∞
GLR	4	8,991	27	5,661	17	59%
	8	6,327	19	0	0	∞
State Eqs	4	4,680	39	2,160	18	117%
	8	2,160	18	360	3	500%
ADI	4	2,907	153	2,109	111	38%
	8	1,444	76	855	45	69%
Integ	4	2,700	27	1,200	12	125%
	8	1,600	16	100	1	1500%
Diff	4	5,700	57	2,100	21	171%
	8	2,100	21	500	5	320%
Prefix	4	6,986	7	3,992	4	75%
	8	5,988	6	0	0	∞
Diff Sums	4	6,986	7	3,992	4	75%
	8	5,988	6	0	0	∞

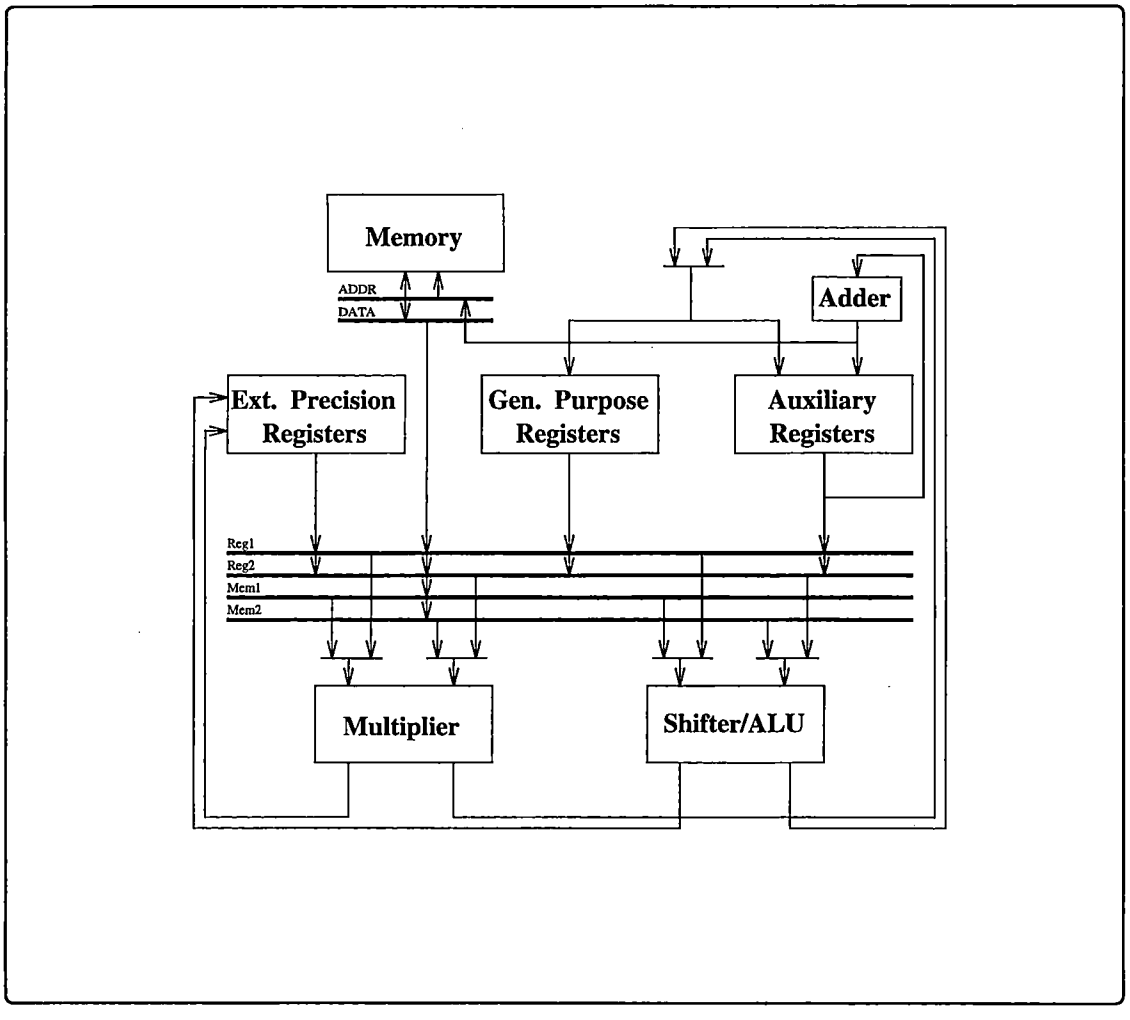


Figure 8.4: A simplified view of the TMX320C44.

Table 8.20: Execution times of the various methods.

	CPU Time			Average Number of Spills per Iteration	
Method	Ave.	Min.	Max.	4 Regs.	8 Regs.
Graph Coloring Gcc	0.06 secs	0.04 secs	0.11 secs	33.8	17.7
BB-Opt	600.0 secs	345.0 secs	2700.0 secs	10.8	—
Loop-Opt	1800.0 secs	480.0 secs	10 hrs.	9.3	—
Heur. Loop-Opt					
Dep=1, Wid=1	0.08 secs	0.06 secs	0.34 secs	9.8	4.7
Dep=1, Wid=2	0.13 secs	0.07 secs	0.48 secs	9.7	4.8
Dep=1, Wid=5	0.34 secs	0.11 secs	0.57 secs	9.8	4.5
Dep=2, Wid=1	0.11 secs	0.08 secs	0.54 secs	11.0	4.8
Dep=2, Wid=2	0.23 secs	0.13 secs	0.65 secs	9.4	4.7
Dep=2, Wid=5	0.39 secs	0.15 secs	0.70 secs	9.3	4.4
Dep=3, Wid=1	0.24 secs	0.16 secs	0.72 secs	11.0	4.7
Dep=3, Wid=2	0.47 secs	0.23 secs	0.89 secs	9.4	4.4
Dep=3, Wid=5	0.93 secs	0.33 secs	1.12 secs	9.3	4.3

assignments produced by the basic block scheme assigned an address variable to a general purpose register near the end of the iteration. This variable was heavily used at the top of the loop, so more spill code (spills of other address variables currently within the Auxiliary Registers) than necessary was generated to accommodate that variable.

Comparison of Loop-Opt and Heuristic Loop-Opt

Table 8.22 presents the results of the spill code produced by the optimal and heuristic algorithms. The last column contains the percentage improvement of Loop-Opt over heuristic Loop-Opt. In a few cases (4 of 14), the heuristic produced results equal to Loop-Opt. For 58% of the cases (7 of 12) results were produced that are within 15% of the optimal. For the rest of the cases (5 of 12), the percentage within optimal is higher, however, the actual difference in spill code produced is only one instruction.

Code Size of Loop Register Assignments

Because code size directly affects the size of the program ROM in an embedded system, the number of iterations produced by the loop assignments is noted.

Table 8.21: Basic block optimal vs. Loop optimal for the TMX320C44.

Benchmark	Number of Registers	BB-Opt Spills per Iteration	Loop-Opt Spills per Iteration	% Impr.
2D-Hydro	2	9	7	29%
	4	2.7	2	35%
Inner	2	4	3	33%
	4	1.7	1	70%
BLE	2	8	7	14%
	4	0	0	—
Tri-diag	2	10	8	25%
(below diag.)	4	2.3	2	15%
Tri-diag	2	11	10	10%
(above diag.)	4	3	2	50%
Prefix	2	7	4	75%
(scan)	4	3	2	50%
Kalman	2	5	4	10%
	4	1.3	1	30%

Table 8.22: Comparison of loop assignments for the TMX320C44.

Benchmark	Number of Registers	Loop-Opt Spills per Iteration	Heuristic Loop-Opt Spills per Iteration	% Impr.
2D-Hydro	2	7	8	14%
	4	2	2	0%
Inner	2	3	4	33%
	4	1	2	50%
BLE	2	7	8	14%
	4	0	1	—
Tri-diag	2	8	8	0%
(below diag.)	4	2	2	0%
Tri-diag	2	10	11	10%
(above diag.)	4	2	3	50%
Prefix	2	4	4	0%
(scan)	4	2	3	50%
Kalman	2	4	5	25%
	4	1	2	50%

Table 8.23: Comparison of loop code sizes for the TMX320C44.

Benchmark	Number of Registers	# Iterations		Equal Spill Code?
		Loop-Opt	Heuristic Loop-Opt	
2D-Hydro	2	3	3	no
	4	2	2	yes
Inner	2	3	4	no
	4	3	3	no
BLE	2	4	4	no
	4	3	4	no
Tri-diag (below diag.)	2	3	3	yes
	4	2	2	yes
Tri-diag (above diag.)	2	2	2	no
	4	2	2	no
Prefix (scan)	2	2	3	yes
	4	2	2	no
Kalman	2	3	3	no
	4	2	2	no

Table 8.23 contains the number of iterations spanned by each of the loop methods for the given number of registers, as well as indication of whether the heuristic method produced an equal amount of spill code as the optimal. In the majority of cases (9 of 12) the heuristic derived assignments which spanned the same number of iterations as the optimal, and, of those, generated the same amount of spill code in three cases (33%). In the other cases (3 of 12), the heuristic assignments spanned one more iteration, producing the same amount of spill code in one case (33%). The overall range of the number of iterations spanned by the assignments is between two and four.

Run-time of the Algorithms

The run-time of the algorithms for the TMX320C44 results is noted in Table 8.24. This table contains the minimum, average and maximum execution times in CPU seconds of the BB-Opt, Loop-Opt and Heuristic Loop-Opt algorithms executing on a Sun 4/30 system running Unix. Included in Table 8.24, are the average number of spills per iteration for each of the register configurations previously examined with the TMX320C44. Although another heuristic is unavailable for comparison, the heuristic loop algorithm derives results that are close to the

Table 8.24: Comparison of running times for the TMX320C44.

Method	CPU Time (in seconds)			Average Number of Spills per Iteration	
	Ave.	Min.	Max.	2 Regs. (6 total)	4 Regs. (12 total)
BB-Opt	372.0	287.0	429.0	8.2	2.1
Loop-Opt	420.0	347.0	488.0	6.7	1.5
Heur. Loop-Opt	0.07	0.05	0.18	7.2	2.2

optimal, while its run-time is efficient enough to be practical.

8.8 Summary of Results

Traditionally, heuristics for optimizations have been designed based upon characteristics of the codes from a given application domain. The viewpoint taken in this thesis is that the register resources are the critical resources and heuristics for optimizations as well as the optimizations themselves should be designed with primary importance given to register requirements while honoring the characteristics of the given application domain.

In this chapter, the observed results for experimentation with the techniques discussed in this thesis are presented. Although not all techniques equally benefit all codes, as, for instance, some codes do not exhibit redundancy in data usage, significant performance improvement is observed when codes exhibit characteristics appropriate for optimization (amenable to the transformations).

The key to improved performance is the increased flexibility provided to the scheduler. When the effects of program transformation on register resources are considered during optimization, this reduces contention for those critical resources, the registers, in the schedule which, in turn, allows the code compactor to examine more options in the selection of values for those registers.

Chapter 9

Conclusions

Traditionally, techniques developed in the context of a parallelizing compiler for a parallel, load/store-type architecture have used an underlying model in which the available registers in the architecture are consolidated into one register file. In the context of embedded systems, many of those techniques are applicable, as the embedded processor in the embedded system resembles a load/store-type of architecture. However, as the available registers in an embedded processor may be partitioned and/or scattered throughout the architecture and may have specialized or restricted usages, conventional register allocation strategies must be modified to produce acceptable performance. Further, as embedded systems applications are often time-critical and are developed once, but remain in the system for its lifetime, high-quality, high-performance code is necessary.

This thesis demonstrates that beyond the specific task of register allocation, there are subtle issues related to register allocation that must be addressed in order to generate high quality code for an embedded application. The contributions of this thesis include:

- a technique which promotes data items from secondary memory to primary memory, thereby substantially reducing the amount of generated memory traffic as observed over the execution of a loop;
- a technique which guides the application of program transformations so that decisions based upon the benefit of the transformation as well as the necessary resource allocation are made globally, rather than locally, resulting

in the overall improvement of resource allocation;

- a technique which provides support for the integration of instruction scheduling and register allocation by eliminating copy instructions which are the result of "on-the-fly" register (re-)allocation during scheduling;
- a technique which optimally allocates registers over loop bodies, resulting in a minimal amount of spill code for a loop as well as minimal memory traffic related to spill code.

These techniques were developed, implemented and integrated into a parallelizing compiler which was used to conduct experimentation with those techniques on a variety of benchmarks in various application domains. The results of that experimentation is presented in this thesis and supports the importance of those techniques in generating high-performance, high-quality code.

Bibliography

[1] I. Ahmad and C. Y. R. Chen. Post-processor for Datapath Synthesis Using Multiport Memories. *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design*, pages 276–279, November 1991. San Jose, California.

[2] A. H. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley. Reading, Massachusetts, 1974.

[3] A. H. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[4] J. Backus. The History of Fortran I, II, and III. In Wexelblat, editor, *History of Programming Languages*, pages 25–45. Academic Press, 1981.

[5] J. L. Baer and D. P. Bovet. Compilation of Arithmetic Expressions for Parallel Computations. *Proc. of IFIP Congress*, pages 34–46, 1968.

[6] M. Balakrishnan et al. Allocation of Multiport Memories in Data Path Synthesis. *IEEE Transactions on the Computer Aided Design of Integrated Circuits and Systems*, 7(4):536–540, April 1988.

[7] F. Balasa, F. Catthoor, and H. De Man. Dataflow-driven Memory Allocation for Multi-dimensional Signal Processing Systems. *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design*, pages 31–34, November 1994. San Jose, California.

[8] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, Norwell, Massachusetts, 1993.

[9] J. P. Banning. An Efficient Way to Find the Side-effects of Procedure Calls and Aliases of Variables. *6th Annual ACM Symposium on Priciples of Programming Languages*, 1979.

[10] D. Bernstein, M. C. Golumbic, Y. Mansour, R. Y. Pinter, D. Q. Goldin, H. Krawczyk, and I. Nahshon. Spill Code Minimization Techniques for Optimizing Compilers. *Proceedings of SIGPLAN Conference on Programming Languages Design and Implementation*, January 1989.

[11] D. A. Berson, P. Chang, R. Gupta, and M. L. Soffa. Integrating Program Optimizations and Transformations with the Scheduling of Instruction Level Parallelism. *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, pages 207–221, August 1996. San Jose, California.

[12] R. Bodik and R. Gupta. Array Data Flow Analysis for Load-Store Optimizations in Superscalar Architectures. *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, August 1995.

[13] D. G.. Bradlee, S. J. Eggers, and R. R. Henry. Integrating Register Allocation and Instruction Scheduling for RISCs. *Proceedings of SIGPLAN Architectural Support for Programming Languages and Operating Systems*, 26(4), April 1991.

[14] D. Brelaz. New Methods to Color the Vertices of a Graph. *Communications of the ACM*, 22(4), April 1979.

[15] R. P. Brent. The Parallel Evaluation of General Arithmetic Expression. *Journal of the ACM*, 21(2), 1974.

[16] P. Briggs. *Register Coloring via Graph Coloring.* PhD thesis, Rice University, April 1992.

[17] P. Briggs, K. Cooper, K. Kennedy, and L. Torczon. Coloring Heuristics for Register Allocation. *Proceedings of SIGPLAN Conference on Programming Languages Design and Implementation*, pages 275–284, June 1989. Portland, Oregon.

[18] D. Callahan, S. Carr, and K. Kennedy. Improving Register Allocation for Subscripted Variables. *Proceedings of SIGPLAN Conference on Programming Languages Design and Implementation*, 25(6):53–65, June 1990. White Plains, New York.

[19] D. Callahan, J. Cocke, and K. Kennedy. Estimating Interlock and Improving Balance for Pipelined Architectures. *Proceedings of the International Conference on Parallel Processing*, 1987.

[20] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. *Proceedings of SIGPLAN Architectural Support for Programming Languages and Operating Systems*, 26(4):40–52, April 1991. Santa Clara, California.

[21] R. Camposano. Path-Based Scheduling for Synthesis. *IEEE Transactions on the Computer Aided Design of Integrated Circuits and Systems*, 10(1), 1991.

[22] R. Camposano and W. Wolf. *High Level VLSI Synthesis.* Kluwer Academic Publishers. Norwell, MA., 1991.

[23] G. Chaitin. Register Allocation and Spilling Via Graph Coloring. *Proc. of SIGPLAN Symp. on Comp. Const.*, 17(6), June 1982.

[24] G. Chaitin, M. Auslander, A. Chandra, J. Coocke, M. Hopkins, and P. Markstein. Register Allocation Via Coloring. *Computer Languages*, 6:47–57, January 1981.

[25] F. Chow and J. Hennessy. The Priority-Based Coloring Approach to Register Allocation. *ACM Transactions on Programming Languages and Systems*, 12(4):501–536, October 1990.

[26] J. Cocke and J. T. Schwartz. *Programming Languages and Their Compilers: Preliminary Notes, Second Revision.* Courant Institue of Mathematical Sciences, New York University, 1970.

[27] J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy Array Dataflow Analysis. *5th Symp. on Principles and Practice of Parallel Programming*, July 1995.

[28] R. Cytron and J. Ferrante. What's in a name? The value of renaming for parallelism Detection and storage allocation. *Proceedings of the International Conference on Parallel Processing*, August 1987.

[29] J. W. Davidson and S. Jinturkar. Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses. *Proceedings of SIGPLAN Conference on Programming Languages Design and Implementation*, 29(6):186–195, June 1994. Orlando, Florida.

[30] E. Duesterwald, R. Gupta, and M. Soffa. A Practical Data Flow Framework for Array Reference Analysis and its Use in Optimizations. *Proceedings of SIGPLAN Conference on Programming Languages Design and Implementation*, 28(6):68–77, June 1993. Albuquerque, New Mexico.

[31] P. Feautrier. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, February 1991.

[32] H. Feuerhahn. Data-Flow Driven Resource Allocation in a Retargetable Microcode Compiler. *International Symposium on Microarchitecture*, 1988.

[33] F. Franssen, M. van Swaaij, F. Catthoor, and H. De Man. Modeling Piece-wise Linear and Data dependent Signal Indexing for Multi-dimensional Signal Processing. *Proceedings of the International Workshop on High-Level Synthesis*, November 1992.

[34] R. A. Freiburghouse. Register Allocation Via Usage Counts. *Communications of the ACM*, 17(11), November 1974.

[35] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers. Norwell, MA., 1992.

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

[37] M. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York, NY, 1980.

[38] J. R. Goodman and W.-C. Hsu. Code Scheduling and Register Allocation in Large Basic Blocks. *International Conference on Supercomputing*, 1988.

[39] G. Goossens. *Optimization Techniques for Automated Synthesis of Application-specific Signal Processing Architectures.* PhD thesis, KU Leuven, 1989.

[40] T. Granlund and R. Kenner. Eliminating Branches using a Superoptimizer and the GNU C Compiler. *Proceedings of SIGPLAN Conference on Programming Languages Design and Implementation*, pages 341–352, July 1992. San Francisco, California.

[41] R. Gupta, M. Soffa, and T. Steele. Register Allocation via Clique Separators. *Proceedings of SIGPLAN Conference on Programming Languages Design and Implementation*, 24(7), July 1989.

[42] R. Gupta, M. L. Soffa, and D. Ombres. Efficient Register Allocation via Coloring Using Clique Separators. *ACM Transactions on Programming Languages and Systems*, 16(3), May 1994.

[43] L. J. Hendren, G. R. Gao, E. Altman, and C. Mukerji. A Register Allocation Framework Based on Heirarchical Cyclic Interval Graphs. *International Conference on Compiler Construction*, pages 176–191, April 1992. Paderborn, Germany.

[44] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, Inc. Palo Alto, Ca., 1990.

[45] L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd. Index Register Allocation. *Journal of the ACM*, 13(1):43–61, January 1966.

[46] W. Hsu, C. Fischer, and J. Goodman. On the Minimization of Loads/Stores in Local Register Allocation. *IEEE Transactions on Software Engineering*, 15(10):1252–1260, October 1989.

[47] Z. Iqbal, M. Potkonjak, S. Dey, and A. Parker. Critical Path Minimization Using Retiming and Algebraic Speed-Up. *Proceedings of the ACM/IEEE Design Automation Conference*, 1993.

[48] K. Kennedy. Index Register Allocation in Straight Line Code and Simple Loops. In R. Rustin, editor, *Design and Optimization of Compilers*, pages 51–63. Prentice-Hall, 1972.

[49] T. Kim and C. L. Liu. Utilization of Multiport Memories in Data Path Synthesis. *Proceedings of the ACM/IEEE Design Automation Conference*, pages 298–302, June 1993. Dallas, Texas.

[50] D. J. Kolson, A. Nicolau, and N. Dutt. Minimization of Memory Traffic in High-Level Synthesis. *Proceedings of the ACM/IEEE Design Automation Conference*, pages 149–154, June 1994. San Diego, California.

[51] D. J. Kolson, A. Nicolau, and N. Dutt. Elimination of Redundant Memory Traffic in High-Level Synthesis. *IEEE Transactions on the Computer Aided Design of Integrated Circuits and Systems*, pages 1354–1364, November 1996.

[52] F. J. Kurdahi and A. C. Parker. REAL: A Program for Register Allocation. *Proceedings of the ACM/IEEE Design Automation Conference*, pages 210–215, June 1987. Miami, Florida.

[53] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. *Proceedings of SIGPLAN Architectural Support for Programming Languages and Operating Systems*, 26(4):63–74, April 1991. Santa Clara, California.

[54] G. Y. Leuh, A. R. Adl-Tabatabai, and T. Gross. Global Register Allocation Based on Graph Fusion. *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, pages 246–265, August 1996. San Jose, California.

[55] C. Liem, T. May, and P. Paulin. Register Assignment through Resource Classification for ASIP Microcode Generation. *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design*, pages 397–402, November 1994. San Jose, California.

[56] J. S. Lim. *Two-Dimensional Signal and Image Processing.* Prentice Hall Signal Processing Series, 1990.

[57] P. E. R. Lippens, J. L. van Meerbergen, W. F. J. Verhaegh, and A. van der Werf. Allocation of Multiport Memories for Hierarchical Data Streams. *Proceedings of the ACM/IEEE International Conference on Computer-Aided Design*, pages 728–735, November 1993. San Jose, California.

[58] D. A. Lobo and B. M. Pangrle. Redundant Operator Creation - An Optimized Scheduling Technique. *28th DAC*, 1991.

[59] D. A. Lobo and B. M. Pangrle. Generating Pipelined Datapaths Using Reduction Techniques to Shorten Critical Paths. *European Design Automation Conference (Euro-DAC)*, 1992.

[60] F. Luccio. A Comment on Index Register Allocation. *Communications of the ACM*, 10(9):572–574, September 1967.

[61] Y. Muraoka. *Parallelism Exposure and Exploitation in Programs.* PhD thesis, University of Illinois, Urbana-Champagne, February 1971.

[62] A. Nicolau and R. Potasman. Incremental Tree Height Reduction for High-Level Synthesis. *Proceedings of the ACM/IEEE Design Automation Conference*, 1991.

[63] A. Nicolau, R. Potasman, and H. Wang. Register Allocation, Renaming and Their Impact on Fine-Grain Parallelism. *4th Int. Wksp on Lang. and Comp. for Par. Comp.*, 1991.

[64] S. Novack and A. Nicolau. Mutation Scheduling: A Unified Approach to Compiling for Fine-Grain Parallelism. *Proc. 7th Int'l Wksp on Lang. and Comp. for Par. Computing*, 1994.

[65] K. O'Brien, M. Rahmouni, and A. Jerraya. DLS: A Scheduling Algorithm for High-Level Synthesis in VHDL. *Proceedings of the European Design Automation Conference (EDAC)*, 1993.

[66] C. Park, T. Kim, and C. L. Liu. Register Allocation for Data Flow Graphs with Conditional Branches and Loops. *Proceedings of the European Design Automation Conference (Euro-DAC)*, pages 232–237, 1993. Hamburg, Germany.

[67] S. S. Pinter. Register Allocation with Instruction Scheduling: A New Approach. *Proceedings of SIGPLAN Conference on Programming Languages Design and Implementation*, 1993.

[68] P. Pöchmüller, M. Glesner, and F. Longsen. High-Level Synthesis Transformations for Programmable Architectures. *Proceedings of the European Design Automation Conference (Euro-DAC)*, 1993.

[69] A. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Rice University, 1989.

[70] R. Potasman. *Percolation-Based Compiling for Evaluation of Parallelism and Hardware Design Trade-Offs.* PhD thesis, University of California, Irvine, April 1991.

[71] R. Potasman, J. Lis, A. Nicolau, and D. Gajski. Percolation Based Synthesis. *Proceedings of the ACM/IEEE Design Automation Conference*, June 1990.

[72] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, second edition, 1992.

[73] T. A. Proebsting and C. N. Fischer. Probabilistic Register Allocation. *Proceedings of SIGPLAN Conference on Programming Languages Design and Implementation*, 27(7), July 1992.

[74] W. Pugh and D. Wonnacott. An Exact Method for Analysis of Value-based Array Data Dependencies. *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, pages 546–566, August 1993. Portland, Oregon.

[75] J.M. Rabey, C. Chu, P. Hoang, and M. Potkonjak. Fast Protoyping of Datapath-Intensive Architectures. *IEEE Design and Test of Computers*, June 1991.

[76] L. Ramachandran, D. D. Gajski, and V. Chaiyakul. An Algorithm for Array Variable Clustering. *Proceedings of the European Design Automation Conference (EDAC)*, pages 262–266, 1994.

[77] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register Allocation for Software Pipelined Loops. *Proceedings of SIGPLAN Conference on Programming Languages Design and Implementation*, 27(6), June 1992.

[78] R. M. Stallman. Using and Porting Gnu CC. *Free Software Foundation*, November 1992.

[79] L. Stok. *Architectural Synthesis and Optimization of Digital Systems.* PhD thesis, Eindhoven University of Technology, 1991.

[80] H. Trickey. Flamel: A High-Level Hardware Compiler. *IEEE Transactions on the Computer Aided Design of Integrated Circuits and Systems*, 6(2), March 1991.

[81] J. Vanhoof, K. Van Rompaey, I. Bolsens, G. Goossens, and H. De Man. *High Level Synthesis for Real Time Digital Signal Processing.* Kluwer Academic Publishers. Norwell, MA., 1993.

[82] W. F. J. Verhaegh, P. E. R. Lippens, E. H. L. Aarts, J. H. M. Korst, J. L. van Meerbergen, and A. van der Werf. Modelling Periodicity by PHIDEO Streams. *Proceedings of the International Workshop on High-Level Synthesis*, 1992.

[83] D. Whitfield and M. L. Soffa. Investigating Properties of Code Transformations. *ICPP*, 1993.