AN EFFICIENT FRAMEWORK FOR PERFORMING EXECUTION-CONSTRAINT-SENSITIVE TRANSFORMATIONS THAT INCREASE INSTRUCTION-LEVEL PARALLELISM

BY

JOHN CHRISTOPHER GYLLENHAAL

B.S., University of Arizona, 1991
M.S., University of Illinois, 1994

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1997

Urbana, Illinois

AN EFFICIENT FRAMEWORK FOR PERFORMING EXECUTION-CONSTRAINT-SENSITIVE TRANSFORMATIONS THAT INCREASE INSTRUCTION-LEVEL PARALLELISM

John Christopher Gyllenhaal, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1997
Wen-mei W. Hwu, Advisor

The increasing amount of instruction-level parallelism required to fully utilize high issue-rate processors forces the compiler to perform increasingly advanced transformations, many of which require adding extra operations in order to remove those dependences constraining performance. Although aggressive application of these transformations is necessary in order to realize the full performance potential, overly-aggressive application can negate their benefit or even degrade performance. This thesis investigates a general framework for applying these transformations at schedule time, which is typically the only time the processor's execution constraints are visible to the compiler. Feedback from the instruction scheduler is then used to aggressively and intelligently apply these transformations. This results in consistently better performance than traditional application methods because the application of transformations can now be more fully adapted to the processor's execution constraints. Techniques for optimizing the processor's machine description for efficient use by the scheduler, and for incrementally updating the dependence graph after performing each transformation, allow the utilization of scheduler feedback with relatively small compile-time overhead.

DEDICATION

To: Liesl, Mom, and Dad.

ACKNOWLEDGMENTS

Without the support of many great people, this dissertation would not have been written. I wish to thank my advisor, Professor Wen-mei Hwu, who provided valuable guidance in many technical and nontechnical areas during my graduate studies; the members of the IMPACT group, both past and present, who developed the fantastic infrastructure that was used for this research; Dr. Rick Hank, Dr. Roger Bringmann, Dr. Scott Mahlke, Dr. Dave Gallagher, Dr. Dan Lavery, Sabrina Hwu, Grant Haab, Brian Deitrich, David August, and Teresa Johnson, who over the years, graciously acted as a sounding board even when they were extremely busy with their own research; Dr. Bob Rau, Dr. Mike Schlansker, Dr. Vinod Kathail, and the rest of the CAR group at HP Labs, who provided valuable insight and discussion concerning countless machine description issues; my parents, Malcolm and Joy Gyllenhaal, who always believed in me; my wife's parents, Donald and Dorothy Little, who made me a part of their family; and my wife, Liesl, who, in addition to being the best thing that has ever happened to me, edited this entire dissertation. I deeply appreciate the help I received from each of you.

TABLE OF CONTENTS

Page

1. INTRODUCTION 1
    1.1 Overview 1

2. THE ILP TRANSFORMATIONS EVALUATED 8
    2.1 Introduction 8
    2.2 Renaming With Copy 8
    2.3 Integer Expression Reformulation 12

3. PROPOSED APPROACH FOR SCHEDULE-TIME TRANSFORMATIONS 22
    3.1 Introduction 22
    3.2 The Schedule-Based Application Algorithm 23
    3.3 The Calculation of Transformation Priorities 27
    3.4 Application of the Two ILP Transformations Evaluated 42
    3.5 Related Work 47

4. PERFORMANCE BENEFITS OF SCHEDULE-TIME TRANSFORMATIONS 49
    4.1 Introduction 49
    4.2 Criteria for Control-Block Level Comparisons 50
    4.3 The Processor Configurations 51
    4.4 The Four Application Approaches Evaluated 53
    4.5 How the Results Were Generated 56
    4.6 Overview of the Rest of This Chapter 57
    4.7 Control-Block Level Analysis of Renaming With Copy 58
        4.7.1 The four-issue processor configuration 58
        4.7.2 The two-issue processor configuration 68
        4.7.3 The one-issue processor configuration 72
        4.7.4 The eight-issue processor configuration 76
    4.8 Control-Block Level Analysis of Integer Expression Reformulation 84
        4.8.1 The four-issue processor configuration 85
    4.9 Control-Block Level Analysis of Both Transformations 95
        4.9.1 The four-issue processor configuration 95
    4.10 Overall Results 100
        4.10.1 Overall results for the four-issue processor configuration 101
        4.10.2 Overall results for the eight-issue processor configuration 109
        4.10.3 Overall results for the two-issue processor configuration 111
        4.10.4 Overall results for the one-issue processor configuration 114

5. INCREMENTAL DEPENDENCE GRAPH GENERATION 117
    5.1 Introduction 117
    5.2 The Register Action Table 118
        5.2.1 Designed to efficiently answer four time-critical queries 119

5.2.2 Handling of overlapping registers 120
5.2.3 Control, memory, and synchronization pseudoregister operands 122
5.2.4 Precalculation of the results for the four time-critical queries 124
5.2.5 Incremental update of the register action table 127
5.3 Building and Updating the Dependence Graph 132
5.3.1 Drawing outgoing destination register dependences 135
5.3.2 Drawing incoming destination register dependences 139
5.3.3 Drawing outgoing source register dependences 140
5.3.4 Drawing incoming source register dependences 141
5.3.5 Drawing control, memory, and synchronization dependences 143
5.3.6 Drawing dependences for overlapping registers 145
5.3.7 Drawing dependences for predicated code 147
5.3.8 Building the dependence graph for the entire control block 149
5.3.9 Incrementally updating the dependence graph 149
5.4 Related Work 154

6. OPTIMIZATION OF MACHINE DESCRIPTIONS FOR EFFICIENT USE 156
6.1 Introduction 156
6.2 Modeling Resource Constraints 160
6.3 A New Representation: AND/OR-Trees 165
6.4 Original MDES Characteristics 168
6.5 Eliminating Redundant or Unused MDES Information 173
6.6 Utilizing Bit-Vector Representations 176
6.7 Optimizing for Bit-Vector Representations 178
6.8 Optimizing AND/OR-Trees for Resource Conflict Detection 182
6.9 Aggregate Effect of All Transformations 184
6.10 Related Work 187

7. COMPILE-TIME IMPLICATIONS OF SCHEDULE-TIME TRANSFORMATIONS 189
7.1 Introduction 189
7.2 Compile-Time Requirements of the HP C Compiler 191
7.3 Time Requirements of DHASY List Scheduling 193
7.4 Time Breakdown for Performing Schedule-Time Transformations 196
7.5 Projection of Future Time Requirements 199

8. CONCLUSIONS AND FUTURE WORK 202

APPENDIX A. PERFORMANCE RESULTS FOR ALL PROCESSOR CONFIGURATIONS 205
REFERENCES 258
VITA 263

LIST OF TABLES

Table Page

4.1: The four processor configurations evaluated. 51
4.2: Operation latencies. 52

6.1: Option breakdown and scheduling characteristics for the four-issue processor configuration. 163
6.2: Option breakdown and scheduling characteristics for the one-issue processor configuration. 169
6.3: Option breakdown and scheduling characteristics for the two-issue processor configuration. 169
6.4: Option breakdown and scheduling characteristics for the eight-issue processor configuration. 170
6.5: Original scheduling characteristics of the machine descriptions for the four processor configurations. 171
6.6: Original MDES memory requirements for the four processor configurations. 172
6.7: MDES memory requirements after eliminating redundant and unused information. 174
6.8: The scheduling characteristics after removing unnecessary options for memory operations. 176
6.9: MDES size characteristics before and after a bit-vector representation is used (one cycle/word). 177
6.10: Scheduling characteristics before and after a bit-vector representation is used (one cycle/word). 177
6.11: MDES memory requirements before and after transforming resource usage times (one cycle per word). 181
6.12: Scheduling characteristics before and after transforming resource usage times and sorting the resulting usages to check time zero first (one cycle per word). 181
6.13: Scheduling characteristics before and after optimizing AND/OR-trees for resource conflict detection. 184
6.14: Aggregate effect of all transformations on MDES resource-constraint representation size. 185
6.15: Aggregate effect of all transformations on MDES scheduling characteristics. 186

7.1: Compile time using the HP C Compiler on the SPEC CINT92 benchmarks. 191
7.2: Compile time using the HP C Compiler on the SPEC CINT95 benchmarks. 191
7.3: Time required to list-schedule all control blocks using the DHASY scheduling heuristic. 194

7.4: A breakdown of the time required by the components of the schedule-time transformation algorithm utilizing the optimized four-issue MDES. 196

7.5: Projection of time requirements for future schedule-time transformation loads for the four-issue processor configuration... 200

LIST OF FIGURES

Figure Page

1.1: Example of the renaming-with-copy transformation. 2

1.2: Example of the integer-expression-reformulation transformation. 4

2.1: Example of the lifetime renaming transformation. 9

2.2: An example of the renaming-with-copy transformation. 10

2.3: An example of renaming with copy increasing dependence height. 11

2.4: Two examples of integer expression reformulation. 12

2.5: The effect of parallelizing expressions on code performance. 15

2.6: Example applications of integer expression reformulation. 16

2.7: Reformulation rules used for integer expression reformulation. 17

2.8: Example of making the reformulation rules assumptions true. 20

2.9: An example of more accurate information being available at schedule time. 21

3.1: Schedule-based application algorithm. 24

3.2: An example dependence graph indicating each operation's (a) early time and (b) late times. 29

3.3: Algorithm for calculating the early time for each operation in a control block. 30

3.4: Algorithm for calculating the late times for each operation in a control block. 32

3.5: Algorithm for calculating transformation priorities. 34

3.6: (a) An example control block and (b) its dependence graph. 35

3.7: Algorithm used to transform each control block. 44

4.1: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the four-issue processor configuration. 59

4.2: Change in cycles for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the four-issue processor configuration. 62

4.3: Percent benefit of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of renaming with copy for the four-issue processor configuration. 64

4.4: Benefit in cycles of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of renaming with copy for the four-issue processor configuration. 66

4.5: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the two-issue processor configuration. 69

4.6: Change in cycles for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the two-issue processor configuration. 71

4.7: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the one-issue processor configuration. 73

4.8: Change in cycles for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the one-issue processor configuration. 75

4.9: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the eight-issue processor configuration. 77

4.10: Percent benefit of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of renaming with copy for the eight-issue processor configuration. 80

4.11: Change in cycles for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the eight-issue processor configuration. 82

4.12: Benefit in cycles of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of renaming with copy for the eight-issue processor configuration. 83

4.13: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of expression reformulation for the four-issue processor configuration. 86

4.14: Percent benefit of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of expression reformulation for the four-issue processor configuration. 90

4.15: Change in cycles for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of expression reformulation for the four-issue processor configuration. 93

4.16: Benefit in cycles of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of expression reformulation for the four-issue processor configuration. 94

4.17: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of both transformations for the four-issue processor configuration. 96

4.18: Speedup due to the application of (a) renaming with copy, (b) expression reformulation, and (c) both transformations for the four-issue processor configuration. 102

4.19: Speedup due to the application of (a) renaming with copy, (b) expression reformulation, and (c) both transformations for the eight-issue processor configuration. 110

4.20: Speedup due to the application of (a) renaming with copy, (b) expression reformulation, and (c) both transformations for the two-issue processor configuration. 112

4.21: Speedup due to the application of (a) renaming with copy, (b) expression reformulation, and (c) both transformations for the one-issue processor configuration. 115

5.1: An example code segment and its register action table. 119
5.2: Example of (a) overlapping registers, (b) the resulting dependences and (c) their action entries. 122
5.3: Algorithm for building enhanced linked lists. 126
5.4: Algorithm for deleting action entry from register action table. 128
5.5: Algorithm for adding action entry to register action table. 130
5.6: Simplified algorithm for building outgoing destination register dependences. 136
5.7: (a) An example code segment and (b) the resulting action entries for Register r1 with the outgoing dependences drawn for the definition of Register r1 in Operation B. 137
5.8: Simplified algorithm for building incoming destination register dependences. 139
5.9: (a) An example code segment and (b) the resulting action entries for Register r1 with the incoming dependences drawn for the definition of Register r1 in Operation F. 140
5.10: Simplified algorithm for building outgoing source register dependences. 141
5.11: (a) An example code segment and (b) the resulting action entries for Register r1 with the only outgoing dependence drawn for the use of Register r1 in Operation D. 141
5.12: Simplified algorithm for building incoming source register dependences. 142
5.13: (a) An example code segment and (b) the resulting action entries for Register r1 with the only incoming dependence drawn for the use of Register r1 in Operation D. 142
5.14: Simplified algorithm for building outgoing source memory dependences. 144
5.15: (a) An example code segment and (b) the resulting memory action entries with the only outgoing memory dependence drawn for Operation B. 144
5.16: Enhanced algorithm for building outgoing source register dependences that supports overlapping registers. 145
5.17: (a) Example of overlapping register and (b) the register anti dependences drawn for the use of r2f in Operation B. 146
5.18: Enhanced algorithm for building outgoing source register dependences that supports predicated execution. 148
5.19: Simplified algorithm for deleting a source register action entry and incrementally updating the dependence graph. 150
5.20: Simplified algorithm for adding a source register action entry and incrementally updating the dependence graph. 150
5.21: Simplified algorithm for deleting a destination register action entry and incrementally updating the dependence graph. 151

5.22: Dependences for Register r1 (a) with the original actions, (b) after Action C is deleted, and (c) after rebuilding dependences for adjacent definitions (Actions A and E). 152
5.23: Simplified algorithm for adding a destination register action entry and incrementally updating the dependence graph. 153
5.24: Dependences for Register r1 (a) with the original actions, (b) after Action C is added, and (c) after rebuilding dependences for adjacent definitions (Actions A and E). 153

6.1: The twenty-four reservation tables that represent the resources used by the four-issue processor configuration's integer literal load operation. 161
6.2: Distribution of options checked during each scheduling attempt using the four-issue processor configuration. 164
6.3: Two methods of modeling the resource constraints of the four-issue processor configuration's integer literal load operation. (a) The traditional OR-Tree representation. (b) The proposed AND/OR-Tree representation........ 166
6.4: An example of how the AND/OR-tree representation can facilitate the sharing of OR-trees. 175
6.5: The OR-tree modeling the resource constraints for the four-issue processor configuration's integer literal load operation, after transforming the resource usage times in order to better utilize the bit-vector representation. 180
6.6: An example of optimizing the order of the OR-trees in an AND/OR-tree for resource conflict detection. (a) Original order specified. (b) After optimizing the order. 183

A.1: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the one-issue processor configuration. 206
A.2: Percent benefit of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of renaming with copy for the one-issue processor configuration. 207
A.3: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of expression reformulation for the one-issue processor configuration. 208
A.4: Percent benefit of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of expression reformulation for the one-issue processor configuration. 209
A.5: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of both transformations for the one-issue processor configuration. 210
A.6: Percent benefit of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of both transformations for the one-issue processor configuration. 211

A.7: Change in cycles for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the one-issue processor configuration. 212
A.8: Benefit in cycles of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of renaming with copy for the one-issue processor configuration. 213
A.9: Change in cycles for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of expression reformulation for the one-issue processor configuration. 214
A.10: Benefit in cycles of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of expression reformulation for the one-issue processor configuration. 215
A.11: Change in cycles for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of both transformations for the one-issue processor configuration. 216
A.12: Benefit in cycles of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of both transformations for the one-issue processor configuration. 217
A.13: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the two-issue processor configuration. 218
A.14: Percent benefit of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of renaming with copy for the two-issue processor configuration. 219
A.15: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of expression reformulation for the two-issue processor configuration. 220
A.16: Percent benefit of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of expression reformulation for the two-issue processor configuration. 221
A.17: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of both transformations for the two-issue processor configuration. 222
A.18: Percent benefit of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of both transformations for the two-issue processor configuration. 223
A.19: Change in cycles for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the two-issue processor configuration. 224
A.20: Benefit in cycles of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of renaming with copy for the two-issue processor configuration. 225
A.21: Change in cycles for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of expression reformulation for the two-issue processor configuration. 226

A.22: Benefit in cycles of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of expression reformulation for the two-issue processor configuration. 227
A.23: Change in cycles for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of both transformations for the two-issue processor configuration. 228
A.24: Benefit in cycles of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of both transformations for the two-issue processor configuration. 229
A.25: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the four-issue processor configuration..230
A.26: Percent benefit of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of renaming with copy for the four-issue processor configuration. 231
A.27: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of expression reformulation for the four-issue processor configuration. 232
A.28: Percent benefit of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of expression reformulation for the four-issue processor configuration..................................233
A.29: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of both transformations for the four-issue processor configuration..234
A.30: Percent benefit of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of both transformations for the four-issue processor configuration...............................235
A.31: Change in cycles for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the four-issue processor configuration. 236
A.32: Benefit in cycles of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of renaming with copy for the four-issue processor configuration. 237
A.33: Change in cycles for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of expression reformulation for the four-issue processor configuration..................................238
A.34: Benefit in cycles of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of expression reformulation for the four-issue processor configuration..................................239
A.35: Change in cycles for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of both transformations for the four-issue processor configuration...............................240
A.36: Benefit in cycles of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of both transformations for the four-issue processor configuration...............................241

A.37: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the eight-issue processor configuration. 242
A.38: Percent benefit of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of renaming with copy for the eight-issue processor configuration. 243
A.39: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of expression reformulation for the eight-issue processor configuration. 244
A.40: Percent benefit of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of expression reformulation for the eight-issue processor configuration. 245
A.41: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of both transformations for the eight-issue processor configuration. 246
A.42: Percent benefit of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of both transformations for the eight-issue processor configuration. 247
A.43: Change in cycles for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the eight-issue processor configuration. 248
A.44: Benefit in cycles of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of renaming with copy for the eight-issue processor configuration. 249
A.45: Change in cycles for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of expression reformulation for the eight-issue processor configuration. 250
A.46: Benefit in cycles of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of expression reformulation for the eight-issue processor configuration. 251
A.47: Change in cycles for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of both transformations for the eight-issue processor configuration. 252
A.48: Benefit in cycles of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of both transformations for the eight-issue processor configuration. 253
A.49: Speedup due to the application of (a) renaming with copy, (b) expression reformulation, and (c) both transformations for the one-issue processor configuration. 254
A.50: Speedup due to the application of (a) renaming with copy, (b) expression reformulation, and (c) both transformations for the two-issue processor configuration. 255
A.51: Speedup due to the application of (a) renaming with copy, (b) expression reformulation, and (c) both transformations for the four-issue processor configuration. 256

A.52: Speedup due to the application of (a) renaming with copy, (b) expression reformulation, and (c) both transformations for the eight-issue processor configuration. 257

1. INTRODUCTION

1.1 Overview

To achieve the best performance from today's processors, compilers must aggressively minimize the dynamic operation count of programs through both traditional optimizations [1] and efficient use of the instruction set [2], [3]. The innate instruction-level parallelism (ILP) of the code, coupled with the use of a few ILP-enhancing transformations that minimally increase dynamic operation count, is usually sufficient to allow effective use of the limited resources in these processors. However, to get the best performance out of future resource-rich processors, aggressive ILP transformations are needed [4]. These transformations attempt to break dependences that limit ILP, often utilizing special hardware such as predicated execution or speculative execution support, and have been shown to be quite effective for future processors [5], [6], [7]. The drawback to these transformations is that they can dramatically increase dynamic instruction count, sometimes by more than a factor of two [8], [9].

For processors with moderate resources, applying these aggressive ILP transformations is currently difficult because of the large performance degradation that can occur when processor resources become overutilized. To illustrate the issues involved, a simple yet effective ILP transformation called *renaming with copy* [10] is applied to the code segment shown in Figure 1.1(a). This code segment is taken from a loop in the SPECcint92 benchmark 085.gcc that searches an array for non-NULL pointers in order to do some processing on them. The code segment is shown scheduled for a one-issue processor (indicated by the single issue slot in the figure), where the cycle in which

Cycle	Issue Slot 1
1	r1<- M[r2+0]
2	
3	bne r1, 0, exit
4	r1<- M[r2+4]
5	
6	bne r1, 0, exit

(a) Original code

Cycle	Issue Slot 1
1	r1<- M[r2+0]
2	t1<- M[r2+4]
3	bne r1, 0, exit
4	r1<- t1
5	bne t1, 0, exit

(b) After renaming with copy

Cycle	Issue Slot 1
1	r1<- M[r2+0]
2	(unavailable)
3	bne r1, 0, exit
4	t1<- M[r2+4]
5	
6	r1<- t1
7	bne t1, 0, exit

(c) Resources overutilized

Figure 1.1: Example of the renaming-with-copy transformation.

each operation is scheduled is shown in the left column. Each load takes two cycles and, as shown in Figure 1.1(a), the total scheduled code length for this segment is six cycles. The highlighted load in Cycle 4 is prevented from executing in Cycle 2 because the Register r1 is used by the branch in Cycle 3 and is redefined by the load. The load’s destination register cannot be simply renamed because the exit code expects the non-NULL pointer to be in r1.

However, the load’s destination register can be renamed, say to t1, if a copy of t1 to r1 is inserted after the load. If this renaming-with-copy transformation is done and the renamed load can be moved into Cycle 2, then the total scheduled code length for this segment can be reduced to five cycles, as shown in Figure 1.1(b). In this case, a 25% increase in dynamic operation count has resulted in a 17% performance improvement for this code segment. However, if after applying the transformation the renamed load could not be moved into Cycle 2 because of a resource constraint, then performance would instead be degraded by 17% as shown in Figure 1.1(c). In this second case, performance is lost and resources are wasted by the addition of the extra copy operation. For resource-rich processors, this second case rarely occurs and so the transformation is often applied liberally. For moderate-resource processors, the second case occurs frequently enough that care must be taken in its application to prevent overall performance loss.

Interest in this area has recently become intense because most next-generation processors fall within this class of moderate-resource processors. Several performance projections using code compiled with the traditional approach of minimizing dynamic operations has shown poor processor utilization on these processors. For example, on a recently announced four-issue SPARC64, this compilation model has been projected to yield, ignoring cache effects, only an average of 1.52 instructions per cycle (IPC) on SPECcint92 code [11]. Although several methods of estimating resource constraints have been proposed that could be adapted to help guide application of ILP transformations [4], [12], [13], it is intrinsically more desirable to be able to use exact resource and schedule layout information. This thesis proposes a new machine-description-driven transformation framework that makes such exact information available by allowing ILP transformations to be efficiently performed and evaluated during instruction scheduling.

The primary mechanism used to evaluate each potential ILP transformation is the effect the transformation has on the scheduled height of the code segment being processed. The *scheduled height* is a common first-order execution-time estimate that takes into account the processor’s resource constraints, the various data and control dependences between operations, and the algorithm used for scheduling [14]. It can be quickly calculated after instruction scheduling using profile information and the cycle in which each operation was scheduled. Although it ignores second-order effects caused by branch prediction and the memory hierarchy, it is one of the best static estimates of

Cycle	Issue Slot 1	Issue Slot 2
1	r1<- r5∧r6	
2	r2<- r1+r7	
3	bgt r2, r8, exit	r3<- r2+r9
4	r4<- r3∧r10	

(a) Original code

Cycle	Issue Slot 1	Issue Slot 2
1	r1<- r5∧r6	t1<- r7 + r9
2	r2<- r1+r7	r3<- r1+t1
3	bgt r2, r8, exit	r4<- r3∧r10

(b) After reformulation

Figure 1.2: Example of the integer-expression-reformulation transformation.

performance available and has been used extensively to show the benefit of ILP transformations [5], [15]. This evaluation mechanism facilitates the use of search algorithms to find the set of transformations that produces a heuristically minimum estimated execution time for each code segment.

In order to evaluate the benefit of the proposed framework, two important ILP transformations were implemented using this framework. Both of these transformations were already being extensively used within the IMPACT compiler [16]. However, despite extensive tuning of their application heuristics, they were found to cause performance degradation in some code segments and miss important opportunities in other code segments. The first ILP transformation, *renaming with copy*, breaks anti and output dependences on destination registers as illustrated earlier in Figure 1.1. The second ILP transformation, *integer expression reformulation*, is a form of critical path reduction [17] that reformulates integer arithmetic expressions in order to improve the performance of a code segment. In Figure 1.2(a), the value in r3 is the result of a summation of r1, r7, and r9 and is used by the calculation of r4 in Cycle 4. The value of r1 is not ready until Cycle 2, which delays the calculation of r3 in its current formulation. However, if the calculation of r3 is changed so that r1 is added last instead of first, the code segment's performance can be improved 25% as shown in Figure 1.2(b). Although sometimes existing operations only need to be modified by this transformation, usually one or more operations need to be created in order to perform this transformation. As with renaming with copy, if these extra operations cause the processor resources to be overutilized, degraded performance can result.

Using these ILP transformations, this thesis examines in detail the performance benefits gained from using the proposed framework's exact schedule information compared to what is currently done, and to what may be possible if approximations were used instead. In order to make fair comparisons between using exact schedule information, using approximations, and using the IMPACT compiler's existing heuristics to guide the application of the ILP transformations, the proposed framework is used for all of the experiments. For the experiments involving approximations, the accuracy of the machine description used to guide the transformations is varied with respect to the processor targeted. This allows existing approaches that utilize only dependence graph information [5] or approximations based on simplified processor models [4] to be evaluated. For the experiments involving the IMPACT compiler's existing heuristics, the transformations' application heuristics were faithfully ported to the new framework from IMPACT's current ILP optimizer, which is typically invoked well before scheduling. These experiments show that only by utilizing the proposed framework's exact schedule information can a compiler achieve consistent performance improvement from the application of ILP transformations.

The first major contribution of this thesis is the proposed schedule-based application framework that facilitates the aggressive and intelligent application of execution-constraint-sensitive transformations. The proposed linear-time transformation selection algorithm yields both significantly and consistently better performance than traditional application methods because the application of transformations can now be more fully adapted to the processor's execution constraints.

A critical issue that this thesis examines in detail is the compile-time cost of using the proposed scheduled-height-based search algorithm. For each combination of ILP transformations considered, the transformed code segment must be rescheduled in order to recalculate the scheduled height. This may result in a code segment being repeatedly scheduled, with a different set of ILP transformations applied each time. If a conventional scheduling framework is used, the compile-time cost may not be acceptable.

The second major contribution of this thesis is a set of novel techniques that drastically reduce the time required to reschedule the code segment after a set of ILP transformations has been performed. The proposed algorithm for incremental dependence graph generation allows near-constant-time updates of the dependence graph after performing each transformation. The proposed machine description optimizations generate an efficient low-level representation of the processor’s complex execution constraints from a high-level machine description language. The compile-time benefits of these techniques are extensively analyzed using a highly-tuned, state-of-the-art implementation of the proposed framework.

Chapter 2 describes in detail the renaming-with-copy and integer-expression-reformulation transformations, which will be used to evaluate the proposed approach for applying transformations. Chapter 3 describes the algorithms proposed for applying transformations at schedule time and Chapter 4 compares the benchmark performance achieved using this approach to several other possible approaches. Chapter 5 describes an algorithm for incremental dependence graph generation that significantly reduces the time required to perform schedule-time transformations. Chapter 6 describes how resource constraints are typically modeled and presents a set of novel techniques that drastically reduce the time required to perform both scheduling and schedule-time transformations. Chapter 7 evaluates the effectiveness of the above techniques in reducing the compile time and estimates the compile-time cost if an extensive set of transformations is applied at schedule-time. Chapter 8 summarizes the results of this research and describes some interesting areas for future application.

2. THE ILP TRANSFORMATIONS EVALUATED

2.1 Introduction

This thesis examines the challenges involved in applying ILP transformations to processors with moderate resources and proposes an new framework for applying these transformations during instruction scheduling. The qualitative and quantitative benefits of applying transformations during scheduling will be analyzed using the two ILP transformations, integer expression reformulation and renaming with copy, that were briefly introduced in Chapter 1. These two transformations were selected for several reasons. The first is that the challenges involved in applying them is representative of a wide array of ILP transformations, such as memory speculation, branch combining, and promotion with copy. The second is that both of these transformations had been previously implemented in the IMPACT compiler, allowing a direct comparison to an existing application technology as well as to approaches based on approximate information. Third, in addition to a more effective application of these transformations directly improving performance, the application of these transformations also should increase the benefit of many other ILP transformations.

2.2 Renaming With Copy

Register anti dependences occur between the redefinition of a register's value and the uses of the register's previous value. For example in Figure 2.1(a), a register anti dependence exists between the use of Register r1 in Cycle 3 and the redefinition of Register r1 in Cycle 3. This dependence prevents the reordering of these two operations

Cycle	Issue Slot 1	Issue Slot 2
1	r1 <- M[r9+0]	
2		
3	r2 <- r1 + r7	r1 <- M[r8+0]
4		
5	r3 <- r1 + r6	

(a) Original code

Cycle	Issue Slot 1	Issue Slot 2
1	t4 <- M[r9+0]	r1 <- M[r8+0]
2		
3	r2 <- t4 + r7	r3 <- r1 + r6

(b) After renaming (no operations added)

Figure 2.1: Example of the lifetime renaming transformation.

in order to prevent the use of r1 in Cycle 3 from receiving an incorrect value. Similarly, *register output dependences* occur between the redefinition of a register's value and the definition of the register's previous value. For example in Figure 2.1(a), a register output dependence exists between the definition of Register r1 in Cycle 1 and the redefinition of Register r1 in Cycle 3. This dependence prevents the reordering of these two definitions in order to prevent the use of r1 in Cycle 5 from receiving an incorrect value. Typically, as in this example, anti dependences are more constraining than output dependences. However both types of dependences can limit instruction-level parallelism by preventing the overlap of computation.

Register renaming refers to a class of transformations that break register anti and output dependences by renaming (changing) a virtual register used by a set of operations [5]. A widely used approach, referred to as *lifetime renaming*, identifies all of the disjoint register lifetimes within a function and assigns each of these disjoint lifetimes a new virtual register. A *register lifetime* is a set of definitions and uses of a particular register where the uses only consume the values produced by definitions in the set and no uses outside the set consume the values produced by these definitions. A register lifetime can be divided up into one or more *disjoint register lifetimes* where each nonoverlapping

Cycle	Issue Slot 1	Issue Slot 2
1	r1<- M[r2+0]	
2		
3	bne r1, 0, exit	r1<- M[r2+4]
4		
5	bne r1, 0, exit	
exit: 1	r4<- M[r1+0]	...

(a) Original code

Cycle	Issue Slot 1	Issue Slot 2
1	r1<- M[r2+0]	t3<- M[r2+4]
2		
3	bne r1, 0, exit	r1 <- t3
4	bne t3, 0, exit	
exit: 1	r4<- M[r1+0]	...

(b) After renaming with copy

Figure 2.2: An example of the renaming-with-copy transformation.

lifetime is the minimal set of definitions and uses required to form a valid register lifetime. An example of a disjoint register lifetime for Register r1 in Figure 2.1(a) is the definition of r1 in Cycle 1 and the use of r1 in Cycle 3. Assigning this disjoint register lifetime a new virtual register (Register t4) breaks the register and output dependences with the redefinition of r1 in Cycle 3, allowing more computation to be overlapped and improving performance 40% as shown in Figure 2.1(b). Because lifetime renaming always operates on disjoint register lifetimes, no fixup code (extra operations) is needed in order to perform the transformation. As one of the few ILP transformations that does not increase the program's dynamic operation count, this transformation is almost universally applied in production and research compilers.

However, register anti and output dependences can also occur between operations within the same lifetime. For example, in Figure 2.2(a), the definition of Register r1 in Cycle 1 and the definition of Register r1 in Cycle 3 reside in the same register lifetime because they both reach the same use of r1 in code labeled 'exit.' As a result, the register anti dependence between the use of r1 in Cycle 3 and the redefinition of r1 in Cycle 3 cannot be removed with lifetime renaming and another register naming transformation

	Cycle	Issue Slot 1	Issue Slot 2
loop:	1	r2 <- r1	r1 <- M[r1 + 8]
	2	beq r2, r3, exit	
	3	bne r1, 0, loop	

(a) Original code

	Cycle	Issue Slot 1	Issue Slot 2
loop:	1	t4 <- M[r1 + 8]	r2 <- r1
	2		
	3	r1 <-t4	beq r2, r3, exit
	4	bne t4, 0, loop	

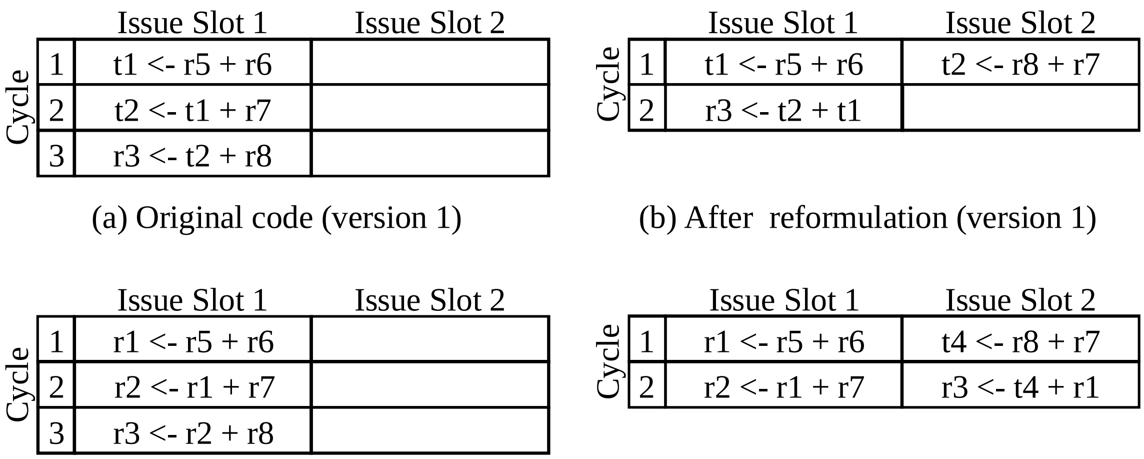
(b) After renaming with copy

Figure 2.3: An example of renaming with copy increasing dependence height.

must be used. Note that in all this thesis' figures, branch operations (e.g., bne r1, 0, exit in Figure 2.2(a)) are assumed to not have any delay slots and operations scheduled after the branch operation (e.g., r1 <- M[r2+4]) only execute if the branch is not taken.

The renaming transformation utilized by the IMPACT compiler when lifetime renaming is not sufficient is *renaming with copy* [5]. This transformation splits a lifetime into two disjoint lifetimes, renames one of them and fixes up the other with a copy (move) operation. Applying this transformation to Figure 2.2(a) yields the code shown Figure 2.2(b). The redefinition of Register r1 in Cycle 3 and use of r1 in Cycle 5 is first renamed to a new virtual Register t3, and then an operation is added to copy the value of t3 to r1 so that the code at 'exit' always gets the proper value for r1. Applying this transformation to this code segment allows more computation to be overlapped, improving performance by 20%.

Unlike the lifetime renaming transformation, renaming with copy increases the program's dynamic operation count. As shown earlier in Figure 1.1(c), care must be taken to prevent overutilization of processor resources. Moreover, the additional copy operation can actually increase the dependence height of a code segment, degrading performance even if enough resources are available as shown in Figure 2.3. In Figure 2.3(a), the register anti dependence between the use of Register r1 and redefinition



Cycle	Issue Slot 1	Issue Slot 2
1	t1 <- r5 + r6	
2	t2 <- t1 + r7	
3	r3 <- t2 + r8	

(a) Original code (version 1)

Cycle	Issue Slot 1	Issue Slot 2
1	t1 <- r5 + r6	t2 <- r8 + r7
2	r3 <- t2 + t1	

(b) After reformulation (version 1)

Cycle	Issue Slot 1	Issue Slot 2
1	r1 <- r5 + r6	
2	r2 <- r1 + r7	
3	r3 <- r2 + r8	

(c) Original code (version 2)

Cycle	Issue Slot 1	Issue Slot 2
1	r1 <- r5 + r6	t4 <- r8 + r7
2	r2 <- r1 + r7	r3 <- t4 + r1

(d) After reformulation (version 2)

Figure 2.4: Two examples of integer expression reformulation.

of r1 in Cycle 1 can be removed by performing renaming with copy, allowing these operations to reorder as shown in Figure 2.3(b). However, the copy operation inserted by the transformation ends up delaying the first branch until Cycle 1 so that the proper value of Register r1 reaches that branch's target. Upon close examination of several benchmarks, this situation was found to occur primarily when this transformation was applied to operations that both define and use the same register, as the definition of r1 does in Figure 2.3(a). For this reason, the IMPACT compiler's existing heuristics prevents application in this case and thus would not apply renaming with copy to the code in Figure 2.3(a). There are, however, situations where applying this transformation to this case does increase performance, making a better mechanism for applying this transformation important.

2.3 Integer Expression Reformulation

Register flow dependences occur between the definition of a register's value and the subsequent uses of that register's value. For example, in Figure 2.4(a), a register flow dependence exists between the definition of Register t1 in Cycle 1 and the use of Register t1 in Cycle 2. Flow dependences usually specify a nonzero delay that indicates that minimum number of cycles that must separate the two operations in order to prevent a hardware interlock. In all of this chapter's examples, each register flow dependence delay will simply be the defining operation's latency, where integer additions and subtractions have a one-cycle latency, and integer load operations have a two-cycle latency. The code segment in Figure 2.4(a) therefore has a dependence height of three cycles. Even with infinite processor resources, this code segment as shown must be scheduled over three cycles. Thus something must be done to reduce the dependence height if better performance is to be achieved on wider-issue processors. It is important to note that in current processors, address generation interlocks, incomplete bypass, fast execution paths, etc., can make a register flow dependence's delay different from the defining operation's latency [2], [18] and these situations can be modeled by the IMPACT compiler's machine description language [19], [20].

The second ILP transformation, *integer expression reformulation*, is a form of critical path reduction [17] that reduces the dependence height along the critical path by reformulating integer arithmetic expressions in acyclic code segments. An example of integer expression reformulation has already been shown in Figure 1.2 and two more examples are shown in Figure 2.4. In all this thesis's figures, including Figure 1.2 and Figure 2.4, the virtual register identifications that start with an 'r' indicate that the register's value is assumed to be used elsewhere, in another code segment not shown. The virtual register identifications that start with a 't' indicate that the virtual register is used only in the code segment shown. This distinction is important when reformulating the code shown in Figure 2.4(a), because it indicates that the values written into virtual Registers t1 and t2 may be safely changed as long as the correct value is still written into virtual Register r3. This allows the summation of Registers r5, r6, r7, and r8 to be reformulated without requiring any extra operations to be added, as shown in Figure 2.4(b). However, if all the virtual registers are used elsewhere, as shown in Figure 2.4(c), the values in these virtual registers may not be changed. This reformulation transformation still can be applied, but an additional operation is required, as shown in Figure 2.4(d). As with any transformation that adds operations, care must be taken to prevent overutilization of processor resources.

This integer-expression-reformulation transformation utilizes the communicative and associative properties of arithmetic to reformulate integer expressions in order to improve the code segment’s performance. Baer and Bovet proposed an earlier algorithm that uses these same properties to generate the most parallel form possible of source-level floating-point expressions [21]. Unfortunately, just parallelizing expressions is not the best choice for critical path reduction and may even degrade performance of the code segment. For example, Figure 2.5(a) shows the same expression as Figure 2.4(a) except that the value of Register r5 is now being loaded from memory. Parallelizing the expression yields the code segment shown in Figure 2.5(b), which is an improvement over the formulation in Figure 2.5(a), but higher-performance formulations are possible. In fact, if the code segment originally has the higher-performance formulation shown in Figure 2.5(c), parallelizing the expression degrades performance, as shown in Figure 2.5(d). Thus, the expression-reformulation algorithm must take into account when each operand in the expression becomes available.

Cycle	Issue Slot 1	Issue Slot 2
1	r5 <- M[r9+0]	
2		
3	t1 <- r5 + r6	
4	t2 <- t1 + r7	
5	r4 <- t2 + r8	

(a) Original code (version 1)

Cycle	Issue Slot 1	Issue Slot 2
1	r5 <- M[r9+0]	t2 <- r7 + r8
2		
3	t1 <- r5 + r6	
4	r4 <- t1 + t2	

(b) After parallelization (version 1)

Cycle	Issue Slot 1	Issue Slot 2
1	r5 <- M[r9+0]	t1 <- r8 + r6
2	t2 <- t1 + r7	
3	r4 <- t2 + r5	

(c) Original code (version 2)

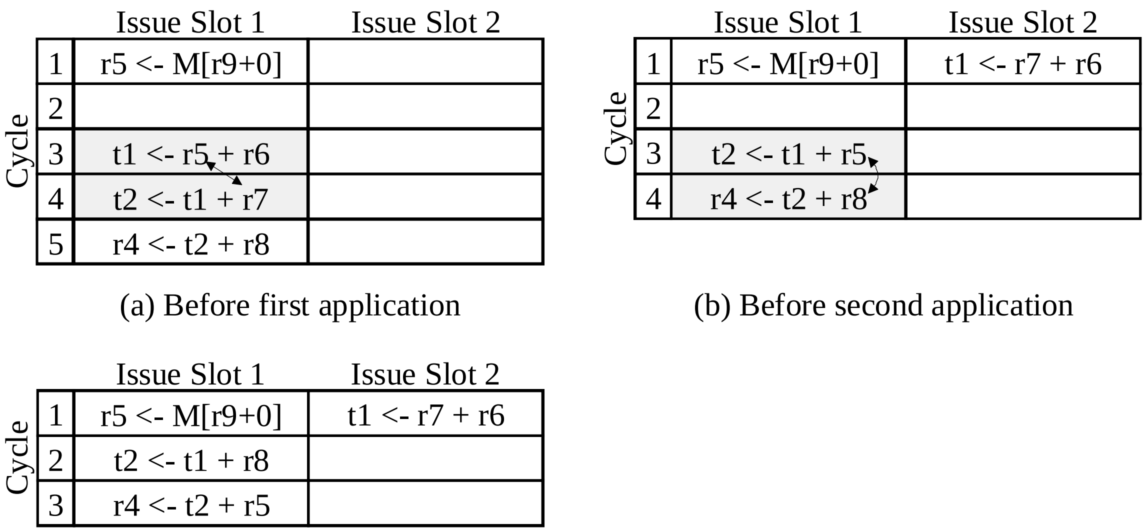
Cycle	Issue Slot 1	Issue Slot 2
1	r5 <- M[r9+0]	t1 <- r8 + r6
2		
3	t2 <- r7 + r5	
4	r4 <- t1 + t2	

(d) After parallelization (version 2)

Figure 2.5: The effect of parallelizing expressions on code performance.

The integer-expression-reformulation algorithm proposed in this thesis is applied to two operations at a time, where the first operation defines a register used by the second operation. The algorithm attempts to reformulate the two selected operations so that both operations can be executed earlier. For example, applying this algorithm to the two highlighted operations in Figure 2.6(a) causes Registers r5 and r7 to be swapped, allowing the first operation to execute two cycles earlier and the second operation to execute one cycle earlier as shown in Figure 2.6(b). The code can then be further improved by applying this algorithm to the two highlighted operations in Figure 2.6(b), causing Registers r5 and r8 to be swapped as shown in Figure 2.6(c). Note that no extra operations were required because the values of Registers t1 and t2, by definition, are not used elsewhere.

One advantage of reformulating just two operations at a time is that the algorithm is straightforward to implement and fully test because there are only a finite number of



Cycle	Issue Slot 1	Issue Slot 2
1	r5 <- M[r9+0]	
2		
3	t1 <- r5 + r6	
4	t2 <- t1 + r7	
5	r4 <- t2 + r8	

(a) Before first application

Cycle	Issue Slot 1	Issue Slot 2
1	r5 <- M[r9+0]	t1 <- r7 + r6
2		
3	t2 <- t1 + r5	
4	r4 <- t2 + r8	

(b) Before second application

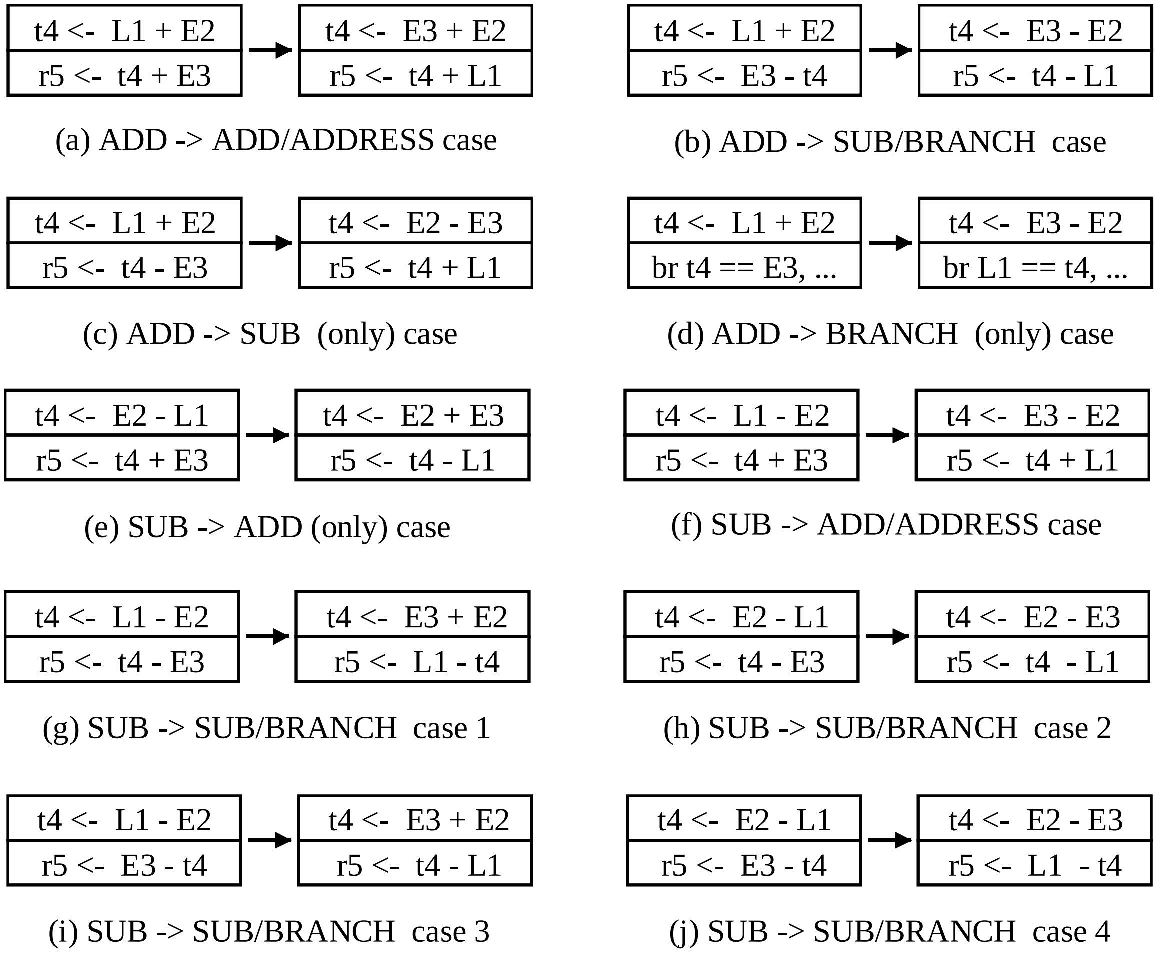
Cycle	Issue Slot 1	Issue Slot 2
1	r5 <- M[r9+0]	t1 <- r7 + r6
2	t2 <- t1 + r8	
3	r4 <- t2 + r5	

(c) After both applications

Figure 2.6: Example applications of integer expression reformulation.

operand/operation type combinations possible (roughly 360 in this thesis's implementation). Another advantage of processing just two operations at a time is that it allows much finer control of the transformation's application. Because each reformulation of two operations can require the addition of one or more extra operations, this fine control allows more conservative application of expression reformulation when resources are limited.

The rules used to reformulate operation pairs are shown in Figure 2.7, where Register L1 is assumed to be available later than Operands E2 and E3 (registers and/or integer constants). These reformulation rules specify how the two operations must be changed in order to move use of Register L1 from the first operation to the second operation. Moving Register L1 to the second operation should allow both operations to execute earlier, assuming execution resources are available. For example, both applications of this transformation shown in Figure 2.6 utilize the transformation rule shown in Figure 2.7(a). The conditions under which this rule applies are summarized by



```
t4 <-  L1 + E2    ->    t4 <-  E3 + E2
r5 <-  t4 + E3          r5 <-  t4 + L1
```

(a) ADD -> ADD/ADDRESS case

```
t4 <-  L1 + E2    ->    t4 <-  E3 - E2
r5 <-  E3 - t4          r5 <-  t4 - L1
```

(b) ADD -> SUB/BRANCH case

```
t4 <-  L1 + E2    ->    t4 <-  E2 - E3
r5 <-  t4 - E3          r5 <-  t4 + L1
```

(c) ADD -> SUB (only) case

```
t4 <-  L1 + E2    ->    t4 <-  E3 - E2
br t4 == E3, ...        br L1 == t4, ...
```

(d) ADD -> BRANCH (only) case

```
t4 <-  E2 - L1    ->    t4 <-  E2 + E3
r5 <-  t4 + E3          r5 <-  t4 - L1
```

(e) SUB -> ADD (only) case

```
t4 <-  L1 - E2    ->    t4 <-  E3 - E2
r5 <-  t4 + E3          r5 <-  t4 + L1
```

(f) SUB -> ADD/ADDRESS case

```
t4 <-  L1 - E2    ->    t4 <-  E3 + E2
r5 <-  t4 - E3          r5 <-  L1 - t4
```

(g) SUB -> SUB/BRANCH case 1

```
t4 <-  E2 - L1    ->    t4 <-  E2 - E3
r5 <-  t4 - E3          r5 <-  t4  - L1
```

(h) SUB -> SUB/BRANCH case 2

```
t4 <-  L1 - E2    ->    t4 <-  E3 + E2
r5 <-  E3 - t4          r5 <-  t4 - L1
```

(i) SUB -> SUB/BRANCH case 3

```
t4 <-  E2 - L1    ->    t4 <-  E2 - E3
r5 <-  E3 - t4          r5 <-  L1  - t4
```

(j) SUB -> SUB/BRANCH case 4

Figure 2.7: Reformulation rules used for integer expression reformulation.

the label “ADD -> ADD/ADDRESS.” This label indicates that this rule handles the case where an add operation produces a result that is used by another add operation or in an address calculation for a load or store operation.

For the case discussed in the previous paragraph, only the operation’s operands need to be swapped. However, for many of the rules shown in Figure 2.7, an operation must be changed from an add to a subtract operation, or vice versa, in order to perform this transformation. For example, the rule shown in Figure 2.7(b) changes the first operation from an add operation to a subtract operation. This rule handles the case where an add operation produces a result that is used by a subtract operation or to determine the branch direction (which conceptually involves subtracting the two operands being compared), where the branch is either a branch equals or a branch not equals operation. (The conditions under which this rule applies are summarized by the "ADD -> SUB/BRANCH label.") In order to achieve the desired result of both L1 and E2 being subtracted from E3, changing the first operation is necessary.

It also is sometimes necessary to change the sense of the second operation, which can be done for subtract operations (Figure 2.7(c)), branch operations (Figure 2.7(d)), and add operations (Figure 2.7e), but not for address calculations, because current processors do not support base minus offset addressing. Note the sense of branch operations can be changed only because the order of the branch operands does not affect the outcome for branch equals and branch not equals operations. The rules that cover the remaining add and subtract combinations are shown in Figure 2.7(f) - Figure 2.7(j). There are no rules for reformulating integer multiplication and division combinations because the associative and communicative properties do not hold for integer division operations. Although it is possible to reformulate expressions involving only integer multiplication operations or logic operations (AND, OR, etc.), in the SPECcint92 and SPECcint95 benchmarks there appear to be very few expressions that would benefit from these types of reformulations. As a result, these logic expressions were not reformulated.

Although the rules shown in Figure 2.7 don't show any operations being added, reformulating the two operations may require the addition of up to two new operations. The first assumption made by the rules shown in Figure 2.7 is that the register defined by the first operation is only used by the second operation so that its value may be safely changed. If this is not naturally the case, a copy of the first operation must be created with a new destination register that is used only by the second operation before the transformation can be performed. This was the reason an additional operation was required in order to transform the code segments shown in Figure 1.2(a) and Figure 2.4(c).

The second assumption that the rules shown in Figure 2.7 make is that Register L1 is not redefined before the second operation and that if Operand E3 is a register, E3 is defined before the first operation. This second assumption allows these operands to be swapped between the two operations without affecting their values. If these assumptions do not initially hold, the operations in the code segment must be reordered, dependences permitting, in order to make this assumption true. In a few important cases, register anti and output dependences prevent an assumption-satisfying ordering from being found and renaming with copy is used to break these dependences. For example, the reformulation rules cannot be directly applied to the two highlighted operations in Figure 2.8(a) because neither assumption holds. The destination of the first operation, Register r1, is used elsewhere (in a code segment not shown) and the first register source of the first operation, Register r1, is redefined before the second operation. The first assumption can be made true by making a copy of the first operation (highlighted) and giving it a new destination register as shown in Figure 2.8(b). Although the second assumption can usually be made true by simply reordering the operations, in Figure 2.8(b) register anti dependences prevent any reordering from occurring. However, if renaming with copy is applied to the load operation defining Register r2, this load operation can then be moved above the redefinition of Register r1, as shown in

Cycle	Issue Slot 1	Issue Slot 2
1	r1 <- M[r9+8]	
2		
3	beq r1, r8, exit1	r1 <- r1 + r2
4	r2 <- M[r7+r1]	
5		
6	bne r2, r6, exit2	

(a) Neither assumption holds

Serial Order
. . .
t3 <- r1 + r2
r1 <- r1 + r2
r2 <- M[r7+t3]
. . .

(b) First assumption made true

Serial Order
. . .
t3 <- r1 + r2
t4 <- M[r7+t3]
r1 <- r1 + r2
r2 <- t4
. . .

(c) Second assumption made true

Cycle	Issue Slot 1	Issue Slot 2
1	r1 <- M[r9+8]	t3 <- r7 + r2
2		
3	beq r1, r8, exit	t4 <- M[r1+t3]
4	r1 <- r1 + r2	r2 <- t4
5	bne t4, r6, exit2	

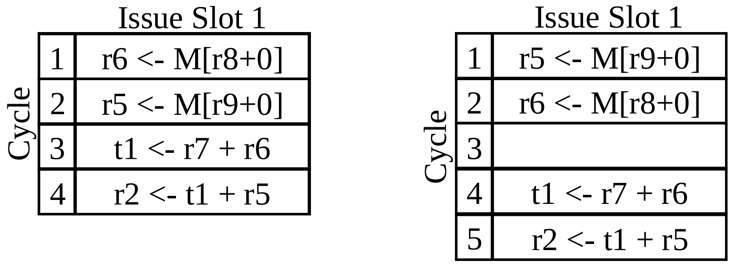
(d) Expression reformulated

Figure 2.8: Example of making the reformulation rules assumptions true.

Figure 2.8(c). Both assumptions now hold and the expression can be reformulated as shown in Figure 2.8(d).

The situations where renaming with copy is needed to make the second assumption true usually involve either a loop variable or a pointer that is incremented with each iteration of the loop. In addition, in loops this situation only occurs for the few cases where the IMPACT compiler's loop optimizer cannot or chooses not to break all the register anti and output dependences of loop variables with other transformations. This case occurs, however, in a few important code segments in the SPEC benchmarks 026.compress, 023.eqntott, and 008.espresso.

One of the advantages of applying transformations such as integer expression reformulation at schedule time is that exact scheduling information can be utilized instead of relying on approximate schedule information (such as dependence height) when



Cycle	Issue Slot 1
1	r6 <- M[r8+0]
2	r5 <- M[r9+0]
3	t1 <- r7 + r6
4	r2 <- t1 + r5

(a) Transformed based on a schedule approximation

Cycle	Issue Slot 1
1	r5 <- M[r9+0]
2	r6 <- M[r8+0]
3	
4	t1 <- r7 + r6
5	r2 <- t1 + r5

(b) The actual schedule

Figure 2.9: An example of more accurate information being available at schedule time.

performing the transformation. For example, if only approximate scheduling information is available and this information indicates that the definition of r6 will occur before the definition of r5, the calculation of r2 will be reformulated to use r5 later as shown in Figure 2.9(a). However, if in the actual schedule the definition of r5 occurs first, as shown in Figure 2.9(b), then this reformulation based on approximate scheduling information would end up degrading performance even though no extra operations were added in order to perform the reformulation. By applying transformations at schedule time (utilizing the algorithms described in Chapter 3) the transformations can adapt to all of the scheduling decisions made during code scheduling, even those that are less than optimal. In addition, schedule-time transformations can also take advantage of opportunities created by these scheduling decisions, such as actual ordering of two load operations that could execute in parallel if there were enough processor resources. It is extremely difficult to estimate schedule information accurately in such cases because the actual ordering often depends on the fine details of the scheduling heuristic used and the processor resources available. By applying transformations at schedule time, information that is much more accurate becomes available.

3. PROPOSED APPROACH FOR SCHEDULE-TIME TRANSFORMATIONS

3.1 Introduction

The overall goal of the proposed transformation framework is to improve performance through the intelligent application of ILP transformations at schedule time. One advantage of applying these transformations at schedule time is that an extremely good first-order approximation of a control block's execution time, the schedule height, can be efficiently computed after the control block is scheduled. The *schedule height* of a control block is defined to be the average number of cycles required to execute the control block per entry as specified by the control block's schedule and profile information. For example, if profile information indicates the top control block shown in Figure 2.2(a) is entered 100 times, and the branch scheduled in Cycle 3 exits the control block 10 times, the schedule height of this control block would be ((10 * 3) + (90 * 5)) / 100 = 4.8 cycles. After applying renaming with copy to the top control block shown in Figure 2.2(b), the schedule height is reduced to ((10 * 3) + (90 * 4)) / 100 = 3.9 cycles. In addition to taking into account the processor's execution constraints, the schedule height also reflects the decisions made by the instruction scheduler. This allows extremely accurate predictions of actual performance to be made. Note that if profile information indicates the control block was not executed, then some reasonable assumption needs to be made (e.g., all branches are equally likely) in order to use the schedule height calculation to guide the application of transformations. This issue is not addressed in this thesis and therefore transformations will only be applied to executed control blocks using the proposed application approach.

3.2 The Schedule-Based Application Algorithm

The algorithm for the proposed schedule-based application approach is shown in Figure 3.1. The parameters passed to this algorithm are the control block to be transformed, the initial schedule height of the control block before transformation, and a set of flags indicating what types of transformations to perform. The algorithm returns the transformed control block (indirectly) and the schedule height of the control block after the transformations have been applied. Note that the initial schedule height and the transformed schedule height are used to prevent unnecessary schedule height calculations (which in turn may require rescheduling the control block).

In order to avoid an exhaustive search, a linear-time greedy search algorithm [22] is utilized. Each transformation is performed one at a time, in priority order, and the effect of the individual transformation is then evaluated. If the control block's schedule height is improved, the transformation is kept. If the schedule height is degraded, the transformation is undone and marked so that it will not be tried again. It is for those cases where the schedule height is unchanged, however, that the interesting trade-offs must be made.

Performing a transformation that does not change the schedule height may consume processor resources that may be better utilized by performing another transformation. If this transformation is undone, the transformation may turn out to be beneficial after another transformation is performed. For the two transformations implemented, benchmark performance is significantly increased if transformations are

```
schedule_based_application (control_block, initial_height, allowed_trans)
{
    best_height = initial_height;                 // Initialize best height to initial height

    // Perform an initial scan of the control block for transformation opportunities
    trans_queue = find_potential_trans (control_block, allowed_trans);

    // Calculate the initial transformation priorities, will update after each transformation
    calculate_trans_priorities (control_block, trans_queue);

    // Test each transformation opportunity in priority order, until all transformations
    // have been tested.  Only examine transformations allowed by 'allowed_trans'.
    while ((trans = select_highest_priority_trans (trans_queue)) != NULL)
    {
        unschedule (control_block);                          // Unschedule control block
        perform_transformation (control_block, trans);       // Apply transformation

        schedule (control_block);                            // Reschedule control block
        new_height = calc_schedule_height (control_block);// Measure the effect

        // Keep if transformation reduces schedule height or
        //        if transformation doesn't change the schedule height and the
        //        transformed operation is not scheduled later than before transformation.
        if ((new_height < best_height) ||
            ((new_height == best_height) && (transformed op not scheduled later)))
        {
            best_height = new_height;          // Keep transformation, update best height

            // Performing trans may expose new opportunities and/or remove others
            update_trans_queue (trans_queue, trans, allowed_trans);

            // Recalculate trans priorities based on new dependence graph
            calculate_trans_priorities (control_block, trans_queue);
        }
        else
        {
            unschedule (control_block);                      // Unschedule control block
            undo_transformation (control_block, trans);      // Return to previous state
        }
    }
    return (best_height);        // Return the new schedule height of control block
}
```

Figure 3.1: Schedule-based application algorithm.

performed even when they don't change the schedule height. This is because a large number of the control blocks require multiple transformations to be applied before the schedule height is improved, independent of the order the transformations are applied. Therefore, if transformations are being tested individually (as in Figure 3.1), transformations should be kept as long as they do not directly degrade performance.

If the algorithm shown in Figure 3.1 simply keeps all transformations that do not degrade the schedule height, almost all of the improved performance would be captured. However, there are a few cases where the schedule height is not affected but the targeted operation is delayed. The *targeted operation* is defined to be the operation that most directly benefits from the performed transformation (i.e., the operation directly constrained by the dependence being transformed). For these cases, if this targeted operation is delayed by performing this transformation, it is unlikely to be beneficial and the algorithm shown in Figure 3.1 causes the transformation to be undone. Otherwise, the transformation could be beneficial and this algorithm therefore keeps the transformation.

It should be noted that the proposed algorithm for selecting transformations to apply is clearly only one of several possible solutions. This researcher implemented and tested a wide array of both linear and binary search algorithms, including several that performed multiple transformations at a time, several that tried each transformation multiple times, and one that exhaustively tried all combinations for up to the first 20 highest-priority transformation opportunities in a control block (1,048,576 combinations). These experiments were performed in an early prototype implementation of the proposed transformation framework and focused on the renaming-with-copy transformation and the SPECcint92 benchmarks. These experiments found that the proposed linear-time search algorithm presented in Figure 3.1 captured all the performance improvement of the exhaustive search in all but a few of the control blocks. Until more transformations are implemented in this schedule-time transformation framework, it is difficult to judge whether or not the proposed linear-time search will continue to do as well or if a more expensive algorithm will be needed. However, for both renaming with copy and integer expression reformulation, the proposed search algorithm has produced near-optimal results.

In addition, this researcher was unable to find a search algorithm that captured almost all the possible performance improvement, and required less compile time, than the proposed linear-time search algorithm. The primary difficulty is that it is nearly impossible to separate the benefit of one transformation from the degradation caused by another transformation if they are both performed at the same time. As a result, search algorithms that evaluate the benefit of more than one transformation at a time can apply clearly degrading transformations and may not apply clearly beneficial transformations, if an unfortunate grouping of transformations occurs during this evaluation. This researcher could not find an intuitive way to prevent such unfortunate groupings from occurring. As a result, in order to capture almost all the possible performance, an algorithm that tests each transformation individually at least once appears to be necessary.

Note that the algorithm presented in Figure 3.1 unschedules the control block before each transformation is performed (or undone) and reschedules the control block after performing the transformation so that the new schedule height can be calculated. This unscheduling and rescheduling of the control block for each transformation evaluated represents most of the compile-time cost of this algorithm. In Section 4.4 two alternatives to rescheduling the control block and making decisions based on changes in schedule height will be described. However, as the results in Chapter 4 will show, these alternatives do not capture all the performance of the proposed approach, so the ability to efficiently reschedule a control block is required. Chapters 5 and 6 will present novel techniques to significantly reduce the compile time required to reschedule control blocks and therefore to significantly reduce the time required to perform this proposed algorithm.

It should be noted that unscheduling a control block is a necessary first step in rescheduling a control block, so its cost should be considered part of the overall rescheduling cost. This unscheduling step is separated out in the proposed algorithm because sometimes scheduling reorders operations in such a way as to prevent some useful transformations from being performed. The actual implementation does allow the transformation of partially scheduled or fully scheduled control blocks, but this ability is not required and is not beneficial for the proposed schedule-time application algorithm.

3.3 The Calculation of Transformation Priorities

The benefit of the proposed schedule-time application algorithm shown in Figure 3.1 is highly dependent on the transformation priority functions utilized. These priority functions determine the order in which the transformations are evaluated (highest priority first). Ideally, these priority functions would give the highest priorities to the most beneficial transformations and the lowest priorities to the least beneficial or the detrimental transformations. Unfortunately, this information is not easily obtained so heuristics must be used to calculate these priorities. The priority functions utilized in this thesis estimate the benefit of a transformation (and thus its priority) by analyzing its effect on the control block's dependence graph.

A control block's *dependence graph* is a compiler representation of all the dependences between operations in the control block. (Please see Sections 2.2 and 2.3 for a description of register anti, register output, and register flow dependences.) In addition to constraining the order of operations in a control block, each dependence also specifies a delay. The dependence's *delay* specifies the minimum number of cycles that must separate the execution of the two operations connected by the dependence. An example dependence graph for a control block is shown in Figure 3.2(a). In this figure, each circle represents an operation, and the shaded Operations C and H are control block exits (branches). The lines between operations represent a dependence and the number next to the line represents the dependence's delay. For example, the dependence between Operation A and Operation B has a delay of one cycle, indicating that Operation B must execute at least one cycle after Operation A. The first operation (i.e., Operation A) will be referred to as the dependence's *source operation* and the second operation (i.e., Operation B) will be referred to as the dependence's *destination operation*. If a dependence has a zero delay (such as the dependence between Operation E and Operation C), the operations may execute in the same cycle as long as the source operation executes before the destination operation. In addition, it is also useful to refer to the *dependences into* an operation. Operation C has two dependences into it, one from Operation B and the other from Operation E.

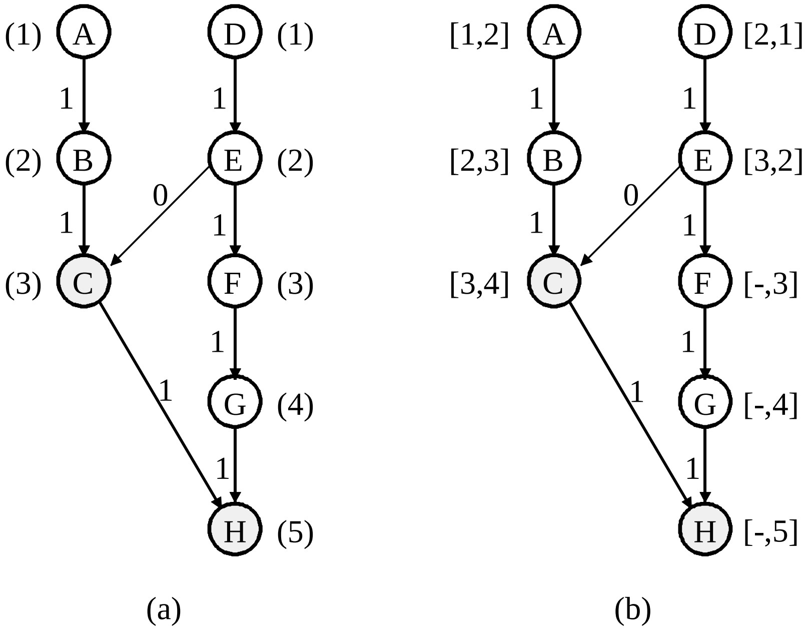


Figure 3.2: An example dependence graph indicating each operation's (a) early time and (b) late times.

One of the useful metrics that can be calculated from the dependence graph is the early time of each operation. An operation's *early time* is the first cycle that the operation could be scheduled in because of dependence constraints, assuming that there are no resource constraints. The early time of each operation is shown in Figure 3.2(a) in parentheses. For example, Operation C's early time is calculated to be cycle 3. The algorithm for determining the early time of every operation in a control block is shown in Figure 3.3. If an operation has no dependences into it (e.g., Operations A and D), its early time defaults to cycle 1. Otherwise, an operation's early time is calculated by taking the maximum of the constraints imposed by the dependences into that operation. The constraint from a dependence is then calculated by adding the early time of the dependence's source operation to the dependence's delay. For example, the constraint caused by dependence between Operations B and C in Figure 3.2(a) is calculated to be

```
calculate_early_times (control_block)
{
    for (operation in control_block)    // Process operations in serial order.
    {
        early_time = 1;                 // If operation is not constrained, its early time is 1.

        // For each dependence constraining the operation, update early time.
        for (dep_in in operation's dependence_into list)
        {
            // Given the early time of the dependence's source operation and
            // the delay of the dependence, calculate the constraint on the early time.
            constraint = dep_in->source_operation->early_time + dep_in->delay;

            if (early_time < constraint)        // Update early time if necessary.
                early_time = constraint;
        }

        operation->early_time = early_time;   // Update operation's early time.
    }
}
```

Figure 3.3: Algorithm for calculating the early time for each operation in a control block.

2 + 1 = cycle 3. For Operation C, the maximum of constraints (3 and 2) then is used to set the early time of this operation to cycle 3.

Another useful metric that can be calculated from a dependence graph is a set of late times for each operation, one for each exit from the control block. (Branch operations are commonly referred to as exits because they can transfer control out of the control block.) An operation's *late time* for a particular exit is the latest cycle that operation can be scheduled in without delaying the exit. Each late-time calculation is based on dependence constraints only; resource constraints are not considered. The late times for each operation are shown in square brackets in Figure 3.2(b). The first number is the late time relative to Operation C (the first exit) and the second number is relative to Operation H (the second exit). For example, the late time for Operation B relative to exit

Operation C is cycle 2 and relative to exit Operation H is cycle 3. The late time for an operation is not defined for an exit if the operation does not need to execute before the exit. For example, Operations F, G, and H do not have to execute before exit Operation C, so a dash is shown for the late time.

The algorithm for calculating the late times for each operation is shown in Figure 3.4. This algorithm assumes there is a single well-defined operation for each exit (no fall-through path) and that each exit is assigned an array index starting from 0. Handling control blocks with a fall-through path is a simple extension of this algorithm (not shown). (Essentially a zero-latency dummy operation can be added to the end of the control block and its late time set to the maximum of all the operation's early times.) For each operation, an integer array is used to store the late time relative to each exit operation. Undefined late times (represented as a dash in the example) are represented by the number 2,000,000,000 in this algorithm. The late times relative to each exit operation, specified by the index variable, are calculated independently.

First, the late times for all operations are initialized to undefined for this index. The next step sets the exit operation's late time to its early time (e.g., the late time for exit Operation C, index 0, is set to cycle 3). The last step propagates the late time constraint backward through the control block. This is done by scanning each dependence into each operation that has a late time specified and updating the late time (if necessary) of the dependence's source operation. For example, the dependence into Operation C from Operation B imposes a constraint that Operation B's late time be at most cycle 2 (for index 0). This constraint is calculated by taking Operation C's late time (cycle 3) and subtracting the dependence's delay (1 cycle). Operation C also constrains the late time of

```
calculate_late_times (control_block)
{
    // Each exit from the control block is numbered.  Calculate the late time relative
    // to each exit individually (late_time is an array of late times)
    for (index=0; index < number_of_exits; index++)
    {
        // Initialize all late times for this index with the special value 2,000,000,000.
        // This value indicates that the late time is currently undefined.
        for (operation in control_block)
            operation->late_time[index] = 2,000,000,000;

        // Set the late time for the exit operation to the early time of the exit operation.
        exit_operation[index]->late_time[index] = exit_operation[index]->early_time;

        // Propagate the late time constraints from this exit operation to all operation
        // before the exit operation in the control block (for this index only).  These
        // operations are processed in reverse serial order.
        for (operation = exit_operation[index]; operation != NULL;
             operation = operation->previous_operation_in_control_block)
        {
            // If this operation's late time is 2,000,000,000, the exit operation can be
            // executed before this operation is executed so there are no constraints
            // to propagate.
            if (operation->late_time[index] == 2,000,000,000)
                continue;              // Goto previous operation in control block

            // For each dependence constraining the operation, update the late time
            // of the dep_in's source operation (if necessary).
            for (dep_in in operation's dependence_into list)
            {
                // To make this operation's late time, the source operation's late
                // time must be delay cycles earlier.
                constraint = operation->late_time[index] - dep_in->delay;

                // Update the source operation's late time, if necessary.
                if (dep_in->source_operation->late_time[index] > constraint)
                        dep_in->source_operation->late_time[index] = constraint;
        }
    }
}
```

Figure 3.4: Algorithm for calculating the late times for each operation in a control block.

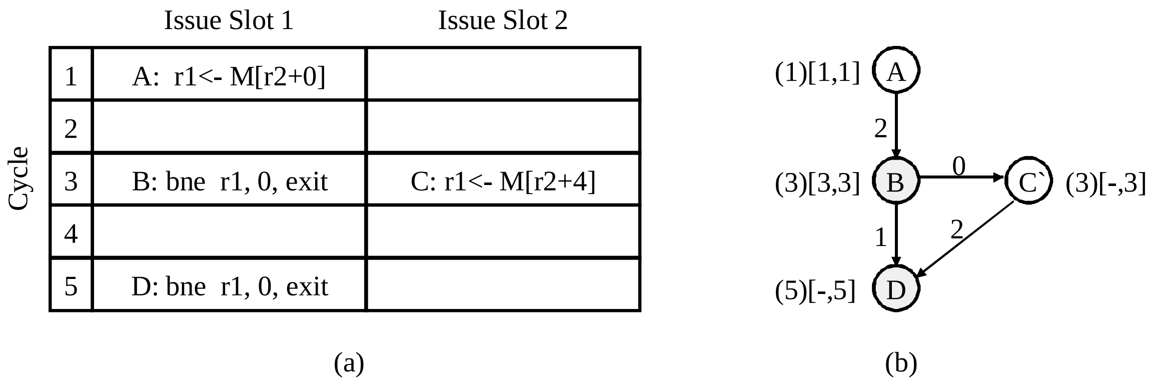
Operation E to cycle 3. After all the constraints from Operation C are propagated, the algorithm then propagates the constraints of the previous operation in the control block. This continues until the constraints of all the operations in the control block before the exit operation have been propagated. Note that the late times for the same operation but for different indexes (exits) can be quite different (they are different for every operation in this example). This happens because operations often constrain the execution of one exit more than another exit.

The algorithm used for calculating each transformation opportunity's priority is shown in Figure 3.5. This priority algorithm was experimentally found to yield the best performance for the four-issue processor configuration described in Section 4.3, which will be discussed in more detail after the algorithm is explained.

In order to facilitate this explanation, the renaming with copy example is revisited in Figure 3.6(a). There is only one transformation opportunity in this control block, the application of renaming with copy to Operation C. This priority algorithm's parameters are the control block being transformed (e.g., Figure 3.6(a)) and the transformation opportunity "trans" that the priority should be calculated for (e.g., renaming with copy on Operation C). This algorithm assumes that the early time and late times for each operation in the control block have been previously calculated. The early and late times for this example control block are shown in Figure 3.6(b).

```
calculate_trans_priority (control_block, trans)
{
    // The targeted operation should benefit most directly from this transformation.
    targeted_op = trans->targeted_operation;

    // Calculate the direct benefit of applying this transformation in terms of the
    // change in the targeted operation’s early time.
    direct_benefit = calculate_change_in_early_time (control_block, trans, targeted_op);

    // Initialize priority to zero, add in priority relative to each exit operation.
    priority = 0.0;
    for (index=0; index < number_of_exits; index++)
    {
        // If exit does not depend on targeted operation, do not increase priority.
        if (targeted_op->late_time[index] == 2,000,000,000)
            continue;           // Skip the rest of the computation for this index.

        // The dependence height before and after target_operation can be quickly
        // calculated using the target_operation’s early and late time.
        before_height = targeted_op->early_time - 1;
        after_height = (exit[index]->early_time  - targeted_op->late_time[index]) + 1;

        // Each transformation uses a different priority calculation.
        if (trans->type == renaming_with_copy)
            exit_priority = 10000.0 * ((min(before_height, after_height) * 10000.0) +
                                       ((before_height + after_height) * 100.0) +
                                        direct_benefit);

        else if (trans->type == integer_expression_reformulation)
            exit_priority = 1.0 * ((after_height * 10000.0) +
                                   ((before_height + after_height) * 100.0) +
                                    direct_benefit);

        // Weight exit priority based on importance of exit (profile based)
        exit_weight = exit_frequency[index] / control_block->entry_frequency;
        if (exit_weight < 0.00001) exit_weight = 0.00001;
        priority = priority + (exit_priority * exit_weight);
    }

    // If transformation not expected to be beneficial, significantly reduce priority.
    if (direct_benefit <= 0)
        priority = priority * 0.001;

    return (priority);
}
```

Figure 3.5: Algorithm for calculating transformation priorities.



Cycle	Issue Slot 1	Issue Slot 2
1	A: r1<- M[r2+0]	
2		
3	B: bne r1, 0, exit	C: r1<- M[r2+4]
4		
5	D: bne r1, 0, exit	

(a)

(b)

Figure 3.6: (a) An example control block and (b) its dependence graph.

The algorithm first determines the operation targeted by the transformation (e.g., Operation C). The algorithm then calculates how much the early time for this targeted operation will change after the transformation is performed. This can be efficiently calculated by scanning the dependences into the targeted operation, ignoring the dependence(s) that will be removed by the transformation. For this example, the early time for Operation C will change from cycle 3 to cycle 1 after renaming with copy is performed, yielding a two cycle change. This change is used to approximate the direct benefit to the targeted operation from this transformation. This is an approximation because resource constraints may prevent this benefit from being realized.

The priority algorithm then calculates the priority contribution from each exit to this transformation's priority. If the exit does not require the targeted operation to be executed (the late time is 2,000,000,000), then that exit makes no contribution to the transformation's priority. Otherwise, the exit's contribution is calculated using the following two statistics. First the position of the targeted operation in the longest dependence chain that goes through both the targeted operation and the exit is calculated. The targeted operation's early time indicates how much of this dependence chain is before the targeted operation (before_height). Note that one cycle is subtracted from this early time to compensate for early times starting at cycle 1. The amount of dependence chain after the targeted operation is calculated by subtracting the targeted operation's late time for this exit from the early time of the exit (after_height). Note that one extra cycle is added to this difference in order to guarantee after_height will be greater than zero. The priority calculations (described below) were found to give undesirable priorities when after_height is zero. For the calculation of exit Operation D's priority contribution, the before_height of Operation C is calculated to be (3 - 1) = 2 cycles and the after_height is calculated to be (5 - 3) + 1 = 3 cycles.

The priority calculation for the transformation then utilizes three ordering heuristics and some scaling constants to determine the transformation's priority. This priority is in effect a sorting key that is used to determine the order transformations will be evaluated in. As such, the magnitude of the priorities are unimportant as long as the proper order of transformations occurs. This property allows several ordering heuristics to be folded into one floating-point number for ease of use. In addition, scaling constants have been chosen for each of these heuristics in order to create a clear hierarchy of importance for determining the order of transformations. The rationale behind each of these ordering heuristics and the choice of scaling constants will now be explained.

The first ordering heuristic, the *direct-benefit metric,* utilizes the direct benefit of the transformation (direct_benefit) in terms of change in early time for the targeted operation. When the direct-benefit metric is zero (or less), performing this transformation is usually not beneficial. When the direct-benefit metric is greater than zero, performing this transformation is significantly more likely to be beneficial. However, this metric does not discriminate between transformations that target the critical path and those that do not. Because in general targeting the critical path is more likely to yield overall benefit, basing the priority on only this metric does not yield good results. However, when combined with other heuristics (described below), it works well when all other considerations are equal. In addition, this is the only metric that can be used to discern between beneficial and unbeneficial transformations. After the transformation's priority using all the ordering heuristics is calculated, this metric is used to scale the priority so transformations more likely to be beneficial are evaluated first. This is implemented by multiplying the priority by 0.001 if direct_benefit is zero (or less); otherwise the priority is not changed.

The second ordering heuristic, the *chain-height metric*, is the dependence height of the targeted dependence chain. The *targeted dependence chain* is the longest chain of dependences that goes through both the targeted operation and the exit operation (before_height + after_height). The larger the dependence height of the targeted dependence chain is, the more likely the targeted dependence chain is constraining performance. Transformations that target these longer dependence chains are therefore more likely to improve overall performance. However, this heuristic does not distinguish between transformations that target the same dependence chain (which frequently occurs in unrolled loops). For transformations like renaming with copy that can (best case) break a dependence chain into two independence pieces, it more desirable to break the dependence chain in the center (which maximizes benefit) than near one end of the dependence chain. The direct_benefit heuristic does not cause this desirable ordering either, so another ordering heuristic is needed.

The next ordering heuristic, the *position metric*, is based on the placement of the targeted operation in the targeted dependence chain. For renaming with copy, the closer the targeted operation is to the center of the targeted dependence chain, the more the dependence chain's height will be reduced if the chain is split into two independence pieces. The distance from the closest edge of the dependence chain, which can be easily calculated by taking the minimum of before_height and after_height, indicates how close to the dependence chain's center the targeted operation is. This minimum is therefore used as the position metric for renaming with copy. For integer expression reformulation, no matter where the targeted operation is, the benefit will be only one cycle (the latency of an integer add or subtract). However, the farther the targeted operation is from the bottom of the targeted dependence chain, the more operations there are below the transformation that potentially could benefit from the reduced dependence height. The distance from the bottom of the targeted dependence chain is simply the after_height metric, which is used as the position metric for integer expression reformulation.

The final combination of ordering heuristics for each transformation was experimentally derived from extensive evaluation of various combinations of the three ordering heuristics described above. Because each of the ordering heuristics is unlikely to have a value larger than 100, each heuristic was multiplied by powers of 100 (1, 100, 10,000) and combined into one floating-point number for easy sorting of transformations. Combinations involving subsets of these three ordering heuristics were also evaluated. In addition, because the transformations do not have the same cost-benefit ratios, various scaling factors were evaluated on the overall priority values based on transformation type.

The aggregate priority value for each exit (exit_priority), as shown in the algorithm, first orders the transformations by the position metric, then by the chain-height metric, and finally by the direct-benefit metric. For renaming-with-copy transformations, this aggregate priority value is then multiplied by 10000.0 so that renaming-with-copy transformations will always have higher priority than integer-expression-reformulation transformations, even if the renaming-with-copy transformation has no direct benefit. (The rationale for this scaling will be explained in Section 3.4.) The final priority value for the transformation (priority) is then a summation of the aggregate priority value for each exit, weighted by the importance of the exit (exit_weight). The importance of an exit is estimated by dividing the number of times the exit is taken by the number of times the control block is executed, as specified by the profile information. To help break ties in a way that helps exits that (according to the profile information) are not taken, exit_weight is forced to have a minimum value of 0.00001. It should be noted that this approach for combining priorities for each exit into one final priority value is identical to that used by the DHASY scheduling heuristic [23]. It is an intuitively good way to combine multiple priorities together.

Note that if there are two or more transformations with the same final priority, the ‘select_highest_priority_trans’ function (called in the transformation selection algorithm shown in Figure 3.1) will pick the transformation that targeted the operation that appeared earliest in the unscheduled control block. (If further tie-breaking is needed, the renaming-with-copy transformation will be chosen first.) Also, whenever a transformation is performed (and kept), the control block is examined for new transformation opportunities that might have been created and the priorities for existing transformation opportunities are recalculated based on the new dependence graph (as shown in Figure 3.1). This ensures that new opportunities are not missed and that the transformation priorities always reflect the current state of the control block. The metric ordering and scaling factors used in the priority calculation shown in Figure 3.5 were chosen because they were found to yield the best performance over the benchmarks at a control-block level (independent of the importance of each control block).

The ranking of the position metric and chain-height metric did not significantly affect the performance of control blocks limited by dependence constraints. Both metrics tend to target the longest dependence chains first (which is desirable) and in these control blocks that do not saturate processor resources, any extra transformations performed usually can be absorbed. However, ranking the position metric first usually yielded higher performance when resource constraints limited the performance of portions of a control block. In order to improve performance in this situation, the transformations applied must facilitate the moving of operations from the portions of the schedule with resource contention into the portions of the schedule that have more processor resources available. Transforming shorter dependence chains so that they can be moved out of the way of the longer dependence chains is sometimes the only way to improve performance. In addition, there tend to be fewer processor resources available, so the order of evaluation tends to be much more important. With the position metric ranked first, well-positioned (and hopefully more promising) transformations targeting shorter dependence chains are evaluated before poorly positioned transformations targeting the longest dependence chain. This was found experimentally to yield good performance for these resource constrained control blocks. If the chain-height metric is ranked first, every transformation targeting the longest dependence chain is evaluated before those targeting even slightly shorter dependence chains. Because of the strategy of keeping transformations as long as they don’t hurt performance, this ranking often allows ineffective transformations to the longest dependence chain to consume all the available resources before the transformations targeting the shorter dependence chains are evaluated. As a result, ranking the chain-height metric first often does not yield as high performance as ranking the position metric first does.

It should be noted that the transformation’s priority is based solely on dependence information. This researcher found that calculating priorities based on scheduling information (e.g., when each operation was scheduled and which dependences were constraining performance in the schedule) tended to generate less optimal priorities. One difficulty with using scheduling information is that the delays caused by resource constraints can mask the importance of some truly constraining dependences. In addition, the priority heuristics must now second-guess the scheduling heuristics. It is difficult to determine whether an operation was delayed because of dependence and resource constraints or because the scheduling heuristics determined that the operation was unimportant. It is also unclear whether an operation was scheduled early because the scheduling heuristics determined the operation was critical or because there were extra resources available so a noncritical operation might as well be scheduled. Although dependence information does not typically specify all the constraints on performance, it is significantly easier to interpret than schedule information. In addition, aggressive ILP scheduling heuristics are typically based on dependence information and profile information. Utilizing this same information allows the transformation algorithms to target those operations the scheduler is likely to consider critical.

3.4 Application of the Two ILP Transformations Evaluated

Traditionally, compilers apply each type of transformation (i.e., renaming with copy and integer expression reformulation) in separate passes over the code. Each of these passes are commonly referred to as a phase. Because applying a transformation can both expose and remove opportunities for future transformations, the order in which the transformations are applied can significantly affect performance. Even with careful tuning of the order, some beneficial transformation opportunities are occasionally missed, thereby reducing performance, which is commonly referred to as the *phase-ordering problem*. The proposed schedule-based application approach (Figure 3.1) allows several different types of transformations to be considered for application at the same time. This approach has the potential to prevent many of the phase-ordering problems encountered with traditional approaches because the application order of the transformations adapts to the transformation opportunities present in the code via the priority functions.

In order to evaluate this potential benefit for the two implemented transformations, each transformation's priority was scaled to allow various degrees of phase-ordering and different relative priorities to be evaluated. However, the experiments clearly showed that for these two transformations, a strict phase-ordering should be used where "free" integer-expression-reformulation transformations are applied first, then renaming-with-copy transformations are applied, and then finally expression-reformulation transformations that require the addition of extra operations are performed.

This is the best way to apply these transformations because the transformations have significantly different cost-benefit ratios. Integer-expression-reformulation transformations that add no extra operations are “free” and as such have a zero cost-benefit ratio. Renaming with copy has a low cost-benefit ratio because a single transformation can halve the height of a dependence chain and the extra operation added can usually be scheduled late in the control block where processor resources are more likely to be available. Integer expression reformulations that require the addition of extra operations have a significantly higher cost-benefit ratio because a single transformation can only reduce dependence height by one cycle, and if an extra operation must be added to make the first reformulation assumption true (see discussion of Figure 2.8) this extra operation must be scheduled early in the control block where processor resources are less likely to be available.

This strict phase-ordering for these transformations was initially implemented by scaling the priority values for the transformations. An artifact of this scaling can be seen in the exit_priority calculations shown in Figure 3.5 where the exit_priority for renaming with copy is scaled (multiplied) by a factor of 10,000.0, whereas the priority for integer expression reformulation is scaled by a factor of 1.0. This particular scaling causes a total phase-ordering where renaming-with-copy transformations are always evaluated first. (Note that this initial scheme did not give “free” integer expression reformulations highest priority.)

This initial priority-based approach then evolved into the approach shown in Figure 3.7, which explicitly performs each of the transformations in individual phases. Each phase utilized the proposed schedule-based application algorithm shown in

```
transform_control_block (control_block, initial_height, max_passes)
{
    // First, apply integer expression reformulations that do not add operations
    best_height = schedule_based_application (control_block, initial_height,
                        INT_EXPR_REFORMULATION | ONLY_FREE_TRANS);
    pass = 0;
    while (pass < max_passes)
    {
        pass++;

        // Apply renaming-with-copy transformations and minimize the number
        // of transformations applied by also doing an undo pass, where unnecessary
        // transformations are backed out.
        rename_height = schedule_based_application (control_block, best_height,
                        RENAMING_WITH_COPY | MINIMIZE_COUNT);

        // Apply integer expression reformulation transformations (both those that
        // require the addition of operations and those that do not).
        expr_height = schedule_based_application (control_block, rename_height,
                        INT_EXPR_REFORMULATION);

        // Stop if schedule height has not changed this pass
        if (expr_height == best_height)
        {
            best_height = expr_height;       // Update best height for return value
            break;                           // Break out of loop
        }

        // Otherwise, update best height and perform another pass of transformations
        best_height = expr_height;
    }
    return (best_height);       // Return the new schedule height of control block
}
```

Figure 3.7: Algorithm used to transform each control block.

Figure 3.1 and the priority functions shown in Figure 3.5. As shown in Figure 3.7, the schedule-based application algorithm is first used to apply “free” integer expression reformulations. Then, in order to maximize the performance of each control block, multiple transformation passes are made until the control block’s performance remains stable or max_passes is reached. For almost all of the control blocks, the first pass captured all the performance improvement (if any) and the second pass ended up not improving performance at all. However, in a few control blocks the second pass did improve performance slightly. This indicates that the priorities for these two transformations are well chosen and that only one pass is currently necessary to capture practically all of the performance benefit. However, after more transformations are implemented, multiple passes may be desirable. Therefore the capability to perform multiple passes is retained in the algorithm.

In each of these passes, renaming with copy is performed first. In addition to performing the schedule-based application algorithm shown in Figure 3.1, an extra pass over the performed transformations is made in an attempt to minimize the number of transformations performed. This extra pass undoes each transformation one at a time and evaluates the effect of undoing the transformation. If undoing the transformation does not harm the control block's performance, the transformation remains undone; otherwise the transformation is redone and truly kept. The algorithm for this extra pass is not shown because it is a simple variation of the algorithm shown in Figure 3.1. The transformations are undone in the same order in which they were performed. This extra pass was found to significantly reduce the number of transformations performed, freeing processor resources for future transformations. In a few cases, undoing unnecessary transformations actually improved the control block's performance. Note that for some transformations, such as renaming with copy, only a minimal amount of state information needs to be maintained in order to allow transformations to be easily undone in a separate pass. However, for other transformations, such as integer expression reformulation, it is difficult to undo arbitrary transformations after several transformations have been applied using only state information because of the extent the code is modified by these transformation. In order to undo transformations for the general case, an inverse transformation is required that analyzes the code for “undo” opportunities and applies the inverse transformation appropriately.

After renaming with copy is performed, integer-expression-reformulation transformations are performed. Both “free” transformations and those that require extra operations are performed in this pass. However, an extra pass to minimize the number of transformations applied was not performed, because of the difficulty in undoing an earlier expression reformulation after other reformulations have been performed. In order to undo expression reformulations properly in this situation, a powerful common subexpression elimination transformation is required that has the ability to rewrite expressions to create common subexpressions which it then can eliminate. Such a transformation is interesting in its own right because it adds dependence constraints while reducing the amount of processor resources utilized. This transformation was not implemented for this thesis but is an interesting area for future work. Note that initial analysis indicates that the inability to minimize the number of expression reformulations performed does not significantly affect performance when the schedule-time algorithm is used. However, some of the other approaches evaluated in Chapter 4 may not perform as well because the number of transformations is not minimized.

3.5 Related Work

Watts, Soffa, and Gupta [24] advocate using feedback from task scheduling to guide parallelizing transformations, either interactively or automatically, in order to improve program performance on a multiprocessor. By analyzing the task schedule, the heuristically-desirable tasks to transform, namely large tasks scheduled at times when at least one of the processors is idle, can be systematically determined. Using these heuristics, loop transformations can then be applied until the task schedule indicates all the processors are fully utilized or some acceptable amount of processor idle time has been achieved. Three algorithms for applying these transformations are proposed. The first approach reschedules the task graph after a heuristically-selected set of transformations is applied. This process continues, applying more transformations with each iteration, until the desired processor utilization is reached. The second approach performs transformations during scheduling, whenever there are no tasks available to schedule on an idle processor. The third approach is a hybrid that first applies the second approach and then does one pass using the first approach. This last approach was advocated because it yielded reasonably good processor utilization while requiring the task graph to be scheduled only twice. The primary benefit of this approach is that fewer transformations are applied. Because each transformation introduces some execution overhead, applying fewer transformations can potentially improve performance. It was observed that blindly applying all possible transformations tends to be time-consuming and often results in task schedules with lower performance.

The fundamental problems involved with applying parallelizing transformations for multiprocessors and for ILP processors as described in this chapter are quite similar.

However, the treatment of these problems in [24] is oversimplified, yielding an application algorithm that ignores the run-time overhead introduced by each transformation and that focuses on maximizing processor utilization, not overall program performance. As a result, applying transformations with this application algorithm may degrade overall program performance. In this chapter's treatment of these problems, both the benefit and the overhead caused by each transformations is evaluated in the context of a detailed model of the processor's execution constraints. As a result, the algorithms presented in this chapter can intelligently apply the transformations in order to maximize the overall program performance. The drawbacks of applying transformations without considering the overhead introduced (as in [24]) will be discussed further in Chapter 4 with the evaluation of the dependence-based approach described in Section 4.4.

4. PERFORMANCE BENEFITS OF SCHEDULE-TIME TRANSFORMATIONS

4.1 Introduction

One advantage of applying transformations at schedule time is that the effect of applying each individual transformation on the resulting code schedule can be accurately evaluated. By utilizing the search algorithm described in Section 3.2 to determine which set of transformations to apply, a near-optimal choice can be made that takes into account the processor's execution constraints, the various dependences between the operations, and the resulting placement of each operation by the scheduler. When compared to the current heuristic-based approach for applying ILP transformations, this schedule-based approach can yield strikingly better performance and more consistent results from ILP transformations.

The proposed schedule-time-transformation framework has been integrated into the IMPACT compiler and it schedules and transforms code one control block at a time. A *control block* is a code segment that has a single entry point and one or more exit points. Control blocks with more than one exit point are called *superblocks* [25] and those with only one exit point are called *basic blocks* [1]. A control block may represent an entire loop, the most frequently executed path through an entire loop, or an acyclic code segment which may or may not reside inside a loop. After scheduling the control block, a good first-order estimate of the processor cycles spent executing the control block can be calculated using profile information and the cycle each exit point is scheduled in. Benchmark performance can then be estimated by adding up all the cycles spent in each control block. Although this first-order estimate assumes no cache misses or branch mispredictions, this estimate is highly indicative of the actual benchmark performance. In addition, the net change in benchmark performance due to cache and branch prediction effects is typically independent of the final code schedule produced. However, there can be large local effects at the control block level that makes comparing control blocks difficult. For this reason, all of the results shown in this section will ignore these effects to allow fair comparisons.

For each SPECcint92 and SPECcint95 benchmark, the highest-performance classical and ILP-optimized superblock code that the IMPACT compiler can currently generate for the HP-PA architecture with extensions for general speculation [15] is used. In the baseline code, the renaming with copy or integer-expression-reformulation transformations are not performed. In addition, each control block is scheduled using an acyclic list scheduler and a scheduling heuristic called *dependence height and speculative yield* (DHASY) [23] that uses dependence and profile information in order to guide compile-time control speculation. This combination of ILP optimized superblock code and DHASY-based list scheduling allows the analysis of a stable and state-of-the-art technology for compiling control-intensive applications for processors that issue four or fewer operations per cycle, allowing a wide range of current and near-future processor configurations to be studied.

4.2 Criteria for Control-Block Level Comparisons

This section's experiments will first evaluate the benefit of applying transformations at schedule time by performing a detailed comparison of this schedule-

Table 4.1: The four processor configurations evaluated.

Processor Resources	Processor Configuration			
	One-issue	Two-issue	Four-issue	Eight-issue
Decoders	1	2	4	8
Branch Units	1	1	1	2
Integer ALUs	1	1	2	4
Floating-Point ALUs	1	1	1	2
Memory (Load/Store) Units	1	1	2	3
Integer Register File Read Ports	2	3	6	12
Integer Register File Write Ports	2	2	3	6

based approach to the current heuristic-based approach at the control block level. The control blocks for each comparison were selected using the following two criteria: the control block was executed at least once and there was at least one opportunity to perform the relevant transformation. In SPECcint92 and SPECcint95 code compiled as described above, there are 6041 control blocks that meet these criteria for just the renaming-with-copy transformation. For just the expression-reformulation transformation these criteria is met in 4455 control blocks. For comparisons involving both transformations, there are 8680 control blocks meeting these criteria where at least one of the two transformations can be performed and in 1816 of these control blocks, both transformations can be performed. The effect of each of these transformations will first be analyzed separately and then the combined effect will be examined.

4.3 The Processor Configurations

In addition, the behavior of heuristic-based approach and the schedule-based approach in the presence of varying levels of resource constraints will be examined using the four processor configurations shown in Table 4.1. All the processors have the

Table 4.2: Operation latencies.

Operation Type	Latency (in cycles)	Comments
Integer Addition/Shift/Logic	1	
Integer Multiplication/Division	2*	Rare after optimization
Integer/Float Load	3	High clock-rate assumed
Branch	1	No delay slots
Floating-Point Addition/Subtraction	2	Very rare in benchmarks
Floating-Point Multiplication	2	Very rare in benchmarks
Floating-Point Division	8/15	Very rare in benchmarks

* Function calls used. Only two cycles of computation are visible to the scheduler.

operation latencies shown in Table 4.2 to facilitate comparisons across configurations. The analysis will center on the four-issue processor configuration because this configuration is well matched to the instruction-level parallelism present in the benchmarks, and the heuristic-based approach was tuned to yield good performance for a four-issue processor configuration. In this four-issue processor configuration, performance is limited by the dependences between operations in some sections of code and by resource constraints in other sections of code, occasionally in the same control block. Although there is opportunity for ILP transformations to significantly improve performance, when applying these transformations the processor's resource constraints cannot be safely ignored.

There are also opportunities for improving performance in the one-issue and two-issue processor configurations by applying ILP transformations, but sensitivity to resource constraints is critical. These processor configurations show how the heuristic-based and schedule-based approaches deal with serious resource constraints. The one-issue processor configuration represents the extreme case where extra operations cannot usually be absorbed at all. Some opportunities do exist, however, because of the three-cycle load latency which can create holes in the schedule. The two-issue processor configuration represents the case where the instruction-level parallelism is high compared to the processor's resources but often a few extra operations can be absorbed. As ILP compilation technology advances, the issue-width at which this situation occurs will continue to increase.

On the other end of the spectrum, the eight-issue processor configuration represents the case where performance is determined almost entirely by the dependences between operations, and the processor's resources are typically underutilized. For this configuration, breaking dependences between operations can lead to larger performance wins than for the four-issue processor configuration because the resources are available to take advantage of the increased instruction-level parallelism. In addition, performing unnecessary transformations is significantly less likely to degrade performance because of the extra resources available. As will be shown, the scheduler still makes some choices that can affect the proper choice of transformations to perform even with abundant resources available.

4.4 The Four Application Approaches Evaluated

In order to evaluate in more depth the various advantages the schedule-based approach has over the heuristic-based approach, two additional variations to the proposed schedule-based approach will also be analyzed. The *dependence-based* variation uses the schedule-time framework to minimize dependence height (maximize instruction-level parallelism) without regard to the processor's resource constraints. The *estimate-based* variation uses the schedule-time framework to minimize schedule height based on an approximate instead of an accurate model of the processor. These variations will be used to provide insight into the behavior of both the schedule-based and heuristic-based approaches. Note that these variations are not intended as “lower-cost” alternatives to the schedule-based approach. These variations require almost all the same compiler framework features as the proposed schedule-based approach and, as will be shown, do not perform as well as the proposed schedule-based approach.

In order to ensure fair comparisons between the heuristic-based, schedule-based, dependence-based, and estimate-based approaches, each approach was implemented using the schedule-time transformation framework proposed in this thesis. In addition, the same code for detecting and performing the two transformations was used by all approaches. The only aspect varied was the algorithm for deciding which of the detected transformations to perform.

The application algorithm for the heuristic-based approach was faithfully ported from IMPACT’s current ILP optimizer. This heuristic-based algorithm applies renaming with copy whenever the transformation will break at least one anti or output register dependence within the control block (a requirement of all four approaches) and when the destination register to be renamed is not also used as a source in the same operation (detailed in the discussion of Figure 2.3). Expression reformulation is applied whenever the transformation may reduce the control block’s dependence height and no additional operations need to be created. Although the original implementation of expression reformulation, written by this researcher in 1992, has a significantly different structure than the new schedule-time implementation, the essence of the algorithm remains the same. However, to allow easier comparisons and to simplify porting, the search algorithm and priority functions described in Chapter 3 are used instead of the original version's ad hoc approach. As a result, this port of the heuristic-based approach consistently applies expression reformulation slightly better than the original version. Note that in order to stay consistent with the original heuristic-based implementation, the search algorithm attempts to minimize the control block's dependence height, not its schedule height.

The schedule-based approach uses the search algorithm and priority functions described in Chapter 3 to select from the set of transformation opportunities which transformations to apply. As described earlier, the set selected represents the minimum schedule height found for the target processor by the search algorithm.

The dependence-based approach is nearly identical to the schedule-based approach. The only difference is that for the dependence-based approach, a 128-issue processor with uniform functional units is specified as the "target" processor while selecting the set of transformations to perform. This effectively results in the minimization of the control block's dependence height without regard to the processor's resource constraints. Note that the same set of transformations will be applied for each of the four processor configurations shown in Table 4.1. It is also worth noting that the schedule-time framework always attempts to minimize the number of renaming-with-copy transformations applied even when nearly unlimited resources are available.

The estimate-based approach is also nearly identical to the schedule-based approach. The only difference is that for the estimate-based approach, a processor configuration with approximate execution constraints is specified as the "target" processor while selecting the set of transformations to perform. For the two-, four-, and eight-issue processor configurations, these execution constraints are relaxed to include only the issue and branch constraints of the actual processor configuration. Utilizing these constraints, the estimate-based approach will not apply transformations that will cause the issue and branch units to become overutilized, but the integer ALU and integer register ports may become inadvertently overutilized. For the one-issue processor configuration, instead of relaxing the resource constraints, all the operation latencies are reduced to one cycle. Utilizing these latencies, the estimate-based approach will tend not to apply transformations that add any operations, because all the schedule holes caused by the load latency have disappeared. This approach emulates the case where the transformations effect on schedule height is approximated instead of being calculated exactly as in the schedule-based approach. An estimate-based approach may be easier to implement in an existing compiler framework and may reduce the compile-time requirements. This estimate-based approach allows evaluation of the importance of using an accurate performance and resource usage metric to guide optimizations.

4.5 How the Results Were Generated

The results presented in this section were generated by first measuring the schedule height (as defined in Section 3.1) of each control block before performing the renaming-with-copy and expression-reformulation transformations. The schedule height is then measured after performing the set of transformations selected using each of the four approaches described in Section 4.4. Note that the control block selection criteria guarantees that all the control blocks analyzed will be entered at least once and therefore have a well-defined schedule height.

The benefit of each approach can then be quantified by subtracting the resulting schedule height from the original schedule height of the control block. In the example used in Section 3.1 to illustrate the calculation of schedule height, applying renaming with copy (using one of the approaches) yields a (4.8 - 3.9) = 0.9 cycle benefit. This metric is informative because it indicates how much the schedule height changed, but unfortunately it doesn't indicate how important the change is. If the control block originally had a schedule height of 100, a 0.9 cycle difference represents only a 0.9% change. However, if the block originally had a schedule height of 4.8 cycles, this difference represents a (0.9/4.8) = 18.8% change. This metric is very informative because it indicates how much the change can effect overall program performance. Unfortunately, without additional information it doesn't indicate how many cycles the schedule height changed. Therefore, to present a complete picture of what is occurring when these transformations are applied, both the percent benefit and the benefit in cycles metrics will be utilized. In addition, performance distributions are used to summarize all this control-block data for ease of comprehension. Where possible, related distributions will be placed in the same figures to facilitate comparisons.

4.6 Overview of the Rest of This Chapter

The remainder of this chapter will analyze each of the four approaches described above at the control block level in the context of the renaming-with-copy transformation (Section 4.7), the expression-reformulation transformation (Section 4.8), and both transformations together (Section 4.9). This analysis will focus on the four-issue processor configuration and use the other configurations to show trends as the resource constraints change. Because of the large number of variations, only a representative subset of these control-block-level distributions will be presented to illustrate points made and trends noted in this chapter. A complete set of these control-block-level distributions is presented in Appendix A for the interested reader. The overall effect on benchmark performance will then be presented and discussed in Section 4.10.

4.7 Control-Block Level Analysis of Renaming With Copy

4.7.1 The four-issue processor configuration

The four-issue processor configuration represents the case where the instruction-level parallelism is well matched to the processor's resource constraints. Although performance is often limited by dependences between operations, resource constraints cannot be safely ignored. The four approaches will be first compared to the untransformed control blocks in Section 4.7.1.1. The schedule-based approach will then be compared to the other three approaches in Section 4.7.1.2.

4.7.1.1 Relative to original control-block performance

Applying renaming with copy to each control block, utilizing the four approaches described in Section 4.4, and scheduling each control block for the four-issue processor configuration yields the four distributions shown in Figure 4.1. These distributions show the percent change in schedule height caused by transforming each control block. The distribution for the heuristic-based approach, shown in Figure 4.1(a), shows a performance improvement in 8.3% of the control blocks and a performance degradation in 3.6% of the control blocks.

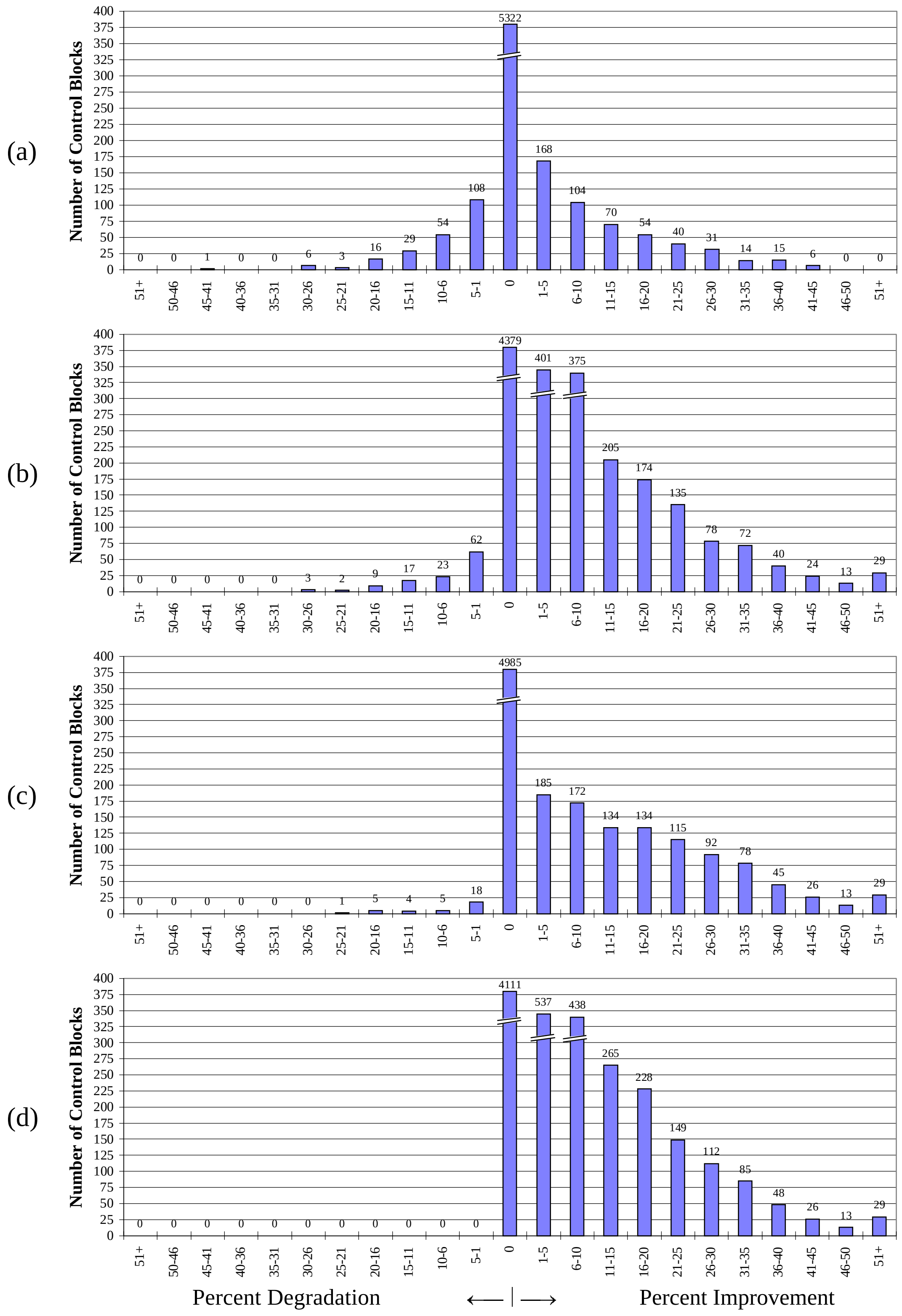


Figure 4.1: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the four-issue processor configuration.

The dependence-based approach reduces the percent of control blocks degraded to 1.9%, as shown in Figure 4.1(b), primarily because renaming with copy is not applied unless it decreases the dependence height. This can significantly reduce the number of extra operations added and often changes a performance degradation to a performance improvement. The percent of control blocks improved also significantly increased to 25.6% from 8.3%. This increase is largely because the dependence-based approach (as well as estimate-based and schedule-based approaches) can pick and choose among the transformation opportunities the heuristic-based approach consider poor risks.

The distribution for the estimate-based approach is shown in Figure 4.1(c). By modeling the issue and branch constraints, this approach prevents application of renaming with copy in the cases where resource constraints are clearly limiting performance, further reducing the percentage of control blocks degraded to 0.5%. However, this model of the processor is not exact and often transformations that are not beneficial for this model turn out to be beneficial for the actual processor model. This reduces the percentage of control blocks improved to 16.9%. However, because of the increased resource awareness, more control blocks are generated that show improvements of 26% or more.

The distribution for the schedule-based approach is shown in Figure 4.1(d). This approach optimizes the control block with exact knowledge of the processor model, and the resulting schedule produced for this model, so a performance degradation can never occur (at least in terms of the first-order performance estimate utilized to generate these results). In addition, the set of transformation opportunities to apply can now be fine-tuned to be a near-perfect fit for the processor configuration and the scheduling algorithm used. As a result, the percentage of control blocks improved surged to 31.9%. In addition, the improvement for each control block was increased, as shown in the distribution by a shift to the right.

As discussed in Section 4.5, it is also useful to examine these changes in schedule height in terms of cycles as shown in Figure 4.2. This view shows that applying renaming with copy often results in a multicycle change in schedule height. At one extreme, both the heuristic-based (Figure 4.2(a)) and the dependence-based (Figure 4.2(b)) approaches degraded performance in several control blocks by six cycles. Because the four-issue processor configuration can execute two integer ALU operations per cycle, this indicates renaming with copy was applied approximately twelve times more than could be absorbed in these control blocks. At the other extreme, the dependence-based, the estimate-based (Figure 4.2(c)), and the schedule-based (Figure 4.2(d)) approaches all improved the performance of 14 control blocks by more than 26 cycles. Many of these are frequently executed control blocks from 026.compress, resulting in a large performance improvements for this benchmark.

Although the one or two cycle changes seem small in comparison to these two extremes, it is through these changes that most of the overall performance is gained or lost. Some of the most frequently executed control blocks are improved or degraded by just one cycle. As described in Section 4.5, a small change in schedule height can translate into a large percentage change in performance for a control block. A one or two cycle change in a large number of moderately executed control blocks can also add up to

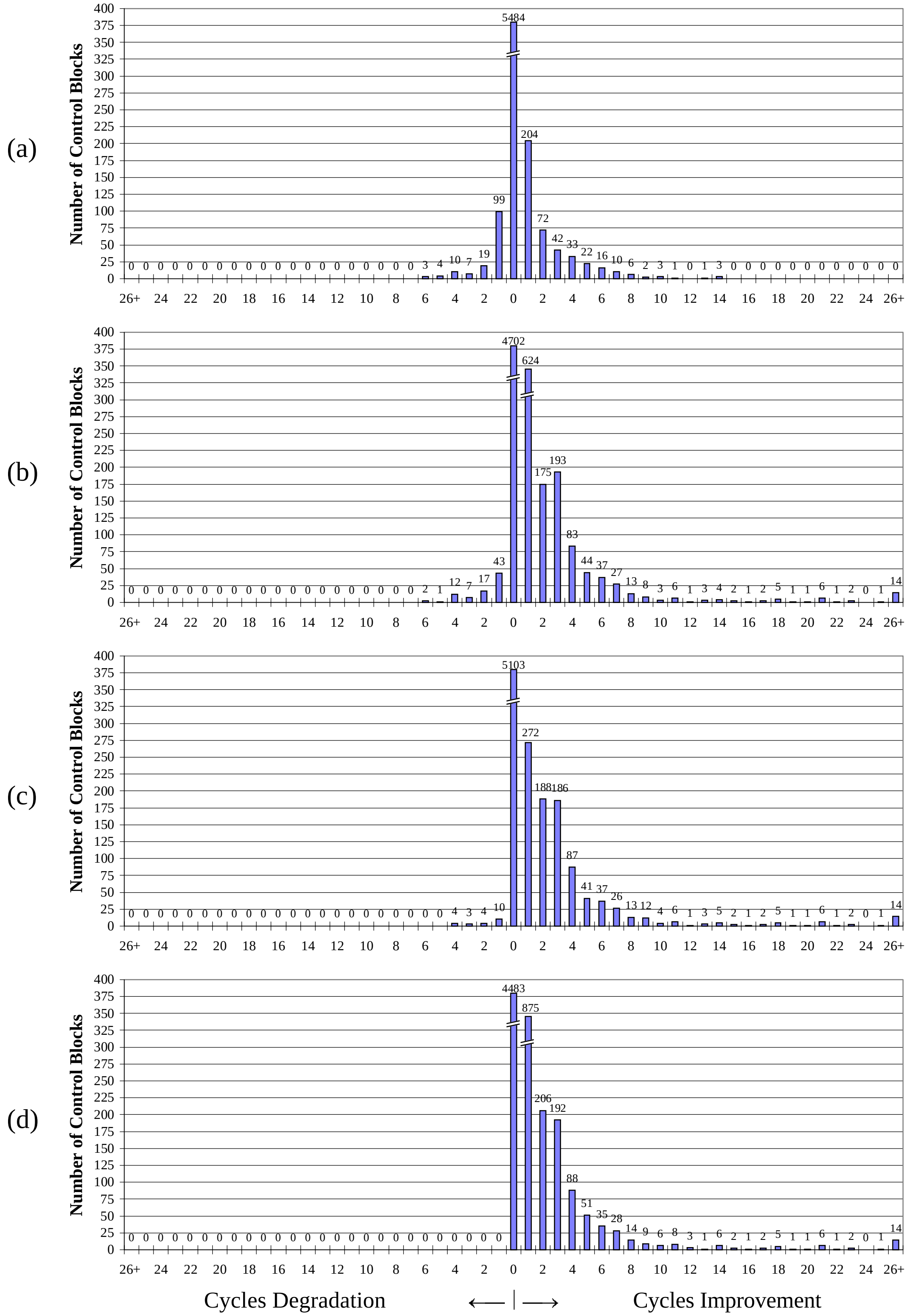


Figure 4.2: Change in cycles for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the four-issue processor configuration.

significant overall performance change. This is one reason the schedule-based approach does better overall than the heuristic-based approach, because it improves the performance of all blocks, not just the blocks it was tuned for.

Note that although Figure 4.2 displays the same schedule height changes as shown in Figure 4.1, there are more control blocks in the “zero” cycle change bin than in the “zero” percent change bin because the “zero” cycle bin is significantly larger than the “zero” percent bin. This is because a 0.4 cycle change (which falls in the “zero” cycle change bin) will be more than a 0.5% percent change (which falls into a 1% or higher percent change bin) if the overall control block schedule height is less than 80 cycles (most are).

4.7.1.2 Relative to the schedule-based performance

Figures 4.1 and 4.2 clearly show that, overall, the schedule-based approach improves the performance of significantly more control blocks than the other approaches. However, it is not clear from these figures how the schedule-based approach performed compared to the other approaches at a control-block-by-control-block basis. Performing this comparison in terms of percent change in schedule height yields the distributions shown Figure 4.3. The schedule-based approach improved performance in 28.1% of the control blocks when compared to the heuristic-based approach (Figure 4.3(a)), 9.4% when compared to the dependence-based approach (Figure 4.3(b)), and 17.1% when compared to the estimate-based approach (Figure 4.3(c)).

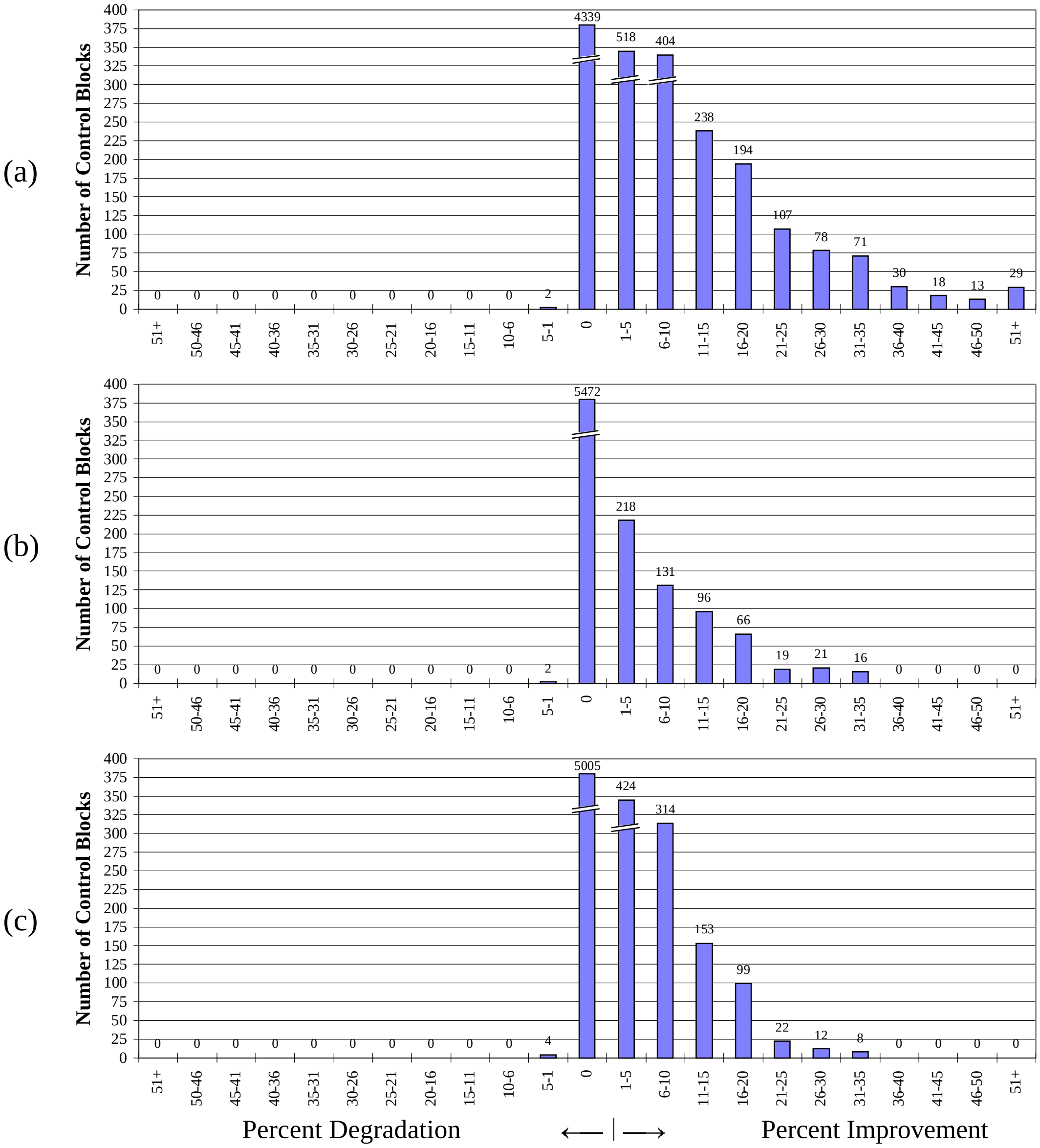


Figure 4.3: Percent benefit of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of renaming with copy for the four-issue processor configuration.

In many of these control blocks, although the other approaches had applied renaming with copy in a way that improved performance, the schedule-based approach found a better way to apply this transformation, yielding even higher performance. Of the 502 control blocks improved by the heuristic-based approach, 90 (17.9%) of these control blocks were further improved by the schedule-based approach. Of the 1546 control blocks improved by the dependence-based approach, 88 (5.7%) were further improved by the schedule-based approach. Of the 1023 control blocks improved by the dependence-based approach, 98 (9.6%) were further improved by the schedule-based approach. These improvements are all because the schedule-based approach can make detailed trade-offs that the other approaches cannot accurately make, further improving the effectiveness of a transformation.

In four of the control blocks, the estimate-based approach produce slightly smaller (thus better) schedule heights. The other two approaches produced slightly smaller schedule heights in two of the control blocks. For the other three processor configurations (not shown), up to seven control blocks have results similar to this for renaming with copy. Such cases are to be expected because the schedule-based approach uses a greedy (thus nonoptimal) search algorithm to minimize the schedule height. The fact that there are so few cases like this strongly indicates that heuristics used to guide the greedy search algorithm for renaming with copy are well chosen.

The same comparisons shown in Figure 4.3 are now shown in Figure 4.4 in terms of the number of cycles the schedule height is changed. In Figure 4.4(a), the distribution shows 1333 control blocks where the schedule-based approach generates a schedule height at least 0.5 cycles shorter (better) than the heuristic-based approach. For 1132 of these control blocks, the schedule-based approach improved the control block's performance and the heuristic-based approach didn't improve or degrade its performance. For 59 of these control blocks, the schedule-based approach improved the control block's performance more than the heuristic-based approach improved it. For 14 of these control blocks, the schedule-based approach improved the control block's performance

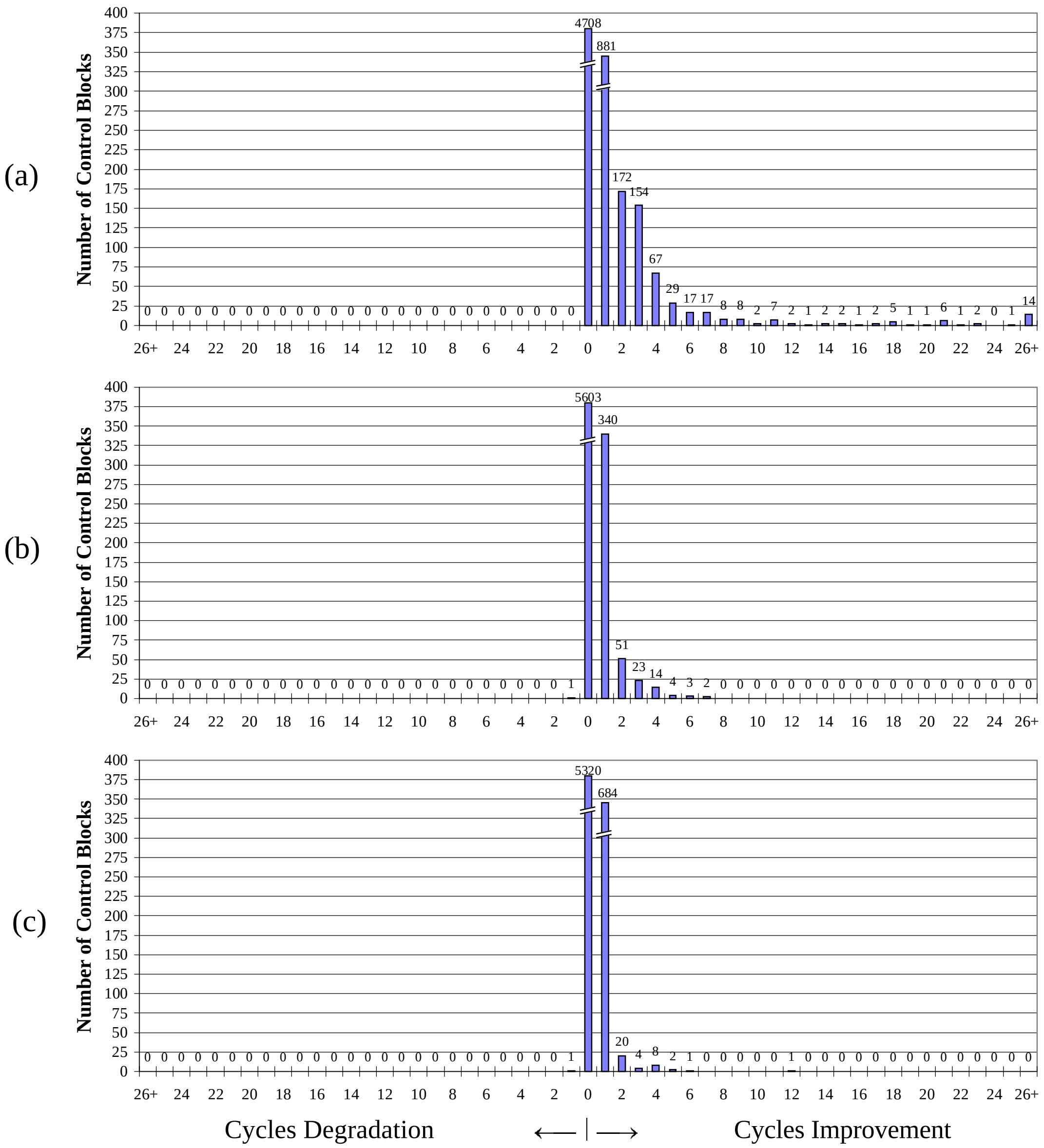


Figure 4.4: Benefit in cycles of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of renaming with copy for the four-issue processor configuration.

and the heuristic-based approach degraded it. One extreme example of this comes from 126.gcc’s flow_analysis function where the original schedule height for the control block was 21.9 cycles. The schedule-based approach reduced this to 15.4 cycles (a 29.7% improvement) and the heuristic-based approach increased this to 23.8 cycles (an 8.7%

degradation). For the remaining 128 of these control blocks, the schedule-based approach didn't improve or degrade the control block's performance but the heuristic-based approach degraded it. It is interesting to note that the schedule-based approach performs almost as well when compared to the heuristic-based approach (Figure 4.4(a)) as when compared to untransformed code (Figure 4.2(d)).

The advantage of the schedule-based approach over the dependence-based and estimate-based approaches is usually just one cycle, as shown in Figures 4.4(b) and 4.4(c) respectively. The dependence-based approach matches the performance of the schedule-based approach more often than the estimate-based approach as shown by the larger number control blocks in the "zero" cycle bin. However, the estimate-based approach has fewer deviations of two or more cycles because it is less likely to overutilize the processor's resource constraints. This conservative nature of the estimate-based approach doesn't always prevent large deviations, as shown by the anomalous case where the schedule-based approach reduced the schedule height twelve cycles more than the estimate-based approach.

Most of the control blocks where the schedule-based approach did achieve lower performance than the other approaches in Figure 4.3 (discussed above) were cases where there was less than 0.5 cycles difference in schedule height. As a result, in Figure 4.4, only two different control blocks show the other approaches yielding better results, one for the dependence-based approach and one for the estimate-based approach. For all the other processor configurations (not shown), there are only five other control blocks total where the other approaches yielded schedule-heights at least 0.5 cycles better.

4.7.2 The two-issue processor configuration

The two-issue processor configuration represents the case where the compiler has exposed more than enough instruction-level parallelism, in general, to keep the processor busy. Although a few extra operations can be usually absorbed, sensitivity to the processor's resource constraints is critical. In order to show how the four approaches perform with these higher resource constraints, the comparisons made in Section 4.7.1 will now be made for the two-issue processor configuration in Sections 4.7.2.1 and 4.7.2.2. The new insights that these comparisons reveal will then be briefly discussed.

4.7.2.1 Relative to original control-block performance

Figure 4.5 presents comparisons identical to those in Figure 4.1, except the two-issue processor configuration is now the target. Figure 4.5(a) shows that the heuristic-based approach now degrades 11.0% of the control blocks and only improves 5.2% of them. Similarly, Figure 4.5(b) shows that the dependence-based approach now degrades 8.2% of the control blocks but still improves 16.0% of them. This significant increase in the number of control blocks degraded and corresponding drop in those improved are due to both of these approaches totally ignoring the processor's resource constraints and applying the exact same set of transformations to each control block.

Figure 4.5(c) shows that the estimate-based approach degrades only 1.5% of the control blocks and improves 14.5% of them, still less than the dependence-based approach. As expected, the schedule-based approach still does significantly better than the other approaches and improves 20.7% of the control blocks as shown in Figure 4.5(d).

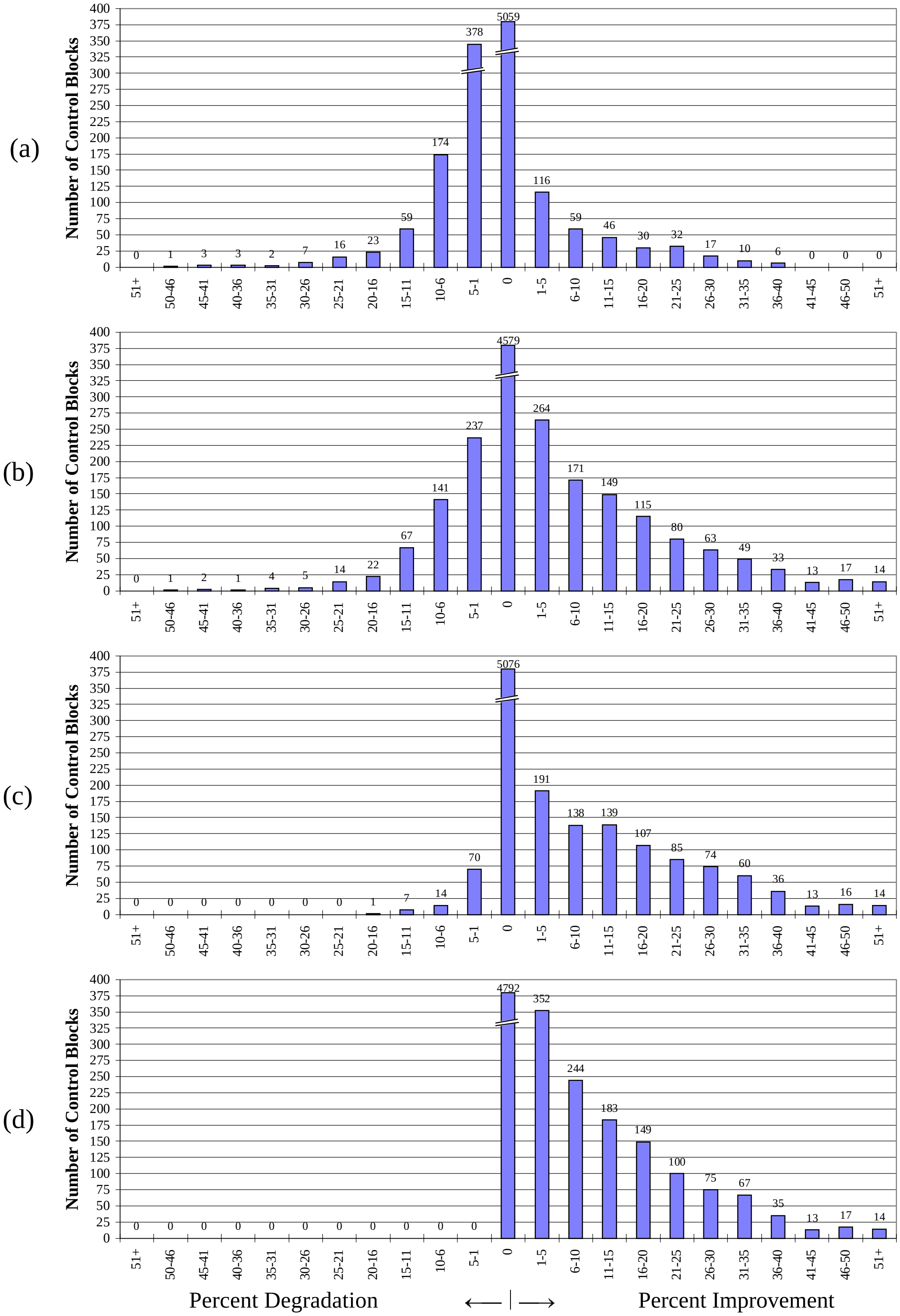


Figure 4.5: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the two-issue processor configuration.

Unlike the heuristic-based and dependence-based approaches, the estimate-based and schedule-based approaches adapt to the processor's resource constraints and apply different sets of transformations to each control block.

As before, examining schedule height changes in terms of cycles allows the low-level effects on the control blocks to be examined. Figure 4.6 presents comparisons identical to those in Figure 4.2, except the two-issue processor configuration is now the target. Comparing these two figures, it is interesting to note that even with processor resources cut in half, many control blocks are still improved by the same or nearly the same number of cycles. However, decreasing processor resources does reduce the opportunities for improvement, reducing the total number of control blocks improved, and can make the performance degradation due to poor application choices larger. This can be seen in both the heuristic-based (Figure 4.6(a)) and dependence-based (Figure 4.6(b)) approaches which can now degrade control blocks by as many as fifteen cycles. Reducing the processor resources does, however, make schedule estimates based on issue width and branch resources more accurate. Although the estimate-based approach (Figure 4.6(c)) degrades more control blocks with the two-issue processor configuration, the additional accuracy prevents control blocks from being degraded more than two cycles. The schedule-based approach (Figure 4.6(d)) still improves significantly more control blocks than the other approaches and yields a performance improvement distribution very similar to the four-issue case (Figure 4.2(d)).

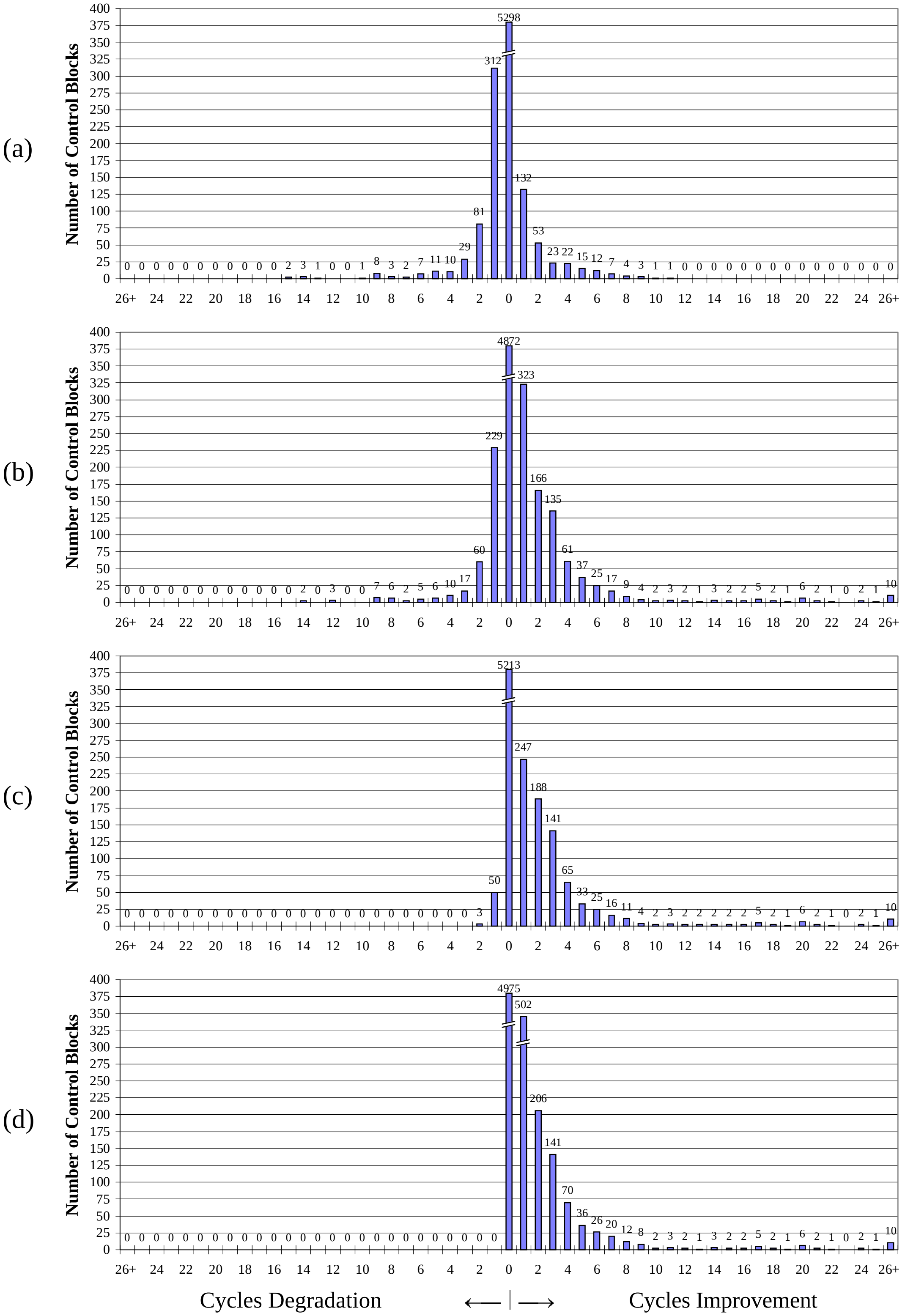


Figure 4.6: Change in cycles for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the two-issue processor configuration.

4.7.2.2 Relative to the schedule-based performance

The comparison of the schedule-based approach to the other approaches yield distributions very similar to those found for the four-issue processor configuration (Figures 4.3 and 4.4). In addition, these comparisons for the two-issue processor provided very little new information and therefore will not be discussed. The interested reader is referred to Appendix A.

4.7.3 The one-issue processor configuration

The one-issue processor configuration represents the case where extra operations cannot usually be absorbed at all, except where holes in the schedule are created because of the three cycle load latency. Although transformations that increase the dynamic number of operations are categorically almost never applied when targeting a resource-limited processor configuration such as this, performing the same comparisons made in Section 4.7.1, except for targeting this one-issue processor configuration, yields some interesting results.

4.7.3.1 Relative to original control-block performance

Figure 4.7 compares each of the four approaches to the original control block performance for the one-issue processor configuration. As was done in Figure 4.1, the distributions are shown in terms of percent change in schedule height. As expected, the heuristic-based approach now degrades 25.9% of the control blocks and improves only 1.4% of them, as shown in Figure 4.7(a). The dependence-based approach, shown in

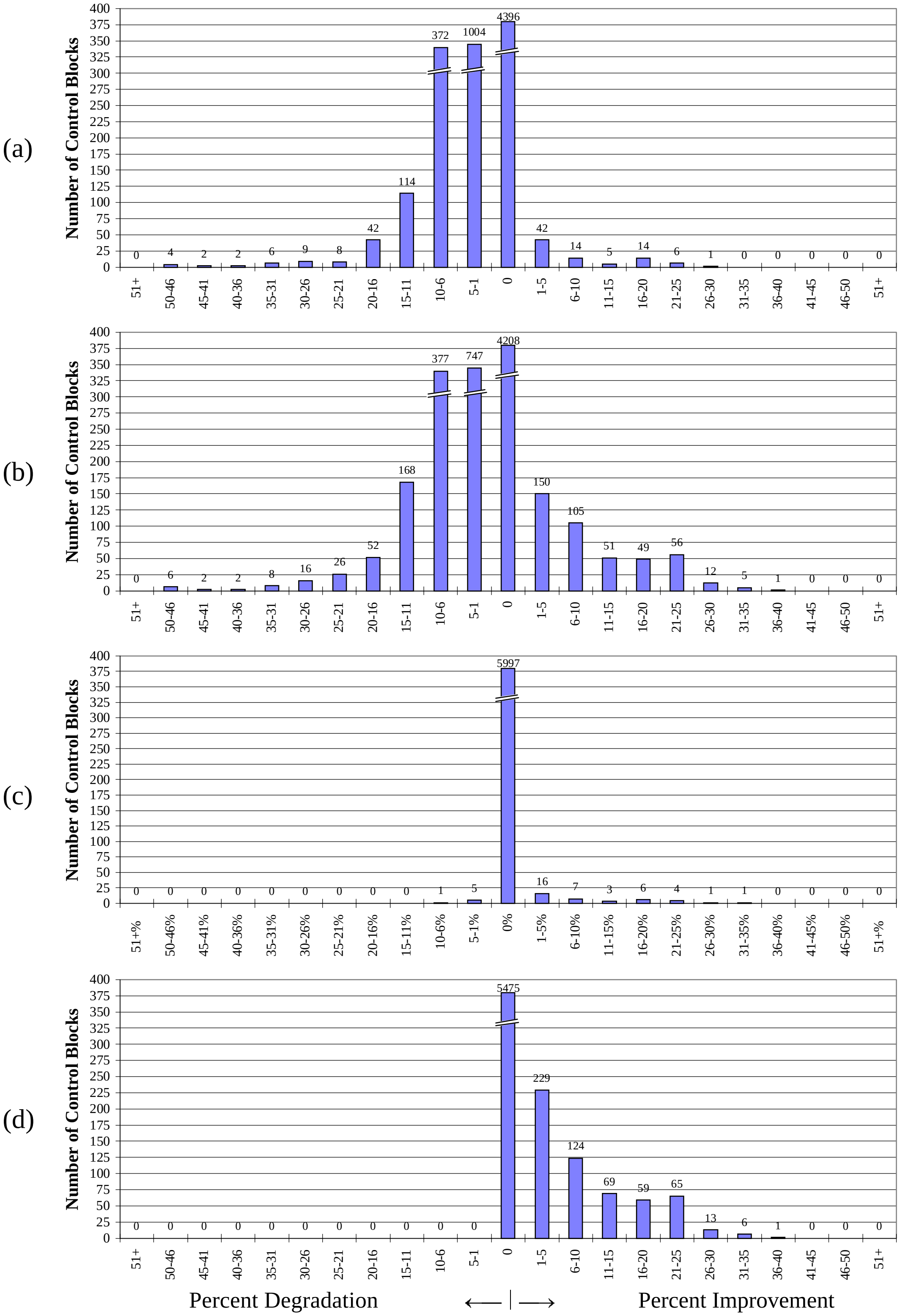


Figure 4.7: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the one-issue processor configuration.

Figure 4.7(b), degrades 23.2% of the control blocks but still manages to improve 7.1% of them. Clearly, the performance of a significant number of control blocks is still limited by dependences between operations. This is further illustrated by the results for schedule-based approach shown in Figure 4.7(d) which improves 9.4% of the control blocks without degrading the performance of any of them. In fact, the careful application of renaming with copy by the schedule-based approach will result in small but significant overall performance improvements for many of the benchmarks.

As described in Section 4.4, the estimate-based approach for the one-issue processor configuration uses an estimate that strongly discourages the insertion of any extra operations. The estimate accomplishes this by assuming that all operations have unit latency, removing all holes in the schedule. As a result, adding an extra operation is guaranteed to delay at least some operations, including the control block's last exit, by one cycle. However, renaming with copy will still be performed if it allows a frequently executed side exit to be scheduled early enough to offset the performance penalty of delaying later operations and exits. This situation does occur for a small number of control blocks, as shown in Figure 4.7(c). Using this extremely conservative approach improves 0.6% of the control blocks and degrades 0.1% of them.

These same schedule-height comparisons are now made in terms of change in cycles in Figure 4.8. The heuristic-based (Figure 4.8(a)) and dependence-based (Figure 4.8(b)) approaches now degrade control block performance by up to twenty three cycles. The performance degradation caused by the estimate-based approach (Figure 4.8(c)) is less than 0.5 cycles, and so does not appear on this distribution.

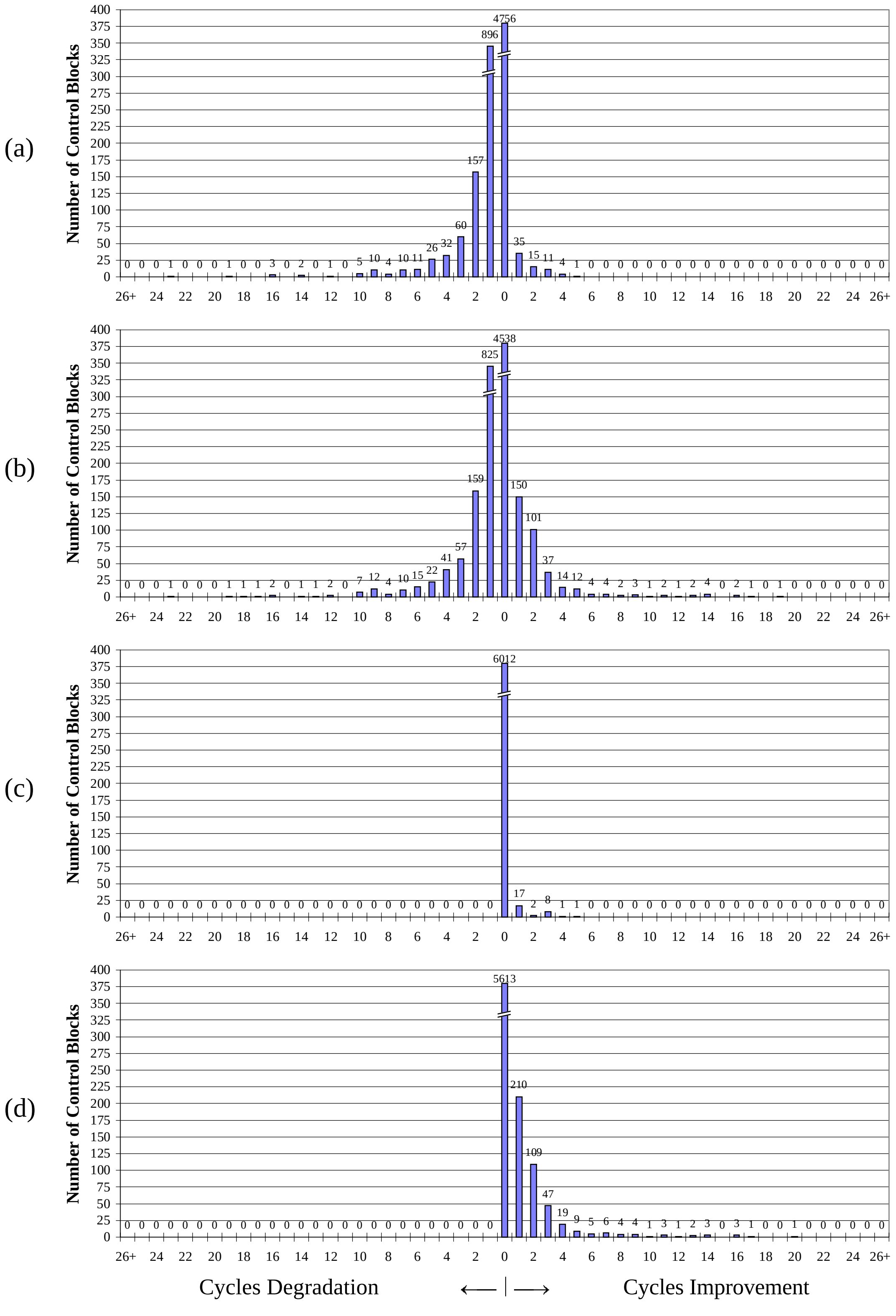


Figure 4.8: Change in cycles for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the one-issue processor configuration.

Interestingly, both the schedule-based (Figure 4.8(d)) and dependence-based approaches still manage to significantly reduce the schedule height of control blocks (up to twenty cycles). This suggests that there will always be opportunities to significantly improve performance by breaking dependences, even when the available instruction-level parallelism is extremely high when compared to processor resources. However, careful application of these dependence-breaking transformations is critical in order to prevent overutilization of resources. Only the schedule-based approach has the resource constraint sensitivity and the feedback from the scheduling algorithm required to consistently generate this performance improvement.

4.7.4 The eight-issue processor configuration

On the other end of the spectrum, the eight-issue processor configuration represents the case where control block performance is almost entirely determined by dependences between operations. The processor’s resources are typically underutilized, providing more opportunities for dependence-breaking transformations to improve performance than with the four-issue processor configuration. In addition, the extra operations inserted by unnecessary transformations are less likely to significantly degrade performance because of the extra resources available. However, as will be shown, careful application of transformations can still lead to higher performance.

4.7.4.1 Relative to original control-block performance

Figure 4.9 compares each of the four approaches to the original control block performance for the eight-issue processor configuration. As in Figure 4.1, the

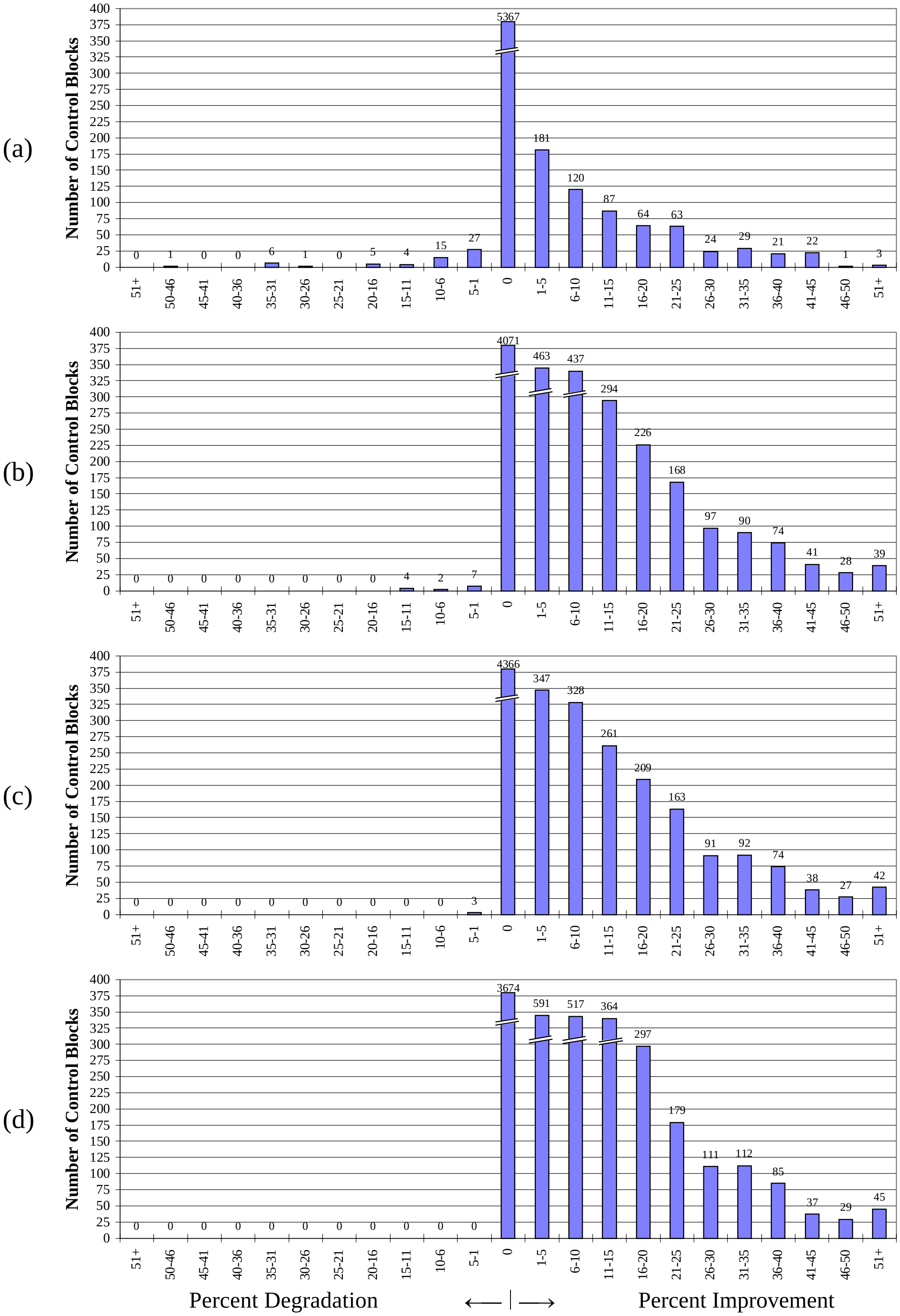


Figure 4.9: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the eight-issue processor configuration.

distributions are shown in terms of percent change in schedule height. The number of control blocks with degraded performance is significantly reduced when compared to the four-issue processor configuration shown in Figure 4.1. The heuristic-based approach (Figure 4.9(a)) degrades just 1.0%, the dependence-based approach (Figure 4.9(b)) degrades only 0.2%, and the estimate-based approach (Figure 4.9(c)) degrades only 0.05% of the control blocks. With the extra processor resources, the effect of poor application choices can almost always be absorbed without degrading performance. As with all the other processor configurations, the schedule-based approach (Figure 4.9(d)) does not degrade the performance of any control blocks.

The number of control blocks with improved performance is also significantly increased. The heuristic-based approach improves 10.2%, the dependence-based approach improves 32.4%, the estimate-based approach improves 27.7%, and the schedule-based approach improves 39.2% of the control blocks. The hand-tuned nature of the heuristic-based approach prevents it from improving as many control blocks as the other approaches. In order to prevent the degradation of a few frequently executed control blocks, the heuristic-based approach also inadvertently prevents the improvement of many less frequently executed control blocks. The heuristic-based approach also inadvertently limits the improvement of some of the important control blocks in 026.compress, resulting in significantly less overall performance improvement when compared to the other approaches. Important missed opportunities like those missed in 026.compress can almost always be solved with further tweaking of the application heuristics, but such solutions only work for the benchmarks the heuristics are tuned for.

The dependence-based approach still improves more control blocks than the estimate-based approach, as it did with all the other processor configurations. The conservative nature of the estimate-based approach prevents the application of transformations that appear not to improve performance based on the estimated processor configuration. However, these transformations sometimes do in fact improve performance for the actual processor configuration, resulting in a loss of opportunity. This illustrates the importance of using an accurate performance metric when using a search algorithm to select the transformations to apply. When the processor resources are underutilized, using only dependence height to estimate performance can yield better results than the estimate-based approach.

The schedule-based approach is compared directly to the other approaches in Figure 4.10. The schedule-based approach improved performance in 1932 (32.0%) control blocks when compared to the heuristic-based approach (Figure 4.10(a)), including 126 control blocks that were already improved by the heuristic-based approach. Compared to the dependence-based approach (Figure 4.10(b)), the schedule-based approach performs better on 517 (8.6%) control blocks, 96 of which were already improved by the dependence-based approach. The schedule-based approach also performs better for 840 (13.9%) control blocks compared to the estimate-based approach (Figure 4.10(c)), including 144 that the estimate-based approach had already improved.

Although dependences between operations is typically what limits overall control block performance in the eight-issue processor configuration, usually there is still heavy contention for processor resources when scheduling the first part of the control block.

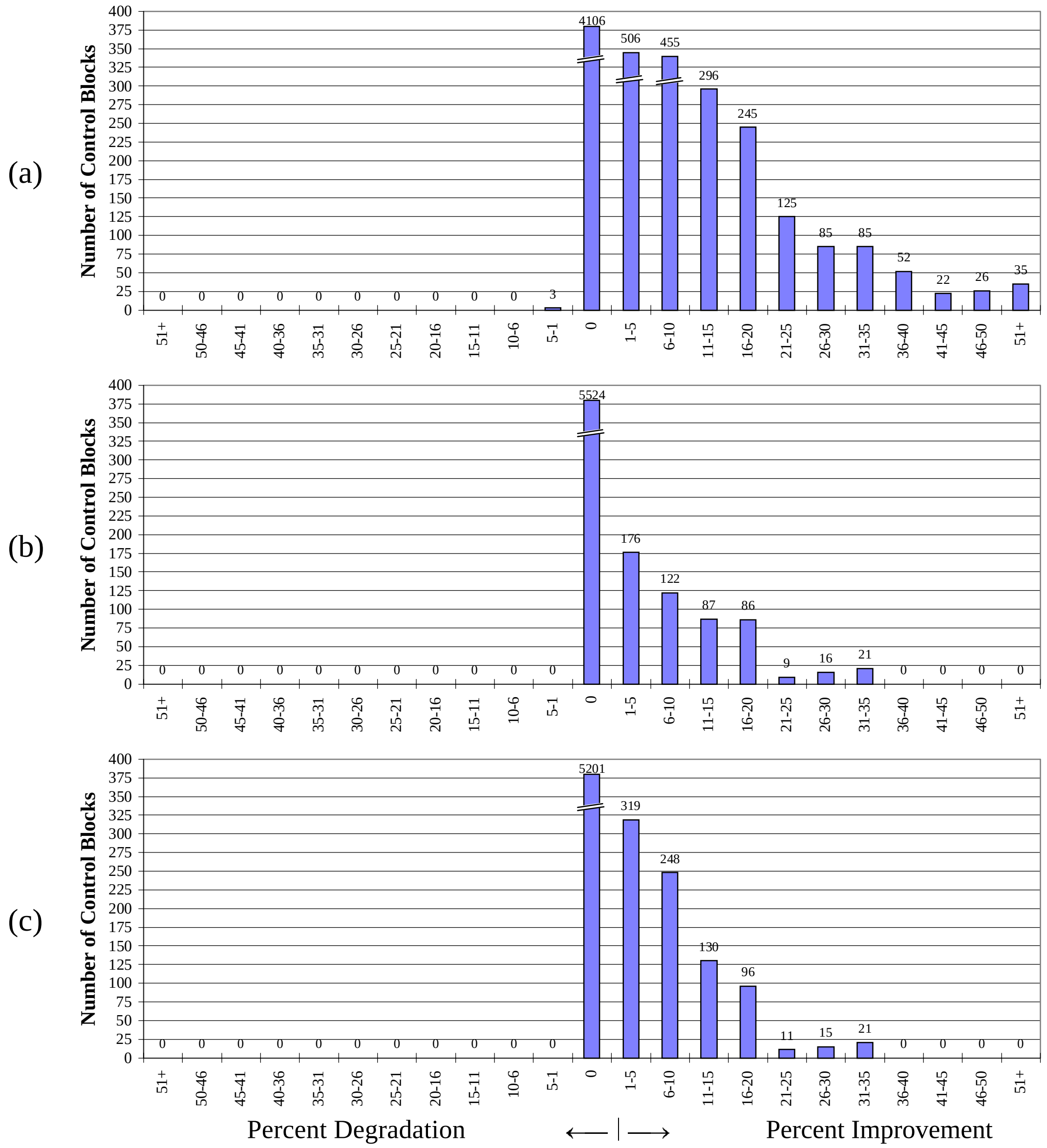


Figure 4.10: Percent benefit of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of renaming with copy for the eight-issue processor configuration.

For example, a control block with a dependence height of 50 cycles and 150 operations (an average ILP of three) may have 20 or more independent operations that can be scheduled in the first cycle. The cycle that these operations end up being scheduled in can significantly affect which of the dependences are important to break later in the schedule. Only the schedule-based approach, which utilizes exact schedule information, can adapt to these scheduler decisions. This allows the schedule-based approach to make fine-grain adjustments that further improve control block performance. In addition, these scheduler decisions can also create opportunities for performance improvement that the other approaches simply cannot detect. This is the primary reason the schedule-based approach improves significantly more control blocks than the other approaches.

The same distributions shown in Figure 4.9 are shown in Figure 4.11, except that the change in schedule height is presented in terms of cycles. As expected, only a few control blocks show more than one cycle of degradation. However, the heuristic-based approach (Figure 4.11(a)) still manages to degrade two control blocks by five cycles. The dependence-based (Figure 4.11(b)), estimate-based (Figure 4.11(c)), and schedule-based (Figure 4.11(d)) approaches all show significantly more control blocks with one or two cycles of improvements when compared to the four-issue configuration (Figure 4.2). For most of the control blocks with the higher improvements, dependences between operations limited performance even for the two-issue processor configuration (Figure 4.6), so adding processor resources changes their behavior very little.

The schedule-based approach is directly compared with the other approaches in terms of cycles in Figure 4.12. Figure 4.12(a) shows that in the three control blocks that the heuristic-based approach found a slightly better set of transformations to apply than the schedule-based approach (Figure 4.10(a)), the difference was less than 0.5 cycles. In addition, as with the other processor configurations, the schedule-based approach performs almost as well when compared to the heuristic-based approach (Figure 4.12(a)) as when compared to untransformed code (Figure 4.11(d)). In the control blocks where

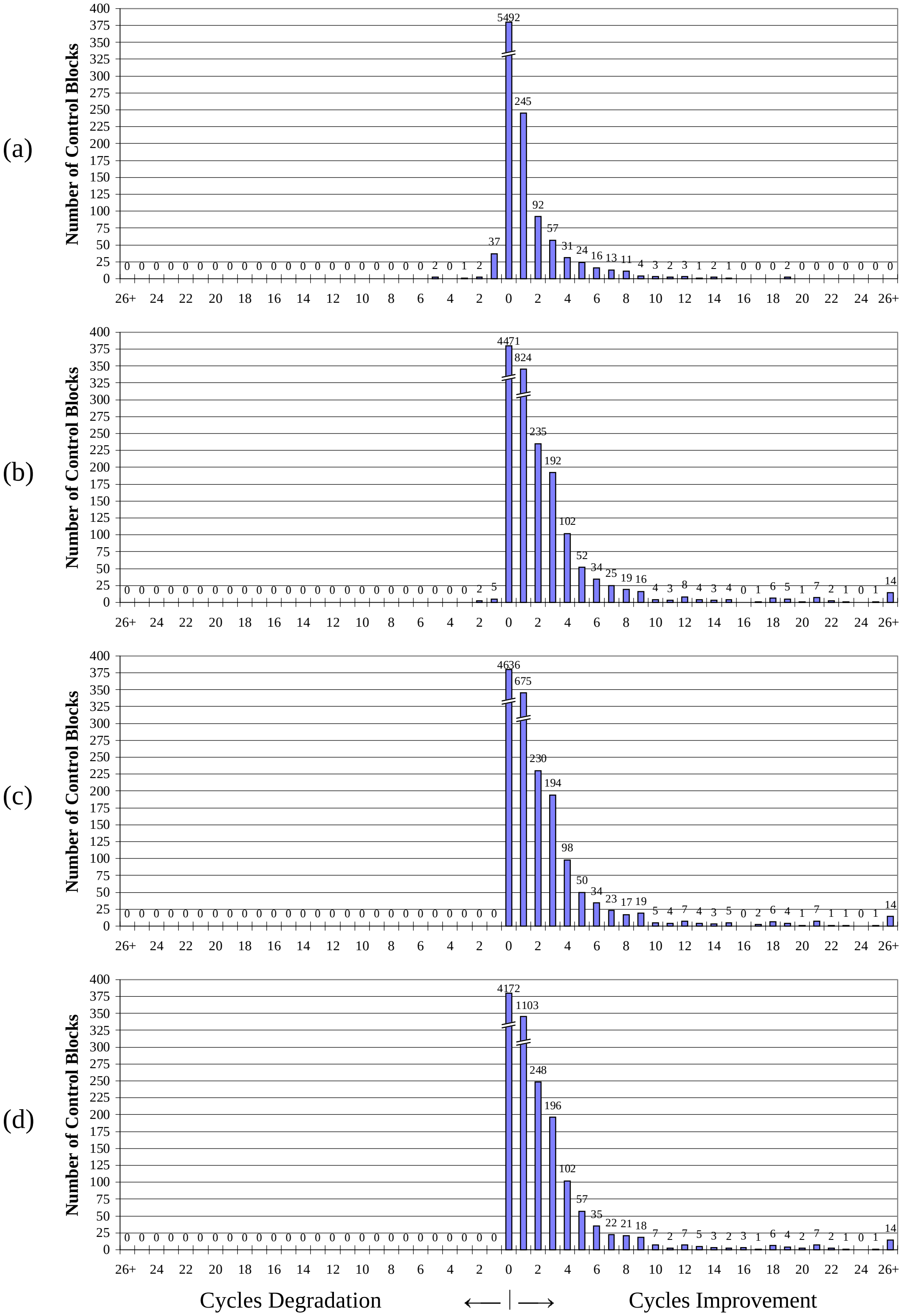


Figure 4.11: Change in cycles for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of renaming with copy for the eight-issue processor configuration.

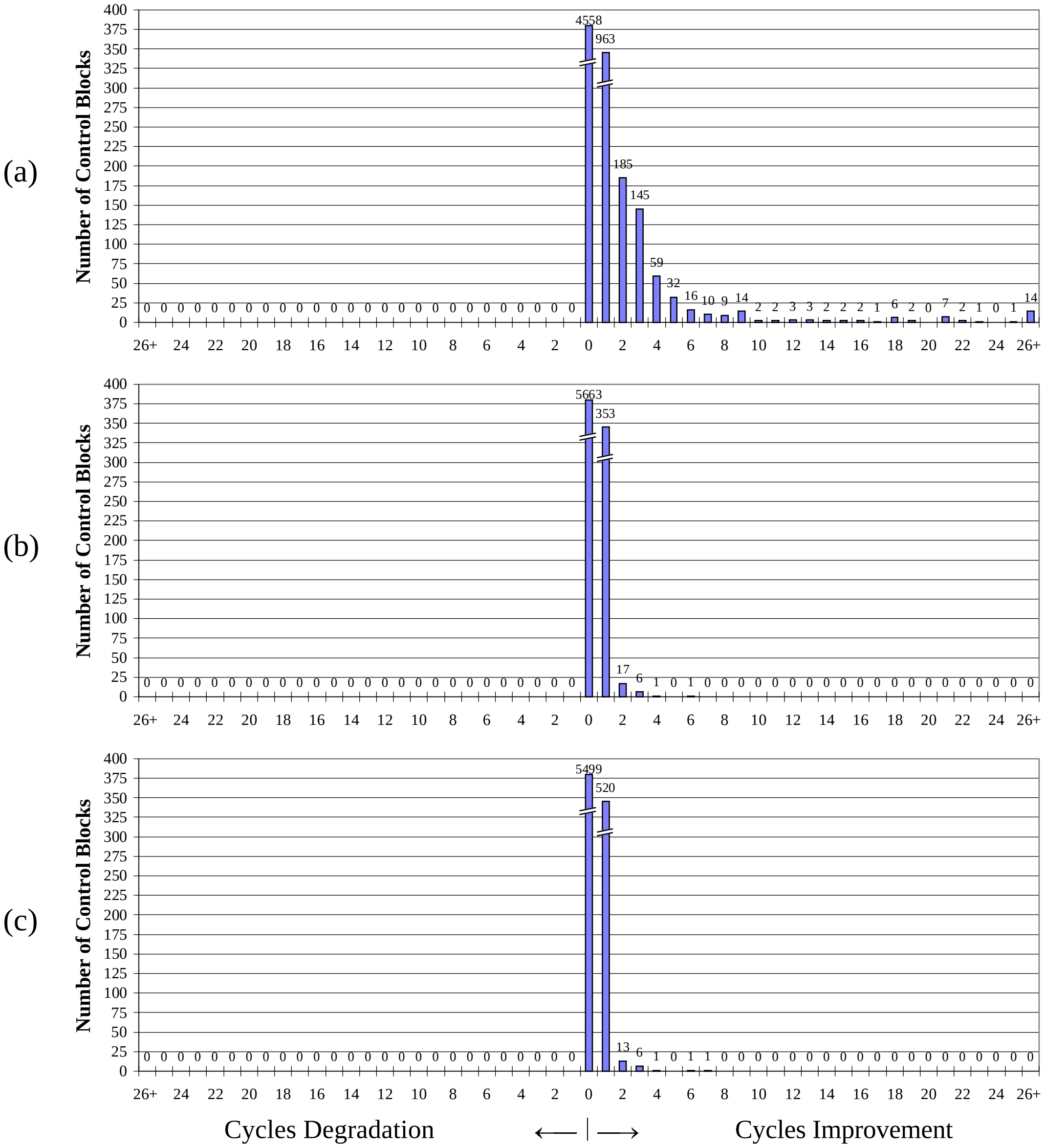


Figure 4.12: Benefit in cycles of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of renaming with copy for the eight-issue processor configuration.

the schedule-based approach performs better than the dependence-based (Figure 4.12(b)) and estimate-based (Figure 4.12(c)) approaches, this improvement is typically around one cycle. However, even with the eight-issue processor configuration, improvements of up to seven cycles are still possible.

4.8 Control-Block Level Analysis of Integer Expression Reformulation

Integer expression reformulation exhibits significantly more complex behavior than renaming with copy. Unlike the renaming-with-copy transformation, the maximum possible benefit from a single application of integer expression reformulation is just one cycle for the processor configurations studied. This benefit stems from reformulating an integer expression in order to remove a one cycle integer add or subtract operation from the critical path. In addition, the extra operations inserted while performing expression reformulation can be more difficult to absorb into the schedule than those inserted by renaming with copy. With renaming with copy, the “copy” operation is inserted after the operation transformed and can be scheduled in later cycles where there usually is less contention for processor resources. With expression reformulation, a newly created definition (see Figure 2.8(b)) must be scheduled earlier in order to be effective, where usually there is significantly more contention for processor resources. These two features of expression reformulation make applying this transformation properly considerably more difficult than applying renaming with copy.

Analyzing integer expression reformulation at the control-block level allows the four approaches described in Section 4.4 to be more clearly differentiated. In addition, this analysis will also provide insight into the behavior of this transformation. The rest of this section will analyze the effect of integer expression reformulation on the 4455 control blocks that meet the criteria presented in Section 4.2.

4.8.1 The four-issue processor configuration

4.8.1.1 Benefit in terms of percent change in schedule height

Figure 4.13 compares each of the four approaches for applying expression reformulation to the original control block performance for the four-issue processor configuration. The performance distributions are shown in terms of percent change in schedule height. Both the heuristic-based (Figure 4.13(a)) and dependence-based (Figure 4.13(b)) approaches utilize a dependence graph to determine which transformations to apply. The only difference between the two approaches (with the experimental setup discussed in Section 4.4) is that the heuristic-based approach prohibits the addition of any extra operations. The heuristic-based approach's strategy allows it to improve 10.9% of the control blocks while degrading only 1.6% of them. The dependence-based approach improves twice as many control blocks (20.8%) but also degrades more than four times as many control blocks (7.5%).

If the additional degraded control blocks were rarely executed and the additional improved control blocks were frequently executed, the dependence-based approach would yield higher performance than the heuristic-based approach. However, for the SPEC benchmarks, which is a subset of benchmarks for which the heuristic-based approach was tuned, some of these degraded control blocks are more frequently executed than the improved control blocks. As a result, the dependence-based approach significantly degrades the overall performance of 023.eqntott and 132.ijpeg while only moderately improving 008.espresso and 026.compress. In addition, for some of the benchmarks

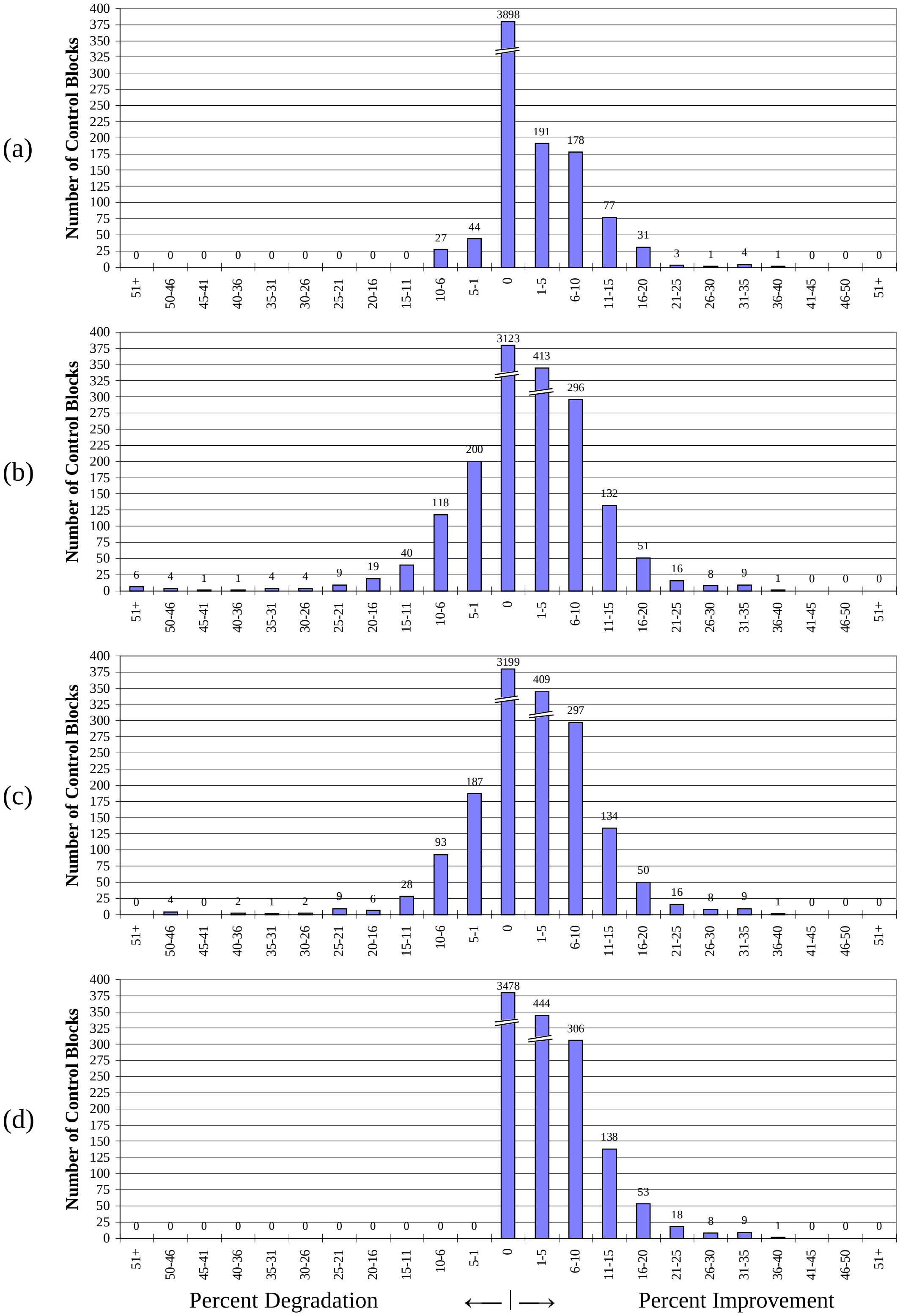


Figure 4.13: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of expression reformulation for the four-issue processor configuration.

studied in 1992 (not shown), even more impressive slowdowns resulted from the dependence-based approach. This is the reason that the heuristic-based approach adopted by the IMPACT compiler prohibits the addition of extra operations.

Although the heuristic-based approach does not add any operations, it is still possible for it to degrade performance as shown in Figure 4.13(a). Similar performance degradation can also occur for the dependence-based and estimate-based approaches when no additional operations are added and when additional operations are added but there should have been enough resources to absorb them. This unexpected degradation is due to the nonoptimal aspects of the DHASY list scheduling heuristic used for these experiments. This heuristic has the unfortunate tendency to delay infrequently-taken side exits unnecessarily when a control block contains a great deal of parallelism [26]. Performing expression reformulation typically increases instruction-level parallelism, providing more opportunities for the DHASY heuristic to make less-than-optimal decisions with regard to these infrequently-taken side exits. Fortunately, a promising solution exists in the form of the *speculative hedge* list scheduling heuristic [26] developed by Deitrich and Hwu. In preliminary experiments utilizing this prototype implementation of the speculative hedge heuristic, rescheduling the degraded control blocks using the speculative hedge heuristic appears to eliminate this problem. In fact, both the original version and the transformed version of these degraded control blocks often performed significantly better than before, further demonstrating the value of this new heuristic. It is expected that after the speculative hedge heuristic is implemented in the proposed transformation framework that it will always be used instead of the DHASY heuristic. In general, the best scheduling heuristics available should always be used to guide the application of transformations. However, no matter which scheduling heuristic is used, nonoptimal schedules will occur. The advantage of the proposed schedule-based approach is that it can adapt to the nonoptimal aspects of the scheduler by choosing a set of transformations that works well with the scheduling heuristics used.

The estimate-based approach, as described earlier, models just the issue and branch resources for the processor, and its performance distribution is shown in Figure 4.13(c). Comparing the estimate-based approach to the dependence-based approach, the estimate-based approach improves about the same number of control blocks (20.7%) but degrades slightly fewer of them (7.5%). In addition, overall benchmark performance is significantly better for several of the benchmarks because the estimate-based approach managed to avoid most of the degradation the dependence-based approach caused to several of the frequently-executed control blocks. The issue-width constraint was a good schedule estimate for these control blocks because they contained a high level of instruction-level parallelism.

Note that both the dependence-based and estimate-based approaches degrade significantly more control blocks for expression reformulation as compared to renaming with copy (Section 4.7.1.1). The major reason, as discussed in the introduction to this section, is that expression reformulation is much more difficult to apply properly. However, there is another, more subtle reason (discussed in Section 3.4). Expression reformulation is also much more difficult to undo after other transformations have been performed. With renaming with copy, after finding a set of transformations that yield the best performance, the transformation framework performs a second search to determine which transformations can be undone without hurting performance. This has the effect of minimizing the operations added even when there are no explicit resource constraints. With expression reformulation, the transformations selected on the first pass are kept, even if they are not needed to improve performance. As a result, extra operations are added even when they don't improve the metric being optimized for, such as dependence-height or the schedule estimate. In order to correctly undo expression reformulation, a sophisticated version of common subexpression elimination is required. Although potentially useful (at least for the dependence-based and estimate-based approaches), this common subexpression transformation was not used (or implemented) in these experiments. Note that keeping "unnecessary" expression reformulations has some beneficial side effects also, which are discussed below.

The schedule-based approach, shown in Figure 4.13(d), improves 21.9% of the control blocks while, as expected, degrading none of them. Unlike for renaming with copy, the heuristic-based approach did not significantly increase the number of control blocks improved over the dependence-based and estimate-based approaches. This difference is primarily due to the fact that the implementation keeps the transformations selected on the first search pass even if they are not needed to improve performance. In general, breaking dependences will result in better schedules even if the overall dependence height (or estimated schedule height) is not reduced. Therefore, by keeping these "unnecessary" expression reformulations, many more control blocks are improved by the dependence-based and estimate-based approaches. However, many more control blocks are also degraded as discussed above. As before, the schedule-based approach continues to generate consistently good results, unlike these other approaches.

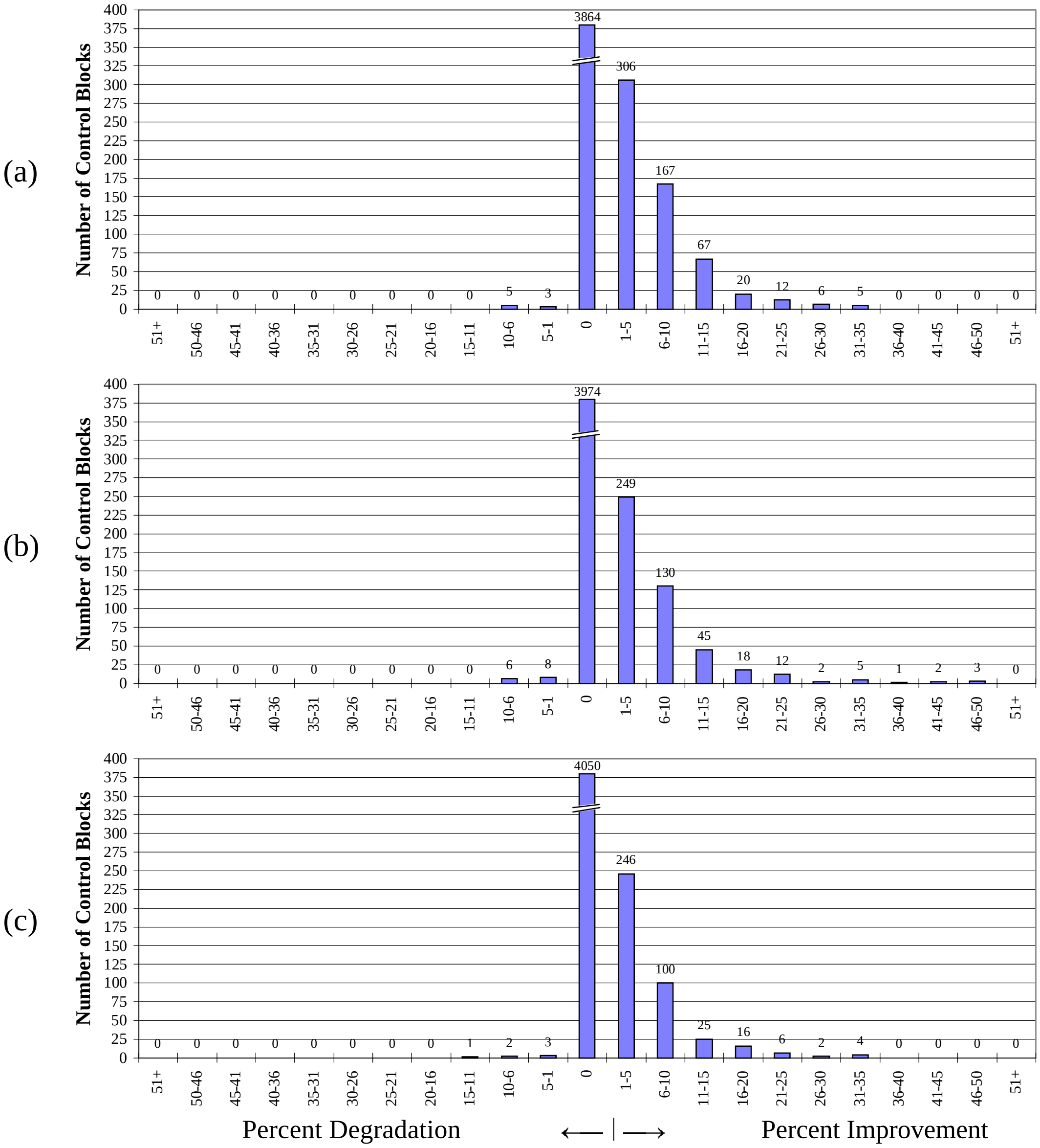


Figure 4.14: Percent benefit of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of expression reformulation for the four-issue processor configuration.

The schedule-based approach is compared to the other approaches in Figure 4.14. The schedule-based approach improves performance in 13.1% of the control blocks when compared to the heuristic-based approach (Figure 4.14(a)), 10.5% when compared to the dependence-based approach (Figure 4.14(b)), and 9.0% when compared to the estimate-based approach (Figure 4.14(c)). The benefit over the heuristic-based approach is primarily from being able to perform transformations that add operations. For the other two approaches, the benefit is primarily from preventing the degradation of any control blocks. However, the schedule-based approach also improved 19 (0.4%) of the control blocks improved by the heuristic-based approach, 33 (0.7%) of the control blocks improved by the dependence-based approach, and 32 (0.7%) of the control blocks improved by the estimate-based approach.

However, the schedule-based approach did not improve performance as much as the heuristic-based approach in 8 (0.2%) of the control blocks, as the dependence-based approach in 14 (0.3%) of the control blocks, and as the estimate-based approach in 6 (0.1%) of the control blocks. Because of the relatively large number of better solutions found by the other approaches, there appeared to be an opportunity to improve the priority function used by the search algorithm. However, further investigating revealed that the priority function selected had nothing to do with it. Given the schedule-time information available to the schedule-based approach, the transformations selected by the other approaches were counterintuitive because they appeared to lengthen, not shorten, the critical path. In fact, the only benefit of applying these transformations turned out to be that they happened to cause the DHASY-based scheduler to make better decisions about infrequently executed side exits, delaying them less (but still more than was necessary). Rescheduling these control blocks with the prototype speculative hedge implementation turned out to make all the benefits of applying these counterintuitive transformations disappear. This again indicates that the speculative hedge heuristic should be used instead of the DHASY heuristic when it becomes available.

In Figure 4.15, each of the four approaches for applying expression reformulation is compared again to the original control block performance for the four-issue processor configuration, except the distributions are shown in terms of change in cycles. The distribution for the heuristic-based approach, which cannot perform reformulations that require adding operations, is shown in Figure 4.15(a). Even without being able to add operations, the heuristic-based approach manages to reduce the schedule height of a control block by 16 cycles, the largest reduction shown for any of the approaches. In addition, many of the control blocks were reduced by more than two cycles. These large benefits, without the need to add any extra operations, illustrates why "free" expression reformulations are performed first before applying any other transformations as described in Section 3.4.

Allowing expression reformulation to add extra operations allows the dependence-based (Figure 4.15(b)), estimate-based (Figure 4.15(c)), and schedule-based (Figure 4.15(d)) approaches to improve about twice as many control blocks as the heuristic-based approach. In addition, the shape of the distribution of the improved control blocks remains approximately the same. This indicates that there is little correlation between requiring operations to be added and the importance of the transformation to the control block's performance. As seen before, adding operations significantly increases the chances of degrading performance, as seen by the degradation of up to four cycles caused by dependence-based and estimate-based approaches.

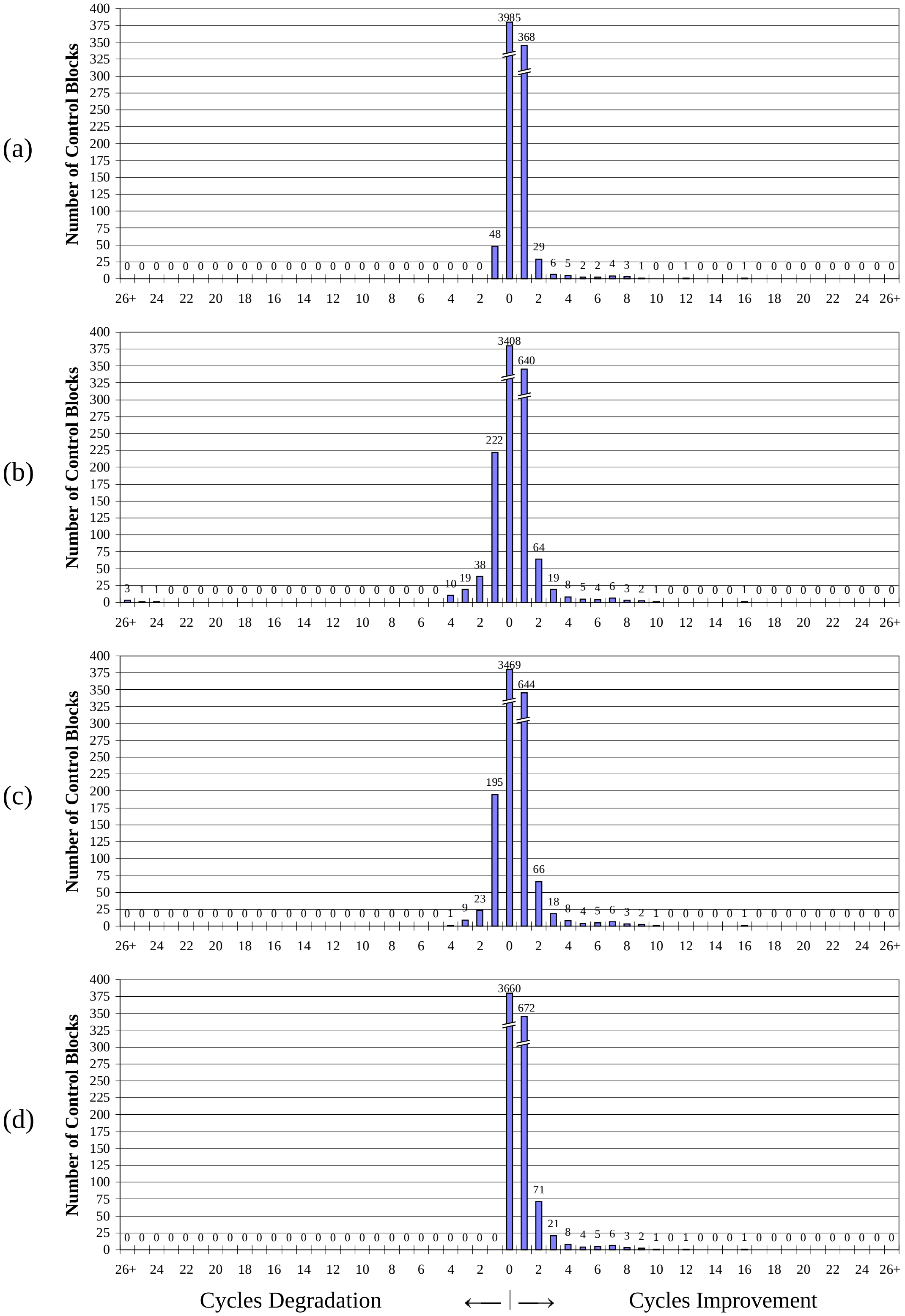


Figure 4.15: Change in cycles for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of expression reformulation for the four-issue processor configuration.

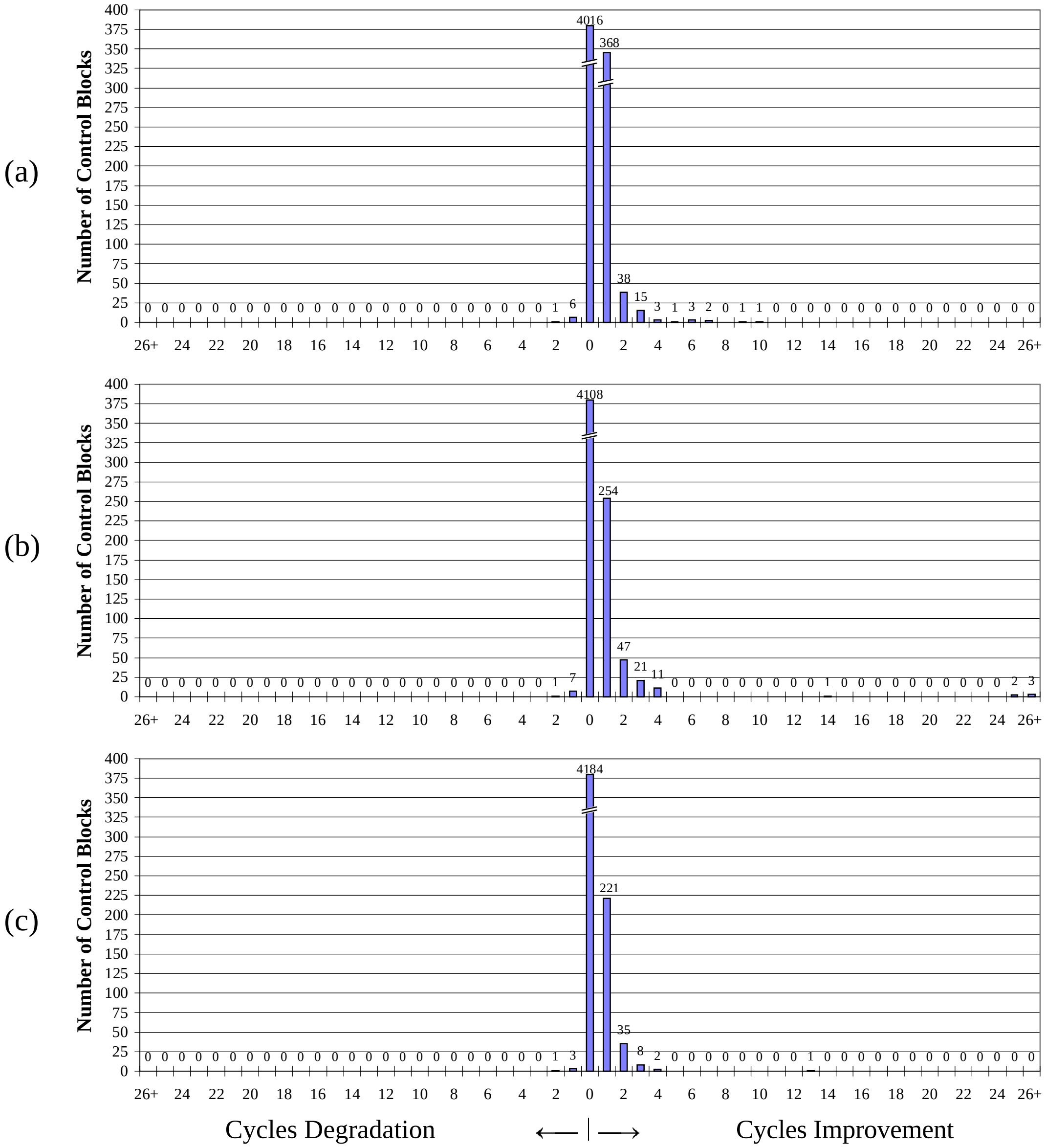


Figure 4.16: Benefit in cycles of schedule-based application over (a) heuristic-based, (b) dependence-based, and (c) estimate-based application of expression reformulation for the four-issue processor configuration.

The schedule-based approach is again compared to the other approaches in Figure 4.16, except the differences in schedule height are now shown in cycles.

4.9 Control-Block Level Analysis of Both Transformations

There are 8680 control blocks where renaming with copy, expression reformulation, or both transformations can be performed. In 4225 (48.7%) of these control blocks, only renaming with copy can be performed. In 2639 (30.4%) of these control blocks, only expression reformulation can be performed. Finally, in the remaining 1816 (20.9%) of these control blocks, both transformations can be performed. The distributions presented in this section indicate how these two transformations perform together.

4.9.1 The four-issue processor configuration

The control-block level performance for both transformations combined is shown in Figure 4.17. These distributions show the percent change in schedule height caused by the transformation of each control block. The heuristic-based approach, shown in Figure 4.17(a), improves 11.1% of the control blocks and degrades 2.9% of them when applying both transformations. The dependence-based approach, shown in Figure 4.17(b), improves 26.3% of the control blocks and degrades 5.4% of them. The estimate-based approach, shown in Figure 4.17(c), improves 21.0% of the control blocks and degrades 3.8% of them. Finally, the schedule-based approach improves 31.2% of them and degrades none of them. For most of these control blocks where performance was improved or degraded, the change in performance was caused by just one of the two transformations. As a result, these distributions are essentially the sum of the individual distributions presented earlier for each of the two transformations. Please see the earlier discussions presented in Sections 4.7 and 4.8 for why each approach generates the above distributions.

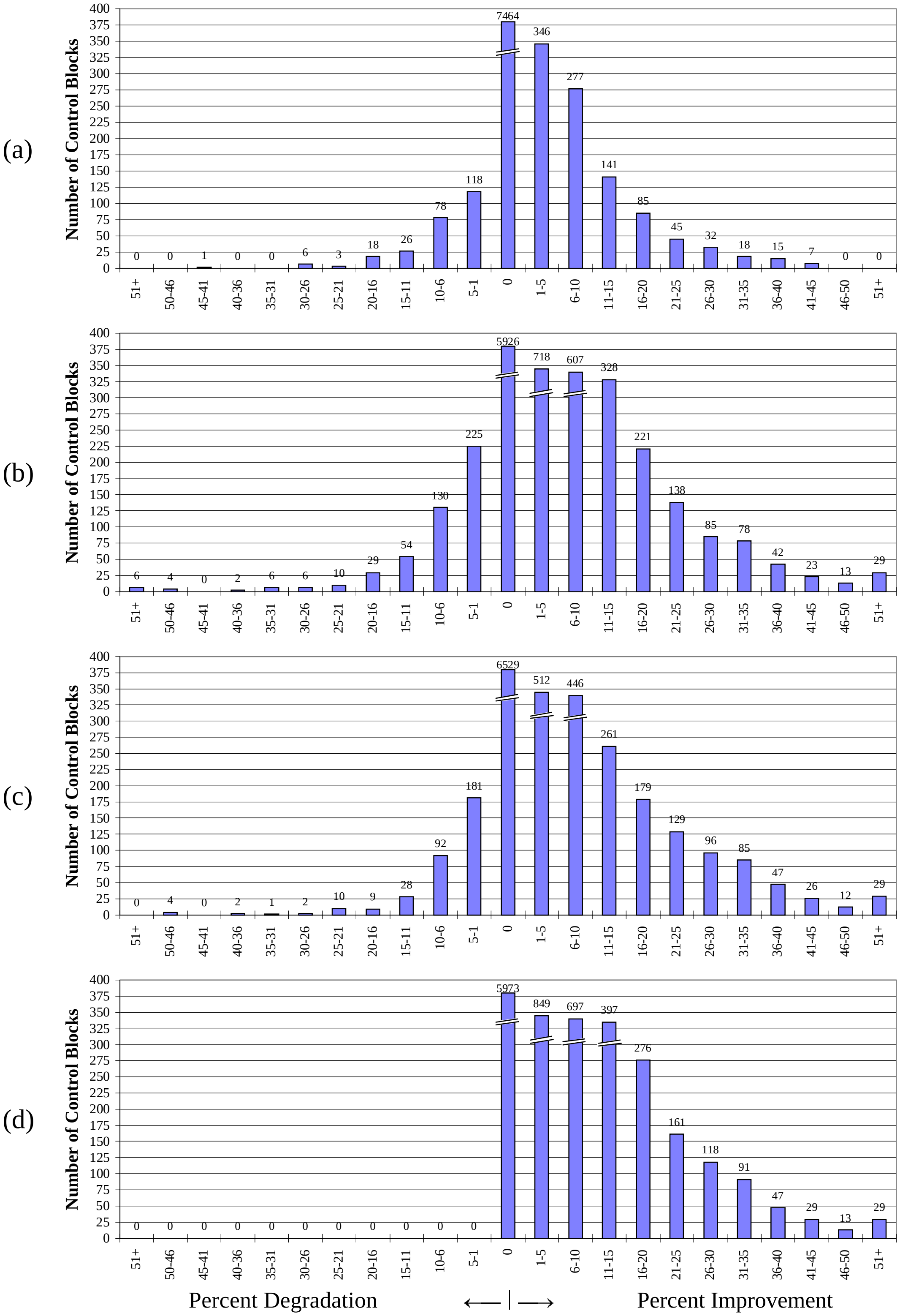


Figure 4.17: Percent change for (a) heuristic-based, (b) dependence-based, (c) estimate-based, and (d) schedule-based application of both transformations for the four-issue processor configuration.

However, there are a few control blocks where earlier experiments indicate that both transformations should affect performance. Examining these control blocks reveals that the effect of these transformations does not always simply add up. We will first focus on the results for the schedule-based approach because each transformation individually always improves performance because of the design of the algorithm. The schedule-based approach improved 2702 control blocks and in 204 (7.5%) of them, earlier experiments indicated both transformations should improve performance. In 82 of these control blocks, the benefit of both transformations individually simply add up when both are performed. In another four of these control blocks, the combined benefit of the transformations is higher (up to 56% more) than the sum of the individual benefits. These superlinear benefits are expected to happen more often as more ILP transformations are applied using the proposed schedule-time framework. Typically, more than one dependence chain prevents a key computation or an exit from being scheduled earlier. Reducing the height of each of these dependence chains is required in order to improve the control block's performance. In these experiments, only a few key computations were constrained by both an anti dependence (renaming with copy) and a flow dependence from an integer computation (expression reformulation). However, after performing both of these transformations, many key computations were found to be constrained by ambiguous memory disambiguation (addressed by data speculation [9], [27]) and many exits were constrained by branch resources (addressed by branch combining [4]). Performing expression reformulation and renaming with copy are important enabling transformations for these two other transformations because they allow memory addresses and branch directions to be calculated much earlier in the control block. Implementing these two other transformations is part of this thesis's future work that will be performed in conjunction with other researchers in the IMPACT group.

In the remaining 118 control blocks, performing both transformations improves performance less than the sum of the individual transformation benefits. In 13 of these control blocks, performing both transformations yields higher performance than the maximum of the individual transformation performances, but is less than the sum of the two performances. For 94 of these control blocks, the combined benefit is the same as the maximum of the two individual benefits. In fact, in 60 of these 94 control blocks, applying either transformation individually or both transformations together generates the same performance benefit. As mentioned above, key computations or exits are often constrained by multiple dependence chains. For these control blocks, the transformations reduce the height of one dependence chain until a different dependence chain is constraining performance. No matter how much the height of that first dependence chain is further reduced, the control block performance will not be further improved. However, if other height-reducing transformations are applied, the additional benefit of applying both transformations may become apparent.

In the last three of these 118 control blocks, performing both transformations generates less benefit than the maximum of the two individual transformation benefits. In two of these control blocks, applying renaming with copy consumed all of the available resources and prevented the more beneficial expression reformulation from being performed at all. In the other control block, expression reformulation was still able to significantly improve performance but was slightly less beneficial because of the resources consumed by renaming with copy. Recall from Section 3.4 that "free" expression-reformulation transformations are applied first, then renaming-with-copy transformations, and then finally expression-reformulation transformations that require the addition of extra operations are performed. This phase ordering was found after extensive testing to yield the highest performance and the fact that the performance of so few control blocks are hurt by this further supports this as the proper choice.

For the heuristic-based, dependence-based, and estimate-based approaches, applying renaming with copy or expression reformulation can degrade performance instead of improving it. As with the schedule-based approach, the effect of performing both transformations is not always simply a summation of the individual performance effects. If both transformations individually degraded control block performance because of overutilization of processor resources, performing both transformations is likely to degrade performance even more than the sum of the two individual effects. When applied individually, the extra operations inserted by each transformation can be partially absorbed by the available processor resources. When both transformations are applied, the full penalty of one transformation will be exposed.

If one transformation individually improves performance and the other one individually degrades performance, applying both transformations is almost equally likely to yield higher, equivalent, or lower performance than the sum of the individual effects. The extra operations added by both transformations sometimes cause extra performance degradation. However, the extra scheduling freedom caused by applying both transformations sometimes allows more operations to be absorbed. There were several control blocks where the effect of the two transformations simply canceled out. As a result, no clear trends can be drawn from the experimental data for these cases.

4.10 Overall Results

This section presents the overall change in benchmark performance due to the application of renaming with copy, expression reformulation, and both transformations utilizing each of the four application approaches. The control-block level results presented in the last three sections show the effect of the various approaches across all the control blocks, independent of how frequently they are executed (see Section 4.2 for the selection criteria). The overall results show the effect of the various approaches on overall performance, which is typically determined by a few, frequently executed control blocks in each benchmark. These frequently executed control blocks are typically unrolled loops that have had aggressive loop ILP transformations applied by the IMPACT compiler such as induction-variable expansion, accumulator-variable expansion, operation folding, and global register renaming [4]. These loop ILP transformations break practically all of the register anti and output dependences between different iterations of the unrolled loop body. In addition, these transformations reformulate most of the expressions involving induction variables and simple accumulation variables in order to break the register flow dependences between different iterations of the unrolled loop body. As a result, most of the large performance gains possible from breaking these types of dependences have already been captured by IMPACT's loop ILP transformations.

The opportunities that remain are significantly smaller in terms of possible performance improvement. Therefore, because of the construction of these experiments (using a high-performance, state-of-the-art ILP transformed code base), the overall performance improvements shown for the benchmarks for these two transformations will be relatively small. However, as additional ILP transformations are implemented in this schedule-time transformation framework, the benefit of the schedule-based approach will continue to grow. Two additional ILP transformations, memory speculation [9], [27] and branch combining [4], appear to be especially promising. After applying renaming with copy and expression reformulation, memory addresses and branch directions can be calculated much earlier in the control block than before which significantly increases the benefit of these additional ILP transformations. Implementing these two additional ILP transformations is part of the future work that will be done with other members of the IMPACT team.

As with the control-block-level analysis, these comparisons are based on a first-order estimate of benchmark performance using profile information and scheduled height of each control block. This first-order approximation allows low-noise comparisons by filtering out second-order effects due to cache misses, branch mispredictions, and register allocation spill decisions. Although these second-order effects are important in other contexts, all of the evaluated approaches are concerned solely with the reduction of schedule height. Therefore, this first-order performance approximation which measures the weighted reduction of schedule height is a more appropriate evaluation metric.

4.10.1 Overall results for the four-issue processor configuration

The overall percentage speedup (or slowdown) for the four-issue processor configuration is shown in Figure 4.18. The results for renaming with copy and expression reformulation individually are shown in Figure 4.18(a) and Figure 4.18(b) respectively. The results for both transformations are shown in Figure 4.18(c). For each

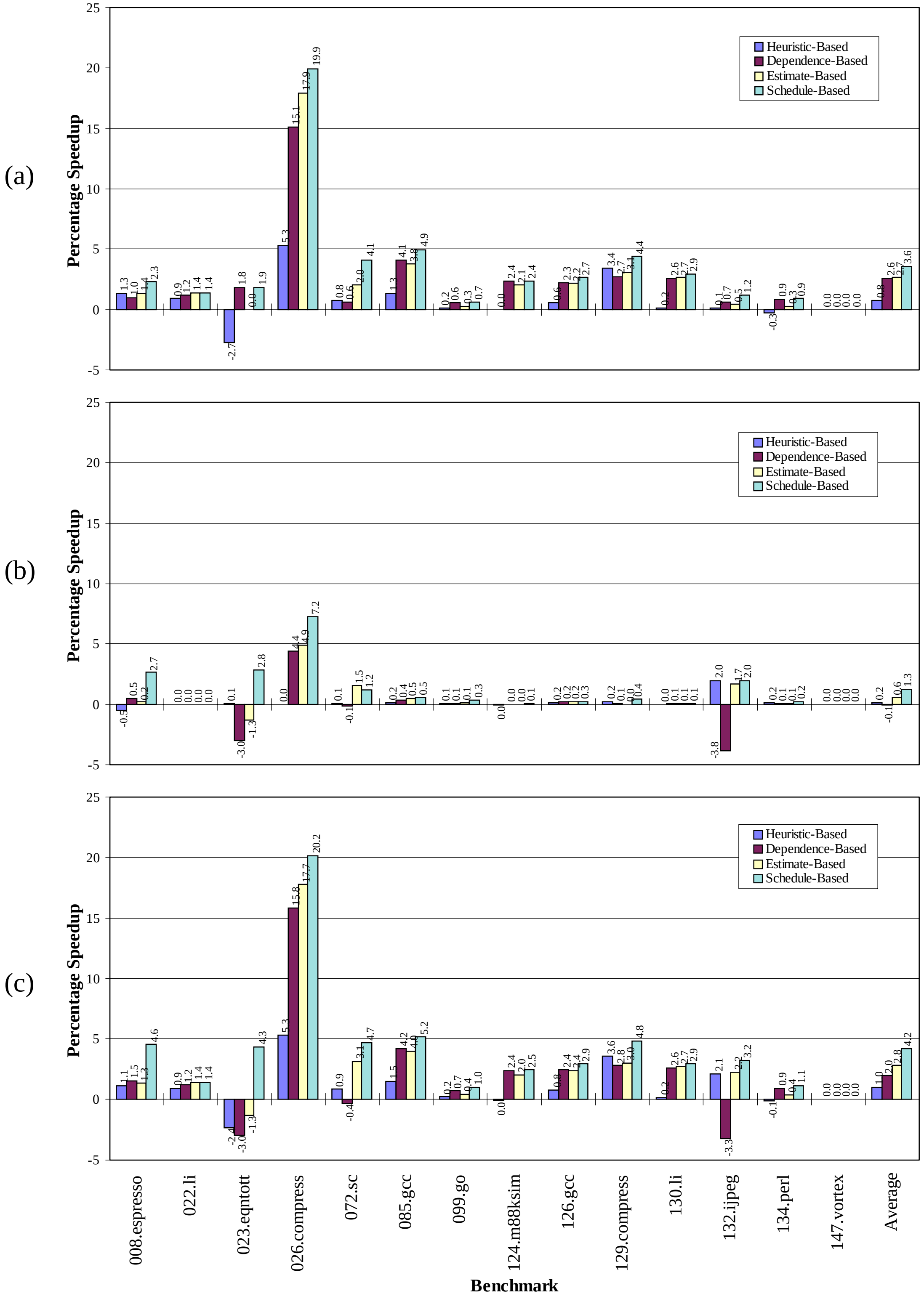


Figure 4.18: Speedup due to the application of (a) renaming with copy, (b) expression reformulation, and (c) both transformations for the four-issue processor configuration.

SPECcint92 and SPECcint95 benchmark, four speedup bars are shown. From left to right, these bars specify the speedup measured for the heuristic-based, dependence-based, estimate-based, and schedule based approaches for applying transformations. In addition, for each of these four approaches, an unweighted arithmetic mean of the speedups across all of benchmarks is shown at the far right.

As described earlier, the heuristic-based approach for applying renaming with copy does not consider dependence or resource constraints in the application decision. The drawback to ignoring these constraints is seen in the slowdown of 023.eqntott and 134.perl in Figure 4.18(a). Renaming with copy was applied too many times to the frequently executed control blocks of these benchmarks, resulting in overutilization of processor resources. Overall, the heuristic-based approach improved performance an average of 0.8%. Although the dependence-based approach does not consider resource constraints, it does attempt to minimize the number of transformations performed by only performing transformations that reduce the control block's dependence height. This results in much better performance overall (2.6%) including on 023.eqntott and 134.perl, but the heuristic-based approach actually does better than the dependence-based approach on 008.espresso and 129.compress. This again illustrates that dependence-height is not the only factor to consider when applying a transformation.

The estimate-based approach adds an approximation of the processor's resource constraints into the application decision, resulting in slightly better overall performance (2.7%). However, the estimate-based approach is sometimes too conservative (such as 023.eqntott and 134.perl), resulting in less speedup than the dependence-based approach. For 129.compress, the estimate-based approach does not even do as well as the heuristic-based approach. In general using an estimate of processor resources helps, but even a fairly accurate estimate (as used here) does not always allow better decisions to be made.

The schedule-based approach uses exact schedule information to guide transformation application. This results in a significantly better overall performance of 3.6%, more than four times the 0.8% average benefit from the heuristic-based approach. For renaming with copy, the schedule-based approach always yields the best performance for each benchmark. This is especially true of 026.compress, where the schedule-based approach improves performance 19.9% versus the heuristic-based approach's 5.3%. Although dependences are the primary factor limiting performance in 026.compress, the schedule-based approach also performs better than the dependence-based (15.1%) and estimate-based (17.9%) approaches. The reason the performance improvement is so much larger for 026.compress is that IMPACT's ILP loop transformations were not able to break cross-iteration register anti and output dependences in several important loops. If IMPACT's ILP loop transformations were not applied to the important loops of other benchmarks, similar performance benefits would be seen for them as well.

The overall results for just expression reformulation for the four-issue processor configuration is shown in Figure 4.18(b). The heuristic-based approach is very conservative, only performing reformulations that do not add any operations. Even without adding operations this approach speeded up 132.ijpeg 2.0%. However, it also slowed down 008.espresso 0.5% because the reduced dependence heights caused the scheduler to make poor decisions regarding side exits. As discussed before, the speculative hedge scheduling heuristic should eliminate this slowdown. The overall performance improvement is just 0.2%.

The dependence-based approach is much more aggressive, performing expression reformulations whenever dependence-height is reduced. This results in a 4.4% speedup for 026.compress but also a 3.8% slowdown for 132.ijpeg and a 3.0% slowdown for 023.eqntott. Expression reformulation has a much higher cost to benefit ratio than renaming with copy, making it much easier to overapply. In addition, unlike the implementation for renaming with copy, the framework does not minimize the number of transformations performed, which forces the dependence-based approach to pay for all of the poor decisions made. On average, the dependence-based approach for expression reformulation slows down the overall performance by 0.1%.

The estimate-based approach adds some restraint to the dependence-based approach, preventing much of the dependence-based-approach's overutilization of resources. This restraint allows the estimate-based approach to improve 026.compress more (4.9%) than the dependence-based approach and to actually improve 132.ijpeg 1.7% instead of degrading it. In fact, for 072.sc, the estimate-based approach (1.5%) even outperforms the schedule-based approach (1.2%). As mentioned in the control-block-level analysis, the linear search algorithm sometimes finds lower-performance solutions than is desired. Although this can happen whenever an exhaustive search is not performed, in these experiments most of the problems are due to the less-than-ideal handling of side exits by the DHASY scheduling heuristic.

However, even with this restraint the estimate-based approach still degrades 023.eqntott 1.3% because of the inaccuracy of the estimate. In addition, the conservative nature of this approach reduces the speedup for 008.espresso and 129.compress. Overall, the estimate-based approach speeds up the benchmarks by an average of 0.6%.

The schedule-based approach significantly improves the performance more than the other approaches for 023.eqntott (2.8%), 008.espresso (2.7%), and 026.compress (7.2%). On average, it speeds up the benchmarks by 1.3% compared to the 0.2% of the heuristic-based approach. Although it would have been nice to see more overall improvements, the control-block-level results have already shown that expression reformulation can significantly improve performance. The ILP loop transformations performed on the code prior to applying these transformations have just managed to capture most of the big wins in the frequently executed loops. From examining the frequently executed control blocks in these benchmarks, extensive expression reformulation was still performed, allowing many addresses and branch directions to be calculated significantly earlier. The transformations to take advantage of this (e.g., memory speculation and branch combining) just have not been implemented yet and other bottlenecks are limiting performance. Once these other bottlenecks have been addressed, the overall performance benefits will then become more tangible.

The overall results after performing both transformations are shown in Figure 4.18(c). The sum of the effects of the two transformations individually approximates the effect of both transformations together, with a notable exception for the benchmark 026.compress. Two factors can prevent the effects for transformations performed individually from adding up. The first factor is that after performing a transformation, performance is often limited by resource constraints (such as branch resources) or other dependences not targeted (such as memory dependences) so that reducing dependence height further with the other transformations is not of much immediate benefit (see discussion in Section 4.9.1).

The second factor is that expression reformulation often performs renaming with copy during the reformulation, and some of this transformation's benefit may be partially from the renaming with copy portion of the transformation. The effect of the individual portions of a transformation are difficult to quantify because the portions are dependent on each other (performing one portion may not be beneficial or possible without the other portions). In addition, from examining the important control blocks of all the benchmarks, the expression reformulations were almost always solely responsible for the performance improvement. The extra operation inserted by renaming with copy was usually part of the cost, not the benefit, of performing this transformation. However, in the case of 026.compress, the renaming with copy was almost always required to be performed in order to perform expression reformulation. In this particular benchmark, performing renaming with copy almost always helps performance, so some portion of the expression reformulations benefit is due to the application of renaming with copy. However, both factors (other constraints and implicit application of renaming with copy) cause the individual benefits of both transformations for 026.compress to not add up at all.

Overall, the schedule-based approach improved the benchmarks on average 4.2%, compared with the 1.0% for the heuristic-based approach. The use of exact schedule information allowed these two transformations to be performed more precisely, tailoring the application of these transformations to the processor's execution constraints. This allows more than four times the amount of improvement from just these two ILP transformations. As more ILP transformations are applied using the proposed schedule-based approach, the benefit to overall performance will continue to grow.

The average overall improvement of 2.0% for the dependence-based approach shows that part of the benefit of the schedule-based approach is from applying only those transformations that actually reduce dependence height (or schedule height). This reduces the number of transformations that are applied, which helps reduce resource overutilization. However, the dependence-based approach is not sufficient, because depending on the processor model, resources may still be overutilized (as in 023.eqntott and 132.ijpeg).

The estimate-based approach approximately models the processor resources, allowing the transformations to be scaled back as processor resources are reduced. For the four-issue processor configuration, the estimate-based approach improved overall performance an average of 2.8%. This shows that part of the benefit of the schedule-based approach is its ability to prevent overutilization of resources. The estimate-based approach performed better overall than the dependence-based approach because it reduced the overutilization of resources in several benchmarks. However, this approach did not capture the dependence-based approach's performance in several benchmarks, such as 134.perl and 008.espresso, because of its conservative nature.

The schedule-based approach applies transformations only when schedule height improves, allowing fine-level trade-offs to be made between reducing dependence constraints and adding extra operations. In addition, only the schedule-based approach is able to take advantage of the opportunities created by scheduling decisions. This allows the schedule-based approach to capture significantly more performance when the program's instruction-level parallelism is well matched to the processor's execution constraints.

4.10.2 Overall results for the eight-issue processor configuration

As processor resources increase relative to the program's instruction-level parallelism, applying transformations whenever they can reduce dependence height becomes more important than preventing resource overutilization. The extra processor resources can absorb many more poor application decisions, minimizing the damage. In addition, the benefit for correct application decisions is significantly increased, because the processor can take more advantage of the increased instruction-level parallelism.

The overall results for the eight-issue processor configuration, shown in Figure 4.19, illustrates these points. Although the exact same set of transformations is applied for the heuristic-based and dependence-based approaches (which do not use processor configuration information), the average overall performance benefits for both transformations (Figure 4.19(c)) are almost triple those shown for the four-issue processor configuration (Figure 4.18(c)). For both renaming with copy (Figure 4.19(a)) and expression reformulation (Figure 4.19(b)) individually, almost all the benchmarks are improved by these approaches. The one exception is that renaming with copy, utilizing the heuristic-based approach, still manages to degrade 134.perl 0.3% by continuing to overutilize resources.

The estimate-based approach loses most of its advantage over the dependence-based approach because resource overutilization is rarely a problem for this processor configuration. In fact, the estimate-based approach's conservative nature causes it to miss several beneficial renaming with copy opportunities in many of the benchmarks. However, for expression reformulation the estimate-based approach performed

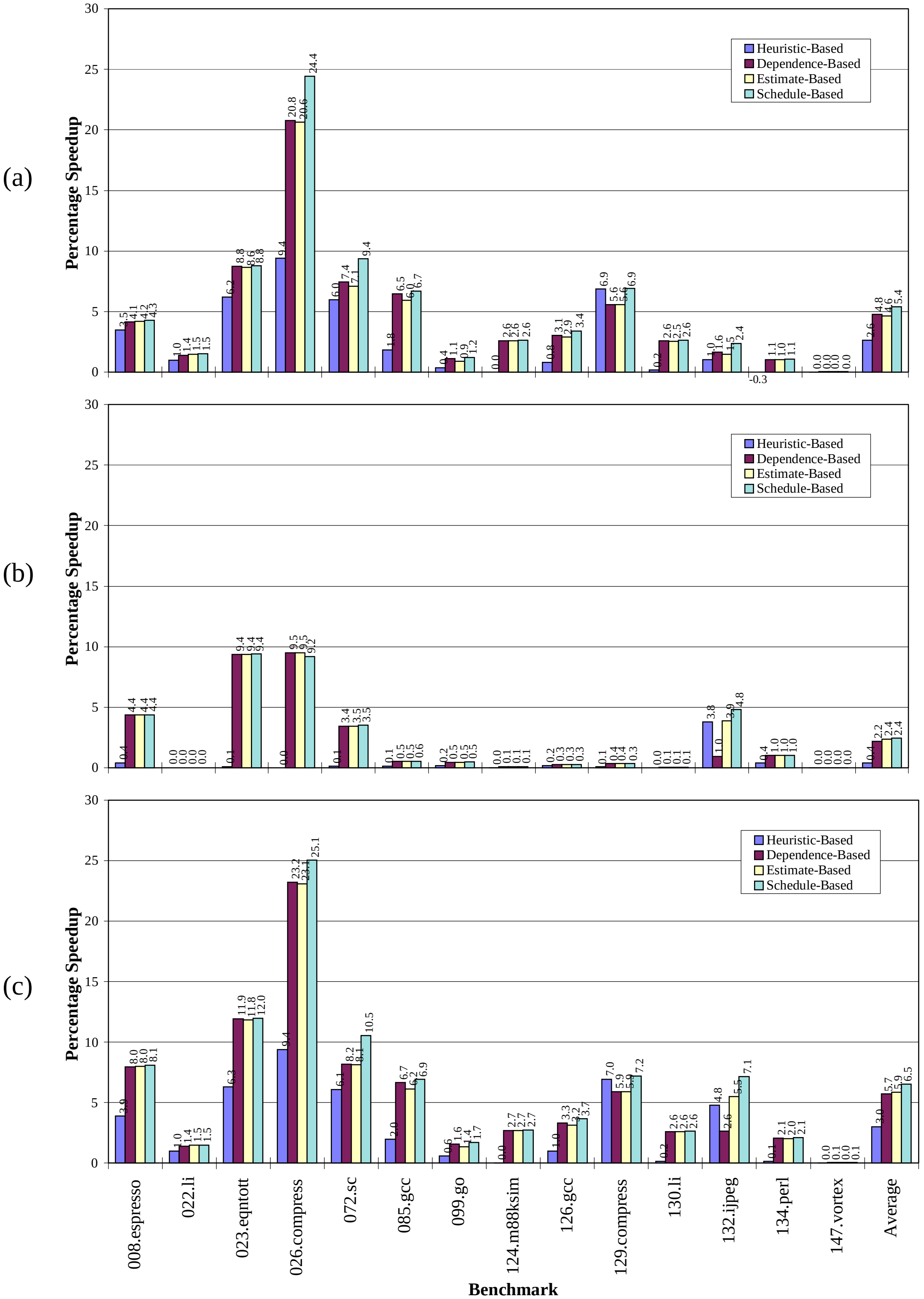


Figure 4.19: Speedup due to the application of (a) renaming with copy, (b) expression reformulation, and (c) both transformations for the eight-issue processor configuration.

significantly better than the dependence-based approach for 132.ijpeg. In several important loops in 132.ijpeg, the instruction-level parallelism is high and preventing resource overutilization is still important.

Overall for both transformations combined, the schedule-based approach improved the benchmarks on average 6.5%, compared to the heuristic-based approach's 3.0%. For most of the benchmarks, the schedule-based approach captured about twice as much performance improvement as the heuristic-based approach. The dependence-based (5.7%) and estimated-based (5.9%) approaches performed almost as well on average as the schedule-based approach for this eight-issue processor configuration. This is because, with the low instruction-level parallelism compared to processor resources, poor application choices do not affect performance as much. As compiler technology improves, the amount of instruction-level parallelism relative to processor resources will increase, causing the results to become more like the four-issue processor configuration.

4.10.3 Overall results for the two-issue processor configuration

As the instruction-level parallelism increases relative to processor resources, preventing resource overutilization becomes the dominant factor for applying ILP transformations. Because of scarce resources, the penalty of adding unnecessary operations can become severe. In addition, the benefit of reducing dependence height is reduced because there are fewer unutilized resources to take advantage of the increased parallelism.

The overall results for the two-issue processor configuration are shown in Figure 4.20. As before, the set of transformations applied by the heuristic-based and

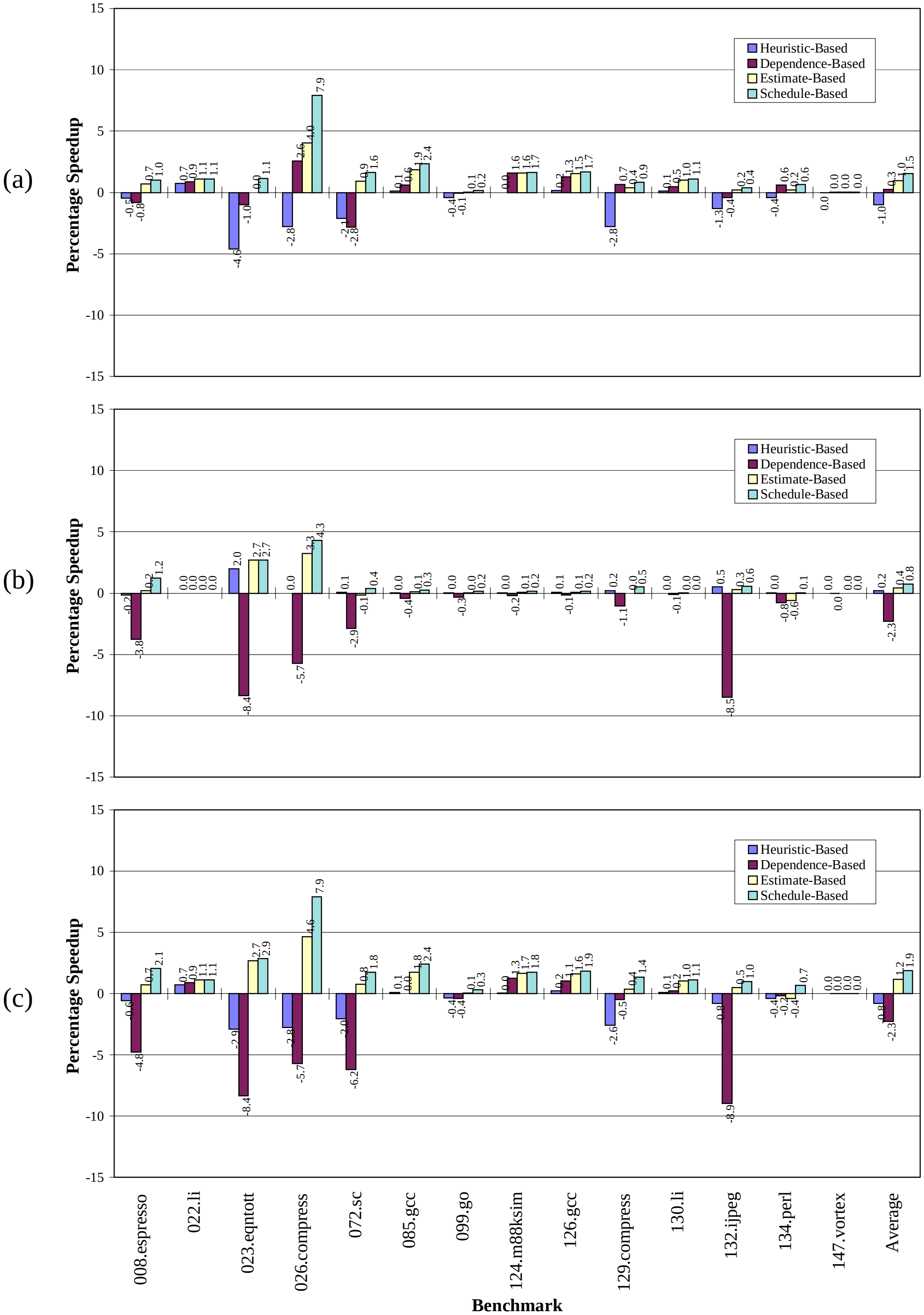


Figure 4.20: Speedup due to the application of (a) renaming with copy, (b) expression reformulation, and (c) both transformations for the two-issue processor configuration.

dependence-based approaches are independent of the processor configuration. As a result, both approaches degrade several benchmarks by overapplying renaming with copy (Figure 4.20(a)). The heuristic-based approach does not perform expression reformulations (Figure 4.20(b)) that add extra operations, so it still performs fairly well for most benchmarks. However, the dependence-based approach does not have this restraint, causing expression reformulation to degrade most of the benchmarks it is applied to. Especially notable are the degradation of 132.ijpeg (8.5%) and 023.eqntott (8.4%). For both transformations combined, the dependence-based approach slows down 072.sc (6.2%) and 008.espresso (4.8%) more than the individual slowdowns added together.

The estimate-based and schedule-based approaches, on the other hand, select the transformations to apply utilizing processor configuration information. As a result, both approaches continue to perform well with the reduced resources. The results for both transformations (Figure 4.20(c)) show that these two approaches significantly improve the performance of 026.compress, 023.eqntott, and 008.espresso whereas the other two approaches degrade performance. Overall, the schedule-based approach improves the benchmarks an average of 1.9% compared to the heuristic-based approach's slowdown of 0.8%. As with the four issue processor configuration, the estimate-based approach captures about two-thirds the performance of the schedule-based approach with an average improvement of 1.2%. Unlike the four-issue processor configuration, the dependence-based approach slows down the benchmarks on average 2.3%. The schedule-based approach still manages to yield significant performance from ILP transformations even when processor resources are limited.

4.10.4 Overall results for the one-issue processor configuration

The one-issue processor configuration represents one extreme, where processor resources are the limiting factor. ILP transformation that add operations such as renaming with copy are typically not performed. However, breaking dependences can still increase performance because there are often holes in the schedule that can be filled because of nonunit latency instructions such as loads. The overall performance results for the one-issue processor configuration are shown in Figure 4.21.

The heuristic-based and dependence-based approaches overapply renaming with copy (Figure 4.21(a)) so, primarily, slowdowns occur. The estimate-based approach has been configured to discourage the addition of any extra operations. As a result, it almost never applies renaming with copy, which is the better default decision when compared to the first two approaches. The schedule-based approach, however, does find that applying renaming with copy in a controlled manner can sometimes improve the benchmark's performance.

For expression reformulation (Figure 4.21(b)), the heuristic-based, estimate-based, and schedule-based approaches all tend to apply only "free" expression reformulations, which do not add any operations. However, the schedule-based approach does find that adding a few operations in order to do expression reformulation can improve the performance of 026.compress. The dependence-based approach, on the other hand, applies expression reformulation aggressively, resulting in a slowdown for every benchmark.

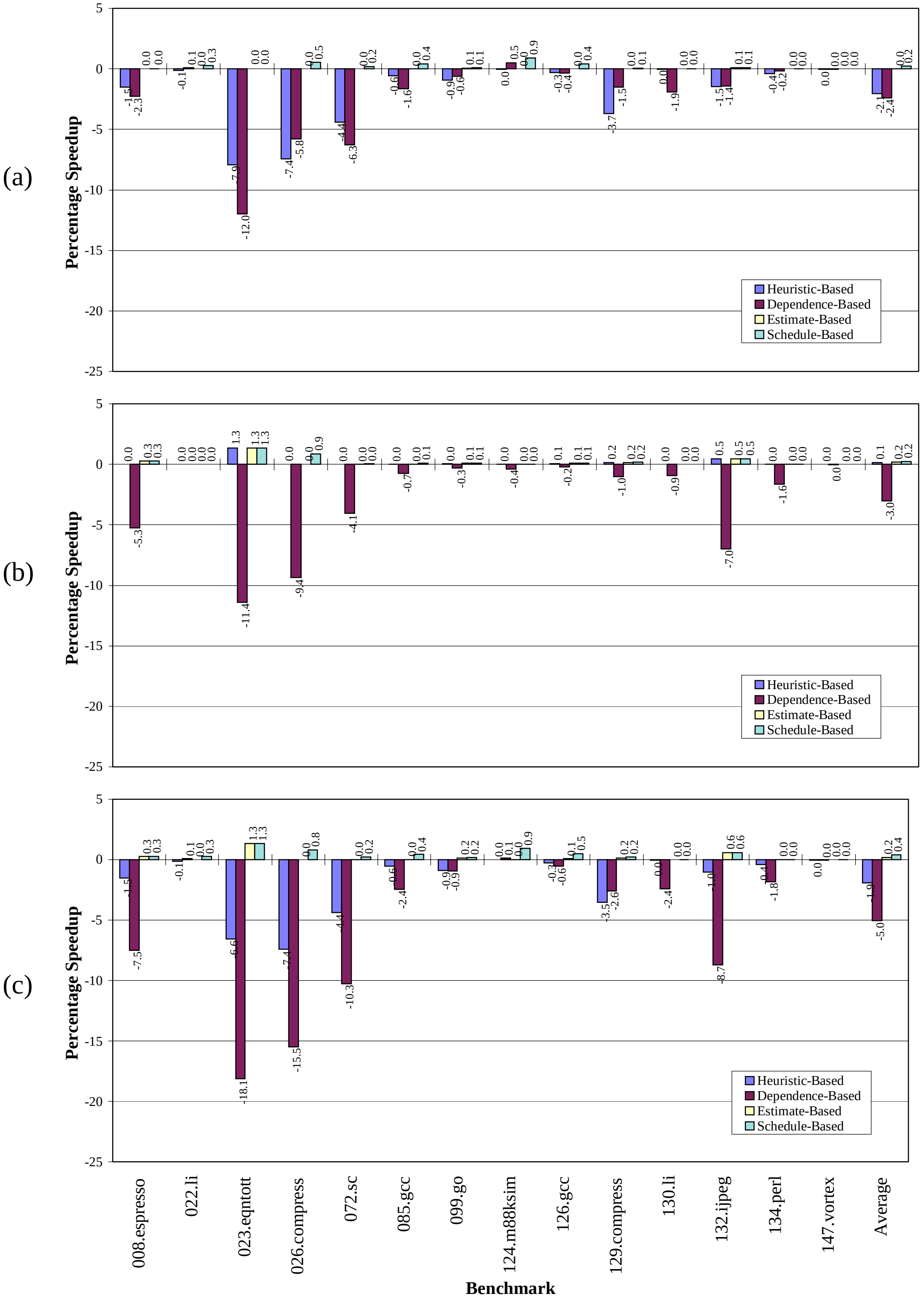


Figure 4.21: Speedup due to the application of (a) renaming with copy, (b) expression reformulation, and (c) both transformations for the one-issue processor configuration.

After applying both transformations, the schedule-based approach on average improves overall performance 0.4%, compared to the heuristic-based approach's slowdown of 1.9%. The estimate-based approach also manages to improve performance 0.2%, primarily from "free" expression reformulations. The dependence-based approach, however, slows down the benchmarks on average a massive 5.0%. For 023.eqntott, the dependence-based approach managed to cause an 18.1% slowdown.

5. INCREMENTAL DEPENDENCE GRAPH GENERATION

5.1 Introduction

The proposed algorithm for applying transformations at schedule time requires the targeted control block to be rescheduled after each transformation is applied. This rescheduling of the control block after each transformation is responsible for almost all of the compile-time cost of this algorithm. One of the most time-consuming steps of the DHASY list-scheduling algorithm [23] (used for this thesis's experiments) is building the control block's dependence graph [1], which represents all the control block's dependence constraints as described in Section 3.3. Because applying a transformation usually affects a relatively small part of the dependence graph, incrementally updating the dependence graph (instead of just rebuilding it) before rescheduling the control block can significantly reduce the compile-time cost of the proposed schedule-time transformation algorithm. However, most dependence graph generation algorithms do not support incremental update because this ability is not, in general, required during scheduling. (Support for control speculation and global scheduling do involve some limited update of the dependence graph but typically this support does not allow operations to be added after the dependence graph is built.) This chapter describes a new dependence graph building algorithm that allows both the efficient building of the initial dependence graph for a control block and efficient general-purpose incremental update of the dependence graph after the application of a code transformation. The key features include the register action table which allows uses and definitions of a particular register to be quickly located (Section 5.2) and dependence-drawing algorithms that can be used for both building and incrementally updating the dependence graph (Section 5.3). The compile-time benefit of allowing incremental update of the dependence graph will be experimentally evaluated in Chapter 7.

5.2 The Register Action Table

After a transformation creates (or modifies) an operation, dependences may need to be added between this operation's register operands and any of the other operations' register operands in the control block. Although a straightforward linear search over all the operations' operands in the control block to determine where to add register dependences would work, it would also be inefficient because an unnecessarily large number of operands would need to be examined. Instead, the proposed technique utilizes a register action table to allow the definitions and uses of a particular register to be efficiently located. This register action table will then be used by the dependence drawing algorithms in Section 5.3 to efficiently draw the appropriate dependences.

The *register action table* contains an *action entry* for every register definition or use in the control block. An example code segment and the register action table summarizing this code segment is shown in Figure 5.1. The notation "r1 <-" indicates an explicit definition of Register r1 and "<- r1" indicates an explicit use of Register r1 by the operation shown in the first column. The notation "(<- r1)" indicates an implicit use of Register r1 by the operation shown in the first column. (Implicit definitions are also possible, primarily for function calls, but are not shown in the figure.) Note that almost all of the Register actions in the register action table will be explicit because the IMPACT compiler's internal representation explicitly represents all the registers defined and used

	r1 Actions	r2 Actions	Control Actions	Memory Actions
op 1: beq r2, 0, exit1	(<- r1)	<- r2 (<- r2)	<- C C <-	
op 2: r1 <- M[r2+8]	r1 <-	<- r2	<- C	<- M
op 3: r1 <- r1 - 1	<- r1 r1 <-		<- C	
op4 : M[r2 + 8] <- r1	<- r1	<- r2	<- C	M <-
op 5: beq r1, 0, exit2	<- r1	(<- r2)	<- C C <-	
op 6: r2 <- M[r2+12]		<- r2 r2 <-	<- C	<- M
op 7: jsr func2			<- C C <-	M <-

Figure 5.1: An example code segment and its register action table.

by an operation, including those registers implicitly defined and used by the operation's opcode. Implicit register actions are used only to specify the registers whose values are used along the taken path (liveout) of a conditional blanch and the registers that may be used or modified by the target of a function call. For example, Registers r1 and r2 are liveout of op 1, so implicit uses of these registers are placed in the register action table as shown in Figure 5.1. Op 1 also explicitly uses Register r2, so an explicit use of r2 is also shown in the register action table.

5.2.1 Designed to efficiently answer four time-critical queries

In order to be able to determine where to add dependences for a new operation or a new register operand in near-constant time, there are four queries about a particular action entry that the register action table was designed to answer in constant time. The first query, *prev_def*, returns the action entry (if any) that represents the previous definition of the same register; otherwise it returns NULL. For example, if given the action entry representing the use of r1 by op 5 (in Figure 5.1), the prev_def query will return the action entry representing the definition of r1 by op 3. The second query, *next_def*, is similar except it returns the next definition (if any). The next_def query will return NULL if given the action entry representing the use of r1 by op 5. In addition, if an operation both uses and defines the same register, the use action entry is defined to occur before the definition action entry. Therefore, if the next_def query is given the action entry representing the use of r1 by op 3, it will return the action entry representing the definition of r1 by op 3.

The third query, *prev_actual*, returns the action entry (if any) that represents the previous definition or use of the same register, otherwise it returns NULL. If the prev_actual query is given the action representing the use of r1 by op 5, the action representing the use of r1 by op 4 will be returned. The fourth query, next_actual, is similar except it returns the next definition or use (if any).

5.2.2 Handling of overlapping registers

Usually dependences are only drawn between action entries for the same exact register. For example, register anti dependences (see Section 2.2) are typically only drawn between the uses of Register r1 and the next definition of Register r1. However, it is sometimes necessary to draw dependences between actions of two different registers, if these two registers overlap in some way. For example, after register allocation two single precision floating-point registers (named r1f and r2f, 32 bits each) might both overlap one double precision floating-point register (named r1d, 64 bits) as shown in Figure 5.2(a). If an use of Register r2f (Operation B) is followed by a definition of r1d (Operation C), a register anti dependence must be drawn between the use of Register r2f and the definition of Register r1d to prevent them from reordering, as shown in Figure 5.2(b). In order to handle overlapping registers efficiently, extra *overlapping action entries* are added to the register action table, denoted with square brackets in Figure 5.2(c). For clarity, the normal action entries (versus the overlapping action entries) are referred to as *actual action entries*. When an actual action entry is added to the register action table for a particular register, an overlapping action entry is also created for each register it overlaps as shown in Figure 5.2(c). (The actual action entry and all the overlapping action entries created are enclosed by a dotted box in this figure.) These overlapping entries (called conflicting entries in the IMPACT compiler's terminology) and the original actual entry itself is placed in a *conflict* list in the actual entry for use by the dependence drawing algorithms. Section 5.3.6 will describe how this conflict list is utilized to draw the appropriate dependences between overlapping registers.

Note that the prev_def, next_def, prev_actual, and next_actual queries will return only actual action entries or NULL (if appropriate). (This is the rationale behind the prev_actual and next_actual names). However, these queries may be made on an overlapping action entry, in which case the actual action returned will be determined by treating the overlapping action entry as if it was an actual action entry for the register it overlaps. For example, if the "[<- r2f]" overlapping action entry in the Register r1d column in Figure 5.2(c) is given to next_def, the "r1d <-" actual action entry will be

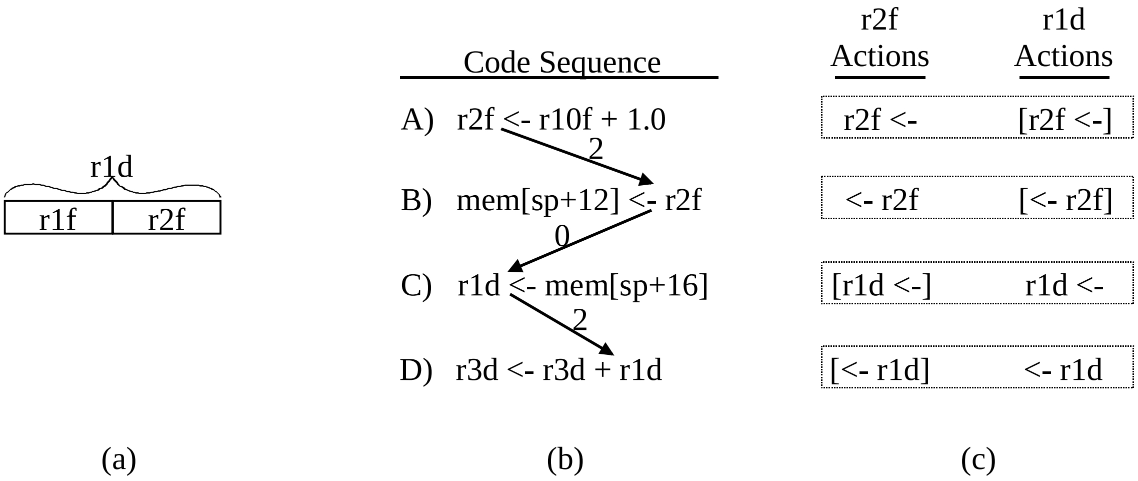


Figure 5.2: Example of (a) overlapping registers, (b) the resulting dependences and (c) their action entries.

returned. In addition, if the "<- r2f" actual action entry is given to next_def, NULL will be returned because there are no actual action entries after it.

5.2.3 Control, memory, and synchronization pseudoregister operands

Up to this point, the focus has been on action entries that facilitate the drawing of register-to-register dependences. Although the register action table was designed primarily to facilitate the drawing of register dependences, it also provides a convenient way of keeping track of those operations that might need control, memory, and synchronization dependences drawn to them. This is done by creating control, memory, and synchronization actions. These actions are handled seamlessly by the register action table by creating three pseudoregister operands (one for each type). These three pseudoregister operands are treated by the register action table as if they were three real register operands. By definition, these three pseudoregister operands will not overlap with any other register operand.

The register action table can be used to quickly locate branch operations by modeling these operations in the action table as a "definition" of the control operand

(C <-). All the operations in the control block can then be modeled in the action table as a "use" of the control operand (<- C). This provides a quick method for determining which branches are above and below any operation through the use of the prev_def and next_def queries. Control dependences are drawn from branch operations to other operations that should not be moved past the branch operation. These are typically other branch operations (to maintain correct control flow), store operations (to maintain the proper memory state), and operations that can throw exceptions such as load operations (invalid addresses) and divide operations (divide by zero). However, control dependences are sometimes drawn to all operations to prevent compile-time control speculation. To support this option, all operations are always modeled as using the control operand. The processors modeled in this thesis support nonexcepting versions of operations that can throw exceptions and compile-time control speculation is allowed, so control dependences are used primarily to prevent the reordering of branch and store operations. An example of how these definition and use control action entries are added is shown in the fourth column of Figure 5.1.

Memory dependences are used to keep load operations from reordering with store operations that may modify the memory value that the load operation is accessing. To allow the register action table to be used to quickly locate store operations, action entries that define the "memory" operand (M <-) are added for store operations. For quick location of load operations, action entries that use the memory operand (<- M) are added for load operations. An example of how these memory action entries are added are shown in the last column of Figure 5.1. Some sort of memory disambiguation [28] must be used to determine which memory dependences need to be drawn.

Synchronization dependences are used when nothing should be moved past a specific operation. These operations are typically operations that access the processor control word or move the processor's register window (on the Sparc architecture [3]). The synchronization operand (not shown in the figure) is used in a way similar to the control operand, except operations requiring synchronization dependences define the "synchronization" operand. The experimental processor configurations used in this thesis do not have any operations that require these synchronization dependences.

5.2.4 Precalculation of the results for the four time-critical queries

In order to be able to answer the four time-critical queries (prev_def, next_def, prev_actual, next_actual) in constant time, the results for these queries are precalculated as the register action table is built and stored in each action entry as pointers to the appropriate action entry for immediate access. For each register (e.g., Register r1) in the control block, there is a *register information entry* that manages all of the action entries for that register in the form of one normal doubly-linked list and two enhanced doubly-linked lists. The normal linked list is simply a doubly-linked list of all (both overlapping and actual) action entries for that register in the order that they occurred in the unscheduled control block. Each action entry has two fields, *prev_complete* and *next_complete*, that are used to connect the action entry to this list. The register information entry has two fields, *first_complete* and *last_complete*, that point to the head and tail of this list.

The first enhanced doubly-linked list contains all the actual action entries for a register in the order that they occurred in the unscheduled control block. Each action entry has two fields, *prev_actual* and *next_actual*, that are used to connect actual action entries to this list. The register information entry has two fields, *first_actual* and *last_actual*, that point to the head and tail of this list. This list is enhanced because overlapping action entries also have their prev_actual and next_actual fields set as if they were in the actual action linked list. The algorithm used to construct these enhanced lists will be described below. This enhancement allows these two fields to be used to efficiently answer the prev_actual and next_actual queries on any action entry.

The second enhanced double-linked list contains all of the actual definition action entries for a register in the order that they occurred in the unscheduled control block. Each action entry has two fields, *prev_def* and *next_def*, that are used to connect the action entry to this list. The register information entry has two fields, *first_def* and *last_def*, that point to the head and tail of this list. This list is enhanced because overlapping and actual use action entries also have their prev_def and next_def fields set as if they were in the actual definition action linked list. As with the actual action enhanced linked list, this allows these two fields to be used to efficiently answer the first_def and last_def queries on any action entry.

An algorithm that can be used for building these enhanced linked lists is shown in Figure 5.3. A forward pass through the complete action linked list is made to calculate the values of prev_actual and prev_def fields for each action entry and to calculate the values of last_actual and last_def fields for the register information entry. Similarly, a backward pass through the complete action linked list is made to calculate the values of the next_actual, next_def, first_actual and first_def fields. As shown, the value of all these fields can be calculated in linear time with respect to the number of action entries in the complete list. Amortized over the complete list of action entries, precalculating these fields' values requires constant time. In order to answer the four time-critical queries discussed earlier, only a constant-time loading of each field's value is required.

```
build_enhanced_linked_lists (reg_info_entry)
{
    prev_actual = NULL;                    // No actual entries before first action entry.
    prev_def = NULL;                       // No actual definitions before first action entry.

    // Make a forward pass through the complete action entry list for prev fields.
    for (action_entry = reg_info_entry->first_complete; action_entry != NULL;
         action_entry = action_entry->next_complete)
    {
        action_entry->prev_actual = prev_actual;      // Set prev_actual field.
        action_entry->prev_def = prev_def;            // Set prev_def field.

        if (is_actual_entry (action_entry))          // Update prev_actual if actual action.
            prev_actual = action_entry;

        if (is_actual_def_entry (action_entry))   // Update prev_def if actual definition.
            prev_def = action_entry;
    }

    reg_info_entry->last_actual = prev_actual;         // Set last_actual field.
    reg_info_entry->last_def = prev_def;               // Set last_def field.


    next_actual = NULL;                    // No actual entries after last action entry.
    next_def = NULL;                       // No actual definitions after last action entry.

    // Make a backward pass through the complete entry list for next fields.
    for (action_entry = reg_info_entry->last_complete; action_entry != NULL;
         action_entry = action_entry->prev_complete)
    {
        action_entry->next_actual = next_actual;      // Set next_actual field.
        action_entry->next_def = next_def;            // Set next_def field.

        if (is_actual_entry (action_entry))         // Update next_actual if actual action.
            next_actual = action_entry;

        if (is_actual_def_entry (action_entry))  // Update next_def if actual definition.
            next_def = action_entry;
    }

    reg_info_entry->first_actual = next_actual;        // Set first_actual field.
    reg_info_entry->first_def = next_def;              // Set first_def field.
}
```

Figure 5.3: Algorithm for building enhanced linked lists.

Although the algorithm described above can be done in linear time, the entire process of building the register action table takes near-linear time because a hash table lookup is required for each action entry to determine which register information entry to add the action entry to. This hash table lookup could take worst-case linear time to perform, but in practice takes near-constant time. (The register id distribution is very well-behaved.) Therefore in practice building the register action table for a control block takes near-linear time to build.

5.2.5 Incremental update of the register action table

When transformations are performed, the register action table must be updated. The algorithm used to delete an action for the register action table is shown in Figure 5.4. Before deleting an action entry from the complete action list, the prev_actual, next_actual, prev_def, and next_def fields of surrounding action entries must be updated. If the action entry to be deleted (action_to_delete) is an actual definition action entry, the surrounding action entries' prev_def and next_def fields must be updated. Fortunately, the values to set these fields to can be determined by simply examining the fields of action_to_delete. The action_to_delete's prev_def field points to the previous action definition (if any), so prev_def field of the entries below the action_to_delete should just be set to this value (as shown). Once the another actual definition entry is encountered (and updated), this update of the prev_def fields is stopped because the remaining entries should be correct. If another actual definition entry is not encountered (action_entry == NULL), then this was the last definition in the complete list and the register information entry's last_def field is updated. Similarly, the action_to_delete's next_def field points to

```
delete_action_entry (reg_info_entry, action_to_delete)
{
    // For actual def entries, need to update prev_def and next_def of adjacent entries.
    if (is_actual_def_entry (action_to_delete))
    {
        // Update prev_def until hit next actual definition entry (update actual def also).
        for (action_entry = action_to_delete->next_complete;  action_entry != NULL;
             action_entry = action_entry->next_complete)
        {
            action_entry->prev_def = action_to_delete->prev_def;
            if (is_actual_def_entry (action_entry)) break;  // Stop after updating actual def.
        }

        // If last definition entry, update reg_info_entry's last_def.
        if (action_entry == NULL)
            reg_info_entry->last_def = action_to_delete->prev_def;

        // Update next_def until have updated previous actual definition entry.
        for (action_entry = action_to_delete->prev_complete; action_entry != NULL;
             action_entry = action_entry->prev_complete)
        {
            action_entry->next_def = action_to_delete->next_def;
            if (is_actual_def_entry (action_entry)) break;  // Stop after updating actual def.
        }

         // If first definition entry, update reg_info_entry's first_def.
        if (action_entry == NULL)
            reg_info_entry->first_def = action_to_delete->next_def;
    }

    (Use same approach as above to update prev_actual and next_actual fields for
     adjacent entries and last_actual and first_actual fields for reg_info_entry.)

    // Remove action_to_delete from reg_info_entry's complete list and delete entry.
    delete_from_complete_action_list (reg_info_entry, action_to_delete);
}
```

Figure 5.4: Algorithm for deleting action entry from register action table.

the next action definition (if any), so the next_def field of the entries above the action_to_delete should be set to this value until another actual definition is encountered and updated (as shown). If this is the first actual definition in the complete list, the register information entry's first_def field is updated. The same approach for updating the prev_actual, next_actual, last_actual, and first_actual fields can be used and it is omitted from the figure because of space limitations. After these fields are updated, the entry is deleted from the complete action linked list by the last function call. In the worst case, two linear passes over the complete action entry list for that register information entry may be required. However, in practice only a few entries to either side of the deleted entry usually need to be updated.

The algorithm for adding an action entry to the register action table is shown in Figure 5.5. The new action entry (action_to_add) is first added to the register information entry's complete action list. This new action entry is placed in this complete action list so that the order of the action entries is the same as the order of the operation in the unscheduled control block (including the new operation). Each operation (and therefore their action entries) has a *serial number* that is used to determine the relative order of operations (and their action entries). A new operation is assigned (algorithm not shown) the appropriate serial number based on its placement in the control block and, if necessary, the adjacent serial numbers are adjusted. (Adjustments are rarely required because the serial numbers are initially spaced 1024 numbers apart.) If the new operation both uses and defines the same register, the action for the use will be placed before the action for the definition in the complete action list.

The algorithm shown in Figure 5.5 then calculates the next_def and prev_def fields for the new action entry. Instead of searching for the adjacent actual definition entries, the next_def and prev_def fields of adjacent action entries (if the adjacent entries exist) or the first_def and last_def fields of the register information entry are utilized to

```
add_action_entry (reg_info_entry, action_to_add)
{
    // Insert into the “proper” place in the complete doubly-linked list.
    add_to_complete_action_list (reg_info_entry, action_to_add);

    // Set next_def and prev_def for action_to_add using adjacent entries’ fields,
    // if adjacent entries exist.  Otherwise, use the reg_info_entry’s fields.
    if (action_to_add->prev_complete != NULL)
        action_to_add->next_def = action_to_add->prev_complete->next_def;
    else
        action_to_add->next_def = reg_info_entry->first_def;

    if (action_to_add->next_complete != NULL)
        action_to_add->prev_def = action_to_add->next_complete->prev_def;
    else
        action_to_add->prev_def = reg_info_entry->last_def;

    // If adding an actual definition, update adjacent entries’ prev_def and next_def fields.
    if (is_actual_def_entry (action_to_add))
    {
        // Update prev_def until have updated next actual definition entry.
        for (action_entry = action_to_add->next_complete; action_entry != NULL;
             action_entry = action_entry->next_complete)
        {
            action_entry->prev_def = action_to_add;
            if (is_actual_def_entry (action_entry)) break;  // Stop after updating actual def.
        }

        // If this is now the last actual definition entry, update reg_info_entry’s last_def.
        if (action_entry == NULL)
            reg_info_entry->last_def = action_to_add;

        // Update next_def until have updated previous actual definition entry.
        for (action_entry = action_to_add->prev_complete; action_entry != NULL;
             action_entry = action_entry->prev_complete)
        {
            action_entry->next_def = action_to_add;
            if (is_actual_def_entry (action_entry)) break;  // Stop after updating actual def.
        }

        // If this is now the first actual definition entry, update reg_info_entry’s first_def.
        if (action_entry == NULL)
            reg_info_entry->first_def = action_to_add;
    }

    (Use the same approach as above for updating prev_actual and next_actual for
      action_to_add and adjacent entries and for updating reg_info_entry, if necessary.)
}
```

Figure 5.5: Algorithm for adding action entry to register action table.

efficiently determine the correct values. This allows these fields to be updated in constant time. If the new action entry is an actual definition entry, then the next_def and prev_def fields of adjacent entries are updated using a similar approach to that used when deleting an entry, described above. If necessary, the first_def and last_def fields of the register information entry are also updated. In the worst case, all the action entries in the complete action list will need to be updated, yielding a linear-time update. In practice, however, only a few adjacent entries typically need to be updated.

The update of the next_actual, prev_actual, first_actual, and last_actual fields for the new entry and adjacent entries can be done using the same approach. The update of these fields is omitted from the algorithm shown in Figure 5.5 because of space constraints.

The implementation of the register action table utilized for this thesis's experiments (and everything else described in this thesis) was built from the ground up to efficiently support schedule-time transformations. Although this framework was extensively tuned to minimize the compile time required to perform various required functions, it was equally important that the framework be robust, easy to test, and easy to maintain. As a result, the same code for incrementally updating a construct (such as the register action table) was also utilized to both build and destroy these constructs whenever feasible. After extensively testing the framework for correctness, the framework was then analyzed for performance bottlenecks, including those created by utilizing the routines for incremental update to perform more mundane tasks. For example, it was found that even after extensive tuning, most of the routines for destroying these constructs were requiring an unreasonable amount of time given the net task being performed (deleting everything) and specialized routines for performing these functions were developed. However, after extensive tuning of the construction routines, the performance to be gained by developing specialized construction routines was found to be minimal. In addition, developing these specialized construction routines would require extensive reimplementation of the core framework functionality, resulting in almost a doubling of the amount of code base that must be maintained. As a result, the easier to understand and more streamlined algorithm shown in Figure 5.3 is not used in the thesis's implementation. Instead, the algorithm for incrementally adding action entries to the register action table shown in Figure 5.5 was also used to build the register action table. This algorithm was extensively tuned and provides almost equivalent overall performance when the register action table is built in a single forward pass through the operations in the control block. A specialized routine is, however, used to delete the register action table. The influence of this approach of utilizing incremental update routines whenever possible will also be apparent in the algorithms for building the dependence graph.

5.3 Building and Updating the Dependence Graph

All the algorithms for building and updating the dependence graph rely on the register action table to provide quick and accurate answers for the prev_def, next_def, prev_actual, and next_actual queries about a register action and to provide information about overlapping registers in an easy to use form. In order to access the register action entries and other information required for scheduling an operation efficiently, an *sm_op* (schedule manager operation) structure is created for each operation in the control block.

This sm_op structure contains an array of action entry pointers for each type of operand (src, dest, pred, etc.) so that the register action for a particular operand can be easily found. Implicit register actions (currently only created for branches) are kept in a separate doubly-linked list to allow easy addition and removal of implicit action entries (to support transformation and scheduling algorithms that move operations between control blocks). In addition, this sm_op structure contains a doubly-linked list of all the action entries (both explicit and implicit) for that operation, so that all the actions for an operation can be efficiently processed. (Otherwise a search of the arrays and the implicit action list would be required.) This design allows the action entries to be accessed in constant time in a manner convenient for the various dependence graph algorithms, scheduling algorithms, and code transformation algorithms implemented in this new framework.

One significant difference between the approach utilized for representing the dependence graph and traditional approaches is that dependences are drawn between the action entries causing the dependence instead of directly between two operations. This approach allows all the dependences caused by a particular operand to be quickly found, which is extremely useful when performing transformations and updating the dependence graph. For example, in order to determine if renaming with copy should be performed on an operation, the destination register action can be examined to see if 1) there are any register anti or output dependences into that register action and 2) there is at least one register flow dependence to an explicit register usage later in the control block. Unless both these conditions are met, there is no possible benefit from performing renaming with copy and there is no need to consider it further. If renaming with copy is applied, the register flow dependences out of the old destination register action can then be used to quickly find all of the register uses that should be changed to the new temporary register (see Section 2.2). After each of these old register uses are changed to the new register, the dependences for the old register will be deleted and dependences from the renamed destination will be drawn to the new register. Although it is possible to do all of the above with dependences drawn directly between operations, this approach is significantly more efficient because only relevant dependences need to be examined.

In Sections 5.3.1-5.3.5, simplified algorithms for drawing the dependences into and out of register, control, and memory actions will be presented. These algorithms are simplified in the sense that they are sufficient only if registers do not overlap (see Section 5.2.2) and predicated execution [4], [27] (conditionally executing each operation based on the value of an extra predicate register source operand) is not supported by the compiler. Although the actual implementation robustly handles both of these complications, these details distract from the initial discussion of the dependence drawing algorithms. After these simplified algorithms have been presented, an overview of the extensions required to handle overlapping registers will be discussed in Section 5.3.6 and an overview of the extensions required to handle predicated execution will be discussed in Section 5.3.7. It is important to note that if a control block is also the body of a loop, these algorithms will only draw intra-iteration (omega = 0) dependences. The framework also supports drawing of cross-iteration (omega != 0) dependences but this feature is not utilized by this thesis's experiments. (See Lavery's thesis [29] for more details about cross-iteration dependences).

When building a dependence graph, simply drawing all the dependences out of each action is sufficient because this will implicitly add all of the dependences going into each of the actions also. However, when updating the dependences after deleting or adding an action entry, the ability to draw both incoming and outgoing dependences for an action entry is required.

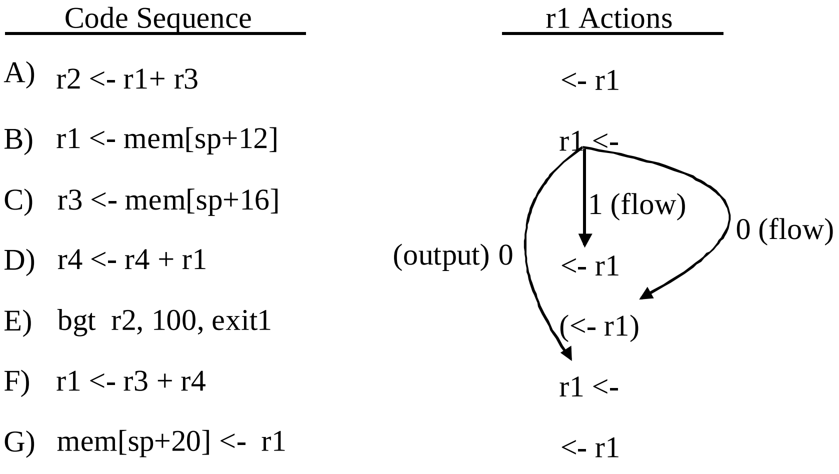
5.3.1 Drawing outgoing destination register dependences

A simplified algorithm for drawing the dependences going out of a destination register's action entry (dest_action) is shown in Figure 5.6. To help illustrate how this algorithm works step by step, the dependences going out of the definition of Register r1 by Operation B in Figure 5.7(a) will be drawn. (The action entry for this definition will be passed to the algorithm, becoming dest_action.) The register action entries for Register r1 are shown Figure 5.7(b) and the soon-to-be-added outgoing dependences for the targeted definition action entry are also shown. The algorithm for adding these outgoing dependences starts with the first actual action entry after dest_action, and scans forward through the actual action entry list using the variable action_after. The appropriate dependence is then drawn based on the characteristics of action_after and dest_action, as described below.

If action_after is a use entry and both action_after and dest_action are explicit actions (see Section 5.2), a normal register flow dependence is drawn from the dest_action to the action_after (as drawn to Operation D's explicit use of Register r1 in Figure 5.7(b)). (See Section 2.3 for a description of register flow dependences.) This dependence is added using the add_dep function call, which requires five arguments. The first argument is the action entry the dependence is going out of and the second argument is the action entry that the dependence is coming into. The third argument is a set of bit flags, combined with a logical OR operation, that specifies the type of dependence to

```
simplified_build_outgoing_dest_register_deps (dest_action)
{
    // Scan forward through the actual action entries after dest_action.
    for (action_after = dest_action->next_actual; action_after != NULL;
         action_after = action_after->next_actual)
    {
        // If an actual use action entry, draw register flow dependence to action_after.
        if (is_actual_use_entry (action_after))
        {
            // If both actions are explicit, draw normal register flow dependence.
            if (is_explicit (action_after) && is_explicit (dest_action))
            {
                // The delay (latency) of the flow dependence is specified by the
                // machine description (MDES).  The dependence's omega is set to 0.
                add_dep (dest_action, action_after,
                          (REG | FLOW | MDES_BASED_DELAY), 0, 0);
            }

            // If one action is explicit and one is implicit (from branch), draw zero-cycle
            // register flow dependence (to prevent reordering but no delay is necessary.)
            else if (is_explicit (action_after) || is_explicit (dest_action))
            {
                // The delay (latency) of the flow dependence is fixed at 0 cycles.
                add_dep (dest_action, action_after, (REG | FLOW | FIXED_DELAY),
                          0, 0);
            }

            // If both are implicit actions (thus for branches), no dependence necessary.
            // This prevents excessive dependences between function calls (jsrs).
        }

        // If an actual definition action, draw register output dependence to action_after.
        else
        {
            // To conserve memory, only draw output deps to explicit actions.
            if (is_explicit (action_after))
            {
                // The delay (latency) of this output dependence is fixed at 0 cycles.
                add_dep (dest_action, action_after, (REG | OUTPUT | FIXED_DELAY),
                          0, 0);
            }

            break;  // After adding this output dependence, no more dependences needed.
        }
    }
}
```

Figure 5.6: Simplified algorithm for building outgoing destination register dependences.



Code Sequence

```
A) r2 <- r1+ r3
B) r1 <- mem[sp+12]
C) r3 <- mem[sp+16]
D) r4 <- r4 + r1
E) bgt  r2, 100, exit1
F) r1 <- r3 + r4
G) mem[sp+20] <-  r1
```

(a) (b)

Figure 5.7: (a) An example code segment and (b) the resulting action entries for Register r1 with the outgoing dependences drawn for the definition of Register r1 in Operation B.

draw. The flags REG and FLOW specify that the dependence is a register flow dependence. The other flag, MDES_BASED_DELAY, specifies that the delay (latency) of dependence is determined by a machine description (MDES) query. (In the example, the query returns one cycle.) (See this researcher's master's thesis [19] for details of how this query is answered by the machine description framework.) The fourth argument specifies an offset to be added to the delay returned by the MDES query. With the DHASY scheduling heuristic this offset is always zero (as shown). For other scheduling algorithms (e.g., modulo schedule [29]) this offset may be nonzero. The fifth argument specifies the omega of the dependence, where a zero omega value indicates that the dependence is not a cross-iteration dependence.

If action_after is a use entry but either action_after or dest_action (but not both) is an implicit action entry, a zero-cycle register flow dependence is drawn from the dest_action to the action_after. This zero-cycle register flow dependence is used to prevent an operation which defines a register's value from moving below a conditional branch when the value is liveout of the branch (as drawn to Operation E's implicit use of Register r1 in Figure 5.7(b)). It is also used to prevent an operation that uses the return value of a function call from moving above the function call (not shown). This zero-cycle dependence is specified by using a FIXED_DELAY flag in the third argument of add_dep call (which sets a base delay of zero instead of querying the MDES) and a zero offset so the total delay is zero.

If action_after is a use, and both dest_action and action_after are implicit actions, no dependence is added. This case currently occurs only between the registers implicitly defined by two function calls (not shown in Figure 5.7). Because a control dependence will be placed between the two function calls to prevent them from reordering, drawing additional register flow dependences is both unnecessary and inefficient.

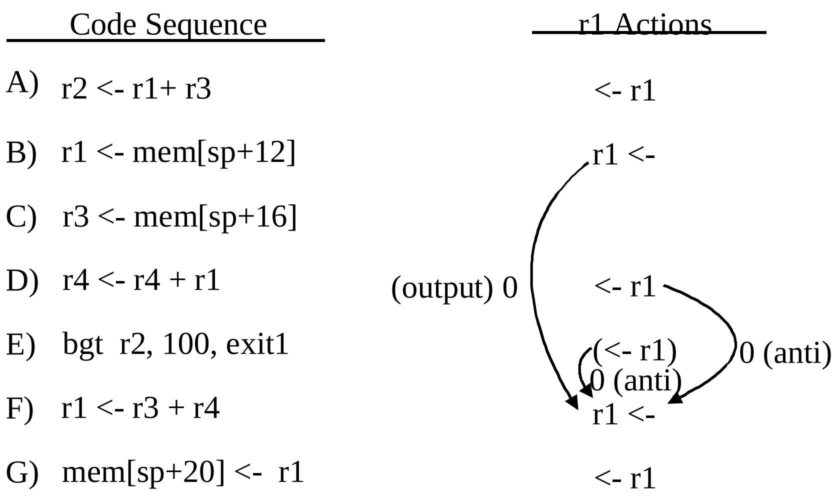
If action_after is a definition and an explicit action, a zero-cycle register output dependence is drawn from the dest_action to the action_after (as drawn to Operation F's explicit definition of Register r1 in Figure 5.7(b)). (See Section 2.2 for a description of register output dependences.) If, instead, action_after is an implicit action, no dependence is drawn because there should already be a zero-cycle register flow dependence to the use version of the implicit action. (Function calls currently both implicitly use and define all the registers they use and/or destroy.) Preventing this dependence from being added is, in essence, an optimization to eliminate unnecessary dependences. After encountering an action_after that is a definition (explicit or implicit), the algorithm terminates because action_after redefines (kills) the register's value defined by dest_action.

```
simplified_build_incoming_dest_register_deps (dest_action)
{
    // Scan backward through the actual entries before dest_action.
    for (action_before = dest_action->prev_actual; action_before != NULL;
         action_before = action_before->prev_actual)
    {
        // If an actual use action entry, draw register anti dependence to dest_action.
        if (is_actual_use_entry (action_before))
        {
            // The delay (latency) of the anti dependence is fixed at 0 cycles.
            add_dep (action_before, dest_action, (REG | ANTI | FIXED_DELAY), 0, 0);
        }

        // If an actual definition action, draw register output dependence to dest_action.
        else
        {
            // To conserve memory, only draw output dependences to explicit actions.
            if (is_explicit (dest_action)
            {
                // The delay (latency) of the output dependence is fixed at 0 cycles.
                add_dep (action_before, dest_action,
                           (REG | OUTPUT | FIXED_DELAY), 0, 0);
            }

            break;  //After adding this output dependence, no more dependences needed.
        }
    }
}
```

Figure 5.8: Simplified algorithm for building incoming destination register dependences.

5.3.2 Drawing incoming destination register dependences

The algorithm for drawing the incoming dependences into a destination register action, shown in Figure 5.8, utilizes an approach similar to that used for drawing outgoing dependences, described in Section 5.3.1. The primary differences are that actual actions before the dest_action are examined (using the action_before variable) instead of those actions after dest_action, and zero-cycle register anti dependences are drawn from use actions to dest_action instead of drawing register flow dependences from dest_action to the use actions. (See Section 2.2 for a description of register anti dependences.)



(a) (b)

Figure 5.9: (a) An example code segment and (b) the resulting action entries for Register r1 with the incoming dependences drawn for the definition of Register r1 in Operation F.

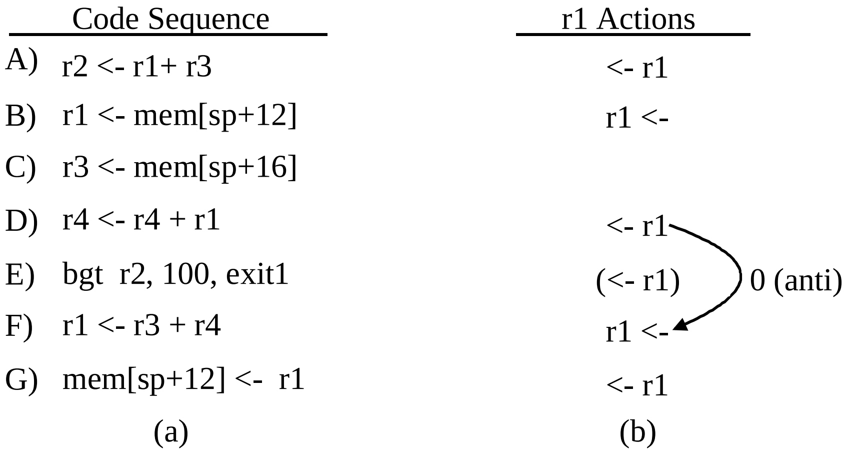
Applying this algorithm to the destination of Operation F in Figure 5.9(a) results in the incoming dependences shown in Figure 5.9(b).

5.3.3 Drawing outgoing source register dependences

A simplified algorithm for drawing dependences going out of a source register's action entry (src_action) is shown in Figure 5.10. This algorithm simply uses the register action table to find the next definition of the register after src_action, and if this definition exists, then the algorithm draws a zero-cycle register anti dependence to this definition. Applying this algorithm to the use of Register r1 by Operation D in Figure 5.11(a) results in the outgoing dependence shown in Figure 5.11(b).

```
simplified_build_outgoing_src_register_deps (src_action)
{
    // Get the next actual definition entry after src_action.
    def_after = src_action->next_def;

    // Draw register anti dependence from src_action to def_after, if def_after exists.
    if (def_after != NULL)
    {
        // The delay (latency) of the anti dependence is fixed at 0 cycles.
        add_dep (src_action, def_after, (REG | ANTI | FIXED_DELAY), 0, 0);
    }
}
```

Figure 5.10: Simplified algorithm for building outgoing source register dependences.



	Code Sequence	r1 Actions
A)	r2 <- r1+ r3	<- r1
B)	r1 <- mem[sp+12]	r1 <-
C)	r3 <- mem[sp+16]	
D)	r4 <- r4 + r1	<- r1
E)	bgt r2, 100, exit1	(<- r1)
F)	r1 <- r3 + r4	r1 <-
G)	mem[sp+12] <- r1	<- r1

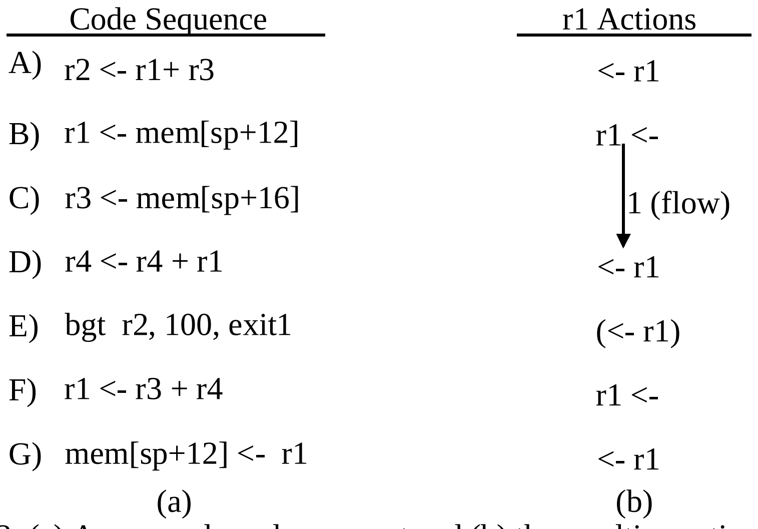
Figure 5.11: (a) An example code segment and (b) the resulting action entries for Register r1 with the only outgoing dependence drawn for the use of Register r1 in Operation D.

5.3.4 Drawing incoming source register dependences

A simplified algorithm for drawing the dependences coming out of a source register's action entry (src_action) is shown in Figure 5.12. This algorithm simply uses the register action table to find the previous definition of the source register, and if this definition exists, draw a register flow dependence from this definition to src_action. As discussed in Section 5.3.1, there are three cases where register flow dependences can be drawn, and in only two of these case are dependences actually drawn. Applying this algorithm to the use of Register r1 by Operation D in Figure 5.13(a) results in the incoming dependence shown in Figure 5.13(b).

```
simplified_build_incoming_src_register_deps (src_action)
{
    // Get the actual definition entry before src_action.
    def_before = src_action->prev_def;

    // Draw register flow dependence from def_before to src_action, if def_before exists.
    if (def_before != NULL)
    {
        // If both actions are explicit, draw normal register flow dependence.
        if (is_explicit (src_action) && is_explicit (def_before))
        {
            // The delay (latency) of the flow dependence is specified by the
            // machine description (MDES).  The dependence's omega is set to 0.
            add_dep (def_before, src_action,
                        (REG | FLOW| MDES_BASED_DELAY), 0, 0);
        }

        // If one action is explicit and one is implicit (from branch), draw zero-cycle
        // register flow dependence (to prevent reordering but no delay is necessary.)
        else if (is_explicit (src_action) || is_explicit (def_before))
        {
            // The delay (latency) of the flow dependence is fixed at 0 cycles.
            add_dep (def_before, src_action, (REG | FLOW | FIXED_DELAY),
                        0, 0);
        }

        // If both are implicit actions (thus for branches), no dependence necessary.
        // This prevents excessive dependences between jsrs.
    }
}
```

Figure 5.12: Simplified algorithm for building incoming source register dependences.



	Code Sequence	r1 Actions
A)	r2 <- r1+ r3	<- r1
B)	r1 <- mem[sp+12]	r1 <-
C)	r3 <- mem[sp+16]	
D)	r4 <- r4 + r1	<- r1
E)	bgt r2, 100, exit1	(<- r1)
F)	r1 <- r3 + r4	r1 <-
G)	mem[sp+12] <- r1	<- r1

Figure 5.13: (a) An example code segment and (b) the resulting action entries for Register r1 with the only incoming dependence drawn for the use of Register r1 in Operation D.

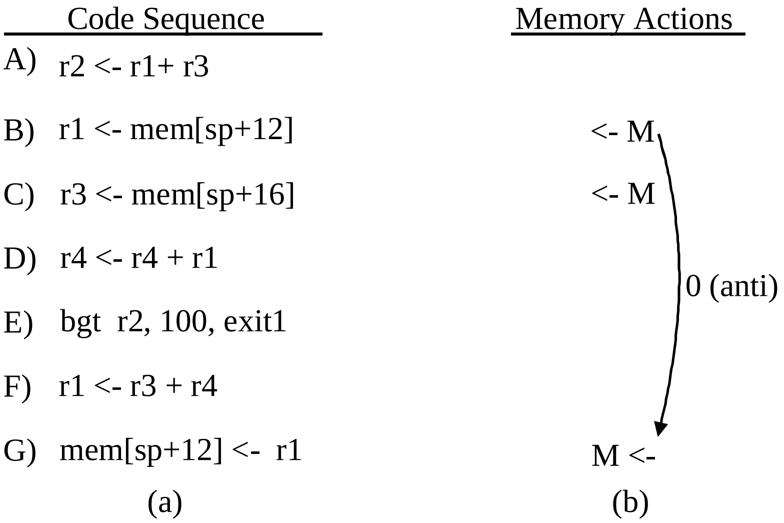
5.3.5 Drawing control, memory, and synchronization dependences

As discussed in Section 5.2.3, the register action table can be used to facilitate drawing control, memory, and synchronization dependences. For each of these dependence types, four algorithms for drawing dependences are required: incoming destination, outgoing destination, incoming source, and outgoing source. These four algorithms use an approach similar to that used in Sections 5.3.1-0 for drawing register dependences. The primary differences are that a pseudoregister operand instead of an actual register operand is passed to the algorithm and dependences are drawn from each use to all of the definitions of this pseudoregister for control and memory dependences, instead of only to adjacent definitions. Although drawing all the transitive control and memory dependences (all uses to all definitions) causes more dependences to be drawn than is strictly necessary, this approach is utilized because it simplifies the implementation of several of the IMPACT compiler's more advanced scheduling and transformation algorithms.

As an illustration of how these algorithms benefit from utilizing the register action table, Figure 5.14 shows the algorithm for drawing outgoing source memory dependences. This algorithm uses the register action table to efficiently find all of the store operations after the load operation's memory action. A memory anti dependence is drawn to every store operation that is not provably writing to a different memory location than the store is reading from. In this thesis's experiments, this determination of independence is done with a quick low-level analysis of the address operands of the two memory operations. If a high-level memory dependence analysis had been performed on

```
simplified_build_outgoing_src_memory_deps (load_action)
{
    // Scan forward though the actual definition entries after load_action
    for (store_after = load_action->next_def; store_after != NULL;
         store_after = store_after->next_def)
    {
        // Unless the store is provably writing to a different memory location than the load
        // is reading (in this iteration of the loop, if any, specified by nonloop_carried),
        // add a zero-cycle memory anti dependence from the load to the store.
        if (!independent_memory_actions (store_after, load_action, nonloop_carried))
        {
            // The delay (latency) of the memory anti dependence is fixed at 0 cycles.
            add_dep (load_action, store_after, (MEM | ANTI | FIXED_DELAY), 0, 0);
        }
    }
}
```

Figure 5.14: Simplified algorithm for building outgoing source memory dependences.



```
Code Sequence
A)  r2 <- r1+ r3
B)  r1 <- mem[sp+12]
C)  r3 <- mem[sp+16]
D)  r4 <- r4 + r1
E)  bgt  r2, 100, exit1
F)  r1 <- r3 + r4
G)  mem[sp+12] <-  r1
```

Figure 5.15: An (a) example code segment and (b) the resulting memory action entries with the only outgoing memory dependence drawn for Operation B.

the code [28], this information would be utilized instead. (This high-level analysis and its internal representation within the IMPACT compiler was undergoing extensive enhancements at the time of this thesis's experiments and, in order to ensure consistent results, was therefore not performed.) Applying this algorithm to Operation B (a load operation) in Figure 5.15(a), draws the memory flow dependence to Operation G as shown in Figure 5.15(b).

```
overlap_enhanced_build_outgoing_src_register_deps (src_action)
{
    // Draw the appropriate dependences into src_action for all of the overlapping action
    // entries (including, by definition, the original src_action).
    for (src_conflict = src_action->first_conflict; src_conflict != NULL;
         src_conflict = src_conflict ->next_conflict)
    {
        // Get the next actual definition entry after src_conflict.
        def_after = src_conflict->next_def;

        // Draw register anti dependence from src_action to def_after, if def_after exists.
        if (def_after != NULL)
        {
            // The delay (latency) of the anti dependence is fixed at 0 cycles.
            // Dependences are only drawn to actual action entries!
            add_dep (src_action, def_after, (REG | ANTI | FIXED_DELAY), 0, 0);
        }
    }
}
```

Figure 5.16: Enhanced algorithm for building outgoing source register dependences that supports overlapping registers.

5.3.6 Drawing dependences for overlapping registers

As described in Section 5.2.2, a register may overlap several other registers in the processor, in which case a special overlapping action entry will be created for each of these overlapping registers. Although the simplified algorithms described in Sections 5.3.1- 0 only draw the dependences for the actual action entry passed to the algorithm as a parameter, by using the same algorithm on the overlapping action entries the appropriate dependences are drawn for the overlapping registers.

To illustrate how this is done, the enhanced version of the algorithm to draw incoming source register dependences is shown in Figure 5.16. The simplified version of this algorithm is shown in Figure 5.10. Note that the same core algorithm is used, but a loop is placed around it that scans though both the actual entry and all the overlapping

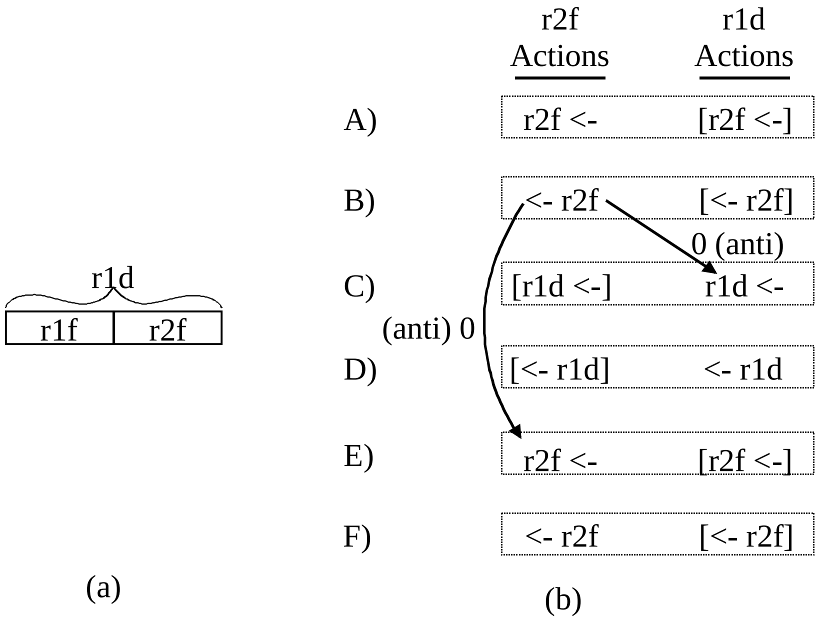


Figure 5.17: (a) Example of overlapping registers and (b) the register anti dependences drawn for the use of r2f in Operation B.

entries (using the src_conflict variable). The dependences, however, are always drawn between the actual action entry (src_action) and another actual entry (def_after) which may be for a different, overlapping, register.

For example, if the Registers r1f, r2f, and r1d overlap as shown in Figure 5.17(a), then the use of r2f in Operation B requires the two register anti dependences shown in Figure 5.17(b) to be drawn. The register anti dependence to Operation E's definition of r2f is drawn when src_conflict points to actual action entry shown for Operation B (<- r2f). The register anti dependence to Operation C is drawn when src_conflict points to the overlapping action entry shown for Operation B ([<- r2f]). Note that the algorithm presented will draw more dependences than is strictly necessary. The techniques used to reduce the number of dependences drawn for overlapping registers is beyond the scope of this thesis. Overlapping registers were not used in this thesis's experiments.

5.3.7 Drawing dependences for predicated code

The predicated execution support in the IMPACT compiler allows a predicate register source operand to be added to any operation in the processor's instruction set [4], [27]. If at run-time the value of this predicate register is one, the operation executes normally; otherwise the operation's execution is suppressed. The use of predicates in the compiler allows control dependences to be converted into data dependences, which may be easier for the compiler to manipulate. For example, the IMPACT compiler uses predication to combine multiple paths through the code into one control block, called a hyperblock [30], [54], in order to facilitate aggressive optimization and scheduling. If the processor does not fully support predicated execution, techniques can then be used to totally [31] or partially [8] remove the predicates from the code.

Two additional queries are used to enhance the simplified dependence drawing algorithms in order to properly draw dependences between predicated operations. The first query, *is_mutually_exclusive*, returns one if it is provably impossible for the two passed actions to both execute in the same invocation of the control block. If one or both of the actions are not predicated, this query will always return zero. If this query returns one, no dependences should to be drawn between the actions to allow maximum scheduling freedom. The second query, *execution_implied*, returns one if the execution of the first passed action implies that the second passed action must also be executed. If the second action is not predicated, this query will always return one. This query is used to determine if an adjacent definition of the action actually starts/terminates the register's lifetime. If this query returns one, then no more dependences need to be drawn. If this

```
predicate_enhanced_build_outgoing_src_register_deps (src_action)
{
    // Scan forward through the actual definition action entries after src_action.
    for (def_after = src_action->next_def; def_after != NULL;
         def_after = def_after->next_def)
    {
        // Draw register anti dependence from src_action to def_after, if def_after exists.
        if (def_after != NULL)
        {
            // Do not draw dependences between mutually exclusive action entries.
            if (!is_mutually_exclusive (src_action, def_after))
            {
                // The delay (latency) of the anti dependence is fixed at 0 cycles.
                add_dep (src_action, def_after, (REG | ANTI | FIXED_DELAY), 0, 0);
            }

            // No more dependences are need if src_action implies the
            // execution of def_after.  Otherwise, continue to scan forward through defs.
            if (execution_implied (src_action, def_after))
                break;
        }
    }
}
```

Figure 5.18: Enhanced algorithm for building outgoing source register dependences that supports predicated execution.

query returns zero, then the scan through the register actions must continue. These queries are described in more detail in Lin’s thesis [32].

To illustrate how these queries are used, an enhanced version of the algorithm to draw incoming source register dependences is shown in Figure 5.18. (The enhancements required for overlapping registers are not shown.) Note that this algorithm degenerates to the simplified algorithm shown in Figure 5.10 when the code is not predicated. It should also be noted that this description is intended only to give a flavor for how predicted execution can be supported. In the actual implementation, some additional techniques are required to handle all the corner cases involved with supporting predicated execution. The details of these techniques are beyond the scope of this thesis because predication support was not utilized in this thesis’s experiments.

5.3.8 Building the dependence graph for the entire control block

When building the dependence graph for the entire control block, only half of the algorithms discussed in Sections 5.3.1-5.3.5 are required. For example, drawing all the dependences into and out of definition action entries will implicitly draw all of the dependences into and out of use action entries. In fact, drawing all the dependences into and out of each definition and use action entry will result in each dependence being drawn exactly twice. This feature was used extensively during regression testing, with mismatches of the dependences drawn automatically reported. (This is essentially N-version programming [33].) During timing runs in this thesis, only the dependences out of each definition and use action entry are drawn. This results in a complete dependence graph, efficiently drawn, with no duplicates.

5.3.9 Incrementally updating the dependence graph

After each code transformation, the dependence graph is incrementally updated for each action entry added to or deleted from the register action table. Four simplified algorithms for incrementally updating register dependences will be described in this section. Similar approaches can be used to incrementally update control, memory, and synchronization dependences, so these algorithms will not be discussed. These algorithms are simplified in the sense that they use the simplified dependence drawing algorithms presented in Sections 5.3.1-5.3.5 and they assume predicated execution is not supported by the compiler. The complications added by predicated execution were discussed in Section 5.3.7. It should be noted that these algorithms are in essence wrappers around the delete_action_entry and add_action_entry algorithms presented in

```
simplified_delete_src_register_action_entry_and_update_deps (src_action)
{
    // Delete all the dependences coming into and going out of this src_action
    delete_all_incoming_dependences (src_action);
    delete_all_outgoing_dependences (src_action);

    // Delete the action entry, incrementally update register action table.
    delete_action_entry (src_action->reg_info_entry, src_action);
}
```

Figure 5.19: Simplified algorithm for deleting a source register action entry and incrementally updating the dependence graph.

```
simplified_add_src_register_action_entry_and_update_deps (reg_info_entry,
                                                          src_action)
{
    // Add to action entry, incrementally update register action table.
    add_action_entry (reg_info_entry, src_action);

    // Build all the dependences coming into and going out of src_action.
    simplified_build_incoming_src_register_deps (src_action);
    simplified_build_outgoing_src_register_deps (src_action);
}
```

Figure 5.20: Simplified algorithm for adding a source register action entry and incrementally updating the dependence graph.

Section 5.2.5, that add the functionality required to incrementally update the dependence graph.

The first simplified algorithm, shown in Figure 5.19, updates the dependence graph when a source register action entry is deleted from the register action table. This algorithm first deletes all of the incoming and outgoing dependences for src_action and then deletes src_action from the register action table. The second simplified algorithm, shown in Figure 5.20, updates the dependence graph when a source register action entry is inserted into the register action table. This algorithm first adds the src_action to the register action table and then builds all of the incoming and outgoing register dependences for that new action. Both these algorithms are trivial because adding or deleting a source (use) register action entry cannot make existing dependences for other

```
simplified_delete_dest_register_action_entry_and_update_deps (dest_action)
{
    // Delete all the dependences coming into and going out of this dest_action
    delete_all_incoming_dependences (dest_action);
    delete_all_outgoing_dependences (dest_action);

    // Get the adjacent definition entries before deleting dest_action
    def_before = dest_action->prev_def;
    def_after = dest_action->next_def;

    // Delete the action entry, incrementally update register action table.
    delete_action_entry (dest_action->reg_info_entry, dest_action);

    // Rebuild dependences going out of previous definition, if it exists.
    if (def_before != NULL)
    {
        delete_all_outgoing_dependences (def_before);
        simplified_build_outgoing_dest_register_deps (def_before);
    }

    // Rebuild dependences coming into next definition, if it exists.
    if (def_after != NULL)
    {
        delete_all_incoming_dependences (def_after);
        simplified_build_incoming_dest_register_deps (def_after);
    }
}
```

Figure 5.21: Simplified algorithm for deleting a destination register action entry and incrementally updating the dependence graph.

action entries invalid or require that new dependences be added to the action entries. This is not the case when adding or deleting destination (definition) register action entries.

The third simplified algorithm, shown in Figure 5.21, updates the dependence graph when a destination register action entry is deleted from the register action table. This algorithm first deletes all the incoming and outgoing dependences for dest_action. Then, the pointers for the adjacent definition entries are stored in def_before and def_after before deleting dest_action from the register action table. After deleting dest_action from the register action table, the dependences going out of the previous definition action entry and those coming into the next definition action entry are rebuilt, if

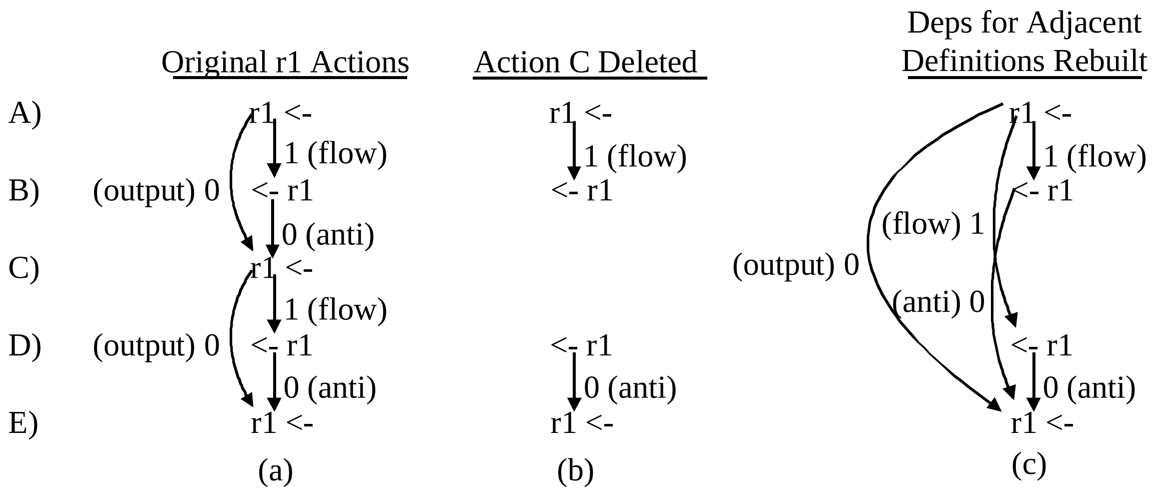


Figure 5.22: Dependences for Register r1 (a) with the original actions, (b) after Action C is deleted, and (c) after rebuilding dependences for adjacent definitions (Actions A and E).

these adjacent definitions exist. This is required because the dependences drawn for these adjacent definitions was determined, in part, by the placement dest_action. After deleting dest_action, rebuilding the dependences for adjacent definitions ensures the dependence graph always represents the current state of the control block.

For example, in Figure 5.22(a), if Action C and all of its incoming and outgoing dependences are deleted, the dependence graph shown in Figure 5.22(b) would result. Rebuilding the outgoing dependences for Action A and the incoming dependences for Action E results in the now-correct dependence graph shown in Figure 5.22(c).

The final simplified algorithm, shown in Figure 5.23, updates the dependence graph when a destination register action entry is inserted into the register action table. The dest_action is first inserted into the register action table and the incoming and outgoing dependences for dest_action then are drawn. The outgoing dependences for the previous definition action entry and the incoming dependences for the next definition action entry are then rebuilt, if they exist.

```
simplified_add_dest_register_action_entry_and_update_deps (reg_info_entry,
                                                           dest_action)
{
    // Add to action entry, incrementally update register action table.
    add_action_entry (reg_info_entry, dest_action);

    // Build all the dependences coming into and going out of dest_action.
    simplified_build_incoming_dest_register_deps (dest_action);
    simplified_build_outgoing_dest_register_deps (dest_action);

    // Rebuild dependences going out of previous definition, if it exists.
    if (dest_action->prev_def != NULL)
    {
        delete_all_outgoing_dependences (dest_action->prev_def);
        simplified_build_outgoing_dest_register_deps (dest_action->prev_def);
    }

    // Rebuild dependences coming into next definition, if it exists.
    if (dest_action->next_def != NULL)
    {
        delete_all_incoming_dependences (dest_action->next_def);
        simplified_build_incoming_dest_register_deps (dest_action->next_def);
    }
}
```

Figure 5.23: Simplified algorithm for adding a destination register action entry and incrementally updating the dependence graph.

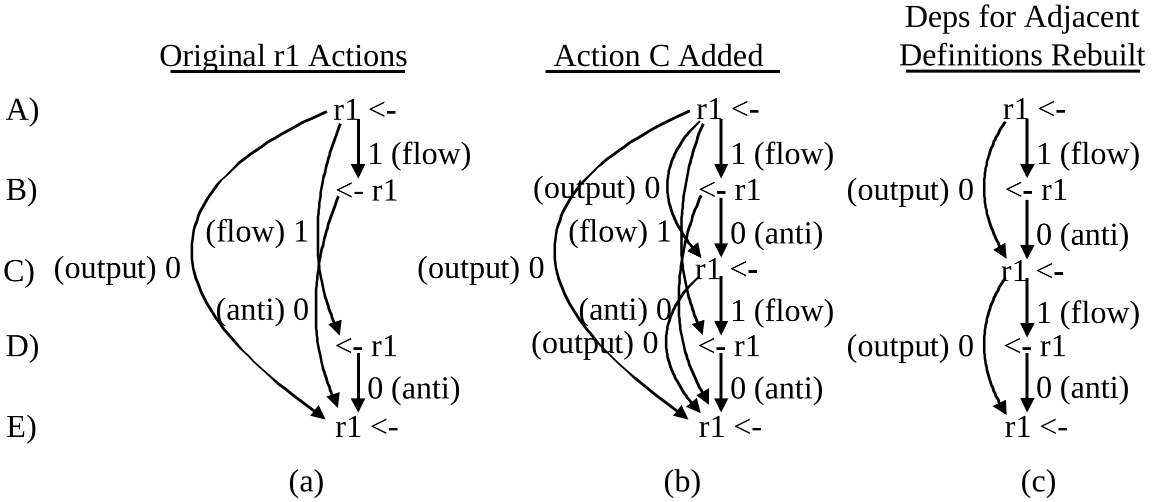


Figure 5.24: Dependences for Register r1 (a) with the original actions, (b) after Action C is added, and (c) after rebuilding dependences for adjacent definitions (Actions A and E).

For example adding a definition of r1 (Action C) between Actions B and D in Figure 5.24(a) must result in the same dependence graph shown in Figure 5.22(a). First

Action C is added to the register action table and the dependences for Action C are built as shown in Figure 5.24(b). If the algorithm stopped here, all the extra invalid dependences would remain, as shown. By rebuilding Action A's outgoing dependences and Action E's incoming dependences and deleting redundant dependences, the dependence graph is now both correct and identical to what would be drawn if the dependence graph was rebuilt from scratch.

5.4 Related Work

Although there is no published work that deals directly with incremental generation of register and control dependences for instruction scheduling, there are several related works that address incremental update of memory dependence information after performing transformations. Gallagher [28] recently developed and implemented within the IMPACT compiler an approach for generating memory disambiguation information at source-level for C programs. This information is then maintained and utilized throughout the rest of the compilation process. As classical optimizations and advanced ILP transformations are performed, this information is updated for the affected operations using specialized routines developed for each transformation. This approach allows the entire compilation process access to more accurate memory disambiguation information than can be generated at the intermediate low-level representation. In addition, this approach improves compiler performance because the memory disambiguation information does not need to be repeatedly regenerated. This chapter's incremental dependence graph generation works in conjunction with the propagated memory dependence information, using this information to determine which memory dependences are necessary.

Smith, Appelbe, and Stirewalt [34] proposed an approach for incremental memory dependence analysis during interactive parallelization of parallel Fortran programs at the source level. The motivation for this work is to reduce the response time of their Parallelizing Assistant Tool to operator specified transformation directives. Using specialized routines for each of the supported parallelization transformations, the operations potentially affected are determined. Memory dependence analysis is then performed again only for these operations. Praveen, Aggarval, and Ghosh [35] then extended this technique to handle more general transformations and formalized the algorithms for regenerating the appropriate memory dependence information. Although some high-level concepts used to perform incremental dependence analysis are similar to those used by the proposed approach for incremental dependence graph generation, the fundamental problems addressed are significantly different. As a result, the final algorithms and their performance characteristics share little resemblance to one another.

6. OPTIMIZATION OF MACHINE DESCRIPTIONS FOR EFFICIENT USE

6.1 Introduction

Machine descriptions have been used to specify execution constraints for several high-performance compilers [36], [37], including the IMPACT compiler [19] utilized in this thesis's experiments. These machine descriptions are primarily used to drive the instruction scheduler, which uses this information to avoid resource conflicts and data dependence interlocks. Because each scheduling decision involves checking execution constraints, the efficiency of such checks can significantly impact the compile time. As a result, compiler writers have faced the choice between two undesirable alternatives. One alternative is to sacrifice portability for accuracy. A compiler designed for a particular processor often uses an accurate, very low-level representation of the machine's description (commonly coded directly into the compiler), which must be tediously modified in order to be effective for subsequent processors. This approach is not desirable in the highly competitive microprocessor industry, where complex new processors are being rapidly designed and brought to the market. Timely development of effective compilers for these new processors is critical to the realization of their full performance potential.

The other alternative is to sacrifice accuracy in favor of portability. Compilers designed to support a wide range of processors, such as gcc, usually describe the machine to their instruction schedulers with easy-to-modify metrics, such as the function unit mix and operation latencies, but these metrics can only approximately model the complex execution constraints in today's superscalar processors. Inaccurate modeling of execution constraints during compilation makes it difficult for the compiler to properly address run-time issues such as resource conflicts and data dependence interlocks. As a result, unexpected execution cycles arise during run time. In processors that exploit high degrees of instruction-level parallelism (ILP), these extra execution cycles can have a significant effect on the overall performance. Accurate modeling of execution constraints is therefore necessary in order to properly utilize these complex processors.

This chapter advocates a model which allows writers to develop a machine description (MDES) in a high-level language, which is then translated into a low-level representation for efficient use by the compiler. The high-level language should be designed to allow the specification of detailed execution constraints in an easy-to-understand, maintainable, and retargetable manner. The low-level representation should be designed to allow the compiler to check execution constraints with high efficiency in both space and time. The discrepancy between the requirements of the high-level language and the low-level representation should then be reconciled with a collection of transformations that derive efficient low-level representations from the easy-to-understand high-level descriptions.

The two-tier model is analogous to using high-level programming languages now that contemporary compiler technology has eliminated the benefits of using assembly language for general purpose programs. The user of a high-level machine description language is not required to be intimately familiar with compiler modules using the machine description, and does not need to manually optimize the description for those modules. In fact, a few of the transformations described in this chapter are adapted from the classical compiler techniques that helped make high-level programming languages so well accepted. There are, however, important transformations introduced in this chapter that have no direct correspondence in the optimizing compiler domain. These transformations take advantage of the unique characteristics of an MDES to increase the efficiency of the resulting low-level representation.

In addition to describing and evaluating the transformations, a novel representation of resource constraints for complex processors is presented. This representation exposes critical information that can be profitably exploited by both these transformations and the compiler modules. This approach to representing machine execution constraints is based on the AND/OR-tree concept used in search algorithms [38].

This new representation for a processor's resource constraints and these MDES transformations have been shown to be highly effective in an earlier study by this researcher [39] that used detailed and precise machine descriptions for the HP PA7100 [40], Intel Pentium [2], Sun SuperSPARC [18], and the AMD-K5 [41]. This chapter evaluates the effectiveness of these techniques utilizing the four experimental processor configurations described in Section 4.3 and used for earlier experiments in this thesis. These processor configurations allow a consistent code base to be utilized, facilitating direct comparisons between varying levels of processor complexity.

The machine descriptions for the four processor configurations are constructed in a high-level language that supports parameterization of all processor features such as issue width, register read and write ports, and the function unit mix. The high-level machine description is based directly on the HP PA7100's machine description and has been used for several of the IMPACT group’s experiments. The low-level representation of these descriptions is then generated and used to schedule the code in order to produce the base (untransformed) scheduled numbers presented in Chapter 4 for each processor configuration. Using a scheduler for the concrete evaluation of these MDES transformations allows the rationale behind and the effect of each transformation to be clearly shown. The primary focus of this chapter will be on the number of resource checks required to determine if an operation can be scheduled. The overall compile-time benefit of using the new representation and the MDES transformations will then be evaluated in Chapter 7.

A description and analysis of a common mechanism used to model resource constraints, which is used by this chapter's machine descriptions, follows this section. A new representation of these resource constraints is introduced in Section 6.3, and is shown to be well-suited for describing today's complex processors. An analysis of each MDES, before any transformations are performed, is presented in Section 6.4. In Section 6.5, the importance of adapting common subexpression elimination, copy propagation, and dead-code removal to clean up machine descriptions is shown, as well as how the new representation facilitates these transformations. Section 6.6 provides a brief overview of the implications of using bit-vectors in the low-level representation of resource constraints in order to set the stage for the Section 6.7, which describes transformations that make the bit-vector representations more effective. Section 6.8 describes and analyzes a set of transformations which makes checking resource constraints more efficient. Section 6.9 summarizes the aggregate effect of all these transformations, with and without the new representation. In Section 6.10, a brief summary of related work is presented and is followed by some concluding remarks.

6.2 Modeling Resource Constraints

This chapter’s machine descriptions model the processor’s resource constraints through the use of a set of *reservation tables* [42], an approach used by several high-performance MDES-driven compilers [36], [37]. In particular, this chapter’s machine descriptions are based on the approach used by the Cydra 5 [36]. Each reservation table specifies a particular way an operation may use a processor’s resources as that operation executes. For example, the resources used by the execution of the four-issue processor configuration’s one-cycle integer literal load operation (which moves a 21-bit literal into an integer register) can be modeled with the twenty-four reservation tables (each called a *reservation table option*, or simply an *option*) that are shown in Figure 6.1. An integer literal load must use one of the four decoders (Decoder), one of the two integer ALUs (Ialu), and one of the three integer write ports (Wr Pt). The literal to be loaded is part of the instruction so none of the register read ports need to be used. All option lists are prioritized (option 1 having highest priority), so for the order shown in this figure, the first available (lowest numbered) decoder, Ialu, and register write port will be used by the integer literal load operation.

The “Cycle” column of these tables indicates the *usage time*, which indicates when each of these resources is used, relative to some chosen time point in the processor’s pipeline. For all the examples and machine descriptions used in this chapter, the point chosen to be time “zero” is the first stage of the execution pipeline. Therefore resources used during decoder stages have negative usage times, and resources used after execution completes, such as result buses and register write ports, have usage times around the operation’s latency. A resource used at a particular usage time will be referred to as a *resource usage*.

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1	☒								
0					☒				
1							☒		

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1	☒								
0					☒				
1								☒	

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1	☒								
0					☒				
1									☒

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1	☒								
0						☒			
1							☒		

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1	☒								
0						☒			
1								☒	

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1	☒								
0						☒			
1									☒

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1		☒							
0					☒				
1							☒		

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1		☒							
0					☒				
1								☒	

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1		☒							
0					☒				
1									☒

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1		☒							
0						☒			
1							☒		

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1		☒							
0						☒			
1								☒	

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1		☒							
0						☒			
1									☒

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1			☒						
0					☒				
1							☒		

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1			☒						
0					☒				
1								☒	

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1			☒						
0					☒				
1									☒

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1			☒						
0						☒			
1							☒		

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1			☒						
0						☒			
1								☒	

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1			☒						
0						☒			
1									☒

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1				☒					
0					☒				
1							☒		

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1				☒					
0					☒				
1								☒	

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1				☒					
0					☒				
1									☒

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1				☒					
0						☒			
1							☒		

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1				☒					
0						☒			
1								☒	

Cycle	Decoder				Ialu		Wr Pt		
	0	1	2	3	0	1	0	1	2
-1				☒					
0						☒			
1									☒

Figure 6.1: The twenty-four reservation tables that represent the resources used by the four-issue processor configuration's integer literal load operation.

It should be noted that the resources modeled often do not represent actual processor resources, but are abstractions used to model the processor's scheduling rules. This approach was used in the construction of the machine descriptions used in this chapter and for the machine descriptions used by the IMPACT compiler in general.

These machine descriptions were designed to accurately and precisely model the processor's scheduling rules, and intuitive resource names were used solely to enhance the clarity of the machine descriptions.

Many of this chapter's examples are drawn from the four-issue processor configuration's MDES, which was briefly described in Section 4.3. The four-issue processor configuration is an in-order superscalar processor that has four full decoders, six integer register read ports (RP), three integer register write ports, two integer ALU (IALU) units, two memory units, one branch unit, and support for one floating-point operation per cycle. The floating-point function units are assumed to have dedicated register ports which do not need to be modeled, but it is important to model the usage of the integer register read and write ports. All of the common integer operations have a one-cycle latency, and load operations have a three-cycle latency.

The number of reservation table options required to model an IALU, branch, or memory operation depends on the number of register source operands that the operation has. For example, an IALU operation with one register source may use any one of the decoders, read ports, IALU units, and write ports, yielding $\binom{4}{1}\binom{6}{1}\binom{2}{1}\binom{3}{1} = 144$ distinct combinations of these resource usages. The same IALU operation with two register sources, requiring two read ports, has $\binom{4}{1}\binom{6}{2}\binom{2}{1}\binom{3}{1} = 360$ distinct combinations of resource usages. Each of these distinct combinations of resource usages is modeled by a reservation table option.

Table 6.1: Option breakdown and scheduling characteristics for the four-issue processor configuration.

Number of Options	% of Scheduling Attempts	Operations Modeled
1	2.12%	Jump ops
3	2.37%	JSR (function call) ops
4	0.13%	Floating-point ALU/multiply/divide/branch ops
6	4.23%	Integer conditional branch/hashing jump ops that use 1 read port
15	3.29%	Integer conditional branch ops that use 2 read ports
18	0.05%	Hashing JSR (address in register)
24	0.22%	Integer literal load ops
48	0.07%	Floating-point load/store ops that use 1 read port
120	4.63%	Integer store, Floating-point load/store ops that use 2 read ports
144	78.26%	Integer ALU/load ops that use 1 read port
360	4.63%	Integer ALU/load ops that use 2 read ports

A breakdown of the number of reservation table options used to model the various operations in the four-issue processor configuration's MDES is shown in Table 6.1. The first column specifies the number of scheduling options. The second column indicates the percentage of the scheduling attempts that the DHASY list scheduler, driven by this MDES and scheduling SPEC CINT92 and CINT95 assembly code, made to try to schedule an operation with that many options during prepass scheduling [14]. The last column gives a brief summary of the types of operation that have that many options. Note that for all the four processor configurations, no nonbranch operation may issue after a branch operation in the same cycle. This model is consistent with the issue-logic rules of current processors. In order to maximize scheduling freedom, branches are modeled as always using the last decoder. (The last two decoders are for the eight-issue processor configuration.)

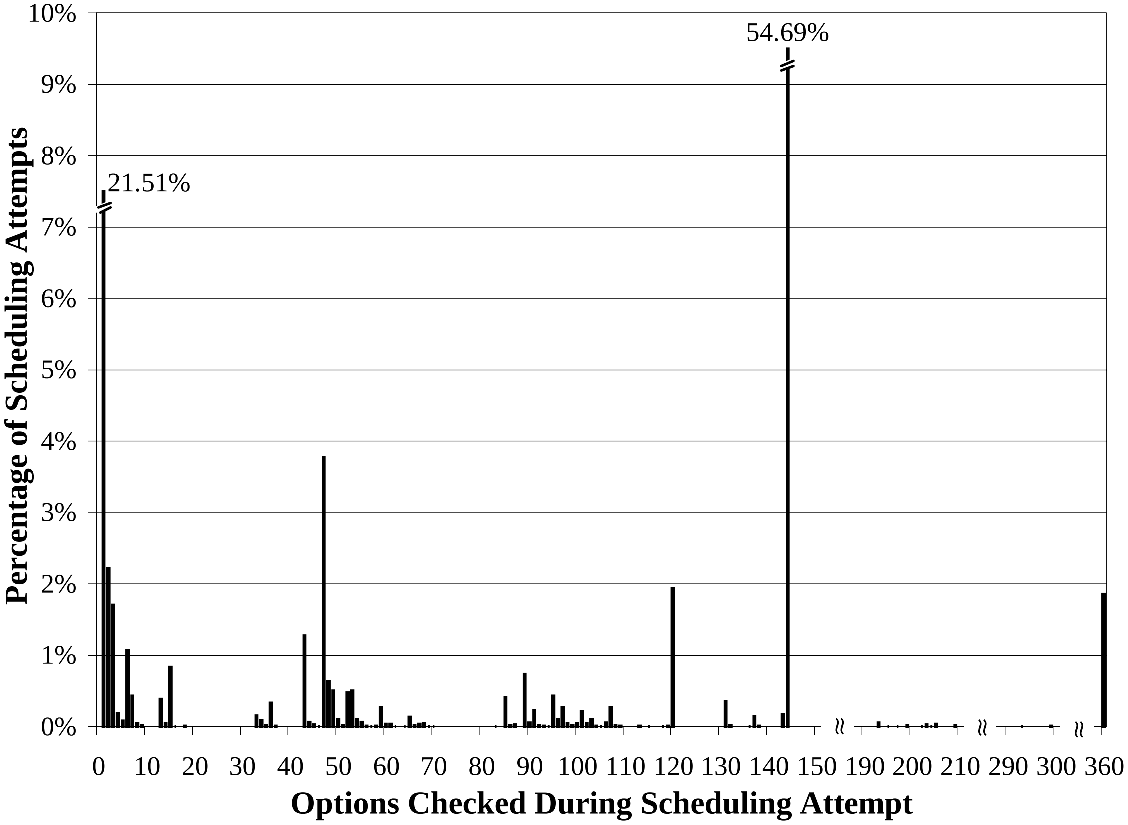


Figure 6.2: Distribution of options checked during each scheduling attempt using the four-issue processor configuration.

Although all of the reservation table options must be tested in order to determine that an operation cannot be scheduled, a variable number of the options need to be tested in order to determine that the operation can be scheduled. Figure 6.2 shows the distribution of options actually checked while scheduling for the four-issue processor configuration. On average, 2.52 scheduling attempts were required per operation, so roughly 60% of the time a scheduling attempt fails. The 54.69% peak at 144 options checked is primarily due to the fact that 89.18% of *unsuccessful* scheduling attempts were on operations with 144 options. The 21.51% peak for one scheduling option checked is due mainly to scheduling attempts that succeed with the first option attempted. For *successful* scheduling attempts, 54.17% succeed with the first option tested, 11.18%

tested between 2 and 9 options, 20.24% tested between 43 and 59 options, and 9.5% tested more than 89 options.

In the following sections, transformations will be presented that make testing each option nearly as efficient as possible. However, unless the number of options checked can be reduced, modeling complex machines exactly will remain expensive in terms of compile time. A new representation, presented in the next section, can dramatically reduce the number of options checked in complex machine descriptions.

6.3 A New Representation: AND/OR-Trees

The primary reason so many options need to be checked for complex processor descriptions is that the traditional representation for resource constraints hides useful information from the compiler. By exposing this useful information with the new representation presented below, the compiler can more efficiently check the resource constraints. This representation can also inherently reduce the MDES size (Section 6.4), facilitate size-reducing transformations (Section 6.5), and facilitate transformations to further optimize for resource conflict detection (Section 6.8). Before describing this new representation, a brief review of the traditional representation is in order.

The traditional representation can be viewed as an OR-Tree, as shown in Figure 6.3(a). This figure shows the twenty-four reservation table options for the four-issue processor configuration's integer literal load operation (the same options that are shown in Figure 6.1). The options are in priority order (with the highest priority first), and if the resources for any of the options are available, the operation can be scheduled. The advantage of this representation is that for OR-trees with a small number of options,

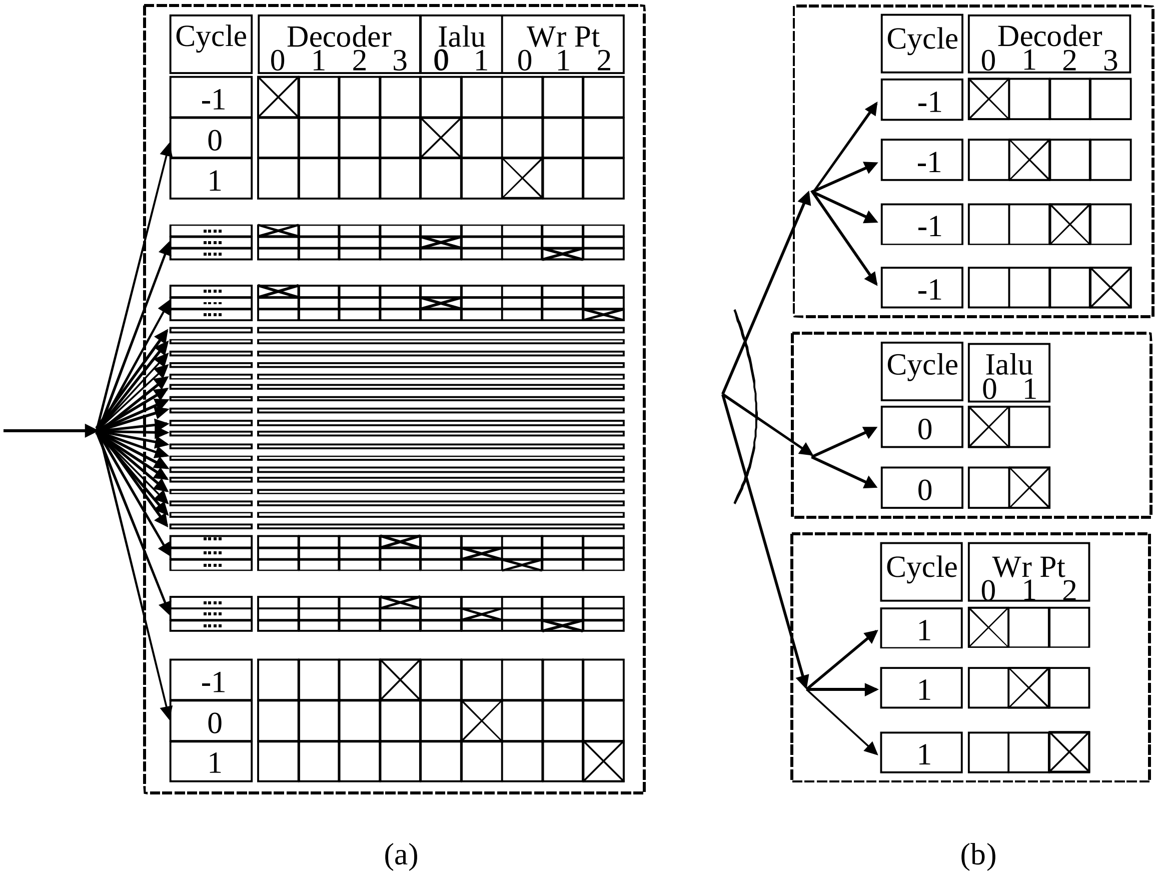


Figure 6.3: Two methods of modeling the resource constraints of the four-issue processor configuration's integer literal load operation. (a) The traditional OR-Tree representation. (b) The proposed AND/OR-Tree representation.

the OR-tree's resource constraints can be quickly and efficiently checked. For processors which have execution constraints that can be modeled with just a few reservation table options, it is difficult to improve upon the efficiency of this OR-tree representation.

The disadvantage of this OR-tree representation is that it does not allow an easy or efficient way of using information about why an option was not available. For example, if Option 1 (the top option) in Figure 6.3(a) is unavailable because Ialu 0 is unavailable, then 11 other options are also guaranteed to be unavailable. Although an inference engine could be programmed into the resource constraint check algorithm to eliminate these options, the overhead would more than negate the benefit.

The solution proposed in this chapter is to use a new representation that is based on the AND/OR-tree concept used in search algorithms [38]. This new representation is, in essence, an AND-tree of OR-trees, allowing multiple OR-trees to be used together in order to represent the resource constraints. An example of this new AND/OR-tree representation is shown in Figure 6.3(b). The AND/OR-tree shown specifies the resource requirements for the four-issue processor configuration's integer literal load operation as requiring one of the four decoders (Decoder), one of the two IALUs (Ialu), and one of the three write ports (Wr Pt). By utilizing the short-circuit properties of AND and OR, the resource constraint check algorithm can quickly determine which of the required resources are available (or if they are not available), without performing any unnecessary checks.

The algorithm overhead incurred by using this new representation is minimal, because it is built upon the OR-tree representation and does not require any new information from the OR-tree resource constraint checker (i.e., which option, if any, is available). In Figure 6.3, each of the OR-trees, which are enclosed in dotted boxes, can have the same internal representation and may have the same resource constraint checker algorithm applied to them. The compiler used for this thesis's experiments does so and, for implementation efficiency, adds an outer loop around the OR-tree's algorithm that processes the array of OR-trees associated with an AND/OR-tree. Although some additional space is required to represent the AND-level of the tree, the use of AND/OR-trees can significantly reduce the size of the resource constraint description in the MDES, as shown in Section 6.4.

6.4 Original MDES Characteristics

In this chapter, the original machine descriptions for the four processor configurations are analyzed to show the rationale behind the transformations presented in the following sections and the advantages of using the AND/OR-tree representation. For this analysis, each of these machine descriptions is used to drive a DHASY list scheduler, which is then used to schedule SPEC CINT92 and CINT95 assembly code that has been aggressively ILP optimized but in which renaming with copy and integer expression reformulation have not been performed (see Section 4.3). (The same exact assembly code is scheduled for each of the four processor configurations.) The four-issue processor configuration was described in Section 6.2, and the reservation table option breakdown of its machine description was shown in Table 6.1. A brief description of the other three processor configurations modeled, and their reservation table option breakdowns, is in order before analyzing the original characteristics of each MDES.

The one-issue processor configuration is an in-order scalar processor that has one decoder, one branch unit, one IALU, one floating-point unit, one memory unit, two integer read ports, and two integer write ports. (The two write ports allow a three-cycle load operation and a one-cycle IALU operation to be scheduled to write to the register file in the same cycle.) Each operation has one, two, or four reservation table options, depending on its register port requirements, as shown in Table 6.2. The two-issue processor configuration is an in-order superscalar processor that is the same as the one-issue processor configuration except it has two decoders and three integer read ports. Each operation has up to twelve reservation table options, as shown in Table 6.3. Operations requiring either one or two integer register read ports have the same number

Table 6.2: Option breakdown and scheduling characteristics for the one-issue processor configuration.

Number of Options	% of Scheduling Attempts	Operations Modeled
1	7.34%	Operations that don’t require an integer write port
2	7.15%	Operations requiring 0 or 2 integer read ports
4	85.51%	Operations requiring 1 integer read port

Table 6.3: Option breakdown and scheduling characteristics for the two-issue processor configuration.

Number of Options	% of Scheduling Attempts	Operations Modeled
1	1.20%	Jump ops
2	1.49%	JSR, floating-point ALU/multiply/divide/branch ops
3	4.30%	Integer conditional branch ops
4	0.12%	Integer literal load ops
6	4.34%	Integer store, hashing JSR, floating-point load/store ops
12	88.55%	Integer ALU/load ops

of options because $\binom{3}{1} = \binom{3}{2} = 3$. The eight-issue processor configuration is an in-order superscalar processor that has eight decoders, two branch units, four integer ALUs, two floating-point units, three memory units, twelve integer register read ports, and six integer write ports. This mixture of resources causes up to 12672 table options to be required to model an operation, as shown in Table 6.4. The most frequently scheduled operations (59.28% of scheduling attempts) are the integer ALU operations that use one integer register read ports and require 2304 table operations to model.

Although it is not necessary for the high-level MDES language to support AND/OR-trees, AND/OR-trees provide a concise way of specifying complex resource

Table 6.4: Option breakdown and scheduling characteristics for the eight-issue processor configuration.

Number of Options	% of Scheduling Attempts	Operations Modeled
2	2.86%	Jump ops
6	3.30%	JSR (function call) ops
16	0.16%	Floating-point ALU/multiply/divide/branch ops
24	5.91%	Integer conditional branch/hashing jump ops that use 1 read port
72	0.07%	Hashing JSR (address in register)
132	4.28%	Integer conditional branch ops that use 2 read ports
192	0.34%	Integer literal load ops
288	0.10%	Floating-point load/store ops that use 1 read port
1584	5.92%	Integer store, Floating-point load/store ops that use 2 read ports
1728	12.37%	Integer load ops that use 1 read port
2304	59.28%	Integer ALU ops that use 1 read port
9504	2.48%	Integer load ops that use 2 read ports
12672	2.93%	Integer ALU ops that use 2 read ports

usages. The machine descriptions used in this chapter’s experiments are written in a high-level MDES language that supports the specification of both OR-trees and AND/OR-trees. The parameterized MDES template utilized to generate the four processor configurations’ machine descriptions uses AND/OR trees extensively. In order to generate the OR-tree MDES representations for this chapter’s experiments, each MDES that uses AND/OR-trees was run through an MDES preprocessor that expanded out each AND/OR-tree specification into the corresponding OR-tree specification.

The scheduling characteristics for these machine descriptions, before any transformations are performed, are shown in Table 6.5. The second and third columns indicate how many operations were scheduled, and how many scheduling attempts were required, on average, before an operation was successfully scheduled. These two numbers will remain constant throughout all of the transformations (with either

Table 6.5: Original scheduling characteristics of the machine descriptions for the four processor configurations.

MDES	Total Ops Sched.	Avg. Sched. Attempts Per Op	OR-Trees		AND/OR-Trees		
			Avg. Options/ Attempt	Avg. Checks/ Attempt	Avg. Options/ Attempt	Avg. Checks/ Attempt	Percent Checks Reduced
1-issue	1264249	7.09	3.55	4.75	1.27	1.78	62.5%
2-issue	1264249	4.43	9.95	18.49	3.74	4.67	74.7%
4-issue	1264249	2.52	99.60	198.32	6.82	8.05	95.9%
8-issue	1264249	1.62	1162.89	2469.08	9.81	11.53	99.5%

representation), and the exact same schedule is produced in each case, because all the execution constraints described in the machine descriptions are being preserved. It should be noted that the number of scheduling attempts required per operation can increase significantly with the use of more advanced scheduling techniques such as iterative modulo scheduling [12] and operation scheduling, and with the application of more ILP optimizations to the assembly code.

The fourth and sixth columns of Table 6.5 show the average number of reservation table options checked for each scheduling attempt, for the OR-tree and AND/OR-tree representations, respectively. The fifth and seventh columns show the average number of resource checks that were required for each scheduling attempt. The last column shows that for complex machine descriptions, before any transformations are performed, the use of the AND/OR-tree representation can reduce the number of resource checks per reservation table option by up to 99.5%.

The memory required to internally represent the resource constraints in the compiler used for this chapter’s evaluation is shown in Table 6.6. Although this internal representation has been extensively tuned to maximize the performance of the resource

Table 6.6: Original MDES memory requirements for the four processor configurations.

		OR-Trees		AND/OR-Trees		
MDES	Number of Trees	Table Options	Size (bytes)	Table Options	Size (bytes)	% Size Reduced
1-issue	30	79	4488	21	2160	51.9%
2-issue	30	259	15096	19	2328	84.6%
4-issue	30	4979	314456	41	2664	99.2%
8-issue	30	146510	9695344	116	4776	99.9%

constraint checking algorithm, it also was designed to minimize memory requirements in ways that incur no performance penalty. To this end, the internal representation allows common information to be shared among AND/OR-trees and OR-trees, but in some cases a small amount of header information per item is duplicated to prevent performance degradation. Both the OR-tree and the AND/OR-tree internal representations have the same number of trees. However, the table shows that the AND/OR-tree representation, because it does not require the explicit enumeration of all the resource usage combinations (OR-tree options), can significantly reduce the memory required (a 99.9% reduction for the eight-issue processor configuration). Thus, before any MDES optimizations, the AND/OR-tree significantly reduces both the internal representation size and the number of checks required per attempt for complex resource constraint descriptions. This advantage will remain definitive after both representations are fully optimized.

The sizes shown in this chapter for the AND/OR-tree representation reflect the extra memory required to store the AND level of the tree. It should also be noted that the common information to be shared is entirely specified by the external MDES representation, in order to minimize the time required to load the MDES into memory. The number of trees and reservation table options shown in the table reflects only what the writer of the MDES specified as being shared. It is easy and natural to specify shared information in the high-level MDES language used, so most of the common information is shared in these machine descriptions. However, common information is often not shared in order to make the machine description more readable or easier to modify. In fact, some of the information in the MDES may not even be used. The transformations presented in the Section 6.5 will deal with these issues.

6.5 Eliminating Redundant or Unused MDES Information

Machine descriptions tend to evolve as a processor's execution constraints become more thoroughly understood, as the compiler's vocabulary of operations increases, and as these machine descriptions are ported to different or experimental processors in the family. As the machine descriptions evolve, the amount of redundant and unused information in the MDES tends to grow because, for an MDES writer, it is typically easier to just make a local copy of the information to be changed than to do the careful analysis required to safely modify or delete existing information. In fact, this was experienced both at Cydrome Inc., with creation and maintenance of the Cydra 5 MDES [39], and by the authors of machine descriptions used by the IMPACT compiler.

This redundant and unused information can be eliminated from the MDES by adapting the classical compiler optimizations (common subexpression elimination, copy propagation, and dead-code removal [1]) to the MDES domain. In this chapter's implementation, common subexpression elimination and copy propagation were

Table 6.7: MDES memory requirements after eliminating redundant and unused information.

	Number	OR-Tree Representation			AND/OR-Tree Representation		
	of Trees	Table	Size	% Size	Table	Size	% Size
MDES		Options	(bytes)	Reduced	Options	(bytes)	Reduced
1-issue	17	46	2928	34.8%	13	1120	48.1%
2-issue	17	153	9536	36.8%	16	1224	47.4%
4-issue	17	3263	211648	32.7%	37	1840	30.9%
8-issue	17	107148	7244960	25.3%	109	4040	15.4%

combined into one step that finds redundant MDES information and points all various references to that information to only one particular copy, and an adaptation of dead-code removal eliminates unreferenced information. These techniques greatly reduced the size required to represent all of the aspects of the MDES, such as resource constraints, operation latency, and operation format. Their effect on the resource-constraint description size in particular is shown in Table 6.7.

It is interesting to note that the AND/OR-tree representation for several of the machine descriptions benefited more from eliminating redundant information than the OR-tree representation. This is because the reservation table options in the AND/OR-tree representation typically specify the resource usages at a finer granularity (fewer usages per option) than the OR-tree options, allowing AND/OR-tree options to be shared more aggressively. In addition, the OR-trees in an AND/OR-tree tend to be more general-purpose, allowing entire OR-trees to be shared by several AND/OR-trees. An example of this second case is shown in Figure 6.4, where the OR-trees for decoder and register write port resource usages are shared by the four-issue processor configuration's integer literal load AND/OR-tree and the integer load (with two register sources) AND/OR-tree. In this

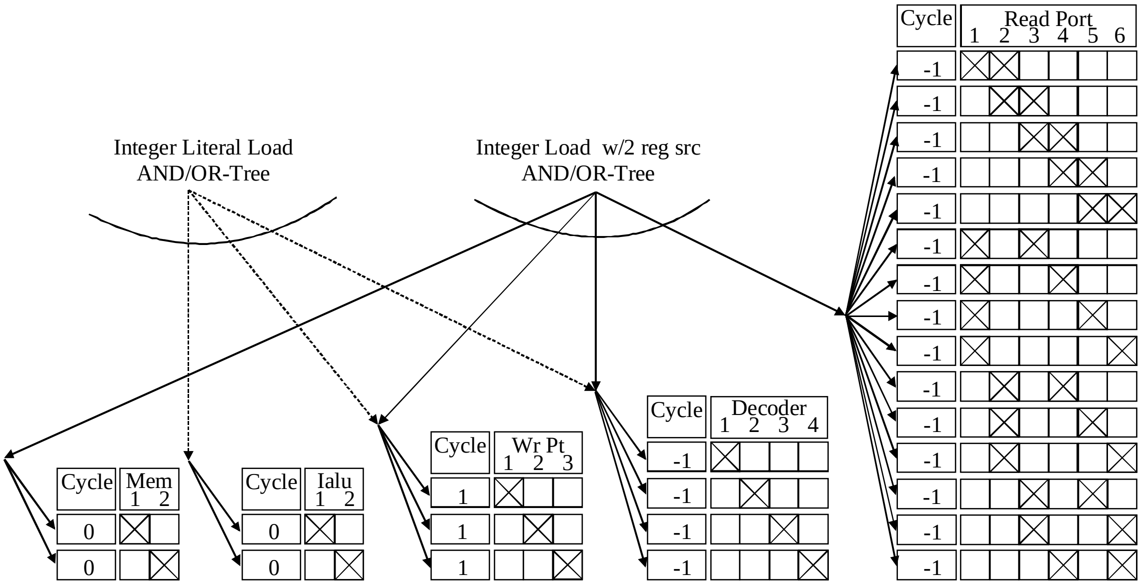


Figure 6.4: An example of how the AND/OR-tree representation can facilitate the sharing of OR-trees.

way the AND/OR-tree representation facilitates further reduction of the MDES size. The amount of reduction drops off as the issue-width increases, which is an artifact of how the parameterized machine description template was written. This machine description template was fairly carefully written and has very little natural redundancy. However, when parameters are set to one (e.g., one branch per cycle) some of the OR-tree specifications become redundant and can be combined with other OR-trees.

The transformations for removing redundant information can also be adapted to more MDES-specific circumstances, such as removing options from an OR-tree that can be determined to be impossible to satisfy. An option can be removed from an OR-tree if its resource usages are identical to, or a superset of, the resource usages for a higher-priority option, because the higher-priority option will always be selected if these resources are available. This case can arise when the use of preprocessor directives enumerates the various OR-tree options, and it can also arise as a machine description evolves, which is the case for the parameterized machine description used in this chapter.

Table 6.8: The scheduling characteristics after removing unnecessary options for memory operations.

MDES	Total Ops Sched.	Avg. Sched. Attempts Per Op.	OR-Tree Avg. Options/ Attempt	OR-Tree Avg. Checks/ Attempt	AND/OR-Tree Average Options/ Attempt	AND/OR-Tree Avg. Checks/ Attempt	AND/OR-Tree Percent Checks Reduced
1-issue	1264249	7.09	3.39	4.56	1.21	1.72	62.3%
2-issue	1264249	4.43	9.56	17.76	3.59	4.49	74.7%
4-issue	1264249	2.52	95.96	189.88	6.63	7.82	95.9%
8-issue	1264249	1.62	1118.18	2334.51	9.53	11.14	99.5%

The parameterized machine description template was derived from an HP PA7100 MDES that was derived from the MDES for an earlier HP PA processor. During the retargeting to the PA7100, two of the reservation table options for the PA7100's memory operations became identical, but the MDES author never realized this because correct output was still generated. This artifact was unintentionally propagated to the parameterized machine description (during its original port many years ago) and the effect this has on the four processor configurations' scheduling characteristics is shown in Table 6.8.

6.6 Utilizing Bit-Vector Representations

The results presented so far have not taken advantage of the fact that most resource-constraint checking algorithms, including the one used in this chapter, use bit-vectors [43], [44], [45] to keep track of the resources used each cycle in what is referred to as a *resource usage map* (RU map). This design allows the RU map size to be minimized and efficiently initialized, and allows multiple resource usages to be checked (and reserved) with a single AND (OR) operation. In addition, using bits in the MDES to represent multiple resource usages can significantly decrease the MDES size. Although it

Table 6.9: MDES size characteristics before and after a bit-vector representation is used (one cycle/word).

MDES	Memory Requirements (in bytes)					
	OR-Tree Representation			AND/OR-Tree Representation		
	Before	After	Diff.	Before	After	Diff.
1-issue	2928	2512	14.2%	1120	1072	4.3%
2-issue	9536	7832	17.9%	1224	1200	2.0%
4-issue	211648	167440	20.9%	1840	1720	6.5%
8-issue	7244960	5666144	21.8%	4040	3512	13.1%

Table 6.10: Scheduling characteristics before and after a bit-vector representation is used (one cycle/word).

MDES	Average Checks Per Scheduling Attempt					
	OR-Tree Representation			AND/OR-Tree Representation		
	Before	After	Diff.	Before	After	Diff.
1-issue	4.56	4.40	3.5%	1.72	1.66	3.5%
2-issue	17.76	15.08	15.1%	4.49	4.43	1.3%
4-issue	189.88	156.54	17.6%	7.82	7.71	1.4%
8-issue	2334.51	1888.68	19.1%	11.14	11.00	1.3%

is possible to pack more than one cycle's resource usages into a single memory word, it is not necessary to do so for the machine descriptions in this chapter. The resource usage time transformation presented in the next section will reduce the number of checks to almost the minimum of one resource check per reservation table option.

The incremental effect of packing each cycle's resource usages into one memory word is shown in Tables 6.9 and 6.10. Before using bit-vectors, each resource usage was represented as a cycle/resource pair (one resource usage per check). After using bit-vectors, the resource usages were represented as a cycle/resource-vector pair (multiple resource usages per check possible, if the usages are in the same cycle). Although both representations require two words to represent each pair, the bit-vector representation typically requires fewer pairs per table. Although all the machine descriptions show some benefit, the resource usages usually do not fall within the same cycle. The only resource usages that naturally fall in the same cycle are the integer register read ports and the instruction decoders. For example, the reservation table options shown in Figure 6.3(a) do not benefit from packing a cycle’s resource usage into a single memory word, because there is only one usage per cycle. However, the resource usage time transformation presented in the next section will resolve this issue.

6.7 Optimizing for Bit-Vector Representations

The use of the actual resource usage times, as in Figure 6.3(a), can significantly reduce the effectiveness of using a bit-vector representation that packs one cycle's worth of resource usages into a single memory word. We address this problem by making use of the theory of pipelined, multifunction unit design [42], [46]. For any ordered pair of reservation table options (A, B), t is a *forbidden latency* (i.e., an operation using reservation table option B cannot be initiated t cycles after an operation that uses reservation table option A) if and only if A and B have resource usages for some common resource at times i and j, respectively, such that i is greater than or equal to j and i-j = t. The set of all forbidden latencies between A and B is termed the *collision vector* for the ordered pair (A, B). A given schedule results in no resource conflicts if and only if, for every pair of operations, the difference in their scheduled times never violates the collision vector for the corresponding pair of reservation tables. Note that the actual reservation table options A and B are not directly important; only the collision vector for (A, B) is. Consequently, we could substitute any reservation table options A' and B' for

A and B, respectively, as long as the collision vector for (A', B') is the same as that for (A, B). Further note that, in computing a forbidden latency, only the difference between the resource usage times i and j matters, not their actual values. In particular, we could add a common constant to both resource usage times without altering the forbidden latency.

With this in mind, the optimization that we use, for each resource, is to subtract a strategically selected constant from the originally specified resource usage times for that resource in every reservation table option, with a view to concentrating resource usages into as few time slots as possible. The constant may be different for each resource. This optimization is related to the one used by Eichenberger and Davidson [47]. Although minimization techniques can be used to find those constants that maximize the benefit, a simple heuristic was found to be highly effective for the forward-scheduling list scheduler and the processors considered in this chapter. The heuristic is, for each resource, to pick the constant to be the earliest resource usage time for that resource (across all reservation table options). The result of this heuristic is to concentrate a far larger number of resource usages than before at time zero, thereby making the bit-vector approach more effective. For a backward-scheduling list scheduler, the constants should be chosen to make the latest usage time to be zero (or some constant). Applying this transformation to Figure 6.3(a) yields the OR-tree shown in Figure 6.5.

In addition to making the bit-vector representation more effective, this transformation also has a subtle effect on the characteristics of the resource usage checks that can be taken advantage of. The resource usages that cause most of the resource conflicts now tend to be concentrated at time zero. The resource usages with times

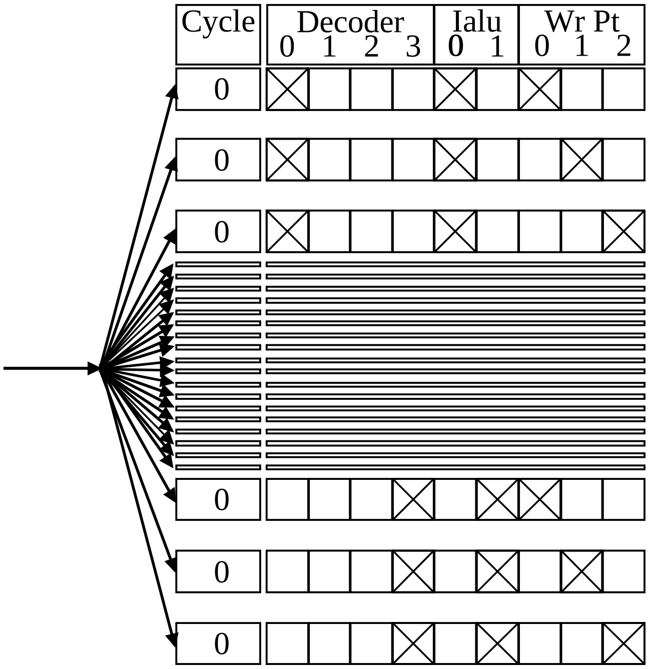


Figure 6.5: The OR-tree modeling the resource constraints for the four-issue processor configuration's integer literal load operation, after transforming the resource usage times in order to better utilize the bit-vector representation.

greater than zero are usually conflict-free and are primarily there to delay the execution of later operations. For example, the nonzero-time divide-unit usages for a divide operation rarely prevent the operation from being scheduled (if the divide unit is available at time zero). However, while scheduling the next divide operation, the divide unit will not be available at time zero until the previous divide completes. Thus for a forward-scheduling list scheduler, the average number of checks before a conflict is detected is minimized by sorting the resulting usage checks so that time zero is checked first. In this manner, the same machine descriptions can be automatically tuned for other types of schedulers by adjusting the heuristic for picking the resource usage time shift constants and for the sorting of the resulting usage checks.

Table 6.11: MDES memory requirements before and after transforming resource usage times (one cycle per word).

MDES	Memory Requirements (in bytes)					
	OR-Tree Representation			AND/OR Tree Representation		
	Before	After	Diff.	Before	After	Diff.
1-issue	2512	1520	39.5%	1072	960	10.4%
2-issue	7832	4520	42.3%	1200	1168	2.7%
4-issue	167440	94712	43.4%	1720	1672	2.8%
8-issue	5666144	4058000	28.4%	3512	3432	2.3%

Table 6.12: Scheduling characteristics before and after transforming resource usage times and sorting the resulting usages to check time zero first (one cycle per word).

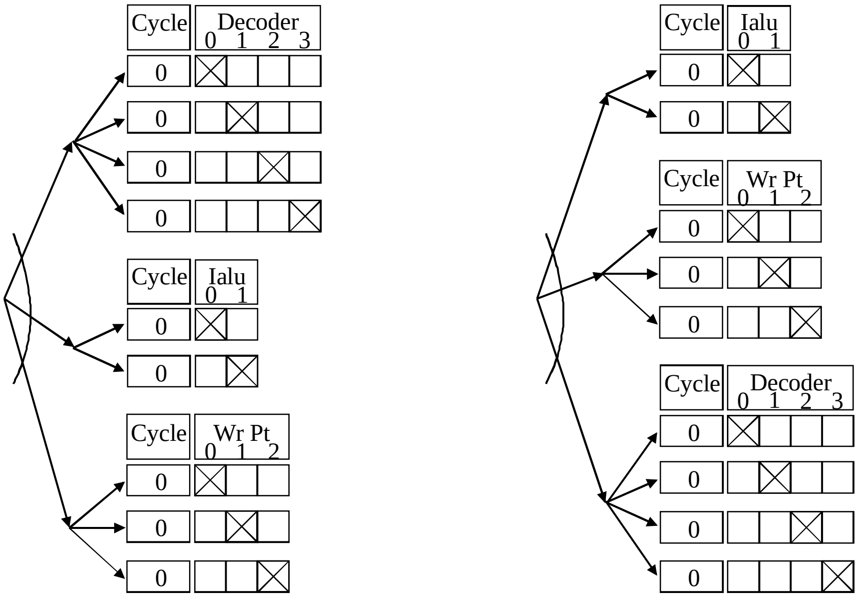
MDES	OR-Tree Rep.				AND/OR-Tree Rep.			
	Avg. Checks/Attempt			Checks/	Avg. Checks/Attempt			Checks/
	Before	After	Diff	Option	Before	After	Diff.	Option
1-issue	4.40	3.42	22.3%	1.01	1.66	1.21	27.1%	1.00
2-issue	15.08	9.61	36.3%	1.01	4.43	3.59	19.0%	1.00
4-issue	156.54	96.70	38.2%	1.01	7.71	6.63	14.0%	1.00
8-issue	1888.68	1288.52	31.8%	1.15	11.00	9.69	11.9%	1.02

The MDES memory requirements after transforming the resource usage times are shown in Table 6.11. The size of the OR-tree representation is reduced up to 43.4% by using this transformation. There is less reduction for the AND/OR-tree representation because this representation tends to have fewer resource usages per option. The transformations presented in the next section do not change the MDES size, so these sizes are the final MDES sizes after full optimization.

The MDES scheduling characteristics after transforming the resource usage times and sorting the resulting usages to check time zero first are shown in Table 6.12. This transformation reduced the average number of resource checks per option to between 1.00 and 1.15, which matches or is close to the ideal case of one check per option. In fact, the integer load operations were the only operations that required more than one check per option because of their use of the integer write port two cycles later than all of the other integer operations. (This prevents their use from being shifting to time zero with all of the other resource usages.) With these near-ideal checks per option, the average number of options checked per attempt is what is truly dictating the number of checks required. Although the AND/OR-tree already has a clear advantage, the number of options checked is further reduced by the transformations in Section 6.8.

6.8 Optimizing AND/OR-Trees for Resource Conflict Detection

The structure of the AND/OR-tree representation allows additional transformations to be performed that can increase the chance of detecting resource conflicts early. The first transformation is to sort the sub OR-trees in the AND/OR-tree so that the OR-tree most likely to have a resource conflict (heuristically determined) is checked first. The following heuristic-based sort criteria were found to produce the most consistent results. The OR-trees are first sorted by the earliest usage time in each tree, because after the resource usage time transformation, most conflicts occur at usage time zero. For OR-trees with the same earliest usage time, sort by the number of options in each OR-tree, so that OR-tree with the fewest options is checked first. To break ties at this point, preference is given to the OR-trees that are shared by the most number of AND/OR-trees, because this gives an indication of which OR-trees have resources that are heavily used. Finally the original order specified is used to break any remaining ties. Figure 6.6(a) shows the OR-tree order originally specified in the MDES (and used for all



(a) (b)

Figure 6.6: An example of optimizing the order of the OR-trees in an AND/OR-tree for resource conflict detection. (a) Original order specified. (b) After optimizing the order.

previous analysis), and Figure 6.6(b) shows the order after sorting the OR-trees using the above criteria. (Only the second criterion applies.)

A second transformation that can be applied is to remove resource usages that are common to all of the OR-tree options and place them in an OR-tree with just one option (creating one if necessary). This transformation works well when a resource common to all options is likely to cause a resource conflict. By pulling it out, this resource conflict can be detected earlier. This transformation can also be used to create some simple AND/OR-trees from OR-tree descriptions. Application of this transformation can actually increase the number of resource checks required, but the following application heuristics were found to yield good results. First, if there is already a one-option OR-tree that has a resource usage with the same usage time as the common usage, apply the transformation. (With bit-vectors, this transformation cannot hurt performance.) Also,

Table 6.13: Scheduling characteristics before and after optimizing AND/OR-trees for resource conflict detection.

MDES	AND/OR-Tree Representation					
	Options Per Attempt			Checks Per Attempt		
	Before	After	Diff.	Before	After	Diff.
1-issue	1.21	1.21	0.0%	1.21	1.21	0.0%
2-issue	3.59	1.76	51.0%	3.59	1.76	51.0%
4-issue	6.63	3.61	45.6%	6.63	3.61	45.6%
8-issue	9.53	6.09	36.1%	9.69	6.25	35.5%

apply the transformation if the common usage is the only usage in the OR-tree with that usage time. (Each option in the OR-tree then has one less check, and in exchange only one check is added.) Otherwise, the transformation should not be applied. In the machine descriptions used in this chapter, all the applications of this transformation occurred because of the first application rule. After the usage time transformation the second case becomes rare or, for these descriptions, nonexistent.

The incremental effect of these transformations on the AND/OR-tree scheduling characteristics is shown in Table 6.13. Most of the AND/OR-trees are reordered so that availability of the function units are checked first (the most constraining resources), which significantly reduces the average number of options checked before a resource conflict is detected. The MDES sizes did not change because of these transformations.

6.9 Aggregate Effect of All Transformations

There are two important machine description aspects that are optimized by the transformations presented in this chapter. The first one is the amount of memory needed by the compiler to represent the processor's resource constraints. Minimizing this size allows more MDES information to fit within the first-level cache during compilation and

Table 6.14: Aggregate effect of all transformations on MDES resource-constraint representation size.

MDES	Memory Requirements (in bytes)				
	Unoptimized	Fully Optimized With Bit-Vector Representation			
	OR-Trees	OR-Trees	Reduction	AND/OR-Trees	Reduction
1-issue	4488	1520	66.1%	960	78.6%
2-issue	15096	4520	70.1%	1168	92.3%
4-issue	314456	94712	69.9%	1672	99.5%
8-issue	9695344	4058000	58.1%	3432	99.9%

also reduces the overall memory requirements of the compiler. The aggregate effect on required memory of all the transformations presented in this chapter is shown in Table 6.14. When applied to an OR-tree representation, these transformations reduce representations by as much as a factor of 3.3. When these transformations are further combined with the AND/OR-trees, representations up to 2800 times smaller than the unoptimized OR-tree representation are produced. As the execution constraints for processors become more flexible, combining these transformations with the AND/OR-tree representation becomes even more effective.

The second aspect of the machine descriptions to be optimized is the number of resource checks per scheduling attempt. Minimizing this number reduces the time required to check resource constraints, making room in the compiler’s time budget for more advanced scheduling or optimization techniques. The aggregate effect of all the transformations presented in this chapter on the average number of resource checks required per scheduling attempt is shown in Table 6.15. As described in Section 6.4, these check-per-attempt statistics were generated using an MDES-driven DHASY list scheduler to schedule SPEC CINT92 and CINT95 assembly code well matched to the

Table 6.15: Aggregate effect of all transformations on MDES scheduling characteristics.

MDES	Average Checks Per Scheduling Attempt				
	Unoptimized	Fully Optimized With Bit-Vector Representation			
	OR-Trees	OR-Trees	Reduction	AND/OR-Trees	Reduction
1-issue	4.75	3.42	28.0%	1.21	74.5%
2-issue	18.49	9.61	48.0%	1.76	90.5%
4-issue	198.32	96.70	51.2%	3.61	98.2%
8-issue	2469.08	1288.52	47.8%	6.25	99.7%

four-issue processor configuration. When compared to the checks-per-attempt of the unoptimized OR-tree representation, these transformations reduced the number of checks required by the OR-tree representation by up to a factor of 2.1. When these transformations are combined with the AND/OR-trees, the number of checks were reduced by as much as a factor of 395.1. As was seen with the MDES-size aspect, combining these transformations with the AND/OR-tree representation is especially effective at reducing the number of checks required as the execution constraints become more flexible.

The trend that these tables show is that as the processors become more powerful and flexible, the AND/OR-tree representation, combined with the described transformations, becomes crucial for keeping under control both the MDES size and the number of checks per scheduling attempt. These results are consistent with what was found in the earlier study with the AMD-K5 K5 MDES and these results are expected to be representative of the latest generation of microprocessors, such as the Intel Pentium Pro and the HP PA8000.

6.10 Related Work

Eichenberger and Davidson [47] recently proposed a minimization algorithm which, for each reservation table option, generates an equivalent reservation table option with a minimum number of resource usages. (This algorithm uses heuristics to avoid exhaustive searches. Although true minimums may not always be found, the results are near optimal.) The total number of resources used to model the processor is also minimized, which facilitates packing multiple cycles of resource usages into a bit-vector. This algorithm, combined with a bit-vector representation, was shown to minimize both the memory required to represent each option and the number of resource checks per option.

The transformations presented in this chapter reduce the number of resource checks and memory required per option to an equivalent level to that obtained using the Eichenberger and Davidson algorithm, although a different and more straightforward approach is used. In addition, Eichenberger and Davidson do not address the problem of reducing the number of option checks per scheduling attempt. This chapter's transformations, when combined with the proposed AND/OR-tree representation, simultaneously optimize the number of options checks per scheduling attempt, the number of resource checks per option, and the memory required to represent the processor's resource constraints.

Proebsting and Fraser [48], Müller [49], and Bala and Rubin [50] have proposed approaches that use finite-state automata, instead of resource reservation tables, to determine if an operation may be scheduled without a resource conflict. These techniques, when compared with the use of unoptimized reservation tables and representations, have shown significant reductions in the number of checks per scheduling attempt and in representation size. However, the combination of the proposed MDES optimizations and AND/OR-tree representation appear to mitigate these advantages, even for complex resource constraints.

In addition, the nature of finite-state automata makes it more difficult, and potentially more time-consuming, to apply some advanced scheduling techniques, such as iterative modulo scheduling [12], which strategically unschedule operations in order to remove the resource conflicts that are preventing an operation from being scheduled. This requires the ability to identify and unschedule the operations that are causing the resource conflicts. Modulo scheduling also uses a cyclic scheduling model that is difficult to efficiently implement with finite state automata. Both these required features can be efficiently implemented in a straightforward manner with reservation tables (implemented in [12]). In addition, reservation tables cannot be totally eliminated even if finite-state automata are used, because analysis of reservation tables in order to calculate scheduling priorities is an integral part of iterative modulo scheduling. Utilizing reservation tables for all scheduling algorithms therefore provides a more consistent and, for several scheduling algorithms, more efficient method of specifying execution constraints.

7. COMPILE-TIME IMPLICATIONS OF SCHEDULE-TIME TRANSFORMATIONS

7.1 Introduction

This chapter experimentally evaluates the compile-time implications of the proposed schedule-time transformation framework (Chapter 3), incremental dependence graph generation (Chapter 5), and machine description optimizations (Chapter 6). These experiments were performed utilizing a complete and highly-tuned schedule-time transformation framework that implements all the algorithms described in this thesis within the IMPACT compiler. This framework was designed to be of production-compiler quality in terms of speed, error checking, stability, and ease of use. In addition to facilitating advanced compiler research with the IMPACT compiler, this design also allows measurement of the true time requirements of the proposed approaches with a solid implementation. However, it should be noted that this framework did not attempt to reduce compile time by focusing only on relatively important control blocks. A production compiler, by making appropriate trade-offs between code quality and compile time, may be able to further reduce the time requirements for schedule-time transformations.

All of the time measurements in this chapter were performed on an unloaded 180MHz HP PA8000 workstation running HP-UX B.10.20 in multiuser mode with 384 Mbytes of main memory. The schedule-time transformation framework was compiled using the HP C Compiler 10.23.03 using the aggressive optimization settings used to generate the SPEC base performance ratios (i.e., +Oall +P). This optimization level requires profiling data to be collected. The framework was profiled optimizing SPEC CINT92 with both the renaming-with-copy and integer-expression-reformulation transformations. In order to measure the time required by various components of the framework, a set of run-time parameters was added that allowed each component to be enabled (invoked normally) and disabled (totally omitted). By carefully measuring the execution time required as each successive component is enabled, an accurate estimate of the time required to perform that portion of the computation can be made.

The time measurements were performed with the times() function that reports both the user and system time used by a process with a resolution of 10 milliseconds. The user and system time are added together for the reported numbers and represent almost all of the wall-time measured. (Time spent waiting for disk I/O and in system processes does not show up in the times() output.) Because of the nature of modern processors and operating systems, small variations in time measurements are unfortunately unavoidable. System processes can perturb instruction caches, data caches, the branch prediction unit, etc., in unpredictable ways. To compensate, each measurement was performed 25 times and results were sorted. The top and bottom five measurements were discarded and the remaining 15 measurements were averaged. The standard deviation among these remaining measurements was often zero and was always less than 0.5%. However, because the difference between two measurements is being used to estimate the execution time for each component, even a one-tenth-of-a-second standard deviation makes it difficult to accurately estimate the time of any component taking less than 1 second. Therefore, in order to increase the overall processing time

Table 7.1: Compile time using the HP C Compiler on the SPEC CINT92 benchmarks.

	Compile Time (in minutes) Using the HP C Compiler	
Benchmark	-O	+Oall +P (Spec Base)
008.espresso	0.61	3.94
022.li	0.19	0.83
023.eqntott	0.12	0.52
026.compress	0.03	0.23
072.sc	0.37	2.67
085.gcc	2.38	35.06
Total	3.70	43.25

Table 7.2: Compile time using the HP C Compiler on the SPEC CINT95 benchmarks.

	Compile Time (in minutes) Using the HP C Compiler	
Benchmark	-O	+Oall +P (Spec Base)
099.go	0.74	4.94
124.m88ksim	1.12	6.33
126.gcc	4.52	79.14
129.compress	0.03	0.12
130.li	0.30	0.95
132.ijpeg	0.92	4.20
134.perl	1.24	44.63
147.vortex	2.11	14.87
Total	10.98	155.18

required by each component, the benchmarks are grouped into two large inputs, SPEC CINT92 and SPEC CINT95.

7.2 Compile-Time Requirements of the HP C Compiler

In order to provide some context for the time requirements for various components of the schedule-time transformation framework, the time required to compile each of the SPEC CINT92 and SPEC CINT95 benchmarks with the HP C Compiler Version 10.23.03 is presented in Tables 7.1 and 7.2. These measurements were taken using the command-line time command (e.g., time cc -O *.c), which has a 100 millisecond resolution. The times shown are the sum of the user and system time reported by the command-line time command. The first column indicates the benchmark being compiled. The second column specifies the compile time, in minutes, for the benchmark using the standard optimization level setting (-O) for the compiler. This setting enforces a strict compile-time budget, which was reached when compiling 085.gcc, 126.gcc, and 134.perl. For these benchmarks, compiler warning messages indicated that some optimizations were skipped because “compiler resource limits” were exceeded. This level of optimization is typically used as an application is being developed. An executable with reasonable performance is generated in a reasonably short amount of time (3.70 minutes for SPEC CINT92 and 10.98 minutes for SPEC CINT95).

The third column specifies the compile time, in minutes, for the benchmark using the optimization settings used to generate the SPEC base performance numbers for SPEC CINT95 (+Oall +P). These settings generate code that runs approximately 30% faster than the standard setting but requires significantly more compile time (43.25 minutes for SPEC CINT92 and 155.18 for SPEC CINT95). These settings also require each benchmark to be profiled on a representative set of inputs first before the code is optimized. (The time required to compile the instrumented executable or to generate the profile information is not included in the compile times shown.) This setting is typically used after development is complete to generate a high-performance executable. Although reducing the compile-time requirements is always desirable, generating a high-performance executable is the top priority at this level of optimization. Users willing to spend more time tuning compiler flags and specifying additional assumptions that the compiler may make about the application (e.g., what’s done for the SPEC peak settings), may be able to generate an executable with even higher performance.

The IMPACT compiler aggressiveness is most closely related to the HP Compiler using the SPEC base optimization settings. Aggressive function inlining, superblock formation, loop unrolling, and ILP transformations are performed that significantly improve executable performance. However, like SPEC base settings, benchmark specific compiler flags are not used to tweak performance. These aggressive optimization techniques also significantly increases static code size and therefore the time required by later compiler modules (e.g., the schedule-time transformation framework) to process the resulting code.

7.3 Time Requirements of DHASY List Scheduling

An efficient scheduler is one of the cornerstones of an efficient schedule-time transformation framework. In Chapter 6, a set of machine description transformations and a new AND/OR-tree representation were shown to significantly reduce the time required to check processor resource constraints. The effect of these MDES optimization techniques on the time required to list-schedule all the control blocks in the SPEC CINT92 and SPEC CINT95 superblock code generated by the IMPACT compiler is shown in Table 7.3. These times are only for the scheduling algorithm and do not include the time required to perform dataflow analysis or build the dependence graph. The compile-time costs of these additional components will be discussed later in this section.

The first column of Table 7.3 specifies the processor configuration modeled by the MDES used for scheduling. The second (SPEC CINT92) and fifth (SPEC CINT95)

Table 7.3: Time required to list-schedule all control blocks using the DHASY scheduling heuristic.

MDES	Time Required to Schedule All Control Block (in seconds)					
	SPEC CINT92			SPEC CINT95		
	Unoptimized OR-Trees	Optimized AND/OR-Trees	Reduction	Unoptimized OR-Trees	Optimized AND/OR-Trees	Reduction
1-issue	8.5	7.4	13%	70.5	69.8	1%
2-issue	8.4	6.2	26%	59.2	50.1	15%
4-issue	18.6	5.5	70%	99.4	37.9	62%
8-issue	122.1	5.0	96%	565.8	33.3	94%

columns specify the time required, in seconds, to schedule all the control blocks utilizing an unoptimized OR-Tree representation for the processor resource constraints. The third and sixth columns specify the time required, in seconds, to schedule all the control blocks utilizing a fully optimizing AND/OR-Tree representation for the processor resource constraints that also takes advantage of bit-vectors. The reduction in schedule-time is shown in the fourth and seventh columns.

Although the MDES optimization techniques presented in Chapter 6 reduced the resource checks by up to a factor of 395, the overall scheduling time is only reduced by up to a factor of 24. This is because, after MDES optimizations, the time required for checking resources is reduced to an average of only about 5% of the total scheduling time. Therefore, after MDES optimizations, the compile-time cost of utilizing a detailed machine description facility (versus hard-coding the processor execution constraints into the compiler) is insignificant. However, without MDES optimizations, describing complex execution constraints would be significantly less practical.

Note that the time required to schedule the code using the optimized MDES increases as the processor issue width decreases. This occurs because the code is ILP optimized for a four-issue processor. For the one-issue processor, the relative instruction-level parallelism is extremely high and it takes an average of 7.09 scheduling attempts to schedule an operation. For the four-issue processor, an average of only 2.52 scheduling attempts is required. (See Table 6.5 for details.)

To add some perspective for these scheduling times, scheduling for the 4-issue processor configuration (5.5 seconds for SPEC CINT92 and 37.9 seconds for SPEC CINT95) takes approximately the same amount of time as building the IMPACT compiler's internal intermediate representation of the benchmark code from the external intermediate representation (binary lcode which has been tuned for performance and size) and significantly less time than most other compiler algorithms implemented in the IMPACT compiler. For example, building the internal intermediate representation from the external representation takes 7.2 seconds for SPEC CINT92 and 27.5 seconds for SPEC CINT95. Performing dataflow analysis (live analysis) takes 19.1 seconds for SPEC CINT92 and 375.4 seconds for SPEC CINT95 (due to some very large functions after ILP optimization). (The time required to perform dataflow analysis will be significantly reduced once region-based compilation [51] is fully integrated into the IMPACT compiler.) Building the dependence graph using the approach described in Chapter 5 takes 12.4 seconds for SPEC CINT92 and 51.8 seconds for SPEC CINT95.

Table 7.4: A breakdown of the time required by the components of the schedule-time transformation algorithm utilizing the optimized four-issue MDES.

Function Calls Utilized By the Schedule-Time Transformation Algorithm	Time Required (in seconds)	
	SPEC CINT92	SPEC CINT95
Calculate initial_height (fixed overhead)	5.5	37.9
find_potential_trans/calculate_trans_priorities	1.0	2.0
perform_transformation / undo_transformation	1.1	3.8
schedule / unschedule	9.2	127.2
Total	16.8	170.9

7.4 Time Breakdown for Performing Schedule-Time Transformations

Chapter 4 examined in detail the performance benefits of performing renaming with copy and integer expression reformulation utilizing the schedule-time transformation algorithm. Table 7.4 shows a breakdown of the time required to perform both transformations using the schedule-time transformation algorithm shown in Figure 3.1. For clarity, the time breakdown is shown beside the function calls used in Figure 3.1 and the time requirements for the optimized four-issue processor MDES are used. The time spent in functions not shown in Table 7.4 is negligible. The time required to initially schedule each control block in order to calculate the control block's initial schedule height is shown in the first row of Table 7.4. This time is an fixed overhead for this algorithm and will not increase as more transformations are performed.

As expected, most of the time is spent list scheduling (9.2 seconds for SPEC CINT92 and 127.2 seconds for SPEC CINT95) the control block using the DHASY scheduling heuristic after each transformation is performed. (These times also include the time required to unschedule the control block, but this takes very little time compared to scheduling the control block.) If the unoptimized OR-tree MDES representation is utilized, the time required by the schedule/unschedule function calls increases to 32.1 seconds for SPEC CINT92 and 335.4 seconds for SPEC CINT95. This again illustrates the benefit of the techniques described in Chapter 6.

Compared to the time required for scheduling the control block, a relatively small amount of time is spent performing and undoing the transformations (1.1 seconds for SPEC CINT92 and 3.8 for SPEC CINT95). Note that these times are a worst-case number where each transformation is rejected and must be undone. (This worst-case assumption allows this component to be executed without having to reschedule the control block, thus allowing time measurements.) This small amount of time required is entirely because the techniques described in Chapter 5 were utilized to incrementally update the dependence graph after each transformation was performed or undone. In fact, very little time is spent in the incremental dependence graph update routines at all. However, if the dependence graph was rebuilt from scratch after performing or undoing each transformation, these times would increase to at least 34.0 seconds for SPEC CINT92 and 209.8 seconds for SPEC CINT95. (These times are for rebuilding the dependence graph only; they assume performing and undoing the transformations take no time.) Thus, incremental dependence graph generation reduces the time required to perform and undo transformations by at least 97%.

The least amount of time is spent finding the best potential transformation opportunity to evaluate next (1.0 seconds for SPEC CINT92 and 2.0 seconds for SPEC CINT95). These times are also worst-case in order to allow the measurement to be taken independently from the rescheduling algorithm. Although transformation priorities are normally recalculated only after a transformation is kept (because the dependence graph has changed), for this measurement the transformation priorities were recalculated after every transformation was attempted (and rejected). About half the time measured was spent recalculating priorities and the other half was spent performing the initial scan of the operations of the control block for potential transformation opportunities. A negligible amount of time is required to incrementally detect new transformation opportunities as transformations are performed (the update_trans_queue function in Figure 3.1). (All of the transformations were rejected, so this functionality was not exercised for this particular measurement.) The priority recalculation can be performed relatively quickly by utilizing the early and late times already calculated during the scheduling of the control block.

Overall, performing both ILP transformations using the scheduling-time transformation framework required 16.8 seconds (0.28 minutes) for SPEC CINT92 and 170.9 seconds (2.85 minutes) for SPEC CINT95 for the four-issue processor configuration. If the techniques proposed for incremental dependence graph generation (Chapter 5) and machine description optimizations (Chapter 6) were not utilized, schedule-time transformations still could be performed. However, the time requirements would increase to at least 85.7 seconds (1.43 minutes) for SPEC CINT92 and 646.6 seconds (10.78 minutes) for SPEC CINT95. Compared to time required by the HP Compiler to aggressively optimize SPEC CINT92 (43.25 minutes) and SPEC CINT95 (155.18 minutes) all these time requirements are relatively small. However, utilizing the proposed techniques reduces the time requirements by up to 80% (for the four-issue processor configuration), allowing significantly more ILP transformations to be performed at schedule-time within the compiler’s time budget.

7.5 Projection of Future Time Requirements

As more execution-constraint-sensitive ILP transformation are implemented in the schedule-time transformation framework, the time requirements will also increase. A projection of these future time requirements can be made utilizing the time requirements for the two ILP transformations implement in this thesis's experiments. For SPEC CINT92, there were 311824 static operations in the intermediate representation, in which 7031 transformation opportunities were examined using the schedule-time transformation framework. This yields 0.0225 transformation opportunities per static operation. For SPEC CINT95, there were 952425 static operations with 14909 transformation opportunities, yielding 0.0157 transformation opportunities per static operation. Utilizing the average time required per transformation, the projection of future time requirements as more transformations are implement can be made.

One aspect of the projected future transformation load is the number of transformation opportunities per operation examined. In the near term, after implementing memory speculation, branch combining, and predicate promotion with copy in the schedule-time transformation framework, the number of opportunities may get as high as 0.1 opportunities per operation examined. In the long term, an upper bound on the number of opportunities might be reasonably set around 1.0 opportunities per operation. To reach this upper bound, a wide array of execution-constraint-sensitive transformations would need to be developed and implemented. In addition, the application code would have to be fortuitously constructed so that many of these transformations could be applied. The actual compile-time requirements are likely to fall

Table 7.5: Projection of time requirements for future schedule-time transformation loads for the four-issue processor configuration.

Potential Future Schedule-Time Transformation Loads	Projected Time Requirement (in minutes)	
	CINT92	CINT95
0.1 opportunities per frequently executed static ops (top 90%)	0.10	1.9
1.0 opportunities per frequently executed static ops (top 90%)	0.93	18
0.1 opportunities per static operation	0.93	15
1.0 opportunities per static operation	8.4	140

between 0.1 and 1.0 opportunities per operation examined and will be significantly closer to 0.1 in the near term.

The other aspect of the projected future transformation load is the number of operations examined for transformation opportunities. Because profile information is available, the compiler may choose to restrict the operations examined in order to reduce compile time. One approach may be to focus on the control blocks representing the top 90% of the executed operations. For SPEC CINT92, this would require only 11.09% of the operations to be examined and for SPEC CINT95, only 12.57% of the operations would need to be examined. The time requirement projections with this approach are shown in the first two rows of Table 7.5. These projections utilize the average time per transformation opportunity measured for the four-issue processor configuration. This yields a time requirement range of 0.10 to 0.93 minutes for SPEC CINT92 and 1.9 to 18 minutes for SPEC CINT95. Time requirements anywhere within these ranges are still relatively small when compared to aggressively optimizing these benchmarks with the HP Compiler. This indicates that this thesis's approach is viable when aggressively optimizing key sections of each program.

On the other end of the spectrum, the transformation framework could be used to transform every static operation. This approach requires significantly more compile time but might be utilized in combination with static branch prediction to aggressively optimize operations that the profiling did not exercise. This yields projected time requirements of 0.93 to 8.4 minutes for SPEC CINT92 and 15 to 140 minutes for SPEC CINT95. The lower end of this range is still quite viable when compared to the HP Compiler's time requirements (43.25 minutes for SPEC CINT92 and 155.18 minutes for SPEC CINT95). However, for the upper end of the range for SPEC CINT95, trade-offs may need to be made to reduce compile time. One possibility is to restrict the transformations considered for control blocks deemed to be less critical.

Note that the projected time requirements in Table 7.5 are shown for one pass of the proposed schedule-time application algorithm. For the two transformations implemented, one pass of the schedule-time application algorithm over the potential transformations captures practically all of the possible performance benefit. As more transformations are implemented, multiple passes may be required to maximize the performance benefit as discussed in Section 3.4. However, with properly chosen priorities, one pass should continue to capture most of the possible performance.

Overall, the time required for future transformation loads are easily in line with the time utilized by the HP production C compiler (when producing highly optimized code). This is in large part due to the proposed techniques for incremental dependence graph generation presented in Chapter 5 and the machine description optimizations presented in Chapter 6.

8. CONCLUSIONS AND FUTURE WORK

This thesis proposes a new algorithm for applying execution-constraint-sensitive transformations that increase instruction-level parallelism. This algorithm applies these transformations at schedule time, so that the effect of each transformation on the execution time of the code can be more accurately evaluated. Each transformation opportunity is first prioritized, based on dependence height information, and then a greedy search algorithm is utilized to select the set of transformation applications that minimize the schedule-based execution time estimate.

Detailed experiments utilizing two execution-constraint-sensitive transformations, renaming with copy and integer expression reformulation, showed that this approach performs consistently better than the current heuristic-based application approach and several other more advanced application approaches. In addition, unlike the other approaches evaluated, the proposed approach performed well across all the processor configurations, from single-issue to eight-issue processors. This makes the proposed approach one of the first transformation application techniques that truly adapts to the processor's execution constraints. Combined with a machine description language that describes the processor's execution constraints, this approach allows aggressive transformations consistently achieve performance improvement over wide variety of processor configurations and application inputs.

This thesis also proposes two significant techniques that facilitate the efficient application of transformations at schedule time. The first is a technique for performing incremental dependence graph generation, which allows the dependence graph to be efficiently updated after each transformation is performed. This makes evaluating the effect of the transformation significantly more efficient because the dependence graph does not need to be rebuilt before rescheduling the code. The second technique consists of a set of machine description optimizations and a new resource constraint representation. This technique allows a compiler writer to develop easy-to-understand, maintainable machine descriptions in a high-level language, which is then translated into an optimized low-level representation for efficient use by the compiler. This significantly reduces both the memory requirements for internal representation of the machine description in the compiler and the time required to check resource constraints during scheduling.

Utilizing a complete and highly-tuned schedule-time transformation framework that implemented all the proposed techniques, the time requirements for the proposed schedule-time transformation algorithm were evaluated. In order to provide context, the time required to aggressively optimize the benchmark set with the HP C compiler was also evaluated. The time required to perform the two implemented transformations was shown to be significantly smaller than that required by the HP C compiler. In addition, rough estimates of future time requirements as more transformations are performed at schedule-time were also found to be reasonable, although aggressively transforming only frequently executed code may be desirable to reduce these future time requirements.

There are several near-term areas for future work. The proposed framework greatly facilitates the application of a wide range of critical path reduction transformations [17]. The first area of future work would be to implement more of these aggressive dependence-breaking transformations within the framework. In addition to creating significantly more instruction level parallelism in the code, these transformations will allow the effectiveness of the search algorithm and transformation priorities utilized to be more fully evaluated. In addition, the potential benefit of the proposed approach, in terms of reducing phase-ordering issues, can also be evaluated.

Other longer-term areas of potential future work include utilizing the proposed framework for more global transformations and scheduling algorithms [52] that move code between control blocks. The framework should facilitate the accurate measurement of the effect of adding instructions to one section of code and removing instructions from another section, allowing accurate trade-offs to be made. Similarly, this schedule-time transformation framework may also be useful when performing even more global code transformations such as loop unrolling [5], [53] and hyperblock formation [4], [30], [54]. Although different trade-offs are involved, these techniques are also sensitive to the processor's resource constraints and may benefit from a more accurate estimate of the effect of the transformations.

APPENDIX A. PERFORMANCE RESULTS FOR ALL PROCESSOR CONFIGURATIONS

For the interested reader, this appendix presents the complete set of control-block-level performance distributions for the four processor configurations analyzed in Chapter 4. For the reader's convenience, the overall benchmark performance distributions for the four processor configurations are presented again at the end of this appendix.

(Please see phd-appendix-john-gyllenhaal.ps for contents of Appendix A.)

REFERENCES

[1] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

[2] Intel, *Pentium Processor Family Developer's Manual*. Mt. Prospect, IL: Intel Corporation, 1995.

[3] D. Weaver, *SPARC-V9 Architecture Specification*. Menlo Park, CA: SPARC International Inc., 1994.

[4] S. A. Mahlke, "Exploiting Instruction Level Parallelism in the Presence of Conditional Branches," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1996.

[5] S. A. Mahlke, W. Y. Chen, J. C. Gyllenhaal, W. W. Hwu, P. P. Chang, and T. Kiyohara, "Compiler code transformations for superscalar-based high-performance systems," in *Proceedings of Supercomputing '92*, November 1992, pp. 808-817.

[6] M. Schlansker, V. Kathail, and S. Anik, "Height reduction of control recurrences for ILP processors," in *Proceedings of the 27th International Symposium on Microarchitecture*, December 1994, pp. 40-51.

[7] W. W. Hwu, R. E. Hank, D. M. Gallagher, S. A. Mahlke, D. M. Lavery, G. E. Haab, J. C. Gyllenhaal, and D. I. August, "Compiler Technology for Future Microprocessors," *Proceedings of the IEEE*, vol. 83, no. 12, pp. 1625-1640, December 1995.

[8] S. A. Mahlke, R. E. Hank, J. McCormick, D. I. August, and W. W. Hwu, "A comparison of full and partial predicated execution support for ILP processors," in *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995, pp. 138-150.

[9] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu, "Dynamic memory disambiguation using the memory conflict buffer," in *Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems,* October 1994, pp. 183-193.

[10] S. A. Mahlke, "Design and implementation of a portable global code optimizer," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[11] N. Patkar, "Sparc64," in *Tutorial Program of the 28th International Symposium on Microarchitecture*, December 1995, pp. 1-45.

[12] B. R. Rau, “Iterative modulo scheduling: An algorithm for software pipelining loops,” in *Proceedings of the 27th International Symposium on Microarchitecture*, December 1994, pp. 63-74.

[13] E. L. Boyd and E. S. Davidson, “Hierarchical performance modeling with MACS: A case study of the convex c-240,” in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993, pp. 203-212.

[14] P. P. Chang, D. M. Lavery, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, “The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors,” *IEEE Transactions on Computers*, vol. 44, no. 3, pp. 353-370, March 1995.

[15] P. P. Chang, N. J. Warter, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, “Three Architectural Models for Compiler-Controlled Speculative Execution,” *IEEE Transactions on Computers*, vol. 44, no. 4, pp. 481-494, April 1995.

[16] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, “IMPACT: An architectural framework for multiple-instruction-issue processors,” in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, June 1991, pp. 266-275.

[17] M. Schlansker and V. Kathail, “Critical path reduction for scalar programs,” in *Proceedings of the 28th International Symposium on Microarchitecture*, December 1995, pp. 57-69.

[18] G. Blanck and S. Krueger, “The SuperSPARC microprocessor,” in *Proceedings of COMPCON*, Spring 1992, pp. 136-141.

[19] J. C. Gyllenhaal, “A machine description language for compilation,” M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1994.

[20] J. C. Gyllenhaal, B. R. Rau, and W. W. Hwu, “Hmdes Version 2.0 Specification,” The IMPACT Research Group, University of Illinois, Urbana, IL, Tech. Rep. IMPACT-96-3, 1996, *http://www.crhc.uiuc.edu/Impact/*.

[21] J. L. Baer and D. P. Bovet, “Compilation of Arithmetic Expressions for Parallel Computations,” in *Proceedings of IFIP Congress*, 1968, pp. 34-46.

[22] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, MA: The MIT Press, 1991.

[23] R. A. Bringmann, “Compiler-Controlled Speculation,” Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[24] T. M. Watts, M. L. Soffa, and R. Gupta, “Techniques for Integrating Parallelizing Transformations and Compiler Based Scheduling Methods,” in *Proceedings of Supercomputing ‘92*, Minneapolis, MN, November 16-20, 1992, pp. 830-839.

[25] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, “The Superblock: An Effective Technique for VLIW and Superscalar Compilation,” in *Instruction-Level Parallelism: A Special Issue of The Journal of Supercomputing*, B. R. Rau and J.A. Fisher, Editors. Boston: Kluwer Academic Publishers, 1993, pp. 229-248.

[26] B. L. Deitrich and W. W. Hwu, “Speculative Hedge: Regulating Compile-Time Speculation Against Profile Variations,” in *Proceedings of the 29th International Symposium on Microarchitecture*, December 2-4, 1996, pp. 70-79.

[27] V. Kathail, M. S. Schlansker, and B. R. Rau, “HPL playdoh architecture specification: Version 1.0,” Hewlett-Packard Laboratories, Palo Alto, CA 94303, Tech. Rep. HPL-93-80, February 1994.

[28] D. Gallagher, “Memory Disambiguation to Facilitate Instruction-Level Parallelism Compilation,” Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, August 1995.

[29] D. M. Lavery, “Modulo Scheduling for Control-Intensive General-Purpose Programs,” Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana IL, May 1997.

[30] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, “Effective Compiler Support for Predicated Execution Using the Hyperblock,” in *Proceedings of the 25th International Symposium on Microarchitecture*, December 1992, pp. 45-54.

[31] N. Warter, “Modulo Scheduling with Isomorphic Control Transformations,” Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana IL, September 1994.

[32] D. C. Lin, “Compiler Support For Predicated Execution in Superscalar Processors,” M.S. thesis, Department of Computer Science, University of Illinois, Urbana IL, 1992.

[33] L. Chen and A. Avizienis, “N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation,” in *Digest of Eighth International Fault-Tolerant Computing Symposium*, IEEE Computer Society, Toulouse, France, 1978, pp. 3-9.

[34] K. Smith, B. Appelbe, and K. Stirewalt, "Incremental Dependence Analysis for Interactive Parallelization," *International Conference on Supercomputing*, vol. 18, no. 3, pp. 330-341, September 1990.

[35] K. V. Praveen, S. K. Aggarval, and R. K. Ghosh, "Incremental Data Dependence Analysis," in *Proceedings of Third International Conference on High Performance Computing*, December 1996, pp. 195-200.

[36] J. C. Dehnert and R. A. Towle, "Compiling for the Cydra 5," in *Instruction-Level Parallelism: A Special Issue of The Journal of Supercomputing*, B. R. Rau and J.A. Fisher, Editors. Boston: Kluwer Academic Publishers, 1993, pp. 181-227.

[37] G. P. Lowney et al., "The Multiflow trace scheduling compiler," in *Instruction-Level Parallelism: A Special Issue of The Journal of Supercomputing*, B. R. Rau and J.A. Fisher, Editors. Boston: Kluwer Academic Publishers, 1993, pp. 51-142.

[38] P. H. Winston, *Artificial Intelligence*. Reading, MA: Addison-Wesley, 1984.

[39] J. C. Gyllenhaal, W. W. Hwu, and B. R. Rau, "Optimization of Machine Descriptions for Efficient Use," in *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996, pp. 349-358.

[40] T. Asprey et al., "Performance features of the PA7100 microprocessor," *IEEE Micro*, pp. 22-35, June 1993.

[41] D. Christie, "Developing the AMD-K5 architecture," *IEEE Micro*, pp. 16-26, April 1996.

[42] E. S. Davidson, L. E. Shar, A. T. Thomas, and J. H. Patel, "Effective control for pipelined computers," in *Spring COMPCON'75 digests of papers*, February 1975, pp. 181-184.

[43] R. L. Kleir, "A representation for the analysis of microprogram operation," in *Proceedings of the 7th Annual Workshop on Microprogramming*, September 1974.

[44] D. J. DeWitt, "A control-word model for detecting conflicts between microprograms," in *Proceedings of the 8th Annual Workshop on Microprogramming*, September 1975.

[45] J. A. Fisher, "The Optimization of Horizontal MicroCode Within and Beyond Basic Blocks; An Application of Processor Scheduling with Resources," Ph.D. dissertation, New York University, 1979.

[46] P. M. Kogge, *The Architecture of Pipelined Computers*. New York: McGraw-Hill, 1991.

[47] A. E. Eichenberger and E. S. Davidson, “A reduced multipipeline machine description that preserves scheduling constraints,” in *Proceedings of the Conference on Programming Language Design and Implementation*, May 1996, pp. 12-20.

[48] T. A. Proebsting and C. W. Fraser, “Detecting pipeline structural hazards quickly,” in *The 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1994, pp. 280-286.

[49] T. Müller, “Employing finite automata for resource scheduling,” in *Proceedings of the 26th Annual International Symposium on Microarchitecture*, December 1993, pp. 12-20.

[50] V. Bala and N. Rubin, “Efficient instruction scheduling using finite state automata,” in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, November 1995, pp. 46-56.

[51] R. E. Hank, “Region-Based Compilation,” Ph.D. dissertation, Electrical and Computer Engineering Department, University of Illinois, Urbana, IL, May 1995.

[52] J. A. Fisher, “Trace Scheduling: A Technique for Global Microcode Compaction,” *IEEE Transactions on Computers*, vol. C-30, pp. 478-490, July 1981.

[53] D. M. Lavery and W. W. Hwu, “Unrolling-Based Optimizations for Modulo Scheduling,” in *Proceedings of the 28th Annual International Symposium on Microarchitecture*, December 1995, pp. 327-337.

[54] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu, “Characterizing the Impact of Predicated Execution on Branch Prediction,” in *Proceedings of the 27th International Symposium on Microarchitecture*, December 1994, pp. 217-227.

VITA

John Christopher Gyllenhaal was born in Abington, Pennsylvania, in 1968. He pursued his undergraduate studies at the University of Arizona in Tucson, Arizona, where he received his B.S. degree in Electrical Engineering in 1991. He continued his studies in Electrical Engineering at the University of Illinois at Urbana-Champaign. In the spring of 1992 he joined the Center for Reliable and High-Performance Computing as a member of the IMPACT project directed by Professor Wen-mei Hwu. He completed his M.S. degree in Electrical Engineering in 1994. After completing his Ph.D. work, he will continue to work on several IMPACT research projects as a Visiting Research Assistant Professor at the University of Illinois.