© Copyright by David Isaac August, 2000

SYSTEMATIC COMPILATION FOR PREDICATED EXECUTION

BY

DAVID ISAAC AUGUST

B.S., Rensselaer Polytechnic Institute, 1993
M.S., University of Illinois, 1996

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, 2000

Urbana, Illinois

SYSTEMATIC COMPILATION FOR PREDICATED EXECUTION

David Isaac August, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 2000
Wen-mei W. Hwu, Advisor

The performance of modern processors depends on their ability to execute multiple instructions per cycle, requiring increasing levels of instruction-level parallelism (ILP) to be exposed in programs. One of the major challenges to increasing available ILP is overcoming the limitations imposed by branch instructions. One paradigm presented to help overcome problems with branches both at run time and at compile time is *predication*. Predication allows the elimination of branch instructions by providing an alternate means of controlling the condition of execution for individual instructions. Effective use of hardware support for predication requires advanced compiler support.

Many existing techniques for compilation using predication make approximations or use heuristics, the inaccuracy of which reduces achievable performance. In some cases, these techniques realize a performance loss compared to codes without predication. This dissertation proposes and investigates a systematic approach to compiling for predication which consistently generates efficient predicated code.

The four key compiler technologies presented here work synergistically to realize the potential of predication. The Partial Reverse If-Conversion Framework provides the first compilation framework to accurately balance control and predication, while providing other compiler components with complete access to the predicated code for further optimization. Though the full potential of the Partial Reverse If-Conversion Framework remains unexplored, current compiler technology justifies its worth.

To operate on predicated code, the optimizer, scheduler, and register allocator require accurate information regarding the relationships among predicates. The Predicate Analysis System is the first efficient predicate relationship database to provide an approximation-free representation.

Optimization, scheduling, and register allocation also require accurate knowledge of the flow of information in the predicated code. Using the Predicate Analysis System, the Predicate Dataflow Graph is built to provide dataflow information. The Predicate Dataflow Graph was developed to provide accurate dataflow information in the presence of predication without requiring a change in the dataflow analysis equations.

In the context of the Partial Reverse If-Conversion Framework, some otherwise unnecessarily complex optimizations become feasible. One example is the Program Decision Logic Optimizer, which uses predicated code and the Predicate Analysis System to represent program decision logic as Boolean expressions. These expressions are minimized, factored, and reformulated to create programs with optimized decision components. Within the given compilation framework, the PDLO has been demonstrated to break many performance bottlenecks due to program decision.

The Partial Reverse If-Conversion Framework, the Predicate Analysis System, the Predicate Dataflow Graph, and the Program Decision Logic Optimizer are demonstrated to work synergistically in the compiler to generate efficient code for nonnumeric programs. Using predication, these techniques effectively optimize codes in the presence of complex branch control flow.

DEDICATION

Dedicated to the fond memory of my grandfather, Dr. Joseph Victor Foa.

ACKNOWLEDGMENTS

For his guidance, encouragement, and support, I thank my advisor, Professor Wen-mei Hwu. I am grateful for his providing the fertile research environment that made this work possible. His inspiration, generosity, and patience have made it possible for me to sustain this research throughout its lengthy course. As he has pointed out, being a research advisor is analogous to being a parent. In this spirit, I will return the generosity he has shown towards me by passing it along to the next generation.

The members of my dissertation committee played a key role in the completion of this work. Professor Farid Najm, Dr. Scott Mahlke, Professor David Padua, and Professor William Sanders each provided valuable comments, suggestions, and insight that helped me address a variety of issues. Prior to becoming a member of my committee, Professor Farid Najm introduced me to techniques from the computer-aided design community that facilitated much of this work.

I feel extremely fortunate to have been a part of the IMPACT Research Group. This research truly would not have been possible without the support, hard work, and friendship of the members of the IMPACT team, who were always there to discuss ideas, debate solutions, practice talks, and develop software.

I thank Dr. Scott Mahlke, who, as a senior member of the IMPACT Group, was the first to guide me through the many intricacies of the IMPACT compiler and its first-generation predication support. While at Hewlett-Packard Laboratories, he continued to be a valuable collaborator on numerous projects. My continuing interaction with him greatly influenced the direction and quality of this work.

I acknowledge and express my gratitude to John Sias. I continue to value highly our intellectual interactions, some of which led to improvements in the quality of this work. His help in taming IMPACT allowed me to focus on studying the techniques described here. His help on the University of Illinois campus allowed me to start my new profession unencumbered as the thesis went through its final administrative paces. I am grateful that I was always able to count on him for friendship, support, and encouragement.

I thank my officemate, Daniel Connors, for his hard work and friendship. His help in debugging and maintaining the main IMPACT compilation path, profiling tools, and a code generator or two when it seemed that the responsibility was ours alone is appreciated. I thank Kevin Crozier for helping with numerous projects. Always ready with a Perl script, he often saved the day by increasing our efficiency. Rick Hank, John Gyllenhaal, Roger Bringmann, Grant Haab, and Ben-Chung Cheng provided technical support in the form of instruction and development of module features. Brian Deitrich and Teresa Johnson provided valuable support while I was taking classes and preparing for the qualifying exam.

I would like to thank my colleagues at Intel and Hewlett-Packard who provided me with valuable summer employment experiences in compilers and architecture. In particular, Sadun Anik, Carole Dulong, Richard Johnson, Jim Pierce, Bob Rau, and Mike Schlansker each personally taught me a great deal.

I would like to express heart-felt thanks to Sheila Hemami. Sheila provided me with perspective which made it possible to know when to call this work complete. Her love and caring allowed me to concentrate on the thesis without distraction near the end. She also gave me the support necessary to balance the many aspects of finishing graduate school and starting a new faculty position.

I would like also to acknowledge the support of my family. I thank my brother, Jonathan, for being my best friend. I thank my parents, Michael and Eugenie, for their love and support, both emotional and financial, and to my grandparents for the same. Their acts of kindness and love remain with me.

Finally, I thank my maternal grandfather, Joseph Victor Foa, to whom this thesis is dedicated. He has truly been an inspiration in my life. As a professor and great instructor, he showed me the joy of teaching when he taught me, his 10-year-old student, the laws of thermodynamics. As a scientist and engineer, his inventions led me to understand the contributions these disciplines can make to our society. As a citizen, by his service as a colonel in the United States Army, as a member of the Technical Industrial Intelligence Committee of the Joint Chiefs of Staff, and by encouraging policymakers to reduce waste in defense spending, he instilled in me a notion of civic responsibility. As a family man, he showed me the importance and rewards of strong family bonds. As a student who,

despite adversity, earned two doctoral degrees, he was the inspiration that made this one possible.

TABLE OF CONTENTS

Page

1. INTRODUCTION 1
 1.1 Branch Control Effects on ILP 2
 1.2 Overcoming Branch Problems with Predication 5
 1.3 Systematic Compilation for Predicated Execution 7
 1.4 Contributions 10
 1.5 Overview 13

2. PREDICATED EXECUTION 15
 2.1 Overview of Predication 15
 2.2 Predicated Execution in the IMPACT EPIC Architecture 17
  2.2.1 Predicate comparison instructions. 18
  2.2.2 Predicate clear/set instructions. 21
  2.2.3 Predicate save/restore instructions. 22
  2.2.4 Predicate register file. 22
 2.3 Other Predicated Architectures 23
  2.3.1 PlayDoh 23
  2.3.2 Predicated execution support in Intel IA-64 25
  2.3.3 Predicated execution support in the Cydra 5 25
  2.3.4 Predicated execution support in ARM 27
  2.3.5 Conditional instructions in TI C6X 28
  2.3.6 Limited predicated execution support in other systems 28

3. COMPILATION FRAMEWORKS FOR PREDICATION 31
 3.1 Beneficial Path Inclusion 32
 3.2 Harmful Path Inclusion 34
 3.3 Phase Ordering Considerations 37
 3.4 The Hyperblock Compilation Framework 39
 3.5 Challenges Posed by Post-If-Conversion Optimization 42
 3.6 Compilation Frameworks 47

3.6.1 The If-Conversion during Scheduling Framework . . . 48
3.6.2 The proposed compilation framework . . . 50

4. PREDICATE ANALYSIS SYSTEM . . . 55
4.1 Predicate Analysis . . . 56
4.2 Previous Mechanisms . . . 59
4.3 The Predicate Analysis System . . . 60
4.3.1 Intrahyperblock analysis and the BDD . . . 60
4.3.2 Interhyperblock relationships . . . 66
4.3.3 Predicate relationship query interface . . . 68
4.4 Evaluation . . . 69
4.5 Significance . . . 73

5. THE PREDICATE DATAFLOW GRAPH . . . 74
5.1 Predicate-Aware Dataflow Analysis . . . 74
5.2 Predicate Flow Graph . . . 76
5.2.1 Active range . . . 79
5.2.2 Predicate flow graph construction . . . 81
5.2.3 The graph width explosion problem . . . 88
5.3 Dataflow Analysis on the Predicate Flow Graph . . . 89
5.4 Predicate Partial Dead Code Elimination . . . 91
5.5 Significance . . . 94

6. BOOLEAN MINIMIZATION OF PROGRAM DECISION LOGIC . . . 96
6.1 Overview of Compiler Techniques . . . 99
6.2 Minimization of Program Decision Logic . . . 106
6.2.1 Optimization of predicate expressions . . . 107
6.2.2 Two-level predicate synthesis . . . 109
6.2.3 Factorization . . . 110
6.3 Architectural Support for Synthesis . . . 115
6.4 Evaluation . . . 117
6.4.1 Processor model and benchmarks . . . 117
6.4.2 Results . . . 118
6.5 Significance . . . 123

7. PARTIAL REVERSE IF-CONVERSION . . . 124
7.1 Partial Reverse If-Conversion . . . 124
7.1.1 Analysis . . . 125
7.1.2 Transformation . . . 127
7.1.3 Policy . . . 130
7.1.4 Code example . . . 137
7.2 Experimental Evaluation . . . 140
7.2.1 Methodology . . . 140
7.2.2 Results . . . 141
7.3 Significance . . . 149

8. SYSTEMATIC COMPILATION FOR PREDICATED EXECUTION 151
8.1 System Overview 151
8.2 System Example 153
8.3 Experimental Evaluation 164
8.3.1 Experimental methodology 164
8.3.2 Derivation of results 168
8.3.3 Results 169

9. CONCLUSION 175
9.1 Summary 175
9.2 Future Research 177

REFERENCES 179

VITA 185

LIST OF TABLES

Table Page

2.1: IMPACT EPIC predicate deposit types. 19
4.1: Predicate deposit logic. 65
4.2: Predicate query operations and their complexity. 69
6.1: Speedup and predicate define count for selected functions. 121
7.1: Application frequency of partial reverse if-conversion. 149
8.1: Machine models used in the evaluation. 168

LIST OF FIGURES

Figure Page

2.1: A simple code segment (a), after if-conversion (b), and after scheduling (c). . . 16
2.2: Example of if-then-else predication in the Cydra 5. 27
3.1: Formation of a hyperblock (b) from consecutive if-then-else constructs(a). . . 33
3.2: Hyperblock formation saturating processor resources. 35
3.3: Early heuristic hyperblock formation estimates final code characteristics. Optimization changes a good hyperblock decision into a poor one. 39
3.4: Hyperblock formation of code (a), including (b) and excluding (c) a path containing a hazard. 42
3.5: Beneficial hyperblock formation of seemingly incompatible paths due to code transformations. The T and F annotations in (a) indicate the taken and fall-through paths for the conditional branch. $r2$ is not referenced outside the T block. (b) shows the hyperblock schedule. 43
3.6: Hyperblock formation of seemingly compatible paths with negative results. Hammock (a) results in a beneficial hyperblock (b). Hammock (c), with less opportunity for cross-path optimization, results in a performance-degrading hyperblock (d). 45
3.7: An efficient hyperblock (b) and tail (c) formed from hammock (a) through the inclusion of a partial path. 47
3.8: The If-Conversion during Scheduling Framework. 49
3.9: The Partial Reverse If-Conversion Predication Framework. 50
4.1: Constant propagation optimization in a predicated region. Branching code (a) and hyperblock (b) in which optimization is invalid, branching code (c) and hyperblock (d) in which it is valid. 57
4.2: Speedup due to post-if-conversion optimization, Predicate Analysis System and approximate methods. 72
5.1: A predicate flow graph example. 76
5.2: A simple if-then-else construct in the original C code (a), the control flow graph for nonpredicated code (b), and the control flow graph for the predicated code (c). 77
5.3: The CFG (b) and PFG (c) for a simple if-then-else construct (a). 79

5.4: An example of incorrectly using live range as active range. 80
5.5: Algorithm to compute predicate active ranges for a hyperblock. 82
5.6: Algorithm to create the predicate flow graph for a hyperblock. 84
5.7: SplitPredBlocks function for creation of Predicate Flow Graph. 85
5.8: MergePredBlocks function for creation of Predicate Flow Graph. 86
5.9: Insertion of operands into the Predicate Flow Graph of a Hyperblock. 88
5.10: Explosion of a predicate flow graph (b) representing independent predicates with overlapping active ranges (a). 89
5.11: Predicate block definition and use computation for live variable analysis. . . . 91
5.12: Merging of dataflow sets for traditional queries. 92
5.13: Partial dead code in a predicate flow graph. 93
6.1: A portion of the inner loop of the Unix utility *wc*. The control flow graph (a), and the corresponding hyperblock formed after complete if-conversion (b). 100
6.2: The *wc* hyperblock after speculation but before logic minimization (a) and its corresponding logic diagram (b). The hyperblock after logic minimization (c) and its corresponding logic diagram (d). 101
6.3: Comparison of the static schedules for the *wc* hyperblock before and after logic minimization. 105
6.4: Example: optimization of *wc* predicate network. 108
6.5: Pseudo-code for performing optimization of predicate expressions 113
6.6: Factorized predicate define optimization. 114
6.7: Various methods of predicate expresssion regeneration. 116
6.8: Speedup from minimization of program decision logic. 119
7.1: Predicate flow graph (b) for predicated code segment (a). Partial dead code elimination is applied, given that $r3$ and $r4$ are not live out of this region. . . 126
7.2: Predicate flow graph (a) and a partial reverse if-conversion of predicate $p1$ located after instructions 1 and 2 (b). 128
7.3: Simple code size reduction on multiple partial reverse if-conversions applied to an unrolled loop. Each square represents an unroll of the original loop. . . . 129
7.4: An algorithm incorporating partial reverse if-conversion into a list scheduler. 132
7.5: The control flow graph from the Unix utility *wc*. 137
7.6: Schedule for *wc* after hyperblock formation and optimization (a), and schedule after partial reverse if-conversion of that hyperblock (b). 139
7.7: Performance increase over superblock exhibited by the hyperblock and partial reverse if-conversion frameworks with no misprediction penalty. 142
7.8: Performance increase over superblock exhibited by the hyperblock and partial reverse if-conversion frameworks with a four cycle misprediction penalty. . 143
7.9: Selected function level performance increase over superblock exhibited by the hyperblock and partial reverse if-conversion frameworks with no misprediction penalty. 146
7.10: Relative static code size exhibited by the hyperblock and partial reverse if-conversion frameworks compared with superblock. 147
8.1: Block diagram of the backend compilation path with hyperblocks. 152
8.2: Control flow graph for a loop (a) and its corresponding hyperblock (b). 154

8.3: The conditions (a) and predicate expressions (b) for the code segment. 155
8.4: The hyperblock before (a) and after (b) an application of the constant propagation optimization. 156
8.5: The hyperblock before (a) and after (b) promotion is performed. 158
8.6: The logic network for the computation of predicates $p3$ and $p6$ before (a) and after (b) optimization. 159
8.7: Hyperblock before (a) and after (b) Program Decision Logic Optimization. . . 160
8.8: Partial Dead Code Elimination after Program Decision Logic Optimization. . 161
8.9: Hyperblock before (a) and after (b) partial reverse if-conversion. 164
8.10: The IMPACT compiler. 165
8.11: System performance on machine 6G_1BL relative to superblock. 170
8.12: System performance on machine 6L_1BL relative to superblock. 170
8.13: System performance on machine 8G_1BL relative to superblock. 171
8.14: System performance on machine 8L_1BL relative to superblock. 171
8.15: System performance on machine 8L_2BL relative to superblock. 172
8.16: Code size on machine 8G1BL relative to superblock. 173

1. INTRODUCTION

In the continuing effort to increase computing power, computer architects have often explored the extraction and use of parallelism. The total execution time of programs can be reduced dramatically by concurrently executing meaningful independent computations. To extract parallelism, a program must be divided into independent parts at one or more levels of granularity. Coarse-grained parallelism involves the presentation of programs as a small number of large, independent code segments to large, weakly connected parallel processor hardware structures. These independent code segments, typically threads or processes, contain hundreds to millions of instructions and are executed by a multi-processor architecture. Medium-grained parallelism breaks computation into segments of tens to thousands of instructions. This parallelism, often at the loop or trace-level, is presented to architectures with high bandwidth interaction among processing structures, such as in multiscalar or vector processors. Fine-grained parallelism exploits independence at the instruction level. Such instruction-level parallelism (ILP) is characterized by the simultaneous presentation of independent instructions to a single processor with multiple data paths and functional units.

Though trade-offs exist between the various levels of granularity, fine-grained parallelism is often most desirable in certain applications. Although a dual processor parallel architecture and a dual instruction issue processor architecture can both theoretically double performance, the dual issue processor requires significantly less hardware to implement. The size of hardware has a dramatic effect on cost and power, making wide-issue ILP processors ideal for embedded devices, portables, and desktop computers, as well as for high-performance workstations. Furthermore, ILP is generally easier to produce for most applications than efficient multiprocessor codes. The suitability of ILP processors to a range of tasks is evident by their recent and increasing proliferation.

1.1 Branch Control Effects on ILP

The performance of current ILP processors depends on the ability consistently to present it with a sufficient number of independent instructions. Studies have shown that by using conventional code optimization and scheduling methods, wide ILP processors have difficulty sustaining a speedup of more than two for nonnumeric programs [1], [2], [3]. These low speedups are a result of numerous challenges encountered in the process of extracting ILP.

Although ILP can be extracted solely by the compiler, or solely by the hardware, the compiler and hardware have individual strengths and the task of extracting ILP should be divided accordingly. For example, the compiler has the ability to perform detailed analyses on large portions of the program, while the hardware has access to information that can only be determined at run time. Though extracting ILP is necessary to achieving large speedups in ILP processors, the architecture must also be able to execute the

extracted ILP efficiently. Hardware often fails to capitalize on extracted ILP due to stalls in the hardware caused by the long latency of operations in certain exceptional conditions. For example, a load from memory may stall the processor while the data is fetched from the relatively slow main memory. Hardware techniques that minimize these stalls, such as cache in case of memory, are essential for realizing the full potential of ILP. One of the major challenges to effectively extracting and efficiently executing ILP code which has not been addressed satisfactorily by hardware-only techniques is overcoming the limitations imposed by branch control flow.

At run time, the branching implementation of control introduces uncertainty of outcome and nonsequentiality in instruction layout, which limit the effectiveness of instruction fetch mechanisms. Sophisticated branch prediction techniques, speculative execution, and advanced fetch hardware are currently necessary to prevent instruction starvation in high-performance processor cores [4], [5], [6], [7].

For nonnumeric benchmarks, researchers report that approximately 20% to 30% of the dynamic instructions are branches, an average of one branch for every three to five instructions. As the number of instructions executed concurrently grows, the sustained rate of branches executed per cycle must grow proportionally to avoid becoming the bottleneck. Handling multiple branches per cycle requires additional pipeline complexity, which includes multiported branch prediction structures. In high issue rate processors, it is much easier to duplicate arithmetic functional units than to predict and execute multiple branches per cycle. Therefore, for reasons of cost, limited branch handling capabilities in ILP processors will likely have limited performance in nonnumeric applications. In

those which do support multiple branches per cycle, the prediction mechanisms are subject to diminishing prediction accuracy created by compounding branch outcomes. This problem is made more significant by the fact that branch prediction mechanisms with good accuracy on a single branch per cycle often have undesirable scaling characteristics. It is thus unlikely that a hardware-only solution to the problem of branch overhead can be found.

Branching control imposes a yet more fundamental limitation on ILP performance than the problem of predictability. An instruction is control dependent on a branch when located such that the outcome of the branch will affect its execution. Control dependences from a branch to other instructions (including other branches) are determined by positional relationships within the code. The set of possible layouts for a program is limited by the need to respect proper control dependence connections. In many cases, respecting these control dependences limits the creation of layouts with desirable ILP characteristics. For example, consider two instructions that have independent conditions of execution and that execute frequently. Because the condition of execution for each instruction is determined by its position relative to the branches that control it, these instructions must remain in separate blocks. A technique that eliminated the positional nature of program control would enable these instructions to execute in parallel.

At compile time, branches present barriers to efficient extraction of ILP in a compiler's back end. Branches impose control dependences that often serialize the execution of surrounding instructions, reducing the number of naturally independent instructions. The compiler must handle control dependences separately from data dependences, complicating aggressive compiler transformations. Typically, compilers avoid this complication by

limiting many optimizations to within a single basic block. Optimization of a code window containing multiple basic blocks requires either transformation of the code, usually with duplication, or application of complex and expensive global optimization techniques. Global basic block techniques often need to make trade-offs between the performance of different paths. The uncertainty of branch outcome forces the compiler to make path selection decisions based on estimates which can often be inaccurate.

1.2 Overcoming Branch Problems with Predication

Many techniques and devices have been proposed to overcome the limitations of branching control, some of which have been discussed above. A significant new approach taken in Explicitly Parallel Instruction Computing (EPIC) architectures allows the compiler to substantially reduce the amount of branching control in programs. This approach is called *predication* [8], [9]. Predication is a model that allows the elimination of control dependences by providing an alternate means of controlling the condition of execution for individual instructions. Predication is typically discussed in two contexts: its use as a representation and its operation during program execution.

The *predicated representation* is a low-level N-address program representation in which each operation has a Boolean source operand, its *guard predicate*, whose value determines whether the operation is executed or nullified. The values of these guard predicates can be manipulated by a set of predicate-defining instructions. The use of predicates to guard instruction execution can reduce or even completely eliminate the need for branch control dependences. When all operations that are control dependent on a branch are predicated to guard execution in the same manner as the branch, that branch

can be removed without changing program semantics. The process of replacing branches with appropriate predicate computations and guards is known as *if-conversion* [10], [11].

The predicated representation provides a useful and efficient model for compiler optimization and scheduling. The use of predicates to guard instruction execution can reduce or even completely eliminate the need for branch control dependences, because branches can legally be removed through the if-conversion process. Control dependences between branches and other instructions are thus converted into data dependences between predicate computation instructions and the newly predicated instructions. In the predicated representation after if-conversion, control flow transformations can be performed as traditional data flow optimizations. In the same way, the predicated representation allows scheduling among branches to be performed as a simple reordering of sequential instructions. Removal of control dependences increases scheduling scope and affords new freedom to the scheduler [12].

Predicated execution is an architectural model in which code expressed in the predicated representation can be executed directly. With respect to a conventional instruction set architecture, the new features are a predicate register file containing the values of the guard predicates, an additional source operand specifying the predicate register guarding each operation, and a set of predicate defining operations used to manipulate the predicate register values. Predicated instructions are fetched regardless of their predicate value. Instructions whose predicate values are true are executed normally. Conversely, instructions whose predicates are false are nullified and thus are prevented from modifying the processor state.

The removal of branches yields performance benefits in the execution of the ILP code, the most notable of which is the elimination of branch misprediction penalties. In particular, the removal of frequently mispredicted branches yields large performance gains [13], [14], [15]. Predicated execution also provides an efficient mechanism by which a compiler can explicitly present the overlapping execution of multiple control paths to the hardware. In this manner, processor performance is increased by the compiler's ability to find ILP across distant multiple program paths. Another, more subtle, benefit of predicated execution is that it facilitates the movement of redundant computation to less critical portions of the code [16].

1.3 Systematic Compilation for Predicated Execution

Since predication allows the compiler tremendous freedom in combining computation from multiple paths of control, compilation takes on some new challenges. High performance depends on forming predicated regions which do not oversubscribe available resources, but which include enough predication to make the transformation worthwhile. Furthermore, since code from multiple basic blocks is combined into a single block, local optimizations must recognize the effects of predication to achieve accurate and efficient results. Effective use of predicated execution thus requires effective compiler support. Many compiler techniques, including those for predication, have in the past been based on complicated estimation and approximation techniques. Unfortunately, these techniques often suffer due to variations in codes that do not conform to the models used to develop these heuristics. With the complexity added by predication, the problem

becomes serious, and the performance potential is often not reached as a result of inaccurate decisions. The contribution of this thesis is to create a systematic approach to compiling for predicated execution which exploits the predicated representation to fully realize predication's potential.

At the most basic level, compilers support predicated execution by performing if-conversion at some point in the compilation process. Previous work has demonstrated the value of introducing predication in an early compilation phase for purposes of later optimization; in particular, the *hyperblock* compilation framework has been shown to generate efficient predicated code [17]. The hyperblock compilation framework is able to deliver better performance than late if-conversion because it allows the compiler to enhance optimization and scheduling through the use of the predicated representation.

The hyperblock framework has some limitations. While early if-conversion allows the compiler to take full advantage of the predicated representation, the decision of what to if-convert is difficult. The over-application of predication with respect to a particular machine will result in the oversaturation of processor fetch and potentially execution resources. A delicate balance between control flow and predication is required in order to maximize predication's potential. Early if-conversion decisions are made based on the code's characteristics prior to further optimization. Unfortunately, code characteristics change dramatically due to optimizations applied after if-conversion, potentially rendering originally compatible traces incompatible. In cases where estimates of final code characteristics are wrong, the final code may perform worse for having had predication introduced. Postoptimization if-conversion yields more stable but much less impressive

performance gains due to the removal of the predicated representation from the compilation path [18].

This work presents a solution to the accurate early if-conversion problem. Early if-conversion is performed in an aggressive hyperblock framework. Optimizations are then performed on the predicated representation. Finally, a new technique called *partial reverse if-conversion* operates at schedule time to balance the amount of control flow and predication present in the generated code, based on the target processor's characteristics. Partial reverse if-conversion replaces some predicated code with branch control flow by reintroducing branches.

Effective optimization, scheduling, and register allocation of code in the predicated representation, however, require the compiler to understand the logical relationships among predicates because execution conditions are no longer solely dependent on branches. Previous work has demonstrated the importance of accurate predicate relation analysis to some phases of compilation, such as register allocation [19]. This work presents a flexible, powerful, and efficient Predicate Analysis System (PAS) and demonstrates its importance in performing optimization of predicated code.

Dataflow analysis requires information from the PAS and from branch control flow in order to generate accurate results. A new structure called the Predicate Dataflow Graph (PDFG) is presented which can accurately represent both predicate and branch execution conditions. Traditional iterative dataflow equation solvers can be applied to the PDFG with little modification.

To further take advantage of the predicated representation's usefulness in manipulating program decision, a new technique called Predicate Decision Logic Optimization

(PDLO) is introduced. PDLO extracts program semantics from the predicate computation logic. Represented as Boolean expressions in a binary decision diagram (BDD), the predicate network is optimized using standard factorization and minimization techniques. PDLO then translates the simplified program decision logic back into more efficient predicate define networks. Combined with aggressive early if-conversion and sophisticated late partial reverse if-conversion, PDLO can be used to optimize the entire decision component in programs.

The Partial Reverse If-Conversion Framework, the Predicate Analysis System, the Predicate Dataflow Graph, and the Predicate Decision Logic Optimizer together form an effective compilation framework for variants of the IMPACT EPIC architecture including IA-64. This framework is extremely effective in diminishing the previously assumed overhead due to program control flow.

1.4 Contributions

The contributions of this work are four key technologies which, taken together, provide an effective, systematic approach to compiling for predicated execution.

- **The Partial Reverse If-Conversion (PRIC) Framework.** In order to apply if-conversion effectively, two major issues must be addressed: what should be if-converted and when should the if-conversion be performed? A compiler's use of predication as a representation is most effective when large amounts of code are if-converted and when if-conversion is performed early in the compilation procedure. On the other hand, efficient execution of code generated for a processor

with predicated execution requires an appropriate balance between control flow and predication. This balance is tightly coupled with scheduling decisions based on detailed processor characteristics. This work presents a compilation framework based on partial reverse if-conversion that allows the compiler to maximize the benefits of predication as a compiler representation while delaying the final balancing of control flow and predication to schedule time.

- **Predicate Analysis System (PAS).** Effective use of the predicated representation requires that the compiler be aware of the logical relations among predicates, which are digests of the control flow graph replaced by if-conversion. This work describes the design and implementation of the accurate and efficient PAS, which provides this support. Previous approaches to predicate analysis have attempted to represent only those relationships encountered in particular modes of use of predicate defining instructions, resulting in a loss of accuracy when more general definition networks are analyzed. Furthermore, databases designed within these assumptions used special-purpose facilities based inherently on the semantics of the predicate defining instructions to represent the relationship among predicates. PAS instead maps predicate relationships to Boolean expressions in a very general and easily extensible manner, enabling both the accurate analysis of arbitrary uses of predicate defining instructions and the application of well-developed computer-aided design (CAD) techniques to the representation problem.

- **Predicate Dataflow Graph (PDFG).** Performing dataflow queries and optimizations on predicated code requires that all queried information be predicate-aware. Dataflow analysis is extremely important to optimization, scheduling, and register allocation. Using information from the PAS, the PDFG is built. A traditional dataflow analysis engine then operates on the PDFG to provide accurate dataflow results. Previous approaches to predicate dataflow analysis have made approximations that may result in performance losses, especially after the code has been heavily transformed. Additionally, the PDFG enables a new transformation called *predicate partial dead code removal*, which demotes instructions to the most restrictive allowable predicates. This technique, similar to partial redundancy elimination, proves valuable in implementing the PRIC, as well as in other applications.

- **Program Decision Logic Optimization (PDLO).** One of the major impediments to achieving high levels of ILP has been inefficient programmatic control flow. Historically, the compiler has translated the programmer's original control structure directly into assembly code with conditional branch instructions. Traditional branch handling techniques cannot significantly alter the program's inherent control structure. Original predicate techniques preserved remnants of the original control flow. However, the PAS models the underlying program logic, not a particular control flow structure. This logical form, referred to as a *program decision logic network*, can represented as a set of Boolean equations. The PDLO minimizes these equations using modified versions of logic synthesis techniques. After

minimization, the more efficient version of the program's original control flow is re-expressed in predicated code.

1.5 Overview

This dissertation is composed of nine chapters.

An overview of predicated execution is first presented in Chapter 2. Predicated execution provides an alternative to branches in representing program control. Hardware support for predication provides the compiler with the flexibility to present control efficiently to the hardware. This chapter presents the architecture and instruction set architecture extensions essential to predicated execution as provided by the IMPACT Explicitly Parallel Instruction Computing (EPIC) architecture. The IMPACT EPIC architecture supports a general form of predicated execution which will be the target of the compiler techniques presented herein.

Chapter 3 describes the challenges in compiling for a predicated architecture. This discussion leads to a review of the hyperblock compilation framework, an earlier compilation model created within IMPACT. It also explores feasibility of alternatives to the hyperblock compilation framework including the one developed in this work, the PRIC.

Chapters 4 through 7 present the four chief contributions of this thesis. Each is presented with individual relevant experimental results. The combined effect of all these mechanisms within the IMPACT compilation framework is presented in Chapter 8.

PAS and PDFG are outlined in Chapters 4 and 5. PAS and PDFG are necessary for optimizations to reach their full potential when operating in the predicated code domain. One advanced predicate-specific optimization, the PDLO, is made possible by PAS, and

is described in Chapter 6. Chapter 7 presents the partial reverse if-converter, an essential element of the PRIC Framework. It then presents an experimental comparison of the hyperblock and PRIC frameworks.

Finally, Chapter 9 develops conclusions and directions for future research based on the results of this thesis.

2. PREDICATED EXECUTION

Discussion and experimental evaluation of the techniques proposed by this work assume the features of the IMPACT EPIC architecture. This chapter presents an overview of architectural support for predicated execution in the IMPACT EPIC architecture. First, an overview of the predicated execution concept is presented. Then, the general form of predication supported in the IMPACT EPIC architecture is detailed. This chapter concludes with a brief survey of predicated execution support in present and past processors which possess a subset of the predication support in the IMPACT EPIC so that the applicability and significance of this work can be appreciated.

2.1 Overview of Predication

Predicated execution refers to the conditional execution of instructions based on the value of a Boolean source operand, referred to as the *predicate*. If the value of the predicate is true (a logical 1), the instruction is allowed to execute normally; otherwise (a logical 0), the instruction is nullified, preventing it from modifying the processor state.

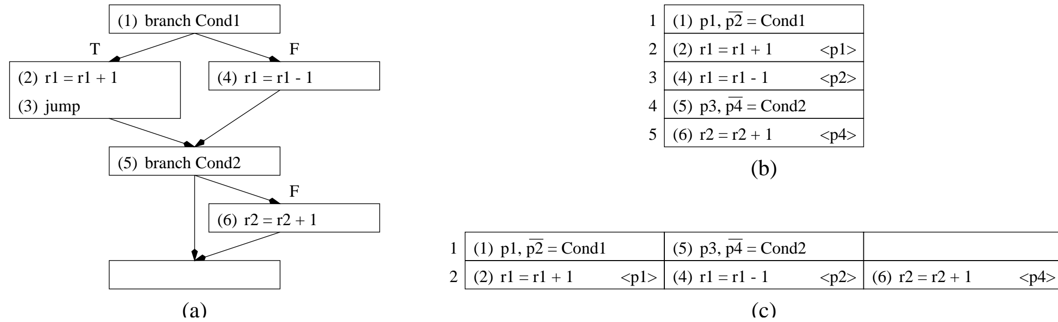


Figure 2.1: A simple code segment (a), after if-conversion (b), and after scheduling (c).

Figure 2.1 contains a simple example illustrating the concept of predicated execution. Figure 2.1(a) shows two sequential if-then-else constructs, called *hammocks*. The branch outcomes are determined by the evaluation of the independent branch conditions $Cond1$ and $Cond2$. Depending on the outcome of the first branch register, $r1$ is either incremented or decremented. Register $r2$ is incremented only when $Cond2$ is false. In order to respect the branch control dependences, the target of the first branch must be selected before the second branch can execute.

The basic compiler transformation used to exploit predicated execution is known as *if-conversion* [10], [11]. If-conversion replaces conditional branches in the code with comparison instructions that define one or more predicates. Instructions that are control dependent on a branch are then converted to predicated instructions, utilizing the appropriate predicate register. In this manner, control dependences are converted to data dependences.

Figure 2.1(b) shows the code segment after if-conversion, and Figure 2.1(c) shows the code after scheduling. Here the two branch conditions are now part of two predicate defines. Four predicates are defined to be true or false depending on the evaluation of the conditions. For example, if $Cond1$ is satisfied, P_1 is set to true and P_2 is set to false.

In the next cycle, instructions 2, 4, and 6 are executed based on the values stored in the predicate registers. These instructions now do not need to respect positional dependence with any branches, and are free to be scheduled simultaneously, as they are here. The final result is that ILP can be increased within a code segment as simple as this. Another benefit is the potential reduction in instruction count. Instruction 3 is no longer necessary since within this block of code execution conditions are no longer determined solely by fetch pattern. Further, fetch hardware now has a single very simple fetch pattern to perform instead of four patterns, of which only one is correct for each execution of the branching code segment.

Predicated code requires hardware which can support predicate defining instructions, predicate registers, and instructions having an additional predicate source operand. The remainder of this chapter presents systems that support predicated execution. First, the IMPACT EPIC architecture is presented, because it is a superset of existing systems and is the architecture chosen as the target to evaluate this work.

2.2 Predicated Execution in the IMPACT EPIC Architecture

An architecture supporting predicated execution must be able to conditionally nullify the effects of a predicated instruction based on the value of its predicate. Additionally, the architecture must support efficient computation of predicate values. The IMPACT EPIC architecture model (a statically scheduled, in-order issue, EPIC processor) is a generalization of the Cydra 5 and the HPL PlayDoh architectures [9], [20] described in Sections 2.3.3 and 2.3.1.

The IMPACT EPIC model of predicated execution contains a set of 64 independently addressable single-bit predicate registers. Predicate register 0 is defined to always hold the value 1. The model has a full set of instructions which have a guard predicate source and an orthogonal set of predicate defining instructions. These predicate defining instructions are classified broadly as predicate comparison instructions, predicate clear/set instructions, and predicate save/restore instructions.

2.2.1 Predicate comparison instructions.

The most common way to set predicate register values is with a set of predicate comparison instructions. Predicate comparison instructions compute predicate values using condition semantics similar to those for compare-and-branch instructions. The major difference is that these instructions write the comparison to one or two destination registers instead of changing flow of control. The predicate comparison instruction format is as follows:

$$\left(P_{dest_1} \overset{type_1}{\longleftarrow},\ P_{dest_2} \overset{type_2}{\longleftarrow}\right)\ (src_0\,[cmp]\,src_1)\ \langle P_{guard}\rangle$$

This computes the condition $C = src_0\,[cmp]\,src_1$ and optionally assigns a value to each of two destination predicates P_{dest_1} and P_{dest_2} according to the predicate types ($type_1$ and $type_2$, respectively), C, and the guarding predicate P_{guard}, as shown in Table 2.1. (In the table, a dash indicates that no value is deposited.) The IMPACT EPIC architecture defines six deposit types that specify the manner in which the result of a condition computation and the guard predicate are deposited into a destination predicate register. The unconditional (U), or (O), and (A), and conditional (C) types are as defined in the HP

P_{guard}	C	UT	UF	OT	OF	AT	AF	CT	CF	$\vee T$	$\vee F$	$\wedge T$	$\wedge F$
0	0	0	0	-	-	-	-	-	-	-	1	0	0
0	1	0	0	-	-	-	-	-	-	1	-	0	0
1	0	0	1	-	1	0	-	0	1	1	1	0	-
1	1	1	0	1	-	-	0	1	0	1	1	-	0

Table 2.1: IMPACT EPIC predicate deposit types.

Labs PlayDoh Specification [20]. The and (A), or (O), conjunctive ($\wedge$), and disjunctive ($\vee$) type predicate deposits are termed *parallel compares* because multiple definitions of a single one of these types can commit simultaneously to the same predicate register. Although simple if-conversion generates only unconditional and or-type defines [11], the other types are useful for optimizing predicate define networks, so an effective predicate analysis system should fully support their general use.

Unconditional predicate register destinations are always written, regardless of the value of P_{guard} and the result of the comparison. If the value of P_{guard} is 1, the result of the comparison (or its complement for UF) is placed in the predicate register. Otherwise, a 0 is written to the predicate register. Unconditional predicates are utilized for blocks that are executed based on a single condition; i.e., they have a single control dependence.

The or-type predicates are useful when execution of a block can be enabled by multiple conditions, such as logical `and` (`&&`) and logical `or` (`||`) constructs in C. Or-type destination predicate registers are set if P_{guard} is 1 and the result of the comparison is 1 (0 for OF); otherwise, the destination predicate register is unchanged. Note that or-type predicates must be explicitly initialized to 0 before they are defined and used. However, after they are initialized, multiple or-type predicate defines may be issued simultaneously and in any order on the same predicate register. This is true because the or-type predicate either writes a 1 or leaves the register unchanged, allowing implementation as

a wired logical `or` condition. This property can be utilized to compute an execution condition with zero dependence height using multiple predicate define instructions.

The and-type predicates are analogous to the or-type predicates. And-type destination predicate registers are cleared if P_{guard} is 1 and the result of the comparison is 0 (1 for AF); otherwise, the destination predicate register is unchanged. The and-type predicate is particularly useful for transformations such as control height reduction [16], [21].

Conditional-type predicate defines have semantics similar to regular predicated instructions, such as adds. If the value of P_{guard} is 1, the result of the comparison is placed in the destination predicate register (or its complement for CF). Otherwise, no actions are taken. Under certain circumstances, a conditional predicate may be used in place of an or-type predicate to eliminate the need for an initialization instruction. However, conditional predicates may not issue in parallel as do the or-type predicates. For this reason, the IMPACT compiler chooses not to generate conditional type predicates.

Two predicate types are introduced to facilitate generating efficient code using the minimization techniques described in Chapter 6. These are referred to as *disjunctive-type* ($\vee T$ or $\vee F$) and *conjunctive-type* ($\wedge T$ or $\wedge F$). Table 2.1 (right-hand portion) shows the deposit rules for the new predicate types. The $\wedge T$-type define clears the destination predicate to 0 if either the source predicate is FALSE or the comparison result is FALSE. Otherwise, the destination is left unchanged. Note that this behavior differs from that of the and-type predicate define, in that the and-type define leaves the destination unaltered when the source predicate evaluates to FALSE. The conjunctive-type thus enables the compiler to form easily and efficiently the logical conjunction of an arbitrary set of conditions and predicates.

The disjunctive-type behavior is analogous to that of the conjunctive-type. With the $\vee T$-type define, the destination predicate is set to 1 if either the source predicate or the comparison result is TRUE (FALSE for $\vee F$). The disjunctive-type is thus used to compute the disjunction of an arbitrary set of predicates and compare conditions into a single predicate.

2.2.2 Predicate clear/set instructions.

The stylized use of or-type, and-type, disjunctive-type, and conjunctive-type predicates described previously requires that the predicates be precleared and preset. Three sets of instructions are provided for these purposes. First, to individually clear and set individual predicates, *pred_clr* and *pred_set* instructions are added to the instruction set. Each takes up to two destination predicate registers and sets those destinations to the value of 0 or 1. Note that unconditional predicate comparison instructions could also be used to initialize individual predicates at the start of or-type, and-type, disjunctive-type, or conjunctive-type predicate computation chains. The clear and set instructions provide either a code-size benefit, by combining multiple initializations into a single operation, or a dependence height benefit, by allowing all condition evaluations in a composite predicate computation to be issued using parallel semantics.

A second set of instructions to clear and set groups of registers using a mask are provided. These instructions are aptly called *pred_clr_mask* and *pred_set_mask*. These instructions set a contiguous group of predicate registers to 0 or 1 using a mask. Any combination of predicates addressed by the immediate mask can be cleared or set.

2.2.3 Predicate save/restore instructions.

Extensions to the base instruction set allow two methods of saving and restoring the contents of the predicate register file. The *pred_ld_blk* and *pred_st_blk* instructions allow the loading and storing of the predicate register file in word-sized blocks. These instructions are primarily used to save/restore the caller-save predicates across subroutine calls and to save/restore the callee-save predicates at function entry and exit points. They also make saving the contents of the predicate register file during a context switch more efficient.

The second method acts on individual predicate registers and is only required if the need arises to spill predicate registers. The IMPACT compiler employs an intelligent allocation algorithm method to avoid spilling predicate registers. In the rare situation in which a predicate register has to be spilled, *pld* and *pst* instructions are used. These instructions allow an individual predicate register to be loaded from and stored to memory. In this manner, the compiler has the freedom to handle predicate registers in the same way as the conventional register types.

2.2.4 Predicate register file.

As previously mentioned, an Nx1 register file is added to the baseline architecture to hold predicates. For several reasons, introducing a new register file to hold predicate values is preferable to using the existing general purpose register file. First, it is inefficient to use a word-sized general register to hold a one bit predicate. Second, register porting is expected to be a significant problem for wide-issue processors. By keeping predicates

in a separate file, additional port demands are not added to the general purpose register file. Within the architecture, the predicate register file behaves no differently than a conventional register file. For example, the contents of the predicate register file must be saved during a context switch. Furthermore, the predicate file is partitioned into caller-save and callee-save sections based on the chosen calling convention.

2.3 Other Predicated Architectures

The IMPACT EPIC architecture is derived from and is a superset of a number of previous architectures supporting predication. These architectures are described in this section.

2.3.1 PlayDoh

The basis for predication in the IMPACT EPIC architecture is the PlayDoh architecture developed at Hewlett-Packard Laboratories [20].

PlayDoh is a parameterized EPIC architecture intended to support public research on ILP architectures and compilation [20]. PlayDoh predicate define instructions generate two Boolean values using a comparison of two source operands and a source predicate. A PlayDoh predicate define instruction has the form

$$\left(P_{dest_1} \overset{type_1}{\longleftarrow},\ P_{dest_2} \overset{type_2}{\longleftarrow}\right)\ (src_0\,[cmp]\,src_1)\ \langle P_{guard}\rangle$$

The instruction is interpreted as follows: pD_0 and pD_1 are the destination predicate registers; $type_0$ and $type_1$ are the predicate types of each destination; src_0 $[cmp]$ src_1 is the comparison, where $[cmp]$ can be *equal (==)*, *not equal (! =)*, *greater than (>)*, etc.;

and $pSRC$ is the source predicate register. The value assigned to each destination is dependent on the predicate type. PlayDoh defines four predicate types, *unconditional* (UT or UF), *conditional* (CT or CF), *wired-or* (OT or OF), and *wired-and* (AT or AF). Each type can be in either normal mode or complement mode, as distinguished by the T or F appended to the type specifier (U, C, O, or A). Complement mode differs from normal mode only in that the condition evaluation is treated in the opposite logical sense.

For each destination predicate register, a predicate define instruction can either deposit a 1, deposit a 0, or leave the contents unchanged. The predicate type indicates which function of the source predicate and the result of the comparison is to be applied to derive the resultant predicate. The eight left-hand columns of Table 2.1 show the deposit rules for each PlayDoh predicate type in both normal and complement modes, indicated as T and F in the table, respectively.

The major limitation of the PlayDoh predicate types is that logical operations can only be performed efficiently among compare conditions. There is no convenient way to perform arbitrary logical operations on predicate register values. While these operations could be accomplished using the PlayDoh predicate types, they often require either a large number of operations or a long sequential chain of operations, or both.

With traditional approaches to generating predicated code, these limitations are not serious, as there is little need to support logical operations amongst predicates. The Boolean minimization strategy described in Chapter 6, however, makes extensive use of logical operations on arbitrary sets of both predicates and conditions. In this approach, intermediate predicates are calculated that contain logical subexpressions of the final predicate expressions to facilitate reuse of terms or partial terms. The intermediate

predicates are then logically combined with other intermediate predicates or other compare conditions to generate the final predicate values. Without efficient support for these logical combinations, gains of the Boolean minimization approach are diluted or lost.

2.3.2 Predicated execution support in Intel IA-64

The Intel IA-64 architecture supports full predication with a bank of 64 predicate registers and a complement of two-destination predicate defining instructions. IA-64 supports the four basic deposit types proposed in the HP PlayDoh model, although the predicate defining instructions have a different format than in the earlier model. IA-64 provides predicate defining instructions which support several combinations of types for the two deposits performed; for example, a template is provided which deposits using an AT semantic into one destination and an OF semantic into the other destination. Not all combinations of deposit types can be combined into a single predicate defining instruction [22]. Like the PlayDoh model, the IA-64 model is a subset of the IMPACT EPIC model, which is capable of combining predicate deposit types arbitrarily as well as of using two new predicate deposit types not defined in IA-64. Thus, techniques presented here for analysis of IMPACT-style predication are directly applicable to IA-64 [22], and techniques presented for predicate optimization are applicable with some modification [23].

2.3.3 Predicated execution support in the Cydra 5

The Cydra 5 system is a very long instruction word (VLIW), multiprocessor system utilizing a directed-dataflow architecture [9], [24]. Each Cydra 5 instruction word contains

seven operations, each of which may be individually predicated. An additional source operand added to each operation specifies a predicate located within the predicate register file. The predicate register file is an array of 128 Boolean (one bit) registers.

The content of a predicate register may only be modified by one of three operations: *stuff*, *stuff_bar*, or *brtop*. The *stuff* operation takes as operands a destination predicate register and a Boolean value as well as an input predicate register. The Boolean value is typically produced using a comparison operation. If the input predicate register contains 1, the destination predicate register is assigned the Boolean value. Otherwise, the destination predicate is assigned the value 0. The *stuff_bar* operation functions in the same manner, except the destination predicate register is set to the inverse of the Boolean value when the input predicate value is 1. The semantics of the unconditional predicates are analogous to those of the *stuff* and *stuff_bar* operations in the Cydra 5. The *brtop* operation is used for loop control in software pipelined loops and sets the predicate controlling the next iteration by comparing the contents of a loop iteration counter to the loop bound.

Figure 2.2 shows the previous example after if-conversion for the Cydra 5. Setting the mutually exclusive predicates for the different execution paths shown in this example requires three instructions. First, a comparison must be performed, followed by a *stuff* to set the predicate register for the true path (predicated on P_1) and a *stuff_bar* to set the predicate register for the false path (predicated on P_2). This results in a minimum dependence distance of 2 from the comparison to the first possible reference of the predicate being set.

```
    mov   r1,0
    mov   r2,0
    ld_i  r3,A,0
L1:
    ld_i  r4,r3,r2
    gt    r6,r4,50
    stuff p1,r6
    stuff_bar p2,r6
    add   r5,r5,1 (p2)
    add   r6,r6,1 (p1)
    add   r1,r1,1
    add   r2,r2,4
    blt   r1,100,L1
```

Figure 2.2: Example of if-then-else predication in the Cydra 5.

In the Cydra 5, predicated execution is integrated into the optimized execution of modulo scheduled inner loops to control the prologue, epilogue, and iteration initiation [25], [26]. When used in conjunction with a rotating register file, predicated execution eliminates almost all code expansion otherwise required for modulo scheduling. Predicated execution also allows efficient modulo scheduling of loops with conditional branches.

2.3.4 Predicated execution support in ARM

The Advanced RISC Machines (ARM) processors consist of a family of processors, targeted for embedded and multimedia applications, that specialize in low cost and very low power consumption [27]. The ARM instruction set architecture supports the conditional execution of all instructions. Each instruction has a 4-bit condition field that specifies the context for which it is executed. By examining the condition field of an instruction and the condition codes in a processor status register, the execution condition

of each instruction is calculated. The condition codes are typically set by performing a compare instruction. The condition field specifies under what comparison result the instruction should execute, such as “equal to,” “less than,” or “less than or equal to.” When the compare instruction result contained in the processor status register matches the condition field, the instruction is executed. Otherwise, the instruction is nullified. With this support, the ARM compiler can eliminate conditional branches from the instruction stream.

2.3.5 Conditional instructions in TI C6X

Texas Instruments’ TMS320C6000 VelociTI architecture supports a form of predicated execution. Five general purpose registers, two in the A bank and three in the B bank, may be used to guard the execution of instructions, either in the positive (execute when the register contains a nonzero value) or negative (execute when the register contains zero) sense. These values may be written using any ordinary instruction, and a set of comparison instructions which deposit into these registers is also provided.

2.3.6 Limited predicated execution support in other systems

Many other contemporary processors offer some form of limited support for predicated execution. A conditional move instruction is provided in the DEC Alpha, SPARC V9, and Intel Pentium Pro processor instruction sets [28], [29], [30]. A conditional move is functionally equivalent to a predicated move. The move instruction is augmented with an additional source operand specifying a condition. As with a predicated move, the contents of the source register are copied to the destination register if the condition is true;

otherwise, the instruction does nothing. The DEC GEM compiler can efficiently remove branches utilizing conditional moves for simple control constructs [31]. The HP PA-RISC instruction set provides each branch, arithmetic, and logic instruction the ability to conditionally nullify the subsequent instruction [32]. This feature has been utilized extensively in the IMPACT compiler to emulate predicated execution support on the HP platform.

The Multiflow Trace 300 series machines supported limited predicated execution by providing *select* instructions [33]. Select instructions provide more flexibility than conditional moves by adding a third source operand. The semantics of a select instruction in C notation are as follows:

select *dest,src1,src2,cond*

dest = (*(cond) ?* src1 *:* src2 *)*

Unlike the conditional move instruction, the destination register is always modified with a select instruction. If the condition is true, the contents of *src1* are copied to the destination; otherwise, the contents of *src2* are copied to the destination register. The ability to choose one of two values to place in the destination register allows the compiler to effectively choose between computations from "then" and "else" paths of conditionals based upon the result of the appropriate comparison.

Vector machines have had support for conditional execution for many years in the form of mask vectors [34]. A mask of a statement S is a logical expression whose value at execution time specifies whether or not S is to be executed. The use of mask vectors allows vectorizing compilers to vectorize inner loops with if-then-else statements.

Although there are a great variety of implementation styles for predicated execution, all can be mapped to the generalized IMPACT EPIC model for purposes of analysis and optimization using the techniques presented here.

3. COMPILATION FRAMEWORKS FOR PREDICATION

Effective use of predicated execution presents a difficult challenge for ILP compilers. Although predication offers the potential for large performance gains when efficiently utilized, an imbalance of predication and control flow in the generated code can lead to dramatic performance losses. The primary determining factor is whether the compiler can select the appropriate sets of paths for inclusion in predicated regions. The flexibility of the *hyperblock* makes it a natural choice for the basic unit of predicated code used in this work. The hyperblock is a structure created to facilitate optimization and scheduling for predicated architectures [17], [35]. A hyperblock is a set of predicated basic blocks in which control may enter only from the top, but may exit from one or more locations. Hyperblocks are formed by applying tail duplication and if-conversion over a selected set of paths.

This chapter begins with a characterization of hyperblock quality, which provides valuable insight into the selection of an appropriate hyperblock formation mechanism and compiler phase ordering. Next discussed is the motivation for performing if-conversion

early in the compilation process. The chapter concludes with an exposition of the problems that arise in developing a heuristic for performing early (pre-schedule-time) hyperblock formation. The predication compilation framework presented in Chapter 7 is motivated by the desire to extract high levels of ILP through the effective use of both predicated execution and the predicated representation despite the complications presented in this chapter.

The principles developed in the following section are demonstrated using specifically designed, hand-crafted code examples. For clarity, these examples are necessarily smaller than hyperblocks typically created during compilation. The ideas illustrated, however, are culled from years of experience compiling real benchmarks in a hyperblock compilation framework. Unless noted otherwise, all schedules are for a three-issue, uniform functional unit machine with unit latencies.

3.1 Beneficial Path Inclusion

Building good hyperblocks entails selecting the appropriate execution paths for inclusion. As might be expected, the scope of hyperblock-based optimizations and the benefit to be derived from run-time effects of predication increase with hyperblock size, as long as included paths are well selected. The following example demonstrates some of the characteristics of a well-formed hyperblock.

Figure 3.1 demonstrates the if-conversion of two consecutive if-then-else constructs, or hammocks. Figure 3.1(a) shows a very simple control flow graph before hyperblock formation. It consists of six instructions, three of which are branches. The instructions in Figure 3.1(a) are given numbers which uniquely identify them and which indicate in

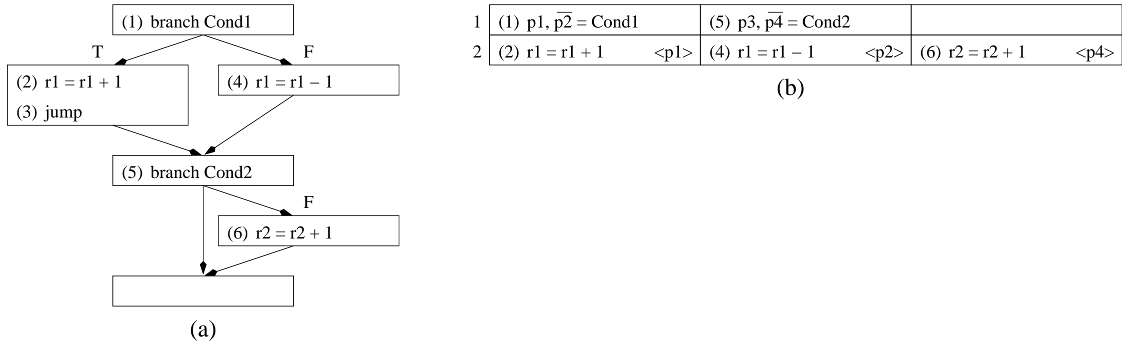


Figure 3.1: Formation of a hyperblock (b) from consecutive if-then-else constructs(a).

what order the code would be laid out. The static basic block schedule for this code on a traditional machine cannot have a height of less than four cycles due to its dependence height. Parallel execution of this code segment is limited solely by control dependences. One subset of these control dependences, which sets a lower bound on the dependence height, consists of the control dependences $1 \rightarrow 2$, $2 \rightarrow 5$, and $5 \rightarrow 6$. Other properties worth noting in the original code are related to the number and types of instructions. There is a total of six instructions: two conditional branches, one unconditional branch, and three computation instructions. The preponderance of branches in this code segment has a parasitic effect on performance, the severity of which depends on the branch-handling capabilities of the target architecture.

Figure 3.1(b) shows the same code segment after hyperblock formation including all paths. Conditional branches 1 and 5 have been converted to predicate define instructions. Branch 3 has been removed by combining the four original control flow paths into one. Elimination of branch 3 reduces the instruction count of this code from six to five. The removal of all branches eliminates their corresponding control dependences. While the resulting hyperblock contains no control dependences, it does contain new

data dependences $1 \rightarrow 2$, $1 \rightarrow 4$, and $5 \rightarrow 6$ created by the sourcing and sinking of predicate registers. However, these data dependences are more desirable than the original control dependences because they allow a two cycle schedule and allow overlapping the consecutive hammocks. The ability to overlap noncomplementary paths can be a source of significant ILP. Assuming the three-issue machine described earlier, neither the dependence height nor the resource subscription force the schedule to more than two cycles. The result is a code schedule that halves the execution time even in the presence of perfect branch prediction.

The benefits of the resulting hyperblock do not stop at the static code schedule. The removal of two conditional branches eliminates their mispredictions and associated misprediction penalties. Any taken branch penalties which may have existed are also removed. The hyperblock's straight-line code segment can be a great improvement over the disjoint blocks of the original code for instruction fetch mechanisms.

3.2 Harmful Path Inclusion

As suggested by the previous section, the benefits of hyperblock formation increase with the inclusion of more paths. However, for practical machines, compilers, and code characteristics there are many constraints on the size of effective hyperblocks.

One reason why including a path can be detrimental to good hyperblock performance is that it causes excessive resource consumption. Figure 3.2 illustrates this problem. Figure 3.2(a) is a single hammock with two control flow paths controlled by branch instruction 1. The taken path consists of five instructions and has a dependence height of 2 cycles. The fall-through path also consists of five instructions and also has a dependence

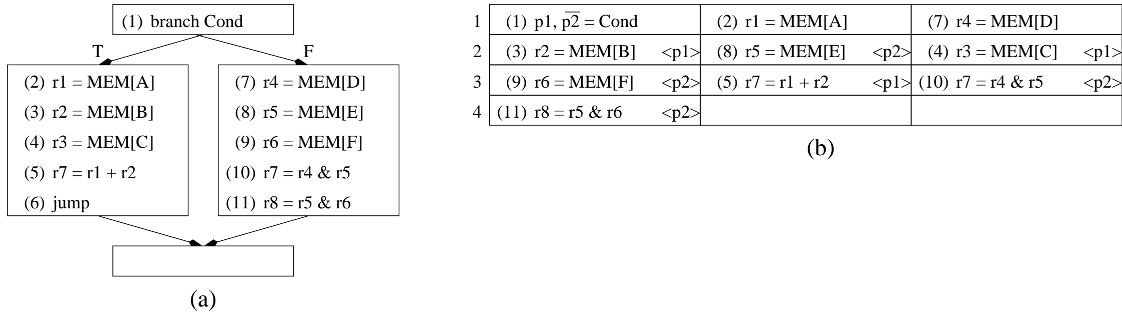


Figure 3.2: Hyperblock formation saturating processor resources.

height of two. Since the machine model assumed for the examples has uniform functional units, instruction count can be an accurate measure of the subscription to machine resources. Scheduling the code of Figure 3.2(a) for this machine model results in each path fitting into two cycles.

Figure 3.2(b) shows the hyperblock formed by the if-conversion of these paths together. As expected, the resulting number of instructions is roughly equal to the sum of the instruction counts of the original paths. Scheduling for the three-issue machine model shows that this hyperblock requires four cycles for completion, compared with the three cycles needed for the original code. This does not take into account the branch and cache penalties eliminated by if-conversion. However, in some machines these benefits would be stripped by the increase in schedule height. Certainly, in actual code where path instruction counts number in the hundreds, resource over-subscription has the potential to negate all benefits of hyperblock compilation.

The problem of predicting the resource consumption due to including a path is not easily solved. In the uniform functional unit machine, the expected resource height is simply the sum of all instructions in each path included divided by the issue width after accounting for the removal of branches and creation of predicate define instructions.

However, in practical machines, the types and number of instructions that can be issued together complicate this calculation.

The example of Figure 3.2 clearly illustrates that resources available in the target machine must be considered when choosing whether or not to include a path. While resource height is important, it is not the only consideration. Dependence height can also cause harmful path interaction. Consider a simple hammock of two paths, one with a dependence height of two cycles and the other with a dependence height of eight cycles. A hyperblock containing both paths cannot complete until all of its constituent paths have completed. Therefore, the overall height of the hyperblock must be the maximum of all the original path's dependence heights. This means that every time the short path would have been executed in two cycles, it is now executed in eight cycles–four times slower–resulting in poor hyperblock performance.

The problem is more complicated than solving independently for resource consumption and dependence height in the hyperblock formation decision process. There can exist many interactions between dependence height and resource consumption in a code schedule. For instance, the number of instructions available for issue in the first cycle may be much greater than the number of instructions available for issue later in the schedule due to flow dependences among the instructions. This is the case in Figure 3.6 on page 45, as will be discussed later. Considering this issue alone suggests that hyperblock formation decisions may belong in the scheduling phase.

3.3 Phase Ordering Considerations

Having established a concept of what make good and bad paths for inclusion in hyperblocks, or the "what to if-convert" question, we now turn our attention to the other question, of when it is most profitable to perform if-conversion in an ILP compiler. Although the previous section concluded with a strong justification for delaying hyperblock formation until schedule time, there are many compelling reasons for wanting to incorporate predication as early as possible in the compiler back-end. The benefits of performing significant optimization subsequent to if-conversion fall into two general categories. The first category consists of optimizations which could be performed in a nonpredicated environment, but which can be effected much more naturally and with less compiler complexity in the predicated domain. The second category consists of optimizations which take particular advantage of the features of predicated execution, and which cannot be implemented without it.

Applying if-conversion to create hyperblocks early enables the full use of the predicated representation by the compiler to facilitate ILP optimizations and scheduling. The formation of relatively large hyperblocks early in the compilation process allows many local transformations to have global scope. Consider the local scheduling of a hyperblock for example, in which instructions from various paths can be scheduled freely with respect to each other. To achieve the same effect in a nonpredicated environment, a global scheduler would have to be employed to move code among the constituent basic blocks. The formation of hyperblocks early in the compilation process also enables local data

flow optimizations to optimize across dependences which were converted from control dependences. This can have a powerful effect on the quality of code generated.

Furthermore, when traditional data flow optimizations are applied to predicate defines themselves, powerful control optimizations can result. For example, common subexpression elimination when applied to predicate defines can have the same effect as branch merging. Put another way, branch merging and many other control flow transformations can be performed by simple data flow transformations which already exist in most compiler infrastructures. The need to implement optimizations that perform control flow transformations can be eliminated. While these benefits of early hyperblock formation are important in simplifying compiler design, a compiler could be built that performs all the necessary optimizations to achieve the same level of code quality.

Finally, there exists a large and growing body of optimizations whose effectiveness depends on being able to generate a predicated result, or which have no practical control flow equivalent. The predicated representation allows complex control flow transformations to be recast into the data dependence domain, making them practical and profitable. Examples of such transformations include branch reordering, control height reduction [16], and branch combining [36], and optimization of program decision logic [23]. Some outcomes of these and other optimizations can only be represented effectively to the hardware using predication. The introduction of predication after most optimizations are complete would make these optimizations impossible or at least would reduce their effectiveness. Work continues in exploring the full potential of predication and the predicated representation and the case for early hyperblock formation can thus be expected to strengthen.

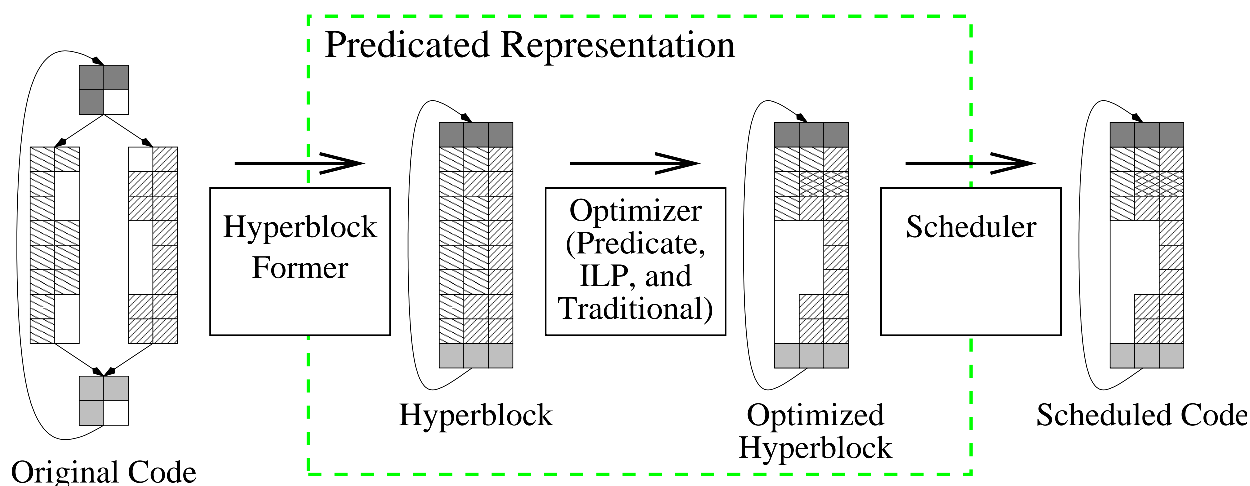


Figure 3.3: Early heuristic hyperblock formation estimates final code characteristics. Optimization changes a good hyperblock decision into a poor one.

3.4 The Hyperblock Compilation Framework

Having examined the rationale for performing if-conversion early, we finally turn to a consideration of the means by which paths could be selected for inclusion in hyperblocks. Balancing control flow and predication in hyperblocks is a relatively easy process when performed after all other code transformations are complete. However, to take full advantage of predication, optimization and scheduling passes must be performed on the hyperblocks. The following discusses various mechanisms for selecting paths, and problems with performing hyperblock path selection before schedule time.

The original approach used in the IMPACT compiler to support predicated execution is called the *Hyperblock Compilation Framework* and is shown in Figure 3.3. In this framework hyperblocks are formed using a path selection heuristic in an early compilation phase. The early phase if-conversion exposes the predicated representation throughout all the back-end compilation phases. Inclusion of a particular path into a hyperblock is

done by considering its profitability. Profitability is determined by five pieces of information: resource utilization, dependence height, hazard presence, execution frequency, and branch behavior. Ideally, a hyperblock contains paths which together fully utilize machine resources, which are balanced in height, and which do not contain hazards. Further, in a model hyperblock most instructions are not nullified often, and only well-behaved branches remain. Heuristic hyperblock formation has been shown to perform well for relatively regular machine models [36]. In these machines, resource consumption, dependence height, and hazards are managed effectively by the carefully crafted heuristics.

Hyperblock formation is a complex decision making process which requires a carefully crafted heuristic. Clearly, the heuristic must be sophisticated enough to consider the resource utilization, dependence height, hazard presence, and execution frequency for each path. Additionally, the heuristic must understand how an included path will interact with other included paths. For example, the inclusion of two paths which share a branch allows for that branch's removal, so the heuristic must also consider the run-time behavior of shared branches in candidate path sets. Path selection is further complicated by the fact that the inclusion or exclusion of particular paths can have a dramatic influence on the effectiveness of further optimizations and scheduling, as well as on how well the code after optimization will utilize the target machine's resources.

Oversubscription to resources and imbalances in dependence height are the two most common causes of poor hyperblock performance. However, other less obvious factors play a role in a hyperblock's effectiveness. One such factor is the presence of hazards. A hazard is any instruction or set of instructions which hinders efficient scheduling of paths

other than its own. Hazards can exists as instructions referencing registers which cannot be renamed, as ambiguous stores, or as `jsr`'s with unknown or conflicting side-effects. Since hazards affect other paths by definition, their negative effects can be eliminated by keeping them separate from other paths. This can be done by excluding them through tail duplication.

Figure 3.4(a) shows the control flow graph of a code segment which contains a hazard. Since the fall-through path contains one instruction and has a dependence height of one, it seems that including it would not adversely affect the hyperblock's dependence height and resource consumption characteristics. Figure 3.4(b) shows the hyperblock formed by including this fall-through path. In this schedule, it is assumed that the `jsr` has an unknown pointer store. Since the called procedure could potentially store to locations B or C, memory dependences $4 \rightarrow 5$ and $4 \rightarrow 6$ need to be created and respected. The result of this hyperblock schedule is that six cycles are required regardless of branch 3's direction. Figure 3.4(c) shows a hyperblock formed by excluding the fall-through path. Here the schedule is only three cycles when the branch is not taken. The destination of instruction 3 is a block containing instruction 4 as well as a copy, or tail duplication, of instructions 5 through 8. This tail duplication makes it possible to obtain the short schedule on the fall-through path. It is important to note that the path containing the `jsr` still has a six cycle schedule, three cycles in the Figure 3.4(c) hyperblock and three cycles in the tail-duplicated code.

As demonstrated, identifying and excluding hazards is very important to resulting code performance. Unfortunately, identifying hazards is a difficult problem. Consider the same example if we had assumed that all of instruction 4's memory operations were

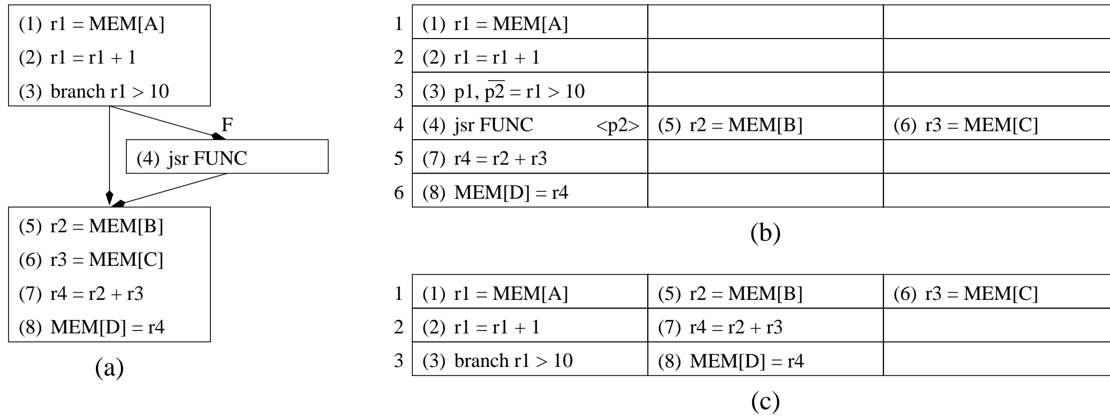


Figure 3.4: Hyperblock formation of code (a), including (b) and excluding (c) a path containing a hazard.

known to be independent with respect to instructions 5 and 6. In that case, the resulting hyperblock would have been four cycles for both paths. Clearly, the `jsr` is only a hazard when it creates dependences between itself and instructions in other paths. The type and number of hazards is often a function of the capabilities of the compiler. For example, the level of memory disambiguation in a compiler determines which store instructions are hazards to efficient code generation.

3.5 Challenges Posed by Post-If-Conversion Optimization

While it might seem reasonable that a heuristic could be designed to take all these effects into account, the following example will show that optimization subsequent to hyperblock formation may render early heuristic-based selection decisions incorrect.

Figure 3.5(a) shows another simple hammock to be considered for if-conversion. The taken path consists of a dependence height of two and a resource consumption of three instructions after if-conversion. The fall-through path consists of a dependence height of six and a resource consumption of six instructions. A simple estimate would indicate that

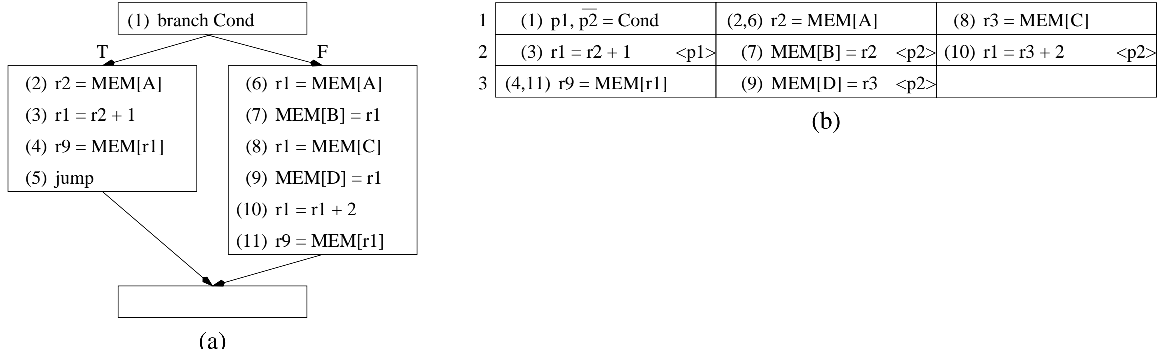


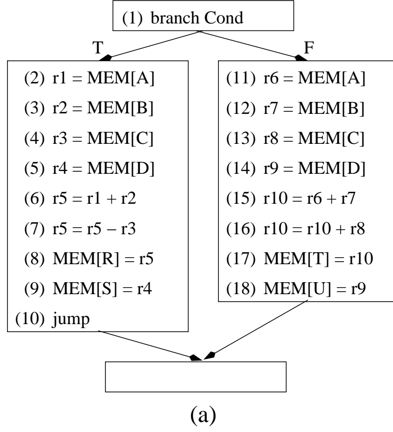
Figure 3.5: Beneficial hyperblock formation of seemingly incompatible paths due to code transformations. The T and F annotations in (a) indicate the taken and fall-through paths for the conditional branch. $r2$ is not referenced outside the T block. (b) shows the hyperblock schedule.

combining these paths together would result in a four-cycle penalty for the taken path due to the fall-through path's long dependence height. Figure 3.5(b) shows this code segment after hyperblock formation and further optimizations. The first optimization performed, renaming, eliminates the false dependences $7 \rightarrow 8$ and $8 \rightarrow 10$. This reduces the dependence height of the hyperblock to three cycles.

If a heuristic could foresee that dependence height would no longer be an issue, it might still choose not to form this hyperblock due to its level of resource consumption. Based on an inspection of Figure 3.5(a), a heuristic might estimate a hyperblock size of 10 instructions. Unfortunately, execution of 10 instructions requires at least four cycles on a three-issue machine, penalizing the taken path by one cycle. Thus a heuristic might decide that the combination of these paths would not be beneficial. After an instruction merging optimization in which instructions 2 and 6 are combined and 4 and 11 are combined, the instruction count becomes 8. The final schedule takes only three cycles, a win for both paths when the control dependence from instruction 1 is considered.

The example of Figure 3.5 shows that even in simple cases an early hyperblock formation heuristic must anticipate the optimizations to be applied afterwards in order to form hyperblocks effectively. In this example, some optimizations, such as renaming, could have been performed before hyperblock formation. Others like operation merging, however, can be performed only after hyperblock formation. In addition, some optimizations may be done differently because of different trade-offs which would be made in the context of the different code characteristics.

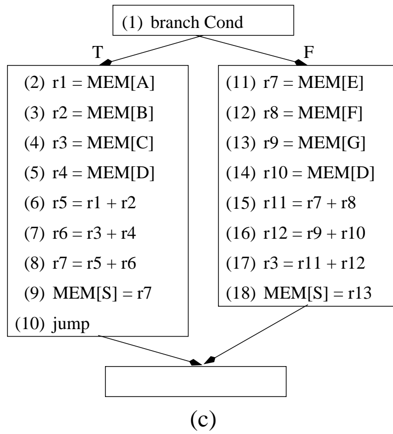
Anticipating exactly how effective future optimizations would be is an intractable problem, but perhaps empirical data could be used to estimate the aggregate effectiveness of optimization. Using this information, the hyperblock former might be able to make intelligent decisions. Assume that empirical data correctly suggests that the resource consumption of the resulting hyperblock will be 66% of the original, and that its dependence height will remain unchanged. Under these conditions the generated estimates may still be misleading, as the example of Figure 3.6 demonstrates. Figure 3.6(a) consists of two paths, each with a dependence height of four. The resource consumption, or equivalently for this machine, the instruction count, is 18 instructions. Using the predicted optimization effects, the resulting hyperblock should contain 12 instructions and have a dependence height of four cycles, for a total cycle count of four cycles. Indeed, Figure 3.6(b) shows that the optimized code meets these expectations. Unfortunately, predicting the outcome of optimization in this way is inaccurate in the majority of cases. Consider the code segment shown in Figure 3.6(c). Like the code in Figure 3.6(a), this segment of code also has two paths with dependence heights of 4 and 18 instructions. Figure 3.6(d) shows this code after optimization. Here the predicted dependence height



(a)

1	(2,11) r1 = MEM[A]	(3,12) r2 = MEM[B]	(4,13) r3 = MEM[C]
2	(1) p1, $\overline{p2}$ = Cond	(5,14) r4 = MEM[D]	(6,15) r5 = r1 + r2
3	(7) r5 = r5 − r3	(16) r10 = r5 + r8	(9) MEM[S] = r4 <p1>
4	(8) MEM[R] = r5 <p1>	(17) MEM[T] = r10 <p2>	(18) MEM[U] = r4 <p2>

(b)



(c)

1	(1) p1, $\overline{p2}$ = Cond	(2) r1 = MEM[A]	(3) r2 = MEM[B]
2	(11) r1 = MEM[E] <p2>	(12) r2 = MEM[F] <p2>	(4) r3 = MEM[C] <p1>
3	(6,15) r5 = r1 + r2	(13) r3 = MEM[G] <p2>	(5,14) r4 = MEM[D]
4	(7,16) r6 = r3 + r4		
5	(8,17) r7 = r5 + r6		
6	(9,18) MEM[S] = r7		

(d)

Figure 3.6: Hyperblock formation of seemingly compatible paths with negative results. Hammock (a) results in a beneficial hyperblock (b). Hammock (c), with less opportunity for cross-path optimization, results in a performance-degrading hyperblock (d).

and resource count was obtained. However, the resulting schedule height is six cycles, 50% higher than the estimate. The discrepancy between the predicated schedule and the actual schedule is caused by the destructive interaction between dependences and resources. This effect demonstrates that to be useful the exact outcome of further optimization must be predicted not just at a high level, but in a very accurate and precise manner. Due to the complex interactions of different optimizations, the only reliable way to predict the exact outcome of optimizations is actually to perform them. Since these

optimizations need to be applied to each candidate path selection set for each hyperblock, the optimization anticipation method is either cost-prohibitive or too inaccurate to be useful.

The challenge of forming efficient hyperblocks does not end there. Further complications result when we consider the fact that including part of a path may sometimes be more beneficial than including or excluding that entire path. This gives a hyperblock formation heuristic many more possibilities to consider. It also gives it the responsibility to be accurate at the instruction level, not just the path level. Taken together, this further complicates the already seemingly impossible task it must perform. If-conversion which can include parts of paths is referred to as *partial if-conversion*. Partial if-conversion is generally effective when the resource consumption or dependence height of an entire candidate path is too large to permit profitable if-conversion, but there is a performance gain to be had by overlapping a part of the candidate path with the other paths selected for inclusion in the hyperblock.

Figure 3.7 shows an example in which partial path inclusion results in better code than full inclusion or exclusion. Figure 3.7(a) shows two otherwise incompatible paths. However, by including all of the taken path and 4 instructions from the fall-through path an efficient hyperblock is created. This hyperblock is shown in Figure 3.7(b). Notice that branch instruction 2 has been split into two instructions: the condition computation, labeled $2'$, and a branch based on that computation, labeled $2''$. The schedule did not benefit from the complete removal of branch instruction 2, as the branch instruction $2''$ has the same characteristics of the original. However, the schedule did benefit from the

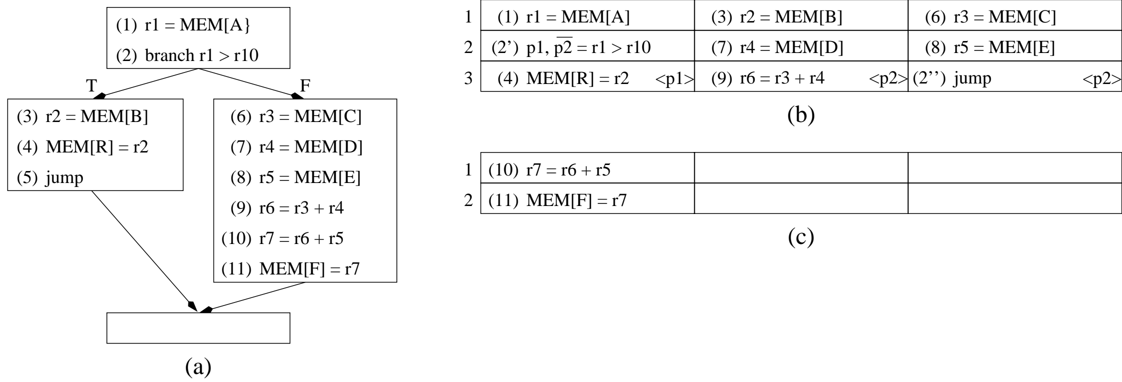


Figure 3.7: An efficient hyperblock (b) and tail (c) formed from hammock (a) through the inclusion of a partial path.

extra speculation drawn from both paths. The destination of branch instruction $2''$ is shown in Figure 3.7(c).

3.6 Compilation Frameworks

Thus far, this chapter has described the challenges of compiling effectively for predicated execution. Efficient code must exhibit a delicate balance between control flow and predication. This balance is highly dependent on final code characteristics and the resource characteristics of the target processor. An effective compilation framework for predicated execution must provide a means of making intelligent trade-offs between control flow and predication so the desired balance can be achieved. The best time to create an accurate balance is after all code transformations have been performed. An over-simplistic solution to this problem is to apply if-conversion at schedule time, after all other optimizations have been performed. While this phase ordering can use predication in a beneficial way, performance will fall short of predication's full potential.

As described in Section 3.3, if-conversion must be done before optimization in order to take full advantage of the predicated representation and of predicate optimizations. In the IMPACT compiler this has been accomplished in the context of the Hyperblock Compilation Framework, as described in Section 3.4. The remainder of this section examines two frameworks, the If-Conversion during Scheduling Framework and the Partial Reverse If-Conversion Framework, which attempt to reconcile the need to perform optimization after if-conversion with the need to form carefully selected regions. The Partial Reverse If-Conversion Framework, the subject of this chapter, is successful because it is designed to capture the desirable properties of both other frameworks.

3.6.1 The If-Conversion during Scheduling Framework

Given the difficulties with forming hyperblocks early in the back-end compilation process, a seemingly natural strategy is to perform if-conversion in conjunction with instruction scheduling [18]. This can be achieved by integrating if-conversion into the scheduling process itself. The scheduler not only accurately models the detailed resource constraints of the processor but also understands the performance characteristics of the code. Therefore, the scheduler is ideally suited to making intelligent if-conversion decisions. In addition, all compiler optimizations are usually complete when scheduling is reached, thus the post-if-conversion path transformation problem is reduced.

Delaying if-conversion until scheduling, however, prevents the compiler from using the predicated representation to perform control flow transformations or predicate-specific optimizations as shown in Chapter 3.3. One solution to this problem, the If-Conversion

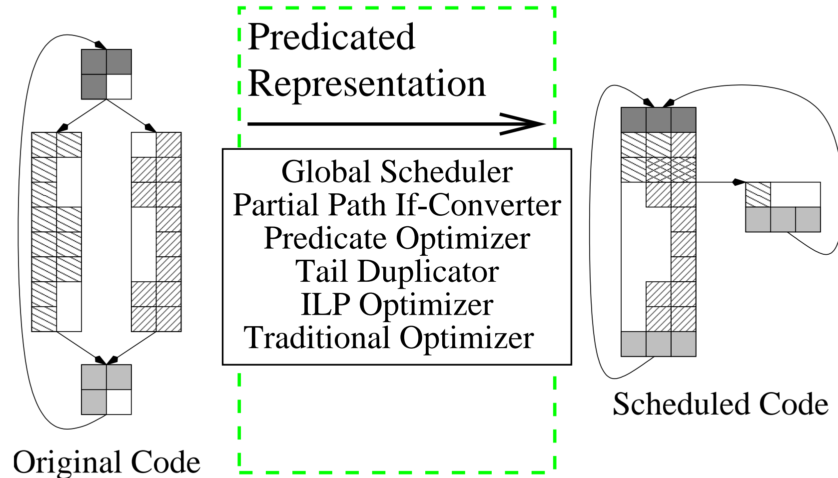


Figure 3.8: The If-Conversion during Scheduling Framework.

during Scheduling Framework shown in Figure 3.8, delays to schedule time the optimizations which require predication. Unfortunately, delaying only some optimizations until schedule time creates a problematic compiler phase ordering. Generally, most transforms have profound effects on one another and must be applied repeatedly in turn to achieve desirable results. For example, one predicate-based transformation, such as control height reduction [16], may expose a critical data dependence edge that should subsequently be broken by expression reformulation. Unless the control dependence height is reduced, however, the potential instruction-count cost of applying expression reformulation combined with the apparent lack of any benefit will prevent the compiler from applying the transform. Thus, for if-conversion during scheduling to yield code of quality comparable to that produced in an early formation scheme, traditional optimizations must be performed along with if-conversion and predicate-specific optimizations, in the scheduler.

For this framework to generate high-quality predicated code, the scheduler must simultaneously perform global scheduling, partial path if-conversion, predicate optimizations, tail duplication, ILP optimizations, and traditional optimizations. While studies

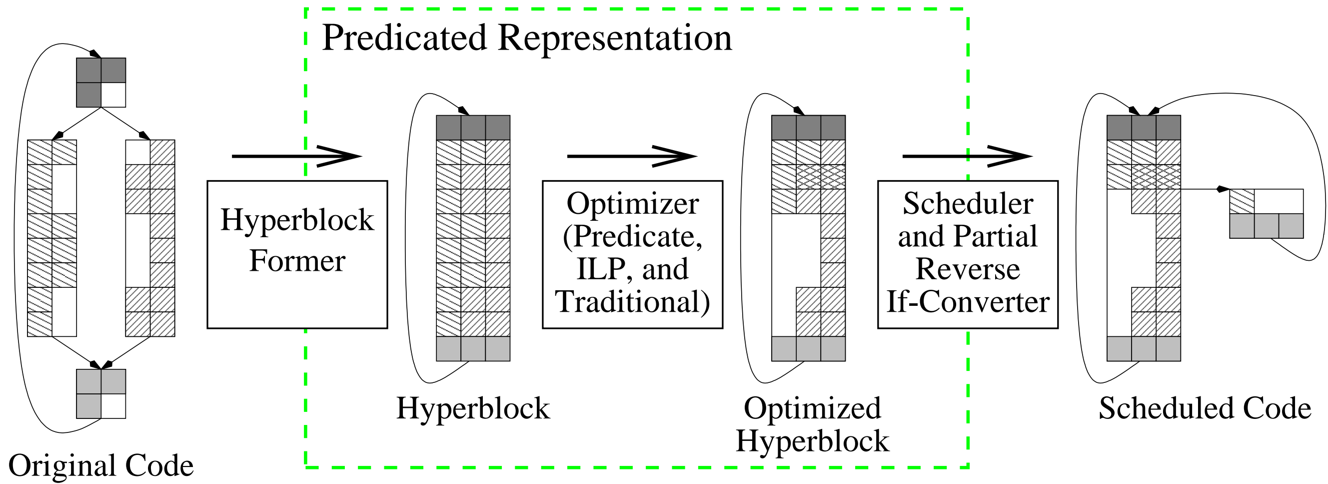


Figure 3.9: The Partial Reverse If-Conversion Predication Framework.

have exhibited the possibility and profitability of performing optimizations at schedule time [37], in the If-Conversion during Scheduling Framework much more complexity is added to an ILP scheduler that must already consider many issues, including control speculation, data speculation, and register pressure. That this additional complexity is tolerable in a commercial instruction scheduler is highly doubtful, but for such a framework to produce quality code it is absolutely necessary.

3.6.2 The proposed compilation framework

Given that performing if-conversion at schedule time limits the use of the predicated representation for optimization and given that if-conversion at an early stage is limited in its ability to estimate final code characteristics, it is logical to look to an alternative compilation framework. This chapter presents such a framework. This framework overcomes limitations of other schemes by utilizing two phases of predicated code manipulation to support predicated execution. Aggressive if-conversion is applied in an early compilation

phase to create the predicate representation and to allow flexible application of predicate optimizations throughout the back-end compilation procedure. Then, at schedule time, the compiler adjusts the final amount of predication to match the target architecture. The Partial Reverse If-Conversion Predication Framework, shown in Figure 3.9, consists of two phases of predicate manipulation surrounding classical, predicate-specific, and ILP optimizations. The first predicate manipulation phase, hyperblock formation, has been addressed thoroughly in [17]. The second predicate manipulation phase, the adjustment of hyperblocks during scheduling termed *partial reverse if-conversion*, was proposed in [38], and is discussed in further detail here.

The first phase of the compilation framework consists of performing aggressive hyperblock formation. The hyperblock former does not need to compute exactly what paths, or parts of paths, will fit in the available resources and be completely compatible with each other. Instead, it forms hyperblocks that are larger than the target architecture can handle. The large hyperblocks increase the scope for optimization and scheduling, further enhancing the benefit derived from the predicated representation. In many cases, the hyperblock former will include almost all the paths. This is generally an aggressive decision because the resource height or dependence height of the resulting hyperblock is likely to be much greater than the corresponding heights of any of its component paths. However, the if-converter relies on later compilation phases to ensure that this hyperblock is efficient. One criterion that is still enforced in the first phase of hyperblock formation is that paths containing hazards are excluded. As was discussed in Chapter 3, hazards reduce the compiler's effectiveness for the entire hyperblock; thus, they should be avoided to facilitate more aggressive optimization.

The second phase of the compilation framework consists of adjusting, via partial reverse if-conversion, the amount of predicated code in each hyperblock as the code is scheduled. Partial reverse if-conversion is conceptually the application of reverse if-conversion to a particular predicate in a hyperblock for a selected set of instructions. Reverse if-conversion was originally proposed as the inverse process to if-conversion. In that process, branching code that contains no predicates is generated from a block of predicated code. This allows code to be compiled using a predicated representation, but to be executed on a processor without support for predicated execution [39].

The scheduler with partial reverse if-conversion operates by identifying the paths composing a hyperblock. Paths which overlap profitably remain unchanged. Conversely, a path that interacts poorly with the other paths is removed from the hyperblock. In particular, the partial reverse if-converter decides to eject certain paths, or parts of paths, to enhance the schedule. To do this, the reverse if-converter inserts a branch that is taken whenever the removed paths would have been executed. This has the effect of dividing the lower portion of the hyperblock into two parts, corresponding to the taken and fall-through paths of the inserted branch. The decision to reverse if-convert a particular path consists of three steps. First, the partial reverse if-converter determines the savings in execution time to be obtained by inserting control flow and applying the full resources of the machine to each of two hyperblocks instead of dividing them between the two paths in the single hyperblock. Second, it computes the loss created by any penalty associated with the insertion of the branch. Finally, if the gain of the reverse if-conversion exceeds the cost, it is applied. Partial reverse if-conversion may be applied repeatedly to the same hyperblock until the resulting code is desirable.

A very important point is that regardless of the best possible balance of predication and control flow, the code has benefited from originally being predicated. Even if the partial reverse if-converter decides to completely restore control flow, the resulting code benefits from the freedom provided to optimizations and scheduling [39]. For example, portions of all paths exist and execute in parallel in various parts of the hyperblock due to the freedom provided to the scheduler.

The strategy used for this compilation framework can be viewed as analogous to the use of virtual registers in many compilers. With virtual registers, program variables are promoted from memory to reside in an infinite space of virtual registers early in the compilation procedure. The virtual register domain provides a more effective internal representation than do memory operations for compiler transformations. As a result, the compiler can perform more effective optimization and scheduling on the virtual register code. Then, at schedule time, virtual registers are assigned to a limited set of physical registers and memory operations are reintroduced as spill code when the number of physical registers was over-subscribed. The framework presented in this chapter does for branches what virtual registers do for program variables. Branches are removed to provide a more effective internal representation for compiler transformations. At schedule time, branches are inserted according to the capabilities of the target processor. The branches reinserted have different conditions, targets, and predictabilities than the branches originally removed. The result is that the branches in the code fit the performance constraints for a particular processor, rather than the code structure decisions made by the programmer.

The key to making this predication and control flow balancing framework effective is the partial reverse if-converter. The mechanics of performing partial reverse if-conversion, as well as a proposed policy used to guide partial reverse if-conversion, are presented in Chapter 7.

4. PREDICATE ANALYSIS SYSTEM

In the Partial Reverse If-Conversion Framework, code is rendered in an early phase by the compiler into the predicated representation through if-conversion. Effective optimization, scheduling, and register allocation of code in the predicated representation, however, require the compiler to understand the logical relationships among predicates because execution conditions are no longer solely dependent on branches. Previous work has demonstrated the importance of accurate predicate relation analysis to some phases of compilation, such as register allocation [19]. This chapter presents a flexible, powerful, modular, and efficient analysis framework that surpasses all previously published systems in accuracy of predicate relationship representation. This framework, called the Predicate Analysis System (PAS), includes a mechanism for incorporating range analysis of conditions into the logical database, where previous techniques do not. This chapter presents results indicating the importance of accurate predicate analysis in performing optimizations, and the efficiency of the selected binary decision diagram (BDD) representation in answering required queries.

4.1 Predicate Analysis

In traditional nonpredicated codes, a program control flow graph specifies the *execution condition* of each instruction. Whether or not a control flow graph is used explicitly, compilers must often consider an instruction's position in the control flow graph relative to others to determine if a particular transformation is legal, or to compute useful dataflow analyses [40]. Because in predicated codes an instruction must be fetched and its guard predicate must hold a value of 1 in order to execute, the control flow graph specifies the *fetch condition* rather than the execution condition, which is the logical conjunction of the fetch condition and the guard predicate condition. Because program state is modified with respect to execution conditions, compiler transformations must be concerned with both the guard predicate and fetch conditions. As such, many notions of code properties must be revisited. For example, let *fetch dominance* (*fdom*) assume the traditional definition of dominance:

I_1 *fdom* I_2 iff every fetch path from the unique entry node $START$ to I_2 includes I_1.

Let *execution dominance* (*edom*) be defined as:

$$I_1 \ edom \ I_2 \text{ iff } I_1 \ fdom \ I_2 \text{ and } P_{I_1} \supseteq P_{I_2}.$$

One of the most significant changes to a compiler in rendering it predicate-aware is the replacement of traditional dominance queries with execution dominance queries.

A simple example demonstrates the application of *edom*. Figure 4.1 shows two code segments which are considered for application of a constant propagation optimization. In Figure 4.1(a), constant propagation cannot be applied because instruction 2, the

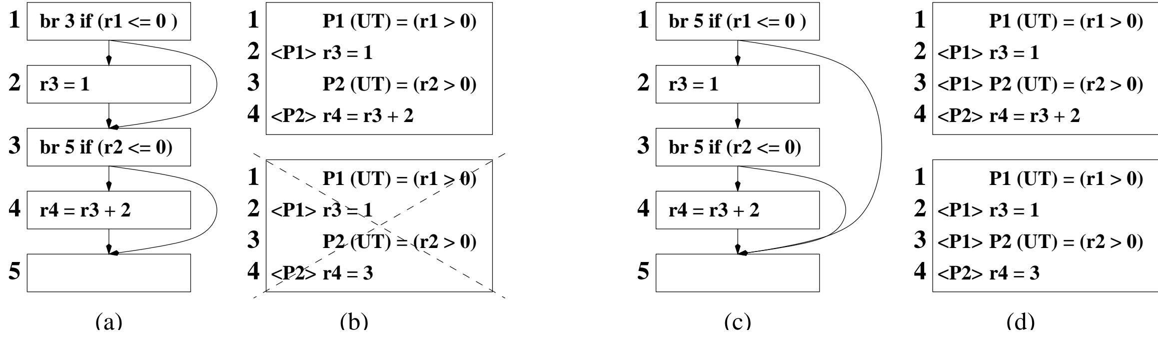


Figure 4.1: Constant propagation optimization in a predicated region. Branching code (a) and hyperblock (b) in which optimization is invalid, branching code (c) and hyperblock (d) in which it is valid.

assignment to $r3$, does not dominate instruction 4, the assignment to $r4$. In Figure 4.1(c), however, the first definition dominates the second, so the optimization is valid. After if-conversion of (a) into (b) and (c) into (d), which does not alter program semantics, instruction 2 fetch-dominates instruction 4 in both cases. A compiler that ignored the effect of predication on execution dominance would erroneously apply the optimization to the code of Figure 4.1(b). On the other hand, a compiler incapable of recognizing execution dominance relations between predicated instructions when they do exist, in this case between instructions 2 and 4 in Figure 4.1(d), would fail to apply valid and profitable optimizations in predicated regions.

To ensure that legitimate optimizations are applied, but that illegal ones are prevented, a predicate relationship query system is required. In particular, a subset query function must be implemented to compute execution dominance for the example in Figure 4.1. One very straightforward (but highly conservative) subset query function could be implemented as

$$Subset\,(P_i, P_j) := (P_j = 1) \vee (P_i = P_j).$$

Unfortunately, the conservative nature of this implementation could adversely affect final code quality. Figures 4.1(c) and 4.1(d) show a code segment in which this is the case. In Figure 4.1(c), constant propagation between instructions 2 and 4 should be performed because the required dominance relation holds. Logically, the same transformation should be applicable after if-conversion, as in Figure 4.1(d). Likewise, I_2 *edom* I_4 since I_2 *fdom* I_4 and $P_1 \supset P_2$. The conservative subset query cannot identify relations between two variable predicates; thus, the execution dominance test fails, and the optimization is not performed. Results presented later will support the suggestion made in this example that conservative predicate analysis results in significant loss of performance.

While the subset relation is important for the computation of the execution dominance relationship, other relationships such as equivalence, nonintersection, inverse, etc. are desirable for other purposes. PAS provides these queries (as indicated in Table 4.2 on page 69) to the rest of the compiler.

The most intuitive approach to supporting these queries is to perform traditional control flow analysis on the code prior to or during if-conversion, and then to retain this information throughout the predicated compiler back end. This method was rejected for several reasons. First, frameworks such as the PDLO, presented in Chapter 6, fundamentally alter the control logic of programs, making it desirable that the analysis should be independent of the means originally used to express control prior to if-conversion. Second, direct analysis of predicated code permits re-optimization or rescheduling of existing binaries, which may be an effective technique for scaling software to new generations of EPIC architectures. Finally, optimizations on the predicated representation, such as instruction merging and predicated branch combining, may create instructions

whose condition of execution does not correspond to that of any instruction in the original control flow graph. The decision to perform direct analysis is further affirmed by others' previous work [41] in the area.

4.2 Previous Mechanisms

Two previous general approaches to predicate analysis have been described in the literature, both of which apply to hyperblock code with restricted predicate define types. The first and most intuitive, the Predicate Hierarchy Graph (PHG), was introduced with the IMPACT hyperblock compilation framework [17], [35]. The PHG relates predicates simply by keeping track of which predicates guarded the definition of each predicate, or of each term component for OR-type expressions. The PHG is inaccurate on code in which predicate subexpressions are reused in ways that generate predicates that do not fit neatly into a hierarchical graph. The PHG is also unable to represent networks that contain and-type, conjunctive-type, and disjunctive-type predicate defines. This is a serious problem for many types of control height reduction optimizations such as those presented in [23] and [42] as it precludes accurate direct analysis.

The second approach, the Predicate Query System (PQS) [41], exists within the Hewlett-Packard Elcor framework. The representational mechanism of PQS, the *partition graph*, can accurately describe only those predicate expressions that can be expressed as logical partitions. (P_2 and P_3 partition P_1 iff $P_1 = P_2 \cup P_3$ and $P_2 \cap P_3 = \emptyset$.) This relation can be formed only for unconditional predicate defines and for or-type predicate defines with disjoint terms. Thus, PQS accurately represents directly if-converted forms, but does not support other predicate types commonly used in modulo scheduling and

predicate optimization [23]. Conservative approximations of relations among these other define types have been proposed; unfortunately, the mechanisms for coping with partition graph inaccuracy introduce the possibility of building for a given set of predicate defining instructions several different partition graphs with varying degrees of accuracy [19]. The primary advantages of PAS over PQS, therefore, are its abilities to perform perfectly accurate direct analysis of code utilizing any desired predicate defining semantics, and the ability to incorporate knowledge of condition relations into its logical database, as will be described shortly.

4.3 The Predicate Analysis System

As described in the previous section, previous approaches have constructed a representation based on characteristics of the subset of predicate define types deemed useful at the time. Today, the full complement of predicate define types is used in various optimizations. Thus, PAS is built upon a general Boolean relationship database commonly used in CAD, a *binary decision diagram (BDD)*, which can accurately represent arbitrary Boolean relations composed using arbitrary predicate defining instructions. First, the derivation of a BDD representation for a single-hyperblock sequence of predicate-defining instructions will be considered; Section 4.3.2 details techniques for extending the presented analysis beyond a single hyperblock.

4.3.1 Intrahyperblock analysis and the BDD

A hyperblock is a single-entry, possibly multiple-exit acyclic region, effectively a straight-line block of predicated code, potentially with side exits. Consider that prior to

analysis, predicates are given single static assignment (SSA) equivalents, as was assumed in PQS. For the moment excluding predicate values live into the hyperblock, a single, sequential pass over the predicate defines in the hyperblock is sufficient to determine the logical relationships among any predicates in the hyperblock, based on an accumulation of the predicate deposit semantics of Table 2.1. While any system that can represent Boolean functions can be used to represent the network of predicate definitions, efficiency of the representation in space and time for useful queries must be considered in selection of a representational form.

In general, a Boolean function $f(x_0, x_1, \ldots, x_n)$ can be represented in a number of forms. A Boolean expression comprised of a conjunction of disjunctions of variables or complements of variables is in conjunctive normal form (CNF). CNF expressions have the form

$$\bigwedge_{j=1}^{m} \left(\bigvee_{i=1}^{n} t_{i,j} \right)$$

where $t_{i,j}$ is either a variable x_i, a variable $\overline{x_i}$, or the constant 1. A Boolean expression comprised of a disjunction of conjunctions of variables or complements of variables is in disjunctive normal form (DNF). DNF expressions have the form:

$$\bigvee_{j=1}^{m} \left(\bigwedge_{i=1}^{n} t_{i,j} \right)$$

where $t_{i,j}$ is either a variable x_i, a variable $\overline{x_i}$, or the constant 0. Although some analysis on DNFs and CNFs is relatively inexpensive, performing a tautology check on DNFs and

testing for satisfiability on CNFs are both NP-hard. A different form which is efficient both in tautology and satisfiability is the if-then-else normal form (INF).

INF uses only the if-then-else (ITE) operator to represent Boolean functions. ITE is defined as

$$ITE\,(x, y, z) = (x \wedge y) \vee (\overline{x} \wedge z)$$

Unlike CNF and DNF which are flat, INF defines Boolean functions recursively using the Shannon expansion:

$$f\,(x_0, x_1, \ldots, x_n) = (x_n \wedge f\,(x_0, x_1, \ldots, x_{n-1}, 1)) \vee (\overline{x_n} \wedge f\,(x_0, x_1, \ldots, x_{n-1}, 0))$$

$$f\,(x_0, x_1, \ldots, x_n) = ITE\,(x_n, f\,(x_0, x_1, \ldots, x_{n-1}, 1)\,, f\,(x_0, x_1, \ldots, x_{n-1}, 0))$$

In INF, the ITE operator is used exclusively so that the expression consists of non-terminal interior ITE nodes and terminal leaf nodes 0 and 1. When no subexpressions are shared in an INF constructed in this way, the INF is called a decision tree. If all equal subexpressions are shared in a decision tree, the resulting INF expression is termed a BDD. Two consequences of the Shannon expansion, which hold for the BDD as well, are that the variables appear in the same order in any path from root to leaf, and that a given variable appears at most once in any such path. A BDD with these properties is an ordered BDD (OBDD). If the OBDD has all identical ITE nodes shared and contains no redundant tests, then the OBDD is a reduced OBDD (ROBDD). The order of the variables in the ROBDD has a large effect on the size of the resulting ROBDD [43].

ROBDDs represent Boolean functions in an efficient and canonical manner. (When referring to BDDs, one usually means ROBDDs.) The canonical nature of ROBDDs makes them very efficient for certain queries. For example, it is possible to test if two functions are identical in constant time. This is useful, especially for testing if a function evaluates to the constant 0 or the constant 1. The cost of this canonicity is that the BDD can become exponentially complex in the number of variables involved in an expression. This is a problem frequently encountered in the use of BDDs in logic synthesis applications, in which circuits such as multipliers require an exponential representation [44]. In this application, however, BDDs do not exceed a reasonable level of complexity, even in aggressively transformed predicated codes.

Much work has been done in the development of efficient ROBDD implementations, mostly intended for use in the domain of Boolean logic circuit optimization [45]. BDDs have also been applied in software problems, usually in the verification domain. One extension created for efficiency is the use of "invert" arcs, which can be used instead of "else" arcs to implement the function:

$$f\left(x_0, x_1, \ldots, x_n\right) = \left(x_n \wedge f\left(x_0, x_1, \ldots, x_{n-1}, 1\right)\right) \vee \left(\overline{x_n} \wedge \overline{f\left(x_0, x_1, \ldots, x_{n-1}, 0\right)}\right) \quad (4.1)$$

This extension allows for constant-time inversion and avoids the addition of extra internal nodes when a complement of an existing subgraph is required [46]. PAS uses the Colorado University Decision Diagram (CUDD) implementation of BDDs [47] with this complement extension.

Within this framework, a "forest" of BDDs is created to represent the interaction of the various predicate functions. The BDD is initialized with a single terminal node having the value 1. PAS then sequentially examines each predicate define in the hyperblock, adding any newly encountered conditions as BDD variables and recording the expression of the destination predicate in the BDD as a function of the predicate source, the previous value of the predicate destination, and the condition. The BDD is extended using the function $ITE(x, y, z)$ to implement the logical relationships between predicates and conditions, where x, y, and z already exist in the BDD. During the addition of the ITE expression, internal functions process the BDD to ensure that it remains in canonical form. Dynamic programming techniques are used to keep the running time of creating an $ITE(x, y, z)$ expression at $O(|x||y||z|)$ where $|f|$ is the number of edges reachable by function node f. While this can create a BDD with exponential size in the worst case, the BDDs in the PAS are extremely well-behaved.

Table 4.1 shows the ITE expressions used to represent the various types of assignment. The expressions are shown in the familiar format used in the C language, where $C \, ? \, A \, : \, B$ indicates that if C is true, the expression takes the value A; otherwise, the expression takes the value B. In the figure, the BDD node associated with the predicate $P_{i,j}$ is represented as $n_{i,j}$. Topologically traversing the acyclic control subgraph ensures that the predicate source and the previous value of the predicate destination are both available as nodes in the BDD at the time a definition sourcing them is encountered.

Nodes representing the conditions evaluated in predicate-defining instructions can be created in several ways. The most straightforward is to create a new variable in the BDD for each unique condition evaluated, ignoring any relationship among conditions

Table 4.1: Predicate deposit logic.

SSA pred. def.	*ITE* Expression
$P_{i,j} \overset{UT}{\longleftarrow} (C)\ \langle P_g \rangle$	$n_{i,j} = C?n_g : 0$
$P_{i,j} \overset{UF}{\longleftarrow} (C)\ \langle P_g \rangle$	$n_{i,j} = C?0 : n_g$
$P_{i,j} \overset{OT}{\longleftarrow} (C)\ \langle P_g \rangle$	$n_{i,j} = C?(n_g?1 : n_{i,j-1}) : n_{i,j-1}$
$P_{i,j} \overset{OF}{\longleftarrow} (C)\ \langle P_g \rangle$	$n_{i,j} = C?n_{i,j-1} : (n_g?1 : n_{i,j-1})$
$P_{i,j} \overset{CT}{\longleftarrow} (C)\ \langle P_g \rangle$	$n_{i,j} = C?n_g : n_{i,j-1}$
$P_{i,j} \overset{CF}{\longleftarrow} (C)\ \langle P_g \rangle$	$n_{i,j} = C?n_{i,j-1} : n_g$
$P_{i,j} \overset{AT}{\longleftarrow} (C)\ \langle P_g \rangle$	$n_{i,j} = n_g?(C?n_{i,j-1} : 0) : n_{i,j-1}$
$P_{i,j} \overset{AF}{\longleftarrow} (C)\ \langle P_g \rangle$	$n_{i,j} = n_g?(C?0 : n_{i,j-1}) : n_{i,j-1}$
$P_{i,j} \overset{\wedge T}{\longleftarrow} (C)\ \langle P_g \rangle$	$n_{i,j} = n_g?(C?n_{i,j-1} : 0) : 0$
$P_{i,j} \overset{\wedge F}{\longleftarrow} (C)\ \langle P_g \rangle$	$n_{i,j} = n_g?(C?0 : n_{i,j-1}) : 0$
$P_{i,j} \overset{\vee T}{\longleftarrow} (C)\ \langle P_g \rangle$	$n_{i,j} = C?1 : (n_g?1 : n_{i,j-1})$
$P_{i,j} \overset{\vee F}{\longleftarrow} (C)\ \langle P_g \rangle$	$n_{i,j} = C?(n_g?1 : n_{i,j-1}) : 1$

that exist in the code. In a graph constructed on this basis, logical relationships between predicates are fully represented, but the conditions on which they are based are treated as independents. A slightly more sophisticated approach is to recognize opposites among the conditions, for example, `(r1 < r2)` and `(r1 >= r2)`, and to treat these two conditions as the normal and complemented senses of a single BDD variable. This provides the BDD with a limited amount of knowledge of logical interdependence among the conditions, so the predicate $P_1 \overset{UT}{\longleftarrow} (\texttt{r1} < \texttt{1})\ \langle \mathbf{T} \rangle$ and the predicate $P_2 \overset{UT}{\longleftarrow} (\texttt{r1} >= \texttt{1})\ \langle \mathbf{T} \rangle$ would be recognized as opposites. Previous work treats conditions at this level.

In PAS it is possible to represent more complex relations among families of conditions that are based on comparing the same register values [48]. Such a system is able to represent, for example, the exclusivity of `(r1 == 1)` and `(r1 == 2)` while indicating that both are subsets of `(r1 > 0)`. This system has been implemented in BDD form

using a technique known as finite domain representation [44]. When using this technique, conditions which can be identified as related are entered into the BDD as related sub-BDDs rather than as independent variables, transparently providing an additional layer of analysis within the PAS framework.

4.3.2 Interhyperblock relationships

In the IMPACT and Elcor compilers, predicated code is generated within the context of hyperblock regions, which are single-entry, multiple-exit acyclic code regions formed by if-conversion and tail-duplication [17]. The hyperblock compilation framework, which includes hyperblock formation routines and support for predication, was developed to simplify global optimization and scheduling. By operating on single-entry regions, code motion and optimization across branches is simplified. For example, because no predecessor merges can exist in control flow within a hyperblock, the need for compensation code when speculating instructions is eliminated. Analysis within hyperblock regions is also simplified, as indicated in the previous section.

Significant benefits, however, may be gained by extending predicate analysis beyond the hyperblock scope. In EPIC environments, control is likely to be implemented as a mixture of branches and predication due to performance constraints, lending appeal to an analysis that could analyze control through a multihyperblock region. This support is essential because the Partial Reverse If-Conversion Framework presented in this work creates predicates with multiple hyperblock scope. Furthermore, rotation of instructions around loop backedges to achieve software pipelining is aided by the ability to analyze

predicates whose live ranges propagate through backedges, though this may pose a more formidable analysis problem.

For the moment disallowing predicate liveness around backedges, we consider the relationships of predicates as defined by predicate defines in the acyclic control flow subgraph. The acyclic control flow subgraph is derived from the program's reducible control flow graph by removing all natural loop back edges. Given the directed control flow graph $G(V, E)$, consider the set of back-edges $B = \{e_i = (v_{tail}, v_{head}) \, | e_i \in E, v_{head} \; fdom \; v_{tail}\}$. The acyclic control flow subgraph in which each node is reachable from the root is then given by $G' = (V, E - B)$. In G', the acyclic control flow graph, all uses of an SSA predicate will appear after all reaching definitions in the topological sort of the instructions. Thus, the desired predicate relation information can be extracted in a single topologically ordered traversal of G'. In this process, the ϕ functions introduced at predicate value merges as part of the SSA representation are processed as predicate defines. $P_{i,l} = \phi\left(P_{i,j}, P_{i,k}\right)$ becomes $n_{i,l} = C_{i,l}?n_{i,j} : n_{i,k}$ where $C_{i,l}$ is a variable representing which path was traversed into the merge.

Previous work related to PQS [19] introduced a conceptual full if-conversion technique that could treat control flow and predication uniformly in an acyclic region, a technique that could relatively easily be incorporated into PAS as well. In PQS, however, no attempt was made to achieve accurate analysis of predicates live around back edges.

Code having predicates live around back edges can be subdivided into two classes. The first class contains situations in which a predicate is live around a back edge, but in which the predicate is initialized before loop entry and unconditionally killed in each iteration. This case would result, for example, if in the process of loop rotation, a predicate-defining

instruction were rotated from the top of the block to the bottom, around the back edge (with duplication in the preheader). This case, because it creates a merge but not a dependence cycle, is handled simply by treating the resulting ϕ as before.

The other, more interesting case occurs when a dependence cycle exists in the predicate data flow graph. This would occur when, for example, a loop contains an and-type predicate assignment with an initialization to 1 in the loop preheader. In this case, representing this construct requires summarizing a Boolean accumulation across all possible loop iterations into a simple structure in the BDD. The ability to handle predicate dependence cycles would enable new means of using predicate-defining instructions to control loops. However, its development is not crucial to the success of the work presented here.

4.3.3 Predicate relationship query interface

The predicate BDD having been built, a query interface is made available to compiler transformations. For this purpose, predicate register operands in the compiler's internal representation are annotated with their SSA equivalents in the predicate analysis system. The query functions internally query the BDD with respect to the nodes associated with these SSA predicates and return results accordingly.

Just as the BDD is constructed to represent logical relationships using $ITE()$, the same function is used to query the BDD as to the relation of two or more functions described by it. For example, consider testing if f_a implies f_b. This implication holds as long as whenever f_a is true, f_b is never false. Thus, the query node $q = ITE(f_a, NOT(f_b), 0)$ is formed and tested for identity to 0. If $q = 0$, the implication holds. Similar queries can be constructed for, among others, the useful relations mentioned previously. A garbage

Table 4.2: Predicate query operations and their complexity. f_i is the Boolean function for P_i.

Query	Complexity
$Equivalent\,(P_i, P_j)$	$O(1)$
$Inverse\,(P_i, P_j)$	$O(1)$
$Subset\,(P_i, P_j)$	$O(\|f_i\|\|f_j\|)$
$Intersect\,(P_i, P_j)$	$O(\|f_i\|\|f_j\|)$
$Exhaust\,(P_i, P_{j0}, P_{j1}, \ldots, P_{jn})$	$O(\|f_i\|\|f_{j0}\|\|f_{j1}\|\ldots\|f_{jn}\|)$

collection mechanism built into CUDD prevents build-up of retired query nodes in the forest.

Table 4.2 shows the basic predicate relation queries supported by this framework, along with an estimate of the average complexity of the underlying BDD algorithms. Due to the canonicity and complement-arc features of the CUDD ROBDD, checks of identity and inverse are performed in constant time. Other checks are made with BDD query functions particular to CUDD which perform very efficient recursive descents into the BDD, extensively applying the ROBDD canonicity properties and memoization techniques to reduce query time [47].

4.4 Evaluation

Unlike PQS or PHG, PAS perfectly represents all the Boolean relations it is able to discern from predicate define networks.

It is necessary to consider the speed of PAS to determine if the benefits of its accurate representation justify the increase in compile time due to implementing it. Regardless of representation, describing an arbitrary Boolean function is provably exponential in the worst case, and BDDs provide no exception. To determine the suitability of BDDs to

the task of representing predicate relations, an experimental evaluation was performed. To assess the efficiency of PAS, its run time was measured across a set of real programs.

The benchmarks *008.espresso*, *022.li*, *026.compress*, and *072.sc*, of SPEC CINT92; *124.m88ksim*, *129.compress*, and *130.li* of SPEC CINT95; and *cccp*, *lex*, and *wc* (which are Unix utilities) were chosen for this evaluation to match those presented in [23]. These programs were compiled with the IMPACT compiler version 990714-R using function inlining, classical optimization, aggressive hyperblock formation, ILP optimization, scheduling, and register allocation.

The PAS analysis function was run once on each of the compiled programs on an HP 9000/785/400 workstation operating at a clock frequency of 400 MHz with 1 GB RAM. The total time required to build the BDDs representing all relationships in the final code of all the above benchmarks was 2.1 s. Typically, the BDD needs to be rebuilt only when predicate definition optimizations are performed, keeping the compile time for BDD construction acceptable for a production compiler. To measure query efficiency, all pairs of predicates within each hyperblock were tested for subset, superset, and disjoint relationships. A total of 1,177,491 queries were made in 3.3 s. This rapid query response is due in part to the canonicity of the BDD and in part to memoization techniques applied in CUDD [47]. This result is expected, as the control flow of structured programs results in predicate relationship equations that are relatively small and well-behaved in comparison to the large circuits the BDD was specifically designed to handle.

Finally, it is important to consider the applicability of PAS. Of the 1 177 491 queries performed in the above experiment, 391 596 (33%) could be resolved trivially to an aggressive result (comparisons to true or of two identical predicates). The intermediate, or

hammock, analysis, resolves only an additional 2338 queries (0.2%) aggressively, although it has some palpable effect on performance for a number of the benchmarks. PAS resolves 67 303 further queries (6%) to aggressive (optimization-enabling) results. A compiler that lacked a predicate analysis system would be unable to identify these relations, and thus would likely overlook legal and profitable optimizations.

To assess the real importance of this, two low-cost predicate analysis solutions were developed. The IMPACT compiler was then run three times on the benchmarks, once with the PAS active and once each with "trivial" and "intermediate" predicate analysis. Trivial predicate analysis returns nonconservative answers to queries regarding pairs that include the TRUE predicate or pairs of identical predicates. Intermediate predicate analysis builds relations between a predicate and its immediate parent only, providing a low-cost answer that is accurate for simple hammocks. The performance effect on the generated code is shown in Figure 4.2, which shows the speedup obtained by applying post-if-conversion (traditional and ILP) optimizations with and without PAS. The reader is referred to previous results that have shown the effectiveness of a predication compilation framework in conjunction with predicated execution support in enhancing performance on ILP processors [38], [49].

Accurate analysis improves the final code quality for all the benchmarks except *wc*. The benchmark *wc*, although it contains a very complex predicate network, has few predicated instructions, reducing the potential effect of accurate predicate analysis. Other benchmarks, such as *008.espresso*, *022.li*, *099.go*, and *lex*, exhibit significantly higher performance with accurate analysis. In a cycle-weighted average, code processed using

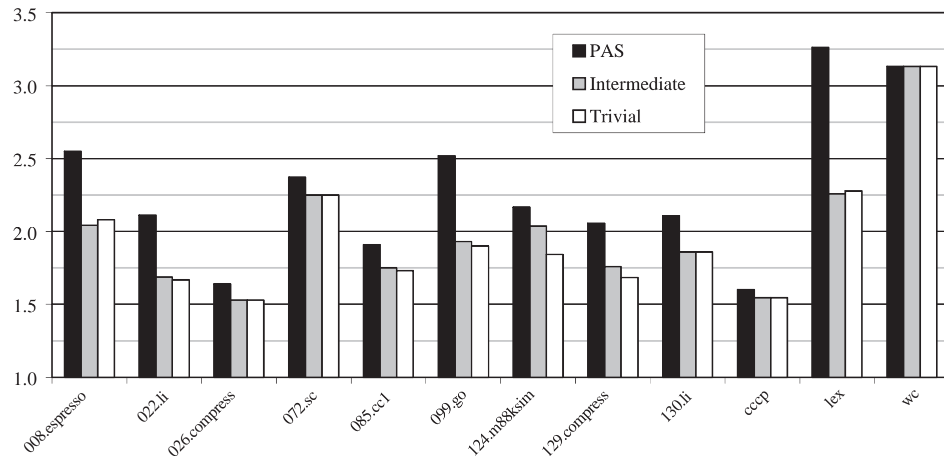


Figure 4.2: Speedup due to post-if-conversion optimization, Predicate Analysis System and approximate methods.

accurate predicate analysis executed 20% faster than code using either simple analysis scheme. These performance differences are due to better application of traditional, predicate, and ILP optimization after if-conversion. In particular, constant manipulation optimizations, copy propagation, dead code removal, code motion, accumulator expansion, strength reduction, and loop invariant code removal were applied at least 10% more frequently with accurate predicate analysis. Note that these optimizations were also applied prior to if-conversion and that these counts are with respect to opportunities exposed by the hyperblock formation process. When PAS was employed, predicate merging, an optimization that replaces two or more instructions with a single instruction predicated on the union of their original execution conditions, was applied three times more often than when it was not. In addition to additional optimization applications, better register allocation and scheduling enabled by the accurate analysis further improved performance [19].

The utility of PAS is not limited to improving existing compiler operations. PAS also plays a key role in PDLO as presented in [23]. PDLO extracts Boolean expressions that describe the predicate define network, performs Boolean minimization on these expressions, and reformulates the result into a new predicate define network. PDLO must use a perfectly accurate representation in order to generate correct code. Approximations used in other systems would result in invalid code. PDLO performance enhancement was excluded from Figure 4.2 in fairness to the compiler without PAS, which could not employ it. As will be documented in Chapter 6, the analysis system described here enables PDLO to achieve significant gains for some benchmarks which do not show significant opportunity for other types of predicate-enabled optimization (notably *wc*).

4.5 Significance

PAS provides efficient and accurate information to support effective compiler optimization and scheduling. The power of PAS comes from the ROBDD's suitability for representing relations of the type present among predicates. While essential for most optimizations, predicate relationships are not the only type of information necessary to perform global optimization. Dataflow information, such as live-ranges, reaching-definitions, and the like, must also be made available. Dataflow analysis is complicated by the the presence of predicated code, but through the use of PAS, a dataflow analysis engine is devised to provide accurate information in a timely manner. This dataflow engine is described in the next chapter.

5. THE PREDICATE DATAFLOW GRAPH

5.1 Predicate-Aware Dataflow Analysis

Program analysis is a prerequisite for important program transformations performed by compilers. Dataflow analysis is the determination of the flow of data through registers in the presence of program control. Dataflow analysis techniques have typically operated on graphs which are used to represent branch control flow. Popular classical program representations used for analysis include the *control flow graph* (CFG) and the *program dependence graph* (PDG) [50]. As originally defined, these representations can only represent conditional execution based on branches. To obtain accurate dataflow information, all sources of conditional execution much be considered. When conditional execution originates from both predication and branches, these representations are not sufficient for accurate analysis.

One means of conducting dataflow analysis in the presence of predication is to extend current iterative dataflow analysis techniques. Accurate results can be obtained when these techniques, modified slightly, are applied on special flow graphs which represent

both predicate and branch conditional execution. These augmented flow graphs are termed *predicate flow graphs* (PFGs).

The advantage of predicate flow graphs is that predicate aware dataflow can be obtained with only minor modification of existing dataflow analysis routines. This section describes and studies the PFG-based approach to predicate dataflow analysis, beginning with the definition of the PFG.

Figure 5.1 shows a basic block and its PFG, which can be thought of as a reverse if-conversion of the predicated region. As an exercise, consider performing dataflow analysis on the basic block in a conservative manner, ignoring the interrelation of the predicates involved. Here, instruction 4 consumes $r3$, and instruction 3 (on a different predicate) produces $r3$. Without knowledge of the interrelation of the predicates guarding instructions 3 and 4, it is impossible to determine if the definition at instruction 3 dominates the consumption at instruction 4; thus, liveness for $r3$ must extend out the entry point of the block. Consider on the other hand analysis performed on the predicate flow graph for the same code segment, shown on the right in the figure. Here, it is apparent that the definition of $r3$ dominates the consumption in 4; thus, it is determined that the live range of $r3$ includes only instructions 3 and 4. This simple example shows the benefit of performing dataflow analysis using the predicate flow graph. Subsequent sections of this chapter discuss the PFG's characteristics and construction in detail.

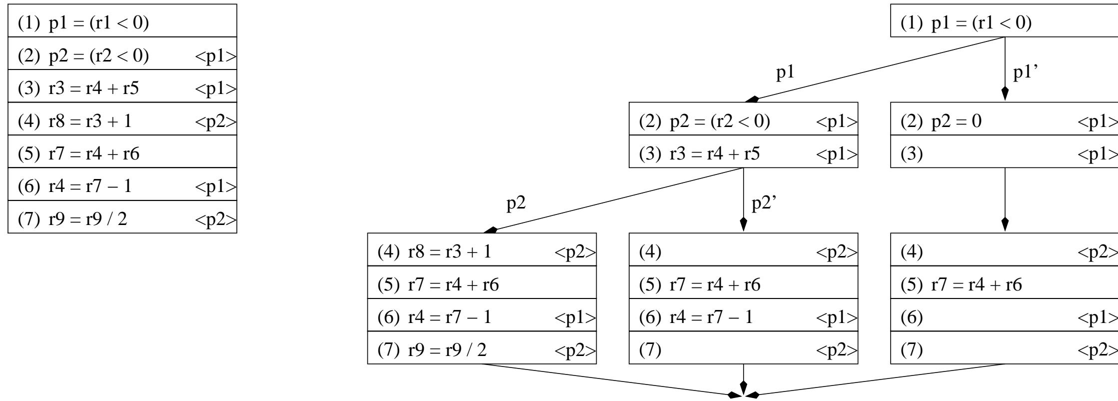


Figure 5.1: A predicate flow graph example.

5.2 Predicate Flow Graph

In this section, we define the PFG by extending the definition of the CFG to the predicated code domain. The node in the CFG, the basic block, is a single-entry single-exit region uniquely identified with a fetch and equivalent execute condition. In the predicate flow graph, a new node type must be defined which accurately represents the now different fetch and execute conditions.

Basic block. A *basic block* is a sequence of consecutive statements in which the flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. Any given instruction exists in exactly one basic block.

The essence of a basic block is that it groups together a set of consecutive, control flow equivalent statements. The execution of the first statement in a basic block guarantees the complete sequential execution of all statements in that basic block before another basic block is executed.

Control flow graph. A *control flow graph* is a directed graph G augmented with a unique *entry* node START and a unique *exit* node STOP such that each node in the graph

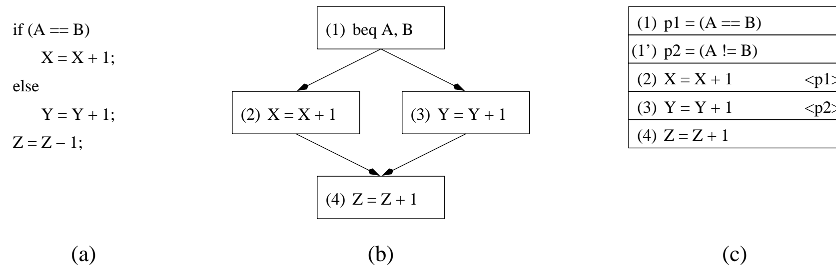


Figure 5.2: A simple if-then-else construct in the original C code (a), the control flow graph for nonpredicated code (b), and the control flow graph for the predicated code (c).

has at most two successors. We assume that nodes with two successors have attributes "T" (true) and "F" (false) associated with the outgoing edges originating at a real or implied branch. We further assume that for any node N in G there exists a path from START to N and a path from N to STOP. All nodes in this graph represent basic blocks.

Figure 5.2 demonstrates the limitation of the traditional control flow graph when applied to predicated code. A simple if-then-else construct is shown in Figure 5.2(a). The CFG for this segment without predication is shown in Figure 5.2(b). Here the CFG clearly shows that one and only one side of the if statement may execute. The CFG of the predicated code is shown in Figure 5.2(c). In this case all the code falls into one basic block because there is no possibility of branching until the end of the set of instructions. Although a CFG-based dataflow algorithm might treat all instructions in Figure 5.2(c) as control equivalent due to their common fetch condition, for compiler transformations the execution condition is a more useful foundation for analysis. The PFG is an extension of the CFG that represents the relations among the execution rather than the fetch conditions of instructions. The CFG represents the fetch behavior of a program, whereas the PFG represents the execution behavior. In codes without predication, where fetch

behavior and execute behavior are equivalent, the CFG is sufficient for analysis. In a manner analogous to the definition of the control flow graph, the definition of the predicate flow graph begins with the definition of the predicate block. The predicate block is the predicated code equivalent of the basic block.

Predicate block. A *predicate block* is a sequence of consecutive statements in which the flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. Additionally, a predicate block has the restriction that all instructions it contains will execute together only under a set of given predicate values. Instructions may exist in one or more predicate blocks.

The *TRUE predicate value set* is a set of predicate register numbers which must be 1 in the associated predicate block. The *FALSE predicate value set* is a set of predicate register numbers which must be 0 in the associated predicate block. Predicates not in the TRUE or FALSE predicate value sets for a predicate block may either assume any value or be undefined during the execution of that predicate block.

Instructions cannot be in the same predicate block if they would not have been in the same basic block. A predicated instruction may only exist in a predicate block if that predicate block's predicate value set specifies a condition under which it would execute.

Predicate flow graph. A *predicate flow graph* is a directed graph G augmented with a unique *entry* node START and a unique *exit* node STOP such that each node in the graph has at most two successors. We assume that nodes with two successors have attributes "T" (true) and "F" (false) associated with the outgoing edges originating at a real or implied branch or the start of a predicate's *active range*. We further assume that

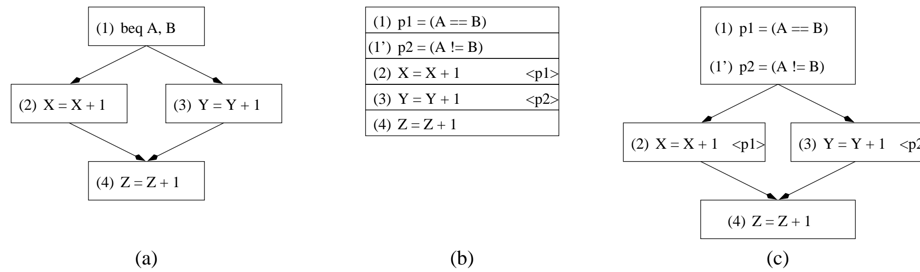


Figure 5.3: The CFG (b) and PFG (c) for a simple if-then-else construct (a).

for any node N in G there exists a path from START to N and a path from N to STOP. All nodes in this graph represent predicate blocks.

Figure 5.3 shows a PFG for a simple if-then-else construct. Figure 5.3(a) shows the original branch code in a CFG. Figure 5.3(b) shows the CFG of the same code after if-conversion. The graph in Figure 5.3(b) provides only fetch information, not the execution information necessary for proper dataflow analysis. Figure 5.3(c), the PFG, represents the conditional execution of instructions 2 and 3 and, as such, has a structure very similar to that of the original branch code. Clearly, in this example, dataflow analysis run on the CFG in Figure 5.3(a) would return the same result as if it were run on the PFG in Figure 5.3(c) except for queries on the newly introduced predicate registers.

5.2.1 Active range

The definition of the PFG uses the term *active range*, which is an essential component of the proper creation of predicate flow graphs. Conceptually, a PFG is a CFG with additional split and merge points. The CFG in Figure 5.3(b) was modified by the addition of a single split point after instruction 1 and by the addition of a single merge point before instruction 4. Split points are placed at the start of any new active range. Likewise, merge

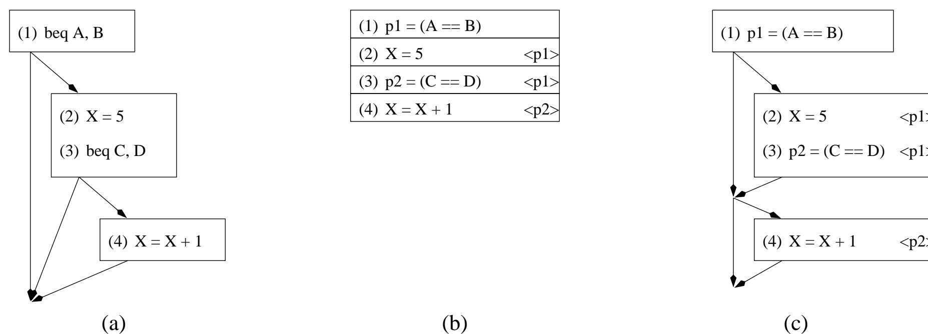


Figure 5.4: An example of incorrectly using live range as active range.

points are placed at the end of any active ranges. In this example, the two active ranges, those for $p1$ and $p2$, had the same start and end points, so only one spilt and one merge were necessary, as shown in Figure 5.3(c). This trivial example suggests that the active range of a predicate could be the same as the live range of that predicate. Unfortunately, accurate dataflow analysis requires a more sophisticated definition.

Figure 5.4(b) shows the if-converted version of the simple code segment in Figure 5.4(a). Using the live range as the active range, the PFG in Figure 5.4(c) is derived. Clearly, there exist paths in the PFG which did not exist in the original CFG. In the PFG, the variable X appears to be live entering the graph because the definition in instruction 2 does not kill all uses in instruction 4. As this example shows, the active range of a predicate must extend to all points which are affected by the outcome of that predicate, often beyond the predicate's live range.

Active range. An *active range* for a predicate pN consists of the set of instructions on all fetch paths from all uses of pN to all instructions whose conditions of execution are *logically* dependent on the value of pN.

The next section presents the PFG construction algorithm. The first step in this process is the computation of the active ranges.

5.2.2 Predicate flow graph construction

In this section, the algorithm used in the creation of PFGs is presented. For the purposes of this discussion, predicates are assumed not to be active across hyperblock boundaries or back edges. With this assumption, an accurate PFG can be created by replacing each hyperblock in the CFG with its PFG. This is possible because all active ranges begin and end within a hyperblock. The splits and merges in the graph do not affect the hyperblock's entries and exits. The remainder of this section presents an algorithm for creating a predicate flow graph segment from a single hyperblock.

Figure 5.5 is the algorithm to compute predicate active ranges, the first step in the process of creating the PFG. In lines 1-5, each instruction which is the last use of a predicate is tagged with that predicate in the *Op.LastUse* field. Because the active range starts at the first use and continues at least until the last use, this information will be useful as it is the earliest point at which that predicate's active range may end. The loop in lines 8-16 traverses the hyperblock to compute *PredsActive*, the set of active predicates, at each operation. This set is stored in *Op.ActiveSet* for each operation in the hyperblock. Lines 9 and 10 add first uses of a predicate to the active set. The predicate then stays in the active set for each subsequent operation until it is removed in lines 11-15. As each predicate's last use is encountered and is added to *PredsDone* set, the set of all predicates whose active range has ended or will end when the loop passes all last uses of predicates related to it. Each predicate in the *PredsActive* set is tested

```
ComputeActiveRanges(Block Hyperblock)
(1)   PredsSeen = ∅;
(2)   FOREACH Op IN REVERSE Hyperblock.Ops DO
(3)       IF (NOT Op.Pred ∈ PredsSeen) THEN
(4)           Op.LastUse = Op.Pred;
(5)           PredsSeen = PredsSeen ∪ {Op.Pred};
          ENDIF
      END

(6)   AllPreds = All preds live in hyperblock;
(7)   PredsActive = ∅;
(8)   FOREACH Op IN Hyperblock.Ops DO
(9)       IF (NOT Op.Pred ∈ PredsActive) THEN
(10)          PredsActive = PredsActive ∪ {Op.Pred};
          ENDIF

(11)      PredsDone = PredsDone ∪ {Op.LastUse};
(12)      PredsRelevant = AllPreds − PredsDone;
(13)      FOREACH Pred IN PredsActive DO
(14)          IF (NOT PredsRelated(Pred, PredsRelevant)) THEN
(15)              PredsActive = PredsActive − {Pred};
              ENDIF
          END
(16)      Op.ActiveSet = PredsActive;
      END
```

Figure 5.5: Algorithm to compute predicate active ranges for a hyperblock.

in line 14. If a predicate is not related to any predicates whose last use has not been encountered, then it can be removed from the active set in line 15. Line 16 stores the state of the active set into each operation in the hyperblock for use in the construction of the predicate graph described next.

The active ranges for hyperblocks determine where splits and merges in the graph are to be made. A split creates two new successor predicate blocks for an existing predicate block. A merge creates a single new predicate block, which is the successor for two existing predicate blocks. As discussed earlier, a predicate block is partially defined by

its predicate value sets. The algorithms presented here utilize the two sets defined earlier, $PRED_TRUE$ and $PRED_FALSE$, to represent this predicate value set. $PRED_DEF$ is the set of predicates which are active throughout the predicate block and is defined as the set union $PRED_TRUE \cup PRED_FALSE$.

Instructions can be duplicated to exist in more than one predicate block. To avoid confusion, the term *operation* is used to refer to the unique operation in the predicated code. The term *instruction* is used to describe the instantiation of an operation in each predicate block.

The algorithm in Figure 5.6 uses the active range information computed in the algorithm in Figure 5.5 to construct the predicate flow graph. To start the construction process, an empty entry predicate block is created with $PRED_DEF$ and $PRED_TRUE$ initialized to contain only the always TRUE predicate (lines 1-3). In line 4, an active predicate block list, $PredBlockList$, is initialized to contain only the entry predicate block as its single entry. The hyperblock is processed top down, operation by operation in the main loop in lines 6-17.

In the main loop, three tasks are performed. The first task is the spliting of the predicate flow graph to handle the new possible predicate outcomes. The graph is split at the start of each predicate active range. Lines 7-9 determine which active ranges have begun and call $SplitPredBlock$ for each newly activated predicate. In a similar manner, lines 10-11 perform the second task, merging paths that are no longer necessary after a predicate's active range has ended. $MergePredBlocks$ is called once for each merge to be performed each point in the operation list. The final task (lines 12-17) involves populating the instructions in the current list of predicate blocks with source and

```
CreatePredicateFlowGraph(Block Hyperblock)
(1)   EntryPredBlock = NewPredBlock();
(2)   EntryPredBlock.PRED_TRUE = {T};
(3)   EntryPredBlock.PRED_DEF = {T};
(4)   PredBlockList = EntryPredBlock;

(5)   PrevActiveSet = ∅;
(6)   FOREACH Op IN Hyperblock.Ops DO
(7)       PredsActivated = Op.ActiveSet − PrevActiveSet;
(8)       FOREACH Pred IN PredsActivated DO
(9)           PredBlockList = SplitPredBlocks(PredBlockList, Pred);
          ENDIF

(10)      PredsDeactivated = PrevActiveSet − Op.ActiveSet;
(11)      PredBlockList = MergePredBlocks(PredBlockList, PredsDeactivated);

(12)      FOREACH PredBlock IN PredBlockList DO
(13)          Inst = NewPredInst(Op, PredBlock);
(14)          PredBlock.Insts = PredBlock.Insts + Inst;
(15)          Op.Insts = Op.Insts + Inst;
          END
(16)      InsertInstructionOperands(Op);
(17)      PrevActiveSet = Op.ActiveSet;
      END
```

Figure 5.6: Algorithm to create the predicate flow graph for a hyperblock.

destination operands. The function *InsertInstructionOperands* places these operands only into predicate blocks with the appropriate predicate value conditions. The functions *SplitPredBlock*, *MergePredBlocks*, and *InsertInstructionOperands* are discussed in detail below.

The algorithm in Figure 5.7 creates one or two new predicate blocks for each predicate block in the current predicate block list. The predicate values in the current predicate block, *PRED_TRUE* and *PRED_DEF*, may allow *Pred*, the newly activated predicate, to assume only 1, only 0, or either value. Calls to the *TrueValue* and *FalseValue*

```
List SplitPredBlocks(List PredBlockList, Operand Pred)
(1)   FOREACH PredBlock IN PredBlockList DO
(2)       IF (TrueValue(Pred, PredBlock.PRED_TRUE, PredBlock.PRED_DEF AND
              FalseValue(Pred, PredBlock.PRED_TRUE, PredBlock.PRED_DEF) THEN
(3)           PredBlockTrue = NewPredBlock();
(4)           PredBlockTrue.PRED_TRUE = PredBlockTrue.PRED_TRUE + {Pred};
(5)           PredBlockTrue.PRED_DEF = PredBlockTrue.PRED_DEF + {Pred};
(6)           PredBlockFalse = NewPredBlock();
(7)           PredBlockFalse.PRED_DEF = PredBlockFalse.PRED_DEF + {Pred};
(8)           PredBlockList = GraphSplit(PredBlockList, PredBlock,
                  PredBlockTrue, PredBlockFalse);
(9)       ELSEIF (TrueValue(Pred, PredBlock.PRED_TRUE, PredBlock.PRED_DEF) THEN
(10)          PredBlockTrue = NewPredBlock();
(11)          PredBlockTrue.PRED_TRUE = PredBlockTrue.PRED_TRUE + {Pred};
(12)          PredBlockTrue.PRED_DEF = PredBlockTrue.PRED_DEF + {Pred};
(13)          PredBlockList = GraphLink(PredBlockList, PredBlock, PredBlockTrue);
(14)      ELSEIF (FalseValue(Pred, PredBlock.PRED_TRUE, PredBlock.PRED_DEF) THEN
(15)          PredBlockFalse = NewPredBlock();
(16)          PredBlockFalse.PRED_DEF = PredBlockFalse.PRED_DEF + {Pred};
(17)          PredBlockList = GraphLink(PredBlockList, PredBlock, PredBlockFalse);
          ENDIF
      END
(18)  RETURN PredBlockList;
```

Figure 5.7: SplitPredBlocks function for creation of Predicate Flow Graph.

functions are used to determine the possible predicate value sets. When 1 is a possible value for *Pred*, a *PredBlockTrue* block is created to represent this execution condition. Likewise, when 0 is a possible value for *Pred*, the *PredBlockFalse* block is created to represent this execution condition. One or both of these new predicate blocks are then linked as a successors to the original predicate block.

The algorithm in Figure 5.8 creates a single successor predicate block for each pair of predicate blocks which assume the same predicate condition value set, *PRED_TRUE* and *PRED_DEF*, disregarding predicates to be deactivated. The algorithm proceeds using one predicate in the *PredsDeactivated* set at a time. Lines 2-8 compare each pair of predicate blocks to see if the predicate value sets would be the same if the predicate being deactivated were no longer relevant. If such a pair is found, a new predicate block

```
List MergePredBlocks(List PredBlockList, Set PredsDeactivated)
(1)   FOREACH Pred IN PredsDeactivated DO
(2)       FOREACH PredBlock1 IN PredBlockList DO
(3)           FOREACH PredBlock2 IN PredBlockList AFTER PredBlock1 DO
(4)               NEW_PRED_TRUE_1 = PredBlock1.PRED_TRUE − Pred;
(5)               NEW_PRED_DEF_1 = PredBlock1.PRED_DEF − Pred;
(6)               NEW_PRED_TRUE_2 = PredBlock2.PRED_TRUE − Pred;
(7)               NEW_PRED_DEF_2 = PredBlock2.PRED_DEF − Pred;
(8)               IF (NEW_PRED_TRUE_1 == NEW_PRED_TRUE_2 AND
                      NEW_PRED_DEF_1 == NEW_PRED_DEF_2) THEN
(9)                   PredBlock = NewPredBlock();
(10)                  PredBlock.PRED_TRUE = NEW_PRED_TRUE_1;
(11)                  PredBlock.PRED_DEF = NEW_PRED_DEF_1;
(12)                  PredBlockList = GraphMerge(PredBlockList, PredBlock1,
                          PredBlock2, PredBlock);
                  ENDIF
              END
          END
      END
(13)  RETURN PredBlockList;
```

Figure 5.8: MergePredBlocks function for creation of Predicate Flow Graph.

is created that has the new predicate value sets (lines 9-11). The new predicate block is then linked as a successor to the two original predicate blocks in line 12.

The final step in the creation of the predicate flow graph is the insertion of operands. This step is necessary to prepare the graph for use in dataflow analysis. The idea is to place source and destination operands in only those predicate blocks whose predicate value conditions make the operands visible. For example, consider the operation $r1 = r2 + 1 < p1 >$. Assume that this operation has two instructions in two predicate blocks in the predicate flow graph. The first predicate block has $p1$ as 1, and the second predicate block has $p1$ as 0. In the first predicate block, the instruction will execute normally; $r2$ and $p1$ are both sampled as source registers, and the destination $r1$ is

modified with the result of the computation. Thus, $p1$ and $r2$ are both source registers, and $r1$ is a destination register. In the second predicate block, however, the situation is different. The predicate $p1$ is still sampled by the machine and found to be 0, causing the instruction to be nullified. Because $r2$ is not (meaningfully) sampled and $r1$ is not written, the instructon in the second predicate block contains only $p1$ as a source register and contains no destination registers.

The $InsertInstructionOperands$ function, shown in Figure 5.9, traverses every instruction of the parameter operation Op. Predicate registers are always sourced, so regardless of the predicate block's predicate value sets, the predicate register is always placed in instruction's source register list (line 2). Most other registers are sourced or written only when the source predicate is 1. Therefore, in line 4 and line 7 the predicate block's $PRED_TRUE$ set is examined for the source predicate. If the source predicate is in the $PRED_TRUE$ set, the operand is inserted into the instruction's operand list lists. Some source and destination operands are unconditionally used or defined, such as in the case of unconditional-type predicate defines. Line 4 and line 7 also test for these special operands and add them to the operand lists as appropriate.

At the completion of predicate flow graph construction for each hyperblock, the hyperblocks in the original CFG are replaced, for purposes of analysis. The result is a PFG for the entire function complete with source and destination operands in the appropriate locations. The PFG is at this point ready for dataflow analysis.

```
InsertInstructionOperands(Op)
(1) FOREACH Inst IN Op.Inst DO
(2)     Inst.Src = Inst.Src + Op.Pred;
(3)     FOREACH Operand IN Op.Src DO
(4)     IF (Op.Pred ∈ Inst.PredBlock.PRED_TRUE OR
            UnconditionalOperand(Operand)) THEN
(5)         Inst.Src = Inst.Src + Operand;
        ENDIF
        END
(6)     FOREACH Operand IN Op.Dst DO
(7)     IF (Op.Pred ∈ Inst.PredBlock.PRED_TRUE OR
            UnconditionalOperand(Operand)) THEN
(8)         Inst.Dst = Inst.Dst + Operand;
        ENDIF
        END
    END
```

Figure 5.9: Insertion of operands into the Predicate Flow Graph of a Hyperblock.

5.2.3 The graph width explosion problem

Depending on the nature of the code, the PFG created by these algorithms can grow exponentially. Consider the example in Figure 5.10(a). Here three independent predicates are defined in operations 1-3 and used in operations 4-6. In the predicate flow graph, each possible execution set must be represented. For n independent predicates there are a total of 2^n execution sets. If these independent predicates have active ranges which overlap, the width of the predicate flow graph is 2^n. This is the case in Figure 5.10(b), where 3 predicates create a PFG of width $2^3 = 8$. At some point the exponential growth of graph width becomes unmanageable. When this occurs a graph width explosion problem is said to have occured.

To manage the graph width explosion problem, the predicate flow graph generation algorithm should contain a limiting function. To accomplish this, a test of the list

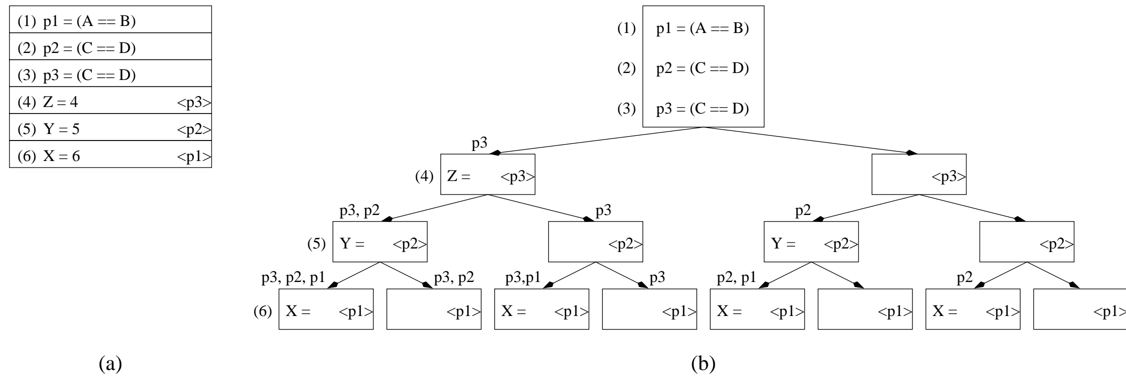


Figure 5.10: Explosion of a predicate flow graph (b) representing independent predicates with overlapping active ranges (a).

size of the *PredBlockList* should be inserted in the the alogorithm of Figure 5.6 to suppress the call to the *SplitPredBlocks* function in line 9 when *PredBlockList* reaches a predetermined size. For predicates for which a split was suppressed, dataflow analysis may be conservative, however, this is a necessary trade-off to bound compile time. In the subsequent description of the dataflow analysis engine, a modification is made to the traditional dataflow analysis equation solver to support conservative analysis for split suppressed predicates.

5.3 Dataflow Analysis on the Predicate Flow Graph

One goal of the predicate dataflow graph was to enable the use of traditional dataflow equation solvers. This goal is met with the exception of two modifications which must be made to all dataflow analyses. The first modification relates to the solution of the path explosion problem presented earlier. Essentially, the dataflow analysis routines must be aware of which predicates are treated conservatively. This modification is the same one that traditional dataflow routines must undergo to handle predicated code in a correct,

albeit conservative, manner. An effect of this is that if a control flow graph is used instead of a predicate flow graph, correct but conservative dataflow results.

When a predicate is not contained in the predicate value sets for a predicate block it must be treated as it can assume any value. As shown earlier in this chapter, this yeilds conservative results, such as extended live ranges. Predicated instructions with predicates in the predicate value sets are treated in an accurate manner.

All definitions of instructions having conservatively treated predicates must be assumed conditional. This means that the definitions cannot be considered to kill a live range. Figure 5.11 shows the definition and use predicate block set computation for live variable analysis. This algorithm is identical to that used for basic blocks in a control flow graph with the exception of the if-statement in line 9 which guards line 10. The if-statement simply tests for the predicate register in the $PRED_DEF$ set. If the predicate exists in the set, then the operand is known to be unconditional. As an unconditional destination operand, the definition can be considered to kill any live variable range. Subsequent steps in the computation of register live ranges are unaffected by the predicate flow graph.

Once the desired dataflow sets have been computed for all instructions in all predicate blocks, this information needs to be combined to create the dataflow information that is associated with each real operation. (Many sophisticated optimizations, one of which is partial dead code removal to be described later, use the information on the predicate flow graph directly.) This is done by using the *confluence operator* to create a single operand set from all instruction sets. Figure 5.12 shows the standard form for collecting this information. Here the confluence operator is repesented by $\bigwedge$. For live variable analysis,

```
(1) FOREACH PredBlock IN Program.PredBlocks
(2)      DEF = ∅;
(3)      USE = ∅;
(4)      FOREACH Inst IN PredBlock.Insts DO
(5)          FOREACH Operand IN Inst.Op.Src DO
(6)              IF (NOT Operand ∈ DEF) THEN
(7)              USE = USE ∪ {Operand};
             END
(8)          FOREACH Operand IN Inst.Op.Dest DO
(9)              IF (Inst.Op.Pred ∈ PredBlock.PRED_DEF OR
                     UnconditionalOperand(Operand)) AND
                     NOT Operand ∈ USE) THEN
(10)                 DEF = DEF ∪ Operand;
                 ENDIF
             END
         END
(11)     Block.USE = USE;
(12)     Block.DEF = DEF;
     END
```

Figure 5.11: Predicate block definition and use computation for live variable analysis.

the confluience operator is set union, $\bigwedge = \cup$. Intuitively, the live range of a variable is the union of its live range in all valid predicate value sets.

5.4 Predicate Partial Dead Code Elimination

Up until now, this chapter has concerned itself with presenting traditional optimizations with accurate dataflow information in the presence of predication. This information, derived by merging of dataflow sets for traditional queries, keeps the optimizations performing well in the presence of multiple overlapped predicate paths. However, the PFG

```
(1) FOREACH Block IN Program.Blocks
(2)     FOREACH Op IN Block.Ops DO
(3)         Op.IN = ∅;
(4)         Op.OUT = ∅;
(5)         FOREACH Inst IN Op.Insts DO
(6)             Op.IN = Op.IN ⋀ Inst.IN;
(7)             Op.OUT = Op.OUT ⋀ Inst.OUT;
            END
        END
    END
```

Figure 5.12: Merging of dataflow sets for traditional queries.

contains additional information about individual predicate paths is critical to the success of the Partial Reverse If-Conversion Framework. This information is extracted and applied in an optimization referred to as predicate partial dead code elimination.

Traditional dead code elimination deletes instructions which do not affect program execution. If values defined by an instruction are never used, then that instruction is dead. This optimization uses each instruction's live-out set to determine whether or not it is dead. If the live-out set does not contain any of the destination registers, that instruction is dead. A related optimization, partial dead code elimination, is the equivalent to dead code elimination on the PFG. Predicate partial dead code removal is related to other types of partial dead code removal [51].

Figure 5.13 shows the hyperblock from Figure 5.1 and its PFG. In this example, we are concerned with performing dead code analysis on the predicate flow graph. Live variable analysis on the predicate flow graph indicates that when the predicate $p1$ is false the result generated by instruction 5 is not used. Using the rule for eliminating dead code, we can delete instruction 5 from the "$p1$ is false" path. Since the instruction

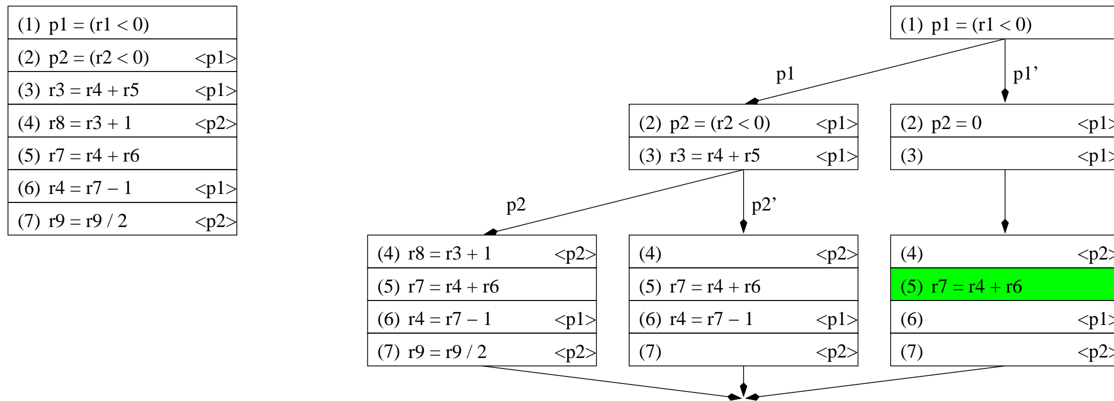


Figure 5.13: Partial dead code in a predicate flow graph.

generates a value which is used in other paths, we cannot delete the instruction in the original hyperblock.

The Partial Reverse If-Conversion Framework uses the fact that instruction 5 is partially dead in two ways. In the first use, partial reverse if-conversion must know when an instruction is live under each predicate in order to determine how to reverse if-convert a hyperblock under a specific predicate. In this example, a reverse if-conversion on $p1$ would not need to have a copy of instruction 5 in the newly created "$p1$ is false" hyperblock. Details of this application are presented in Chapter 7.

The second use of partial dead code is for demotion. An optimization, called promotion, is applied aggressively to all instructions in hyperblocks. Promotion reduces the strength or even completely removes predicates when it is possible to do so without affecting program outcome. This is done so that the scheduler can have the freedom to schedule instructions prior to the predicate defines which once guarded them. This is a form of speculation since the instructions are as a result executed in more situations than in the original program. After prepass scheduling, it is wise to demote, or increase the

strength of, guard predicates. By increasing the strength of guard predicates, instructions will be made to execute less frequently. In the case of instructions with side effects, this can improve program performance. Demoting loads, for example, eliminates unnecessary memory accesses. Demoting after prepass preserves the promotion necessary to improve the code schedule, while eliminating unnecessary speculation.

Performing demotion is a two-step process. The first step, partial dead code elimination is performed as described above. The second step is selecting the strongest possible predicate which can legitimately guard the instruction. To do this, each predicate path in which the instruction exists is examined. The intersection of all the PRED_TRUE sets for predicate blocks containing the instruction is formed. This set is the set of predicates that can guard the instruction. Since the best candidate for demotion is the predicate which most strongly guards the instruction, the predicate that is true in the fewest number of predicate blocks is chosen. Consider instruction 5 in Figure 5.13. Predicates $p1$ and $p2$ are defined at instruction 5; however, only predicate $p1$ is in the intersection of the PRED_TRUE sets for two leftmost paths. Predicate $p1$, being the only predicate in the intersection, is necessarily the most restrictive. Demotion occurs when instruction 5 is predicated on $p1$. This is intuitive since the value $r7$ is consumed only by instruction 6, which in turn executes only when $p1$ is true.

5.5 Significance

The PFG is essential to the success of the proposed framework. Using the information extracted from the PAS, a PFG is built representing the true flow of information within a hyperblock by means of a reverse if-conversion. Predicate-aware dataflow analysis

can then be performed simply by applying standard iterative dataflow analysis to the PFG. The resulting dataflow information is as accurate as traditional analysis applied to the equivalent nonpredicated code. Partial dead code elimination, performed on the PFG, is necessary to enhance scheduling freedom and to enable efficient partial reverse if-conversion.

6. BOOLEAN MINIMIZATION OF PROGRAM DECISION LOGIC

One fundamental limitation of most branch handling techniques is that they do not significantly alter the program's control flow logic. As the compiler translates high-level language control constructs into assembly-level branches, it does not alter the basic control structure. Instead, techniques focused on exposing and increasing ILP within a fixed control structure are applied. With control speculation, this is obvious. Control dependences are removed to enable the motion of instructions above branches. The branches themselves are not altered. Likewise, when predication is applied by the process of if-conversion, branches are transformed into predicate computations and control dependent instructions are rendered conditional by the addition of guarding predicates. This process converts control flow and control dependences into data flow and data dependences, but preserves the original program's control structure.

Restricting a compiler to use the program's unaltered control structure is undesirable for several reasons. First, a high-level language such as C or C++ represents program control flow in a sequential manner through the use of nested if-then-else statements, switch statements, and loop constructs. Each control construct is fully evaluated before

proceeding to the next. This sequential computation often defines critical paths that constrain the available ILP. Second, programmers often represent control flow for understandability or for ease of debugging rather than for efficient execution on the target architecture. As a result, software often contains redundant control constructs that are difficult to detect with traditional compiler techniques. These may involve evaluating the same conditions multiple times or evaluating conditions that partially overlap. An effective ILP compiler should be capable of transforming the program control structure to eliminate these problems.

The ability to restructure code aggressively is a critical feature of an effective ILP compiler. The most obvious situation where aggressive transformation is regularly applied is on arithmetic expressions. Compilers often completely restructure the programmer's arithmetic computations into more parallel forms using a variety of transformations. These include expression re-association, tree height reduction [52], and blocked back substitution [53]. Although ILP compilers may aggressively restructure computation, they typically preserve the program's original control structure. This conservative approach can seriously limit the level of efficiency as well as the level of ILP achieved in branch-intensive programs.

Approach. Motivated by the potential of aggressive techniques for transforming arithmetic expressions, this chapter presents a new approach to optimizing program control flow. The goal is to develop a systematic methodology for reformulating program control flow for more efficient execution on an ILP processor. Control expressed in branches and predicate define instructions is first extracted and represented as a *program decision logic network*. Then, a new, more efficient network is synthesized with the goals

of reducing dependence height and redundancy. To accomplish the desired optimization and synthesis, the program decision logic network is modeled as a Boolean equation. Boolean minimization techniques are then applied to simplify and optimize the equation. Finally, the optimized network is re-expressed in the form of predicated assembly code. One unique feature of this approach is that all branches and predicates within a segment of code are treated jointly in a systematic manner [49].

Previous Work. Previous research in the area of control flow optimization can be classified into three major categories: branch elimination, branch reordering, and control height reduction. Branch elimination techniques identify and remove those branches whose direction is known at compile time. The simplest form of branch elimination is loop unrolling, in which instances of back edge branches are removed by replicating the body of the loop. More sophisticated techniques examine program control flow and data flow simultaneously to identify correlations among branches [54], [55]. When a correlation is detected, a branch direction is determinable by the compiler along one or more paths, and the branch can be eliminated. In [55], an algorithm is developed to identify correlations and to perform the necessary code replication to remove branches within a local scope. This approach is generalized and extended to the program-level scope in [54]. The second category of control flow optimization work is branch reordering. In this work, the order in which branches are evaluated is changed to reduce the average depth traversed through a network of branches [56].

The final category of control flow optimization research focuses on the reduction of control dependence height. This work attempts to collapse the sequential evaluation of linear chains of branches in order to reduce the height of program critical paths [21].

In an approach analogous to a carry look-ahead adder, a look-ahead branch is used to calculate the taken condition of a series of branches in a parallel form. Subsequent operations dependent on any of the branches in the series need only to wait for the lookahead branch to complete. The control dependence height of the branch series is thus reduced to that of a single branch. The mechanisms introduced herein also serve to reduce control dependence height. This chapter, however, introduces an approach to minimization and re-expression of control flow networks that is far more general than those proposed in previous work.

6.1 Overview of Compiler Techniques

This section presents a conceptual overview of the program decision logic minimization process, starting with the conversion of code to the predicated representation for subsequent optimization. In order to simplify the extraction and manipulation of control expressions, the compiler applies if-conversion and reformulation of nonbranch control constructs to transform all programmatic control flow into the predicated representation. In the IMPACT compiler, this conversion is fully performed within acyclic code regions formed using *hyperblock* formation heuristics [17]. To a great extent, the ability of these control logic optimization techniques to improve performance depends on the scope of these regions, as only the control structure transformed into the predicate domain is available for subsequent optimization. In order to promote effective hyperblock formation, aggressive function inlining is performed.

An example extracted from the Unix utility *wc* illustrates the application and benefit of the described techniques. Figure 6.1 shows the code segment before and after complete

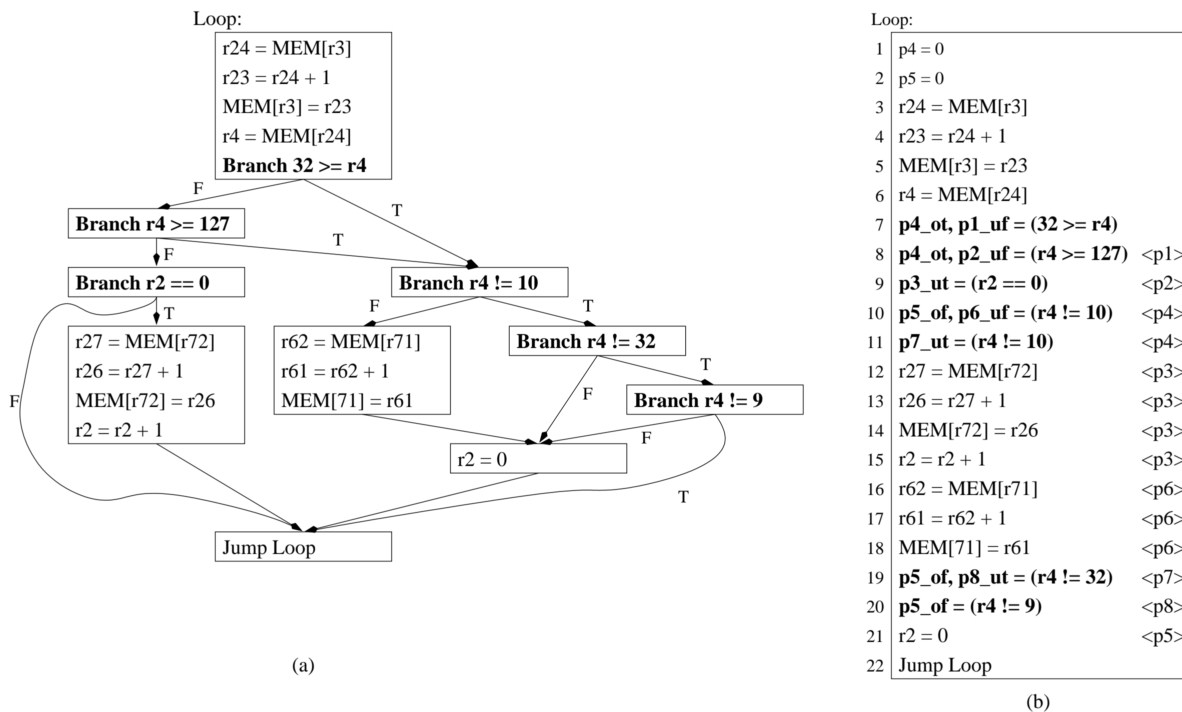


Figure 6.1: A portion of the inner loop of the Unix utility *wc*. The control flow graph (a), and the corresponding hyperblock formed after complete if-conversion (b).

if-conversion. As shown in Figure 6.1(a), the code before if-conversion consists of basic blocks and conditional branches (shown in bold) which direct the flow of control through the basic blocks. As shown in Figure 6.1(b), the code after if-conversion consists of only a single block of sequential instructions, a hyperblock [36]. The conditional branches have been replaced with predicate define instructions (shown in bold), and the predicate registers defined have been placed as source operands on all guarded instructions in accordance with their execution conditions.

After if-conversion, control speculation is performed to increase opportunities for optimization. Control speculation is a means of breaking a control dependence by allowing an instruction to execute more frequently than is necessary. In a predicated representation, this is performed in *predicate promotion*, the process by which predicate flow

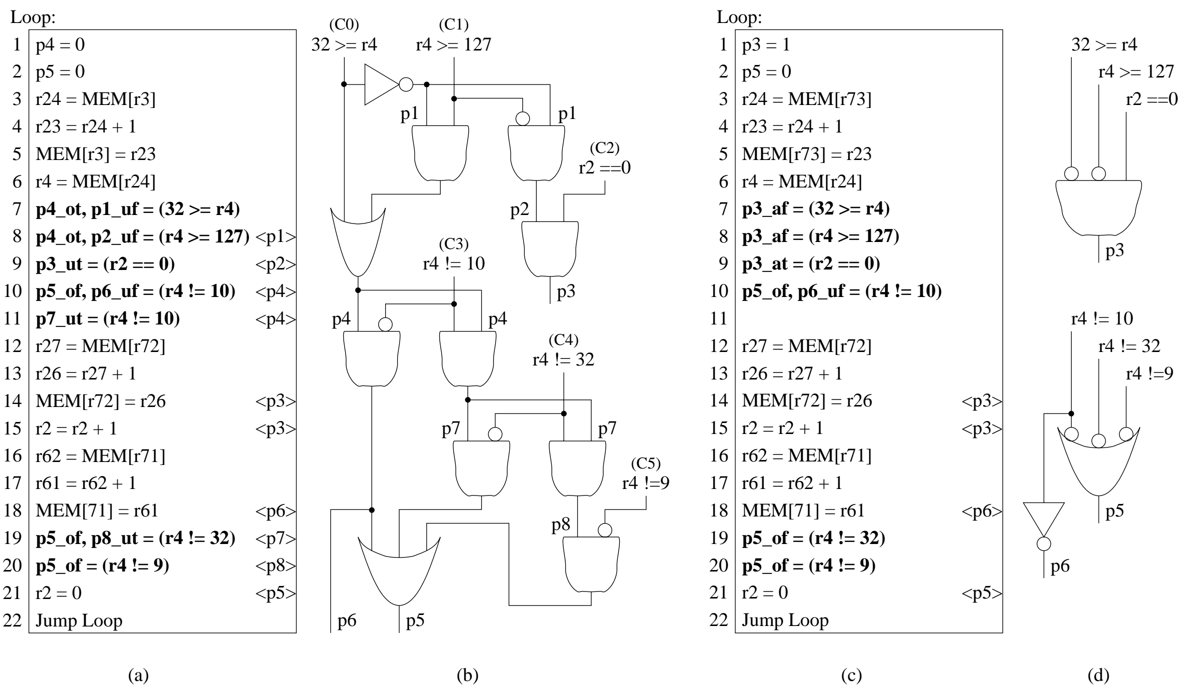


Figure 6.2: The *wc* hyperblock after speculation but before logic minimization (a) and its corresponding logic diagram (b). The hyperblock after logic minimization (c) and its corresponding logic diagram (d).

dependences are broken and instructions are made to execute speculatively by changing an instruction's guard predicate to another predicate, whose expression subsumes that of the original [17]. When instructions are aggressively promoted, some predicates may no longer be utilized as guards on computation. When a predicate is no longer necessary, the program decision logic is simplified. Figure 6.2(a) shows the *wc* hyperblock segment after predicate promotion. Comparison with Figure 6.1(b) shows that four instructions (12, 13, 16, and 17) have had their predicates promoted to the TRUE predicate, denoted in the figure as the absence of a source predicate. However, no predicates were rendered completely unused by this process.

Next, the program decision logic network is constructed. Since predicates can only assume Boolean values, predicates and predicate defines in an acyclic region can be viewed

as a combinational logic circuit. To derive the Boolean functions from a hyperblock, the compiler needs only to examine the predicate define instructions. Consider instructions 7 and 8 in Figure 6.2(a), in which the expression for $p1$ can be written as $p1 = \overline{C}_0$ and $p2$ can be written as: $p2 = p1\overline{C}_1$, where C_0 is the condition: (32 $\geq$ r4) and C_1 is the condition: (r4 $\geq$ 127). The expression for $p2$, in terms of conditions, is $p2 = \overline{C}_0\overline{C}_1$. In the course of this complete back substitution, expressions based on condition variables are formulated for all predicate define instructions. The composition of all these expressions is the program decision logic network. This network can be modeled as a logic circuit that represents all the decisions made in the program. The logic circuit has conditions as its input and the predicates which control computation as its output. The multiple-output Boolean logic circuit for the *wc* code segment is shown in Figure 6.2(b).

Once the logic circuit has been derived, many computer-aided design (CAD) techniques can be employed to simplify the program decision logic network. In the IMPACT compiler, the derived Boolean function is represented with a BDD [43]. The BDD algorithms used are described in [45]. The *predicate BDD* contains the relationship among predicates as defined by the network of predicate define operations. The predicate BDD is used throughout the compiler as a database for queries made by optimizations when operating on predicated code. For example, one common query is to determine if one instruction executes only when another instruction has executed. This query is equivalent to the dominance relationship in the control flow domain. Here, the BDD is queried to determine if the predicate expression of one instruction subsumes the predicate expression of another. Queries to the BDD are made in IMPACT by the optimizer, the scheduler, and dataflow analysis.

For the purposes of decision logic minimization, the BDD provides a simple method by which expressions describing the hyperblock logic can be derived. The only expressions requested from the BDD are those expressions describing the *essential predicates*. Essential predicates are those predicates that guard real computation instructions (any instruction that is not a predicate define). In Figure 6.2a, the essential predicates are $p3$, $p5$, and $p6$. Predicates $p1$, $p2$, $p4$, $p7$, and $p8$ are *nonessential predicates* as they are used only as intermediates in evaluation of the essential predicates.

The BDD maintains a canonical representation of the decision logic functions, from which a Boolean sum-of-products expression can be produced for any represented function. Note that the expression thus generated reflects the canonical nature of the BDD's internal representation and is usually not optimal for expressions with multiple product terms. Therefore, it is necessary to optimize the derived expression before attempting to synthesize a predicate defining structure.

The expressions describing the evaluation of the essential predicates are optimized using techniques which eliminate redundant terms in the function and which reexpress the Boolean function in a more parallel form. The resulting expression is reformulated back into predicate define instructions in the hyperblock. Section 6.2 presents the details of the Boolean logic optimizers and reformulators studied in this work. These optimizers and reformulators must balance the reduction of dependence height with the number of predicate defines that can be accommodated in the code schedule. This involves making an accurate estimate of how much time is available for computation of control functions based on the availability times of conditions and when predicates need to be consumed.

These and other considerations make the design of an optimizer and a reformulator nontrivial.

Figures 6.2(c)and 6.2(d) show the reformulated hyperblock and corresponding logic circuit after the minimization process is complete. The number of logic gates in the circuit implementation is reduced from ten to three. In addition, the six-level gate network in Figure 6.2(b) is reduced to a single-level gate network in Figure 6.2(d). All nonessential predicates were also eliminated as part of this process. An example optimization performed on the logic circuit takes the form: $C_0 + C_1\overline{C}_0 \rightarrow C_0 + C_1$. An application of this optimization occurs between instructions 7 and 8 when computing $p4$.

The values of variables in the decision logic network are supplied by evaluating conditions on predicate define instructions. It is important to recognize that these variables are not necessarily independent, and that knowledge of the relationships between these variables can allow for significant further optimization of the predicate define structure. Consider the computation of $p6$ in Figure 6.2(a). Instruction 10 computes $p6_uf = C_3$ $\langle p4 \rangle$. Logically, this leads to the expression $p6 = \overline{C}_3(C_0 + C_1)$, where $C_0 = (32 \geq r4)$, $C_1 = (r4 \geq 127)$, and $C_3 = (r4 \neq 10)$. Here, since $\overline{C}_3$ implies C_0 and excludes C_1, the expression for $p6$ can be simplified to $p6 = \overline{C}_3$. In this approach, the relationships between conditions are represented in a BDD, termed the *condition BDD*, which can be queried to determine if logical implications exist between conditions and, if so, what they are. The current implementation of this mechanism identifies "families" of integer register-constant comparisons which are based on the same definition of a given register. Then, within each family, a number line is created and divided into disjoint segments

Cycle	Instructions issued					
0	op1	op2	op3	op12	op16	
1	op4	op6	op13	op17		
2	op5	**op7**				
3	**op8**					
4	**op9**	**op10**	**op11**			
5	op14	op15	op18	**op19**		
6	**op20**					
7	op21	op22				

(a) Schedule for the hyperblock in Figure 6.2(a).

Cycle	Instructions issued					
0	op1	op2	op3	op12	op16	
1	op4	op6	**op9**	op13	op17	
2	op5	**op7**	**op8**	**op10**	**op19**	**op20**
3	op14	op15	op18	op21	op22	

(b) Schedule for the hyperblock in Figure 6.2(c).

Figure 6.3: Comparison of the static schedules for the *wc* hyperblock before and after logic minimization.

from which the set of register values yielding a "TRUE" evaluation for any member condition can be composed by union [48]. Finally, the relationships between the comparisons are described in BDD form using a finite domain technique [44]. Various elements of the optimizer query this BDD to determine the inherent relationships between conditions, which are the decision network's input variables.

The overall effectiveness of the program decision logic minimization process on the *wc* example is best shown by comparing the schedules of the code before and after optimization. For illustration purposes, a six-issue processor with no restrictions on the combination of instructions that may simultaneously be issued is assumed. Furthermore, all instructions are assumed to have a latency of one cycle. Figure 6.3 presents the schedules for the example hyperblock before and after optimization. The instructions

in bold correspond to the predicate defines in each hyperblock. The schedule for the pre-optimization hyperblock (Figure 6.3(a)) is relatively sparse due to the sequentiality of the predicate defines. The overall schedule length is eight cycles. The schedule after logic minimization is reduced by a factor of two. The chain of predicate define instructions in the original hyperblock is replaced by a parallel, more efficient computation in the optimized hyperblock. The reformulated hyperblock requires only a single level of predicate defines to compute the essential predicates as opposed to the five-level network used in the original code, yielding a significant increase in performance.

6.2 Minimization of Program Decision Logic

The previous section provided an overview of the process of program control height minimization through the optimization of the predicate define network. This section describes in detail the mechanisms by which the predicate define optimizer generates new predicate define instructions to evaluate more efficiently the program's essential predicate functions. The discussion in this section assumes that the program's decision logic has been represented by the predicate BDD and the condition BDD, and that sum-of-products (SOP) expressions for the essential predicates have been extracted as described in the previous section. Once the program decision logic has been extracted, program control is optimized and reexpressed in three steps. First, SOP expressions are optimized using condition analysis and traditional Boolean logic minimization techniques. The resulting optimized expressions are then optionally factorized based on condition availability times and resource constraints. Finally, program control is reexpressed in

predicate define instructions, either in a two-level network or in a multilevel network, depending on whether or not factorization was performed.

The generation of an efficient predicate define network begins with the extraction and subsequent optimization of the SOP for the predicate functions. Figure 6.4(b) shows the expressions extracted for the essential predicates in the *wc* example, as well as the conditions to which the variables in the expressions correspond. Figure 6.4(a) shows the original predicate define network for reference. Because the control expressions are completely represented by the predicate BDD in terms of conditions, the nonessential predicates are eliminated from consideration. This process maps the predicate define structure, in this case five stages of predicate define instructions, into a SOP which can be synthesized into a two-cycle sequence of predicate define instructions. However, this expression can exhibit a large number of redundant and constant-FALSE products, and must be refined before use in define regeneration. From Figure 6.4(b), two-level regeneration of the unoptimized expressions of the *wc* example would require 13 predicate defines in the first level and six in the second, far more than the seven required in the initial network.

6.2.1 Optimization of predicate expressions

Predicate expressions are optimized in two steps. First, expressions are reduced using condition BDD information. For example, conditions that imply or exclude each other (i.e., (r1 < 4) *implies* (r1 < 5) and *excludes* (r1 >= 7)), can cause predicate expressions to contain redundant or constant-FALSE products, as well as redundant literals in useful

p4_ot, p1_uf = (32>=r4)		
p4_ot, p2_uf = (r4>=127) <p1>		
p3_ut = (r2 == 0) <p2>	p5_of, p6_uf = (r4 != 10) <p4>	p7_ut = (r4 != 10) <p4>
p5_of, p8_ut = (r4 != 32) <p7>		
p5_of = (r4 != 9) <p8>		

(a) Original predicate define structure.

C_0	(32>=r4)
C_1	(r4>=127)
C_2	(r2==0)
C_3	(r4!=10)
C_4	(r4!=32)
C_5	(r4!=9)

$p3$	$\overline{C}_0\overline{C}_1C_2$
$p6$	$C_0\overline{C}_3+\overline{C}_0C_1\overline{C}_3$
$p5$	$C_0\overline{C}_3+\overline{C}_0C_1\overline{C}_3+$ $C_0C_3\overline{C}_4+\overline{C}_0C_1C_3\overline{C}_4+$ $C_0C_3C_4\overline{C}_5+\overline{C}_0C_1C_3C_4\overline{C}_5$

(b) Conditions and original predicate expressions.

$p3$	$\overline{C}_0\overline{C}_1C_2$
$p6$	$\overline{C}_3$
$p5$	$\overline{C}_3+\overline{C}_4+\overline{C}_5$

(c) Optimized predicate expressions.

p3_af = (32 >= r4)	p3_af = (r4 >= 127)	p3_at = (r2 == 0)	...
...	p5_of, p6_uf = (r4 != 10)	p5_of = (r4 != 32)	p5_of = (r4 != 9)

(d) Optimized predicate define structure.

Figure 6.4: Example: optimization of *wc* predicate network.

products. These extraneous features are removed in this phase. One such case from the benchmark *wc* was examined in Section 6.1.

Once redundant and constant-FALSE products and literals have been removed from the predicate expressions, the general Boolean logic optimization program *espresso* is used to generate a minimal SOP implementation.

Figure 6.4(c) shows the expressions to which the essential predicates of the *wc* example are reduced in the logic optimization phase. These expressions are both less complex and more parallel than the original functions.

6.2.2 Two-level predicate synthesis

Following optimization of the predicate expressions, the control logic can be synthesized most intuitively as a two-level predicate define network which directly evaluates the minimized SOP expression. In this approach, two levels of predicate define instructions are used for each predicate. The first level consists of and-type predicate defines of the form $p_i_at = C_i\langle T\rangle$, where one predicate p_i is defined for each product term in the predicate expression, and T is the TRUE predicate, which always has the value 1. The second level consists of or-type predicate defines of the form $p_j_ot = (condT)\langle p_i\rangle$, where there is one such predicate define for each product (p_i) and $condT$ is an invariant TRUE condition (e.g., $(0 == 0)$). Thus, a predicate expression having L literals and M products consumes $M + 1$ predicates and performs $L + M$ predicate assignments.

Continuing the *wc* example in Figure 6.4(d), note that the two special cases of two-level predicate synthesis occur, in which the computation of functions containing a single product and functions that are disjunctions of single-literal products can be performed in a single cycle. Note also that predicates which have products in common can share intermediate predicates, allowing for some savings through reuse. In most cases, however, two-level synthesis generates an enormous number of predicate define instructions, because redundancy between products is not reduced. Furthermore, since the evaluation of such a predicate define network usually takes at least two cycles after the last condition becomes available (one for the and-level and one for the or-level), the result may also be suboptimal in latency, even when scheduled for infinite issue. Results demonstrating both

these phenomena are presented in Section 6.4. Clearly, a more sophisticated technique is required.

6.2.3 Factorization

In the example of the previous section, the code sample from *wc* exhibited a large ratio of control height to computation height, and the computation was nearly completely dependent on the outcome of the decision mechanisms. Thus, it was important to compress the height of the entire decision structure as much as possible, as any reduction in the decision height improved performance. Furthermore, since the predicate conditions were strongly related, the resulting predicate define structure actually reduced the predicate and predicate define count. In many other situations, however, predicates are based on more independent conditions and the number of predicate define instructions required to generate a two-level network may be quite large. Factorization seeks to use the code's computation or datapath height to hide some portions of the decision latency which are not on the critical path. Thus, the optimizer is free to focus on reducing implementation size rather than delay when implementing these noncritical sections, saving valuable predicate registers and instruction issue resources.

The factored generation method determines how much factoring can be performed at no cost. The availability times of conditions and the time at which predicate values are needed by the computation component drive the factorizer. If parallel computation height, rather than predicate define height, is the critical path through the code segment, then it is beneficial to perform factorization instead of full expression flattening.

To measure the availability times of conditions and the time at which predicate values are needed, a special version of the code is scheduled. This version of the code has all the predicate dependences between predicate defines removed. For each condition, a predicate destination is added for each predicate whose function depends on that condition. In the resultant code, predicate define instructions are placed as early in the schedule as their condition availability will allow. Also, all uses of a predicate are placed as early as possible, but after all the conditions which may be needed to compute it. By extracting the issue time of these predicate defines and predicate uses, the amount of time the new predicate network has to compute predicates without performance penalty is ascertained. This information is then used together with the previously extracted predicate expressions in later stages of optimization.

With factorization, the goal is to form intermediate predicates as the conditions to compute them become available, and then to reuse these intermediate predicates in the computation of the essential predicates. This activity factors the optimized sum-of-products expression or its products so that the resulting define structure may take more cycles, but can reuse more intermediate predicates, thus saving predicate defines and predicate registers.

In certain cases, when resource utilization is very high and predicate functions are very complex, factorization becomes critical for performance. In some cases, generation of code which would optimally generate the predicate results on an infinitely wide machine could actually degrade performance in a real machine due to excessive width. In these situations, an additional factorization preprocessing stage is applied, in which

predicates are selectively factored on subexpressions available in essential predicates generated earlier in the original code. This activity, shown in lines 2 though 8 of *Factorize* in Figure 6.5, has the effect of moderating the restructuring of control in cases where reordering of the predicate expressions would generate a define network too wide for the target architecture.

Figure 6.6 shows an example extracted from the function *cofactor* of the *008.espresso* benchmark. The minimal sum-of-products is computed for each of the final predicates, as shown in Figure 6.6(a). Next, with the help of condition availability and predicate use times from Figure 6.6(a) and 6.6(b), all useful predicates are factorized, and common expressions are shared. Figure 6.6(c) shows the result of this method. This factoring results in the reduction of the number of predicate define instructions from 37 to 13. Furthermore, the useful predicates ($p1$ and $p2$) are available a single cycle after the last condition is evaluated, sooner than would be possible using a two-level synthesis of the predicate expressions, two cycles after the last condition evaluation.

In the direct sum-of-products conversion, the computation of $p1$ and $p2$ begin respectively at cycle 5 and cycle 6, at the availability time of their latest conditions; results are available two cycles later. With the factorization method, however, predicates $p1$ and $p2$ can be evaluated in a single cycle after the availability of C_5 and C_6. Thus, in some cases, the factorization method is able to reduce predicate latency by one cycle compared to the result of the direct sum-of-products conversion.

```
Factorize(func_list, sched)
1  factor_list = Empty_list();
2  FOREACH func_x IN func_list DO
3      FOREACH func_y IN func_list BEFORE func_x DO
4          IF Factor_simplifies(func_y, func_x) THEN
5              IF Resource_constrained(func_x.id) THEN
6                  IF NOT (List_member(func_y, factor_list) THEN
7                      List_insert_last(factor_list, func_y);
8                  func_x = Factor_SOP(func_x, func_y);
9  FOR cycle = sched.min_cycle TO sched.max_cycle DO
10     FOREACH func IN func_list DO
11         FOREACH product IN func DO
12             ready_prod = Ready_product(product, cycle);
13             match_prod = Match_term(ready_prod, factor_list);
14             IF match_prod THEN
15                 ready_factor = match_prod;
16             ELSE
17                 ready_factor = ready_prod;
18                 ready_factor.id = Unique_token();
19                 List_insert(factor_list, ready_factor);
20             Factor_term(product, ready_factor);
21         List_insert_last(factor_list, func_list);
22     Factor_common_disjoint_subexpr(factor_list, func_list);
23 RETURN func_list, factor_list;

Factor_common_disjoint_subexpr(factor_list, func_list)
1  FOREACH func IN func_list DO
2      product_factor_list = Extract_ready_products (func);
3      fact_func = Find_factor(product_factor_list, func);
4      IF fact_func THEN
5          match_fact = Match_factor(fact_func, factor_list);
6          IF NOT (match_fact) THEN
7              fact_func.id = Unique_token();
8              List_insert(factor_list, fact_func);
9              match_fact = fact_func;
10         Factor_term(func, match_fact);
```

Figure 6.5: Pseudo-code for performing optimization of predicate expressions

Pred	Expression	Use Cycle
$p1$	$C_0C_2C_4\overline{C}_5+$ $C_0C_2C_3\overline{C}_5+$ $C_0C_1\overline{C}_5$	6
$p2$	$C_0C_2C_4\overline{C}_5C_6+$ $C_0C_2C_3\overline{C}_5C_6+$ $C_0C_1\overline{C}_5C_6$	7

(a) Optimized predicate expressions.

C_0	C_1	C_2	C_3	C_4	C_5	C_6
1	1	2	3	4	5	6

(b) Condition availability.

Time	Predicate expression
1	$p3_ut = C_0$ $p4_at = C_0$ $p4_at = C_1$
2	$p5_ut = C_2$ $p6_ut = C_2 \ \langle p3 \rangle$
3	$p7_ut = C_3 \ \langle p6 \rangle$
4	$p8_ut = C_4 \ \langle p6 \rangle$
5	$\mathbf{p1}_of = C_5 \ \langle p7 \rangle$ $\mathbf{p1}_of = C_5 \ \langle p8 \rangle$ $\mathbf{p1}_of = C_5 \ \langle p4 \rangle$
6	$\mathbf{p2}_ut = C_6 \ \langle p1 \rangle$

(c) Factoring with schedule time information.

Figure 6.6: Factorized predicate define optimization.

6.3 Architectural Support for Synthesis

Description of the predicate optimization in previous sections has disregarded the means by which Boolean expressions are converted back into predicate defining instructions. This section examines the instruction set considerations that evolved in supporting an effective predicate synthesis system.

Implementation of two-level predicate synthesis is straightforward in the HPL Playdoh predicate architecture. For example, in Figures 6.2 and 6.4(c), a simple sum-of-products expression is converted into a small set of predicate defines.

Synthesis of multilevel factored functions is not as simple as product-of-sums or sum-of-products expressions, but yields significant improvements in both performance and predicate define count. When an expression is factored out of one or more predicate expressions, its value is computed and stored in a predicate for later use. After factoring, expressions to be synthesized thus contain predicates as well as conditions. To illustrate the use of factoring, the example in Figure 6.7 is presented. In Figure 6.7(a), predicate $p1$ is a subexpression of $p2$. Factoring $C_1 + C_2$, or $p1$, out of $p2$ allows more sharing of predicate defines between predicate computations. As can be seen in Figure 6.7(b), this subexpression can be computed in cycle 1 using or-type predicate defines. The availability of this expression before the computation of $p2$ allow an efficient application of factorization. In cycle 3, the conjunction of the subexpression stored in $p1$ with the previous value of $p2$ and C_3 is required. This expression is awkward to compute using the PlayDoh predicate define semantics because the logical combination of predicates is not directly supported. With the extension to the PlayDoh predicate define semantics, this

Pred	Expression	Use Cycle
$p1$	$C_1 + C_2$	3
$p2$	$C_0C_1C_3 + C_0C_2C_3$	4

(a) Optimized predicate expressions.

C_0	C_1	C_2	C_3
1	2	2	3

(b) Condition availability.

Time	Predicate expression
1	$p2_\wedge t = C_0$
2	$p1_ot = C_1$ $p1_ot = C_2$
3	$p2_\wedge t = C_3 \langle p1 \rangle$

(c) Factorization with conjunctive-type predicate defines.

Time	Predicate expression
1	$p3_at = C_0$ $p4_at = C_0$
2	$p1_ot = C_1$ $p1_ot = C_2$ $p3_at = C_1$ $p4_at = C_2$
3	$p3_at = C_3$ $p4_at = C_3$
4	$p2_ot = TRUE \langle p3 \rangle$ $p2_ot = TRUE \langle p4 \rangle$

(d) No factorization

Time	Predicate expression
1	$p2_at = C_0$
2	$p1_ot = C_1$ $p1_ot = C_2$ $p3_af = C_1$ $p3_af = C_2$
3	$p2_at = C_3$ $p2_af = TRUE \langle p3 \rangle$

(e) Factorization without conjunctive-type predicate defines.

Figure 6.7: Various methods of predicate expresssion regeneration.

expression can be computed with a single conjunctive-type predicate define. Figure 6.7(c) shows the final set of predicate defines used to compute the factored predicate expressions. The two expressions are computed using a total of two predicates and four predicate defines. The last predicate define conjoins $p1$ and C_3 to the previous contents of $p2$ (C_0) to finish the computation of the $p2$ expression.

The primary use of the conjunctive-type predicate defines is to reduce the number of instructions required to compute factored expressions. This reduction is best illustrated when the generation of the predicate expressions is done without the conjunctive type. Figures 6.7(d) and 6.7(e) show two generation options that do not use the conjunctive

type. In Figure 6.7(d), no factorization is performed and the direct sum-of-products expressions are computed. This approach requires a total of ten predicate defines, six more instructions than was required in Figure 6.7(c). Further, the two-level nature of the sum-of-products generation adds an extra level of dependence height. In Figure 6.7(e), factorization is performed, but the conjunctive-type is not used. Here, a total of seven predicate defines, three extra instructions, is necessary. Of these, two predicate defines are needed to compute the complement of the factored expression. This is done by applying DeMorgan's theorem. Another method of complementing $p1$ could have been used, but it would have cost a cycle of latency. The third extra predicate define is used to nullify $p2$ if the complement of the factored predicate is TRUE. Note that the disjunctive-type predicate defines are analogously useful when product-of-sums expressions are used.

6.4 Evaluation

The effectiveness of the Boolean minimization techniques for generating predicated code are evaluated in this section. These techniques have been implemented within the IMPACT experimental compiler framework and applied to a set of benchmarks.

6.4.1 Processor model and benchmarks

The processor modeled is an eight-issue processor with in-order execution and register interlocking. The processor has no limitation on the combination of instructions that may be issued each cycle, except that only one branch may be executed per cycle. The instruction latencies assumed match those of the HP PA-7100 microprocessor. The instruction set contains a set of nontrapping versions of all potentially excepting

instructions, with the exception of branch and store instructions, to support aggressive speculative execution. The instruction set also contains support for predicated execution as described in Chapter 2.

The execution time for each benchmark was obtained using the IMPACT emulation-driven simulator. Some dynamic effects such as branch mispredictions, cache misses, and TLB misses were not measured. This decision was made to ensure that the experimental results highlight the effects of the techniques being evaluated. Since the reformulation of the predicate decision logic does not affect the basic nature of memory access patterns and branch histories, any change in these dynamic effects between the original and optimized codes would be spurious in nature.

The benchmarks used in this experiment consist of 13 nonnumeric programs: four of the SPECINT 92 benchmarks, *008.espresso*, *022.li*, *026.compress*, *072.sc*; six of the SPECINT 95 benchmarks, *099.go*, *124.m88ksim*, *126.gcc*, *129.compress*, *130.li*, *132.ijpeg*; and three Unix utilities, *cccp*, *lex*, *wc*.

6.4.2 Results

The first set of results presented compare the performance of a code set transformed with the described techniques to the performance of a baseline code set. The baseline code consists of the best code generated by the IMPACT compiler (version 990123-I) for a predicated architecture using hyperblock compilation techniques. The transformed code corresponds to the baseline hyperblock code after Boolean minimization techniques are used to restructure the predicate defines, and after the code is rescheduled. Performance is derived by computing the ratio of the execution cycle count for the baseline code to

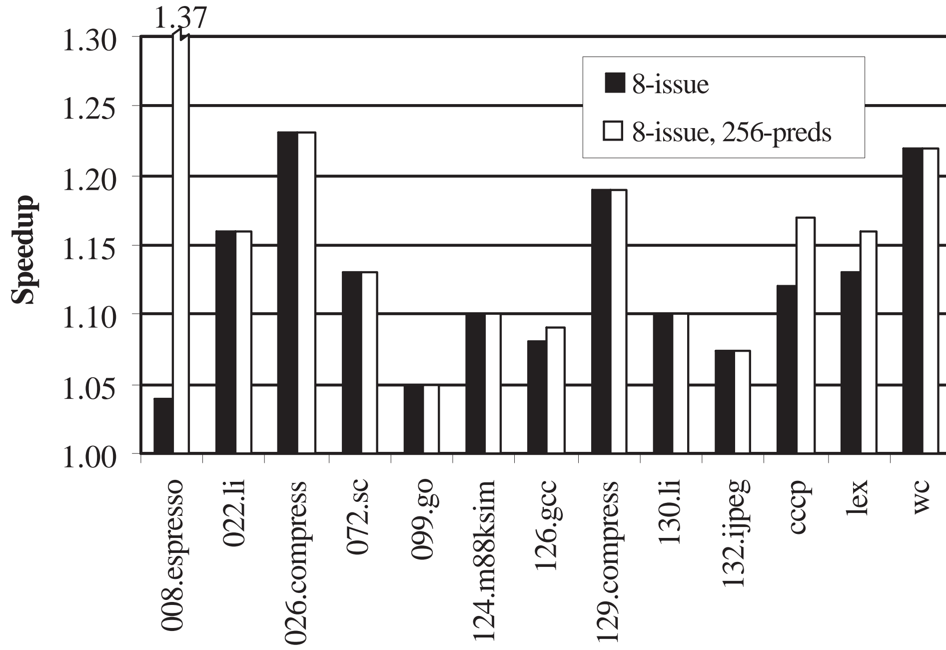


Figure 6.8: Speedup from minimization of program decision logic.

that of the transformed code. The performance is examined at two levels, first at the overall benchmark level and then at the function level.

The overall benchmark speedups are presented in Figure 6.8. For each benchmark, two results are reported. The first is the benchmark speedup on the target architecture. The unweighted average speedup for all the benchmarks is 1.13. For some benchmarks, such as *022.li*, *026.compress*, *129.compress*, and *wc*, the program decision height was significantly limiting performance throughout the most frequently executed portions of the code; when this height is reduced by these techniques, speedups of around 1.2 are achieved.

The second result presented for each benchmark, labeled "8-issue, 256-preds," is the speedup on a hypothetical machine capable of issuing eight non-predicate-define instructions and up to 256 predicate defines per cycle. The significance of the second set of

numbers is that they reflect only the dependence height of predicate defines, while eliminating their resource consumption characteristics. These results suggest a logical upper bound for gains possible with more effective factorization techniques. In most benchmarks, the optimizer produced a number of predicate defines that was appropriate for the schedule and machine model. However, in four benchmarks, *008.espresso*, *cccp*, *126.gcc*, and *lex*, the optimizer was unable to balance height reduction with resource consumption and performance was penalized. This effect was very dramatic in *008.espresso* because it is very decision height limited. Unfortunately, the excessive optimization opportunity available in *008.espresso* allowed the current minimization heuristic to be overly aggressive in reducing height. With more advanced factorization techniques, the number of predicate defines could be reduced in these instances, more closely approximating the "8-issue, 256-preds" results.

Overall, the full benchmark results are encouraging. In most cases, the benefit of these technique was limited solely by the bottleneck created by program computation height. During experimental exploration, I observed that as optimizations targeting computation height were improved, the decision logic became dominant, and relative speedups improved. In particular, data and memory dependences seemed to hide much of the program decision height reduction in many important hyperblocks. As the various components of compiler technology mature, the overall effectiveness of Boolean minimization will improve, as will the importance of an intelligent height versus count selector in the factorizer.

Table 6.1: Speedup and predicate define count for selected functions.

Benchmark, Function	Original Pred. Defines	Two-Level Synthesis Pred. Defines	Two-Level Synthesis (∞)	Two-Level Synthesis (8)	Factored Synthesis Pred. Defines	Factored Synthesis (∞)	Factored Synthesis (8)
008.espresso, essen_parts	39	1293	1.29	0.39	49	1.24	1.16
022.li, xleval	48	485	1.07	0.66	80	1.10	1.10
022.li, mark	42	67	1.48	1.48	53	1.50	1.48
026.compress, compress	60	456	1.20	1.03	221	1.23	1.23
072.sc, update	141	240	1.15	1.15	159	1.23	1.23
099.go, getefflibs	98	1083	1.06	0.98	204	1.07	1.07
124.m88ksim, execute	41	47	1.12	1.12	40	1.12	1.12
124.m88ksim, goexec	176	175	1.10	1.09	155	1.09	1.08
124.m88ksim, load_data	42	54	1.30	1.30	53	1.30	1.30
124.m88ksim, loadmem	84	88	1.13	1.13	84	1.13	1.13
126.gcc, invalidate	89	202	1.27	1.24	125	1.22	1.21
126.gcc, flow_analysis	64	92	1.77	1.69	58	1.86	1.86
126.gcc, canon_hash	89	149	1.88	1.20	116	1.90	1.74
129.compress, compress	63	154	1.21	1.21	98	1.26	1.26
130.li, mark	55	148	1.15	1.14	101	1.19	1.19
132.ijpeg, forward_DCT	31	47	1.46	1.35	32	1.46	1.43
cccp, skip_if_group	157	208	1.23	1.05	190	1.32	1.24
lex, cgoto	236	330	1.31	1.10	260	1.18	1.14
wc, main	56	48	1.22	1.31	48	1.22	1.22

To better understand the effect program decision logic minimization has on complete programs, I measured the performance and code size characteristics of a number of selected functions. Table 6.1 examines the performance of one or more functions from each of the benchmarks. These functions were chosen based on two criteria: significant program execution time and potential for optimization (e.g., the control height was significant relative to the computation height). The table compares the effectiveness of two strategies for program logic transformation: two-level predicate synthesis and factorization. For each strategy, the static number of predicate define instructions, the performance gain on an eight-issue processor with unconstrained predicate define resources (∞), and the performance gain on the eight-issue processor are reported. In addition, the static number of predicate define instructions in the code before minimization is reported.

From the table, the two-level synthesis approach shows mixed results. For the unconstrained machine, the reduction in height translates directly into large speedups. However, the unconstrained performance does not always translate into the same performance gain on the eight-issue processor. This is most pronounced in *008.espresso, essen_parts* where the 1.16 speedup is sharply reduced to 0.39. The primary reason for this behavior is the large increase in the number of predicate define instructions, which oversaturates processor resources and results in loss of performance. Correspondingly, when the number of predicate defines is not increased by a large amount, the unconstrained performance does indeed translate directly into performance on the eight-issue processor. Clearly, factored synthesis is necessary for successful optimization of program decision logic.

As shown in the table, the factored approach yields both larger and more consistent speedups. Both methods reduce the predicate computation height, but the factored approach dramatically reduces the number of predicate defines required for the optimization. The function *126.gcc, canon_hash* provides a good example of this behavior. Both methods achieve good speedup for the unconstrained processor. However, the two-level synthesis approach requires 149 predicate defines to accomplish the improvement. For the eight-issue processor, most of the performance gain is lost due to this increase in instructions. The factored approach reduces the number of predicate defines to 116, increasing the eight-issue speedup to 1.74. The number of predicate defines is still more than the original 89. Note, however, that simply increasing the number of predicate defines from the original code is not necessarily viewed as a negative. Boolean minimization approaches do this systematically to improve performance by identifying

condition subexpressions that can be computed early. This allows the final predicate to be made available as soon as possible after the final condition is ready. However, the factored approach is consistently more effective because it factors predicate expressions into multiple-level structures which are less demanding of processor resources than two-cycle evaluations. Another interesting result is that for some functions such as *update* from *072.sc* the factored synthesis method outperforms the two-level method, even at infinite issue. This is due to the ability of the factorizer to generate expressions in one cycle rather than the two usually required by the two-level synthesis approach.

6.5 Significance

The PDLO is a new method for optimizing programmatic control flow. It provides a systematic methodology for reformulating program control flow for more efficient execution on ILP processors. Control expressed through branches and predicate defines is extracted and represented as a *program decision logic network*. Boolean minimization techniques are applied to the network both to reduce dependence height and to simplify the component expressions. Redundancy is controlled by employing a schedule-sensitive factorization technique to identify intermediate logical combinations of conditions that can be shared. After optimization, the network is reformulated into predicated code.

As compiler technology progresses to make use of predication more extensive and effective, minimization of program decision logic is likely to become an increasingly more important part of total program optimization.

7. PARTIAL REVERSE IF-CONVERSION

As discussed in Chapter 3, an effective compiler strategy for supporting predicated execution must address the “what” and “when” questions of if-conversion. This chapter presents the mechanism called *partial reverse if-conversion* that operates at schedule time to balance the amount of control flow and predication present in the generated code, based on the characteristics of the target processor [38]. The Partial Reverse If-Conversion Framework enables the compiler to extract the full benefits of the predicated representation by applying aggressive if-conversion early in the compilation procedure, while at schedule time reducing the or eliminating the effect of overaggressive hyperblock formation.

7.1 Partial Reverse If-Conversion

The partial reverse if-conversion process consists of three components: analysis, transformation, and decision. These components are integrated into an instruction scheduler in the Partial Reverse If-Conversion Framework. This section consists of a description

of these components and their integration into a hyperblock scheduler. The section concludes with a realistic code example to illustrate the operation of the framework.

7.1.1 Analysis

Before any manipulation or analysis of execution paths can be performed, these paths must be identified in the predicated code. Execution paths in predicated code are referred to as *predicate paths*. Immediately after hyperblock formation, the structure of the predicate paths is identical to the control flow graph of the code before hyperblock formation. The structure of the predicate paths can be represented in a form called the *predicate flow graph* (PFG) as described in Chapter 5. Recall that the predicate flow graph is simply a control flow graph (CFG) in which predicate execution paths are also represented.

Figure 7.1 shows a predicated code segment and its PFG. The PFG shown in Figure 7.1(b) is created in the following manner. The first instruction in Figure 7.1(a) is a predicate definition. At this definition, $p1$ can assume TRUE or FALSE. A path is created for each of these possibilities. The complement of $p1$, $p2$, shares these paths because it does not independently create new conditional outcomes. The predicate define instruction 2 also creates another path. In this case, the predicates $p3$ and $p4$ can only be TRUE if $p1$ is TRUE because their defining instructions is predicated on $p1$; therefore, only one more path is created. The creation of paths is determined by the interrelations of predicates, which are provided by mechanisms addressed in other work [19], [41]. For the rest of the instructions, the paths that contain these instructions are determined by the predicate guarding their execution. For example, instruction 3 is based on predicate

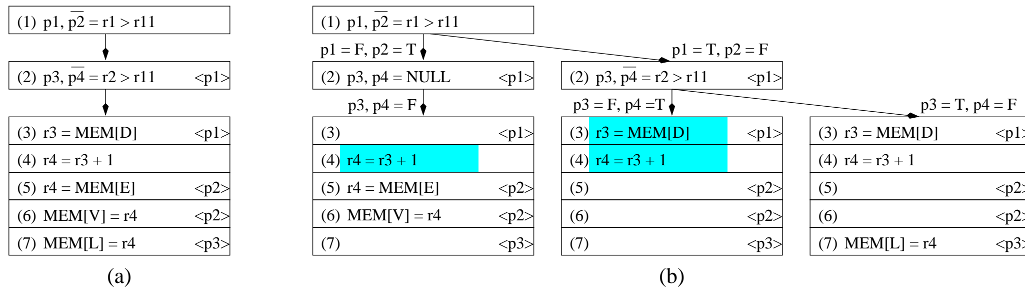


Figure 7.1: Predicate flow graph (b) for predicated code segment (a). Partial dead code elimination is applied, given that $r3$ and $r4$ are not live out of this region.

$p1$ and is therefore only placed in paths where $p1$ is TRUE. Instruction 4 is not predicated and therefore exists in all paths. The type of predicate defines used in all figures in this chapter are unconditional, meaning they always write a value [20]. Because they write some value regardless of their predicate, their source predicates can be ignored, and the instruction's destinations must be placed in all paths.

Paths in a PFG can be merged when a predicate is no longer used and does not affect any other predicate later in the code. However, this merging of paths may not be sufficient to solve all potential path explosion problems in the PFG. This is because the number of paths in a PFG grows exponentially with respect to the number of independent predicates whose live ranges overlap. Fortunately, in practice this never becomes prohibitively expensive except during code scheduling. After code scheduling, a complete PFG will have a large number of paths and may be costly. How the partial reverse if-converter overcomes this problem is described in Chapter 5.

With a PFG, the compiler has the information necessary to know which instructions exist in which paths. In Figure 7.1, if the path in which $p1$ and $p3$ are TRUE is to be extracted, the instructions which would be placed into this path would be 3, 4, and 7.

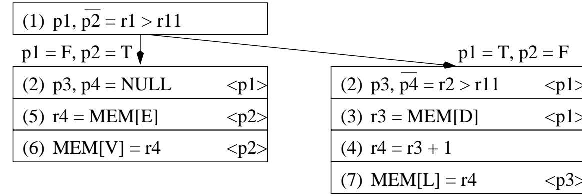
The instructions that remain in the other two paths seem to be 3, 4, 5, and 6. However, inspection of the dataflow characteristics of these remaining paths reveals that the results of instructions 3 and 4 are not used, given that $r3$ and $r4$ are not live out of this region. This fact makes these instructions dead code in the context of these paths. Performing traditional dead code removal on the PFG, instead of the CFG, determines which parts of these operations are dead. Because this application of dead code removal indicates that these instructions are dead only under certain predicate conditions, this process is termed *predicate partial dead code removal* and is described in Chapter 5. The result of partial dead code removal indicates that instructions 3 and 4 would generate correct code and would not execute unnecessarily if they were predicated on $p3$.

At this point, all paths have been identified, and unnecessary code has been removed by partial dead code removal. The analysis and possible ejection of these paths now becomes feasible.

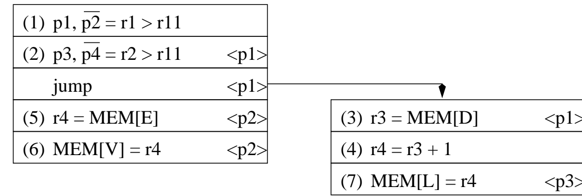
7.1.2 Transformation

Once predicate analysis and partial dead code elimination have been completed, performing reverse if-conversion at any point and for any predicate requires a small amount of additional processing. This processing determines whether each instruction belongs in the original hyperblock, the new block formed by reverse if-conversion, or both. Figure 7.2 is used to aid this discussion.

The partial reverse if-converted code can be subdivided into three code segments. These are: the code before the reverse if-converting branch, the code ejected from the hyperblock by reverse if-conversion, and the code which remains in the hyperblock below



(a)



(b)

Figure 7.2: Predicate flow graph (a) and a partial reverse if-conversion of predicate $p1$ located after instructions 1 and 2 (b).

the reverse if-converting branch. Instructions before the location of the partial reverse if-converting branch are left untouched in the hyperblock. Figure 7.2(b) shows the partial reverse if-conversion created for $p1$ after instructions 1 and 2. This means that instructions 1 and 2 are left in their originally scheduled locations, and the reverse if-converting branch, predicated on $p1$, is scheduled immediately following them. The location of instructions after the branch is determined by the PFG. To use the PFG without experiencing a path explosion problem, the PFGs generated during scheduling are done only with respect to the predicate being reverse if-converted. This keeps the number of paths under control since the single-predicate PFG can contain no more than two paths. Figure 7.2(a) shows the PFG created for the predicate to be reverse if-converted, $p1$. Note that the partial dead code has already been removed as described in the previous section. Instructions that exist solely in the "$p1$ is FALSE" path, such as 5 and 6, remain in the original block. Instructions that exist solely in the "$p1$ is TRUE" path, such as 3,

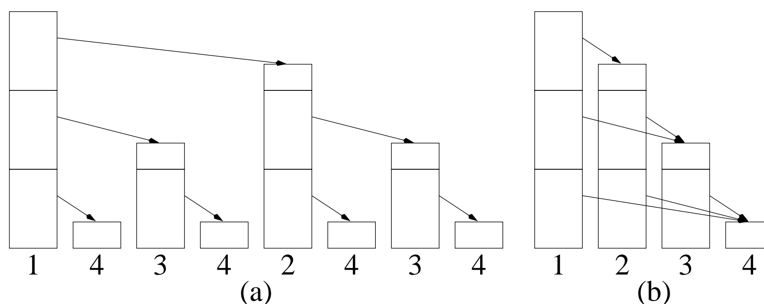


Figure 7.3: Simple code size reduction on multiple partial reverse if-conversions applied to an unrolled loop. Each square represents an unroll of the original loop.

4, and 7, are moved from the original block to the newly formed region. An instruction which exists in both paths must be placed in both regions.

Notice that the hyperblock conditionally jumps to the code removed from the hyperblock, but there is no branch from this code back into the original hyperblock. While this is possible, it was not implemented in this work. Branching back into the hyperblock would violate the hyperblock semantics because it would no longer be a single entry region. Violating hyperblock semantics may not be problematic since the benefits of the hyperblock have already been realized by the optimizer and prepass scheduler. However, the postpass hyperblock scheduler may experience reduced scheduling freedom since each side entry into the hyperblock effectively divides the region into two smaller hyperblocks.

The advantage of branching back into the original hyperblock is a significant reduction in code size through elimination of unnecessarily duplicated instructions. However, as will be shown in the experimental section, code size was generally not a problem. One code size optimization that was performed merges targets of partial reverse if-conversion branches if the target blocks are identical. This resulted in a significant code size reduction in codes where loop unrolling was performed. If a loop in an unrolled hyperblock

needed to be reverse if-converted, it is likely that all iterations needed to be reverse if-converted. This creates many identical copies of the loop body subsequent to the loop being reverse if-converted. Figure 7.3(a) shows the original result of repeated reverse if-conversions on an unrolled loop. Figure 7.3(b) shows the result obtained by combining identical targets. While this simple method works well in reducing code growth, it does not eliminate all unnecessary code growth. Eliminating all unnecessary code growth would require a mechanism capable of inserting a reentry target at an opportune location in the hyperblock.

7.1.3 Policy

After creating the predicate flow graph and removing partial dead code, the identity and characteristics of all paths in a hyperblock are known. With this information, the compiler can make decisions on which transformations to perform. The decision process for partial reverse if-conversion consists of two parts: deciding which predicates to reverse if-convert and deciding where to reverse if-convert the selected predicates. To determine the optimal reverse if-conversion for a given architecture, the compiler could exhaustively try every possible reverse if-conversion, compute the optimal cycle count for each possibility, and choose the one with the best performance. Unfortunately, there are an enormous number of possible reverse if-conversions for any given hyperblock. Consider a hyperblock with p predicates and n instructions. This hyperblock has 2^p combinations of predicates that could be selected for reverse if-conversion. Each of these reverse if-conversions can then locate its branch in up to n locations in the worst case. Given that each of these possibilities must be scheduled to measure its cycle count, this can

be prohibitively expensive. Obviously, a heuristic is needed. While many heuristics may perform effective reverse if-conversions, only one is studied in this chapter. This heuristic may not be the best solution in all applications, but for the machine models studied in this work, it achieves a desirable balance between final code performance, implementation complexity, and compile time.

The process of choosing a heuristic to perform partial reverse if-conversion is affected greatly by the type of scheduler used. Because partial reverse if-conversion is integrated into the prepass scheduler, the type of information provided by the scheduler and the structure of the code at various points in the scheduling process must be matched with the decision of what and where to reverse if-convert. An operation-based scheduler may yield one type of heuristic, and a list scheduler may yield another. The policy presented here was designed to work within the context of an existing list scheduler described in [57] and [58]. This list scheduler performs a cycle-by-cycle greedy schedule using an instruction priority function. The partial reverse if-conversion algorithm as integrated in this scheduler is shown in Figure 7.4.

The first decision addressed by the proposed heuristic is where to place a predicate selected for reverse if-conversion. If one location can be shown to be generally more effective than the rest, then the number of locations to be considered for each reverse if-conversion can be reduced from n to 1, an obvious improvement. Such a location exists under the assumption that the reverse if-converting branch consumes no resources and the code is scheduled by a perfect scheduler. It can be shown that there is no better placement than the first cycle in which the value of the predicate to be reverse if-converted

```
1 Initialize ready_priority_queue;
2 ric_queue = NULL;
3 cycle = 0;
4 num_unsched = Number of operations;
5 sched_no_ric = Compute_dynamic_cycles_for_hyperblock;
  // Each trip through this loop is a new cycle
6 WHILE num_unsched != 0 DO
    // Handle reverse if-converting branches first
7   FOREACH ric_op IN ric_queue DO
8     IF Schedule_Op(ric_op, cycle) THEN
9       Compute location for each unscheduled op;
10      sched_ric_taken = Compute_dynamic_cycles_in_ric_taken_path;
11      sched_ric_hb = Compute_dynamic_cycles_in_ric_hyperblock;
12      mipred_ric = Estimate_ric_mispreds * miss_penalty;
13      ric_cycles = sched_ric_hb + sched_ric_taken;
14      ric_cycles = ric_cycles + mispred_ric;
15      IF (sched_no_ric > ric_cycles) THEN
16        sched_no_ric = sched_ric_hb;
17        Place all ops in their no_ric schedule location;
18      ELSE
19        Unschedule_OP(ric_op);
20      Remove ric_op from ric_queue;
    // Then handle regular operations
21  FOREACH regular_op IN ready_priority_queue DO
22    IF Schedule_Op(regular_op, cycle) THEN
23      Remove regular_op from ready_priority_queue;
24      num_unsched = num_unsched - 1;
25      IF Is_Predicate_Define(regular_op) THEN
26        Add reverse if-converting branch to ric_queue;
27  cycle = cycle + 1;
```

Figure 7.4: An algorithm incorporating partial reverse if-conversion into a list scheduler.

is available after its predicate define instruction.[1] Since the insertion of the branch has the same misprediction or taken penalty regardless of its location, these effects do not favor one location over another. However, the location of the reverse if-converting branch does determine how early the paths sharing the same resources are separated and given the full machine bandwidth. The perfect scheduler will always do as well or better when

---

[1]There exist machines where the placement of a branch a number of cycles after the computation of its condition removes all of its mispredictions [59]. In these machines, there are two locations which should be considered, immediately after the predicate define instruction and in the cycle in which the branch mispredictions are eliminated.

the full bandwidth of the machine is divided among fewer instructions. Given this, the earlier the paths can be separated, the fewer the number of instructions competing for the same machine resources. Therefore, a best schedule will occur when the reverse if-converting branch is placed as early as possible.

Despite this fact, placing the the reverse if-converting branch as early as possible is a heuristic. This is because the two assumptions made, that the scheduler is perfect and that there is no cost for the reverse if-converting branch, are in general not valid. It seems reasonable, however, that this heuristic would do well despite these imperfections. Another consideration is code size, since instructions existing on multiple paths must be duplicated when these paths are separated. The code size can be reduced if the reverse if-converting branch is delayed. Depending on the characteristics of the code, this delay may have no cost or a small cost which may be less than the gain obtained by the reduction in code size. Despite these considerations, the placement of the partial reverse if-converting branch as early as possible is a reasonable choice.

The second decision addressed by the heuristic is what to reverse if-convert. Without a heuristic, the number of reverse if-conversions that would need to be considered with the heuristic described above is 2^p. The only way to optimally determine which combination of reverse if-conversions yields the best results is to try them all. A reverse if-conversion of one predicate can affect the effectiveness of other reverse if-conversions. This interaction among predicates is caused by changes in code characteristics after a reverse if-conversion has removed instructions from the hyperblock.

In the context of a list scheduler, a logical heuristic is to consider each potential reverse if-conversion in a cycle-by-cycle fashion, in the order in which the predicate

defines are scheduled. This heuristic is used in the algorithm shown in Figure 7.4. This has the desirable effect of making the reverse if-conversion process fit seamlessly into a list scheduler. It is also desirable because each reverse if-conversion is considered in the context of the decisions made earlier in the scheduling process.

In order to make a decision on each reverse if-conversion, a method of evaluating its profitability must be employed. For each prospective reverse if-conversion, three schedules must be considered: the code schedule without the reverse if-conversion, the code schedule of the hyperblock with the reverse if-converting branch inserted and paths excluded, and the code schedule of the paths excluded by reverse if-conversion. Together they yield a total of $3p$ schedules for a given hyperblock. Each of these three schedules needs to be compared to determine if a reverse if-conversion is profitable. This comparison can be written as: $sched_cycles_{no_ric} > sched_cycles_{ric_hb} + sched_cycles_{ric_taken} + (mispred_{ric} * miss_penalty)$ where $sched_cycles_{no_ric}$ is the number of dynamic cycles in the schedule without reverse if-conversion applied, $sched_cycles_{ric_hb}$ is the number of dynamic cycles in the schedule of the transformed hyperblock, $sched_cycles_{ric_taken}$ is the number of dynamic cycles in the target of the reverse if-conversion, and $mispred_{ric}$ is the number of mispredictions introduced by the reverse if-conversion branch. The $mispred_{ric}$ can be obtained through profiling or static estimates. $miss_penalty$ is the branch misprediction penalty. This comparison is computed by lines 10 through 14 in Figure 7.4.

While the cost savings due to the heuristic is quite significant, $3p$ schedules can still be costly for the more complicated machine models. To reduce this cost, it is possible to reuse information gathered during one schedule in a later schedule.

The first source of reuse is derived from the cycle-by-cycle operation of the list scheduler itself. At the point which each reverse if-conversion is considered, all previous instructions have been scheduled in their final locations by lines 8 or 22 in Figure 7.4. Performing the scheduling on the reverse if-conversion and the original scenario only needs to start at this point. The number of schedules is still $3p$, but the number of instructions in each schedule has been greatly reduced by the removal of instructions already scheduled.

The second source of reuse takes advantage of the fact that, for the case in which the reverse if-conversion is not done, the schedule has already been computed. At the time the previous predicate was considered for reverse if-conversion, the schedule was computed for each outcome. Because the resulting code schedule in cycles is already known, no computation is necessary for the current predicate's $sched_cycles_{no_ric}$. This reuse reduces the total number of schedules computed to $2p + 1$, with each schedule considering only the unscheduled instructions at each point due to the list scheduling effect. This reuse is implemented in Figure 7.4 by lines 5 and 16.

Another way to reduce the total number of instructions scheduled is to take advantage of the fact that the code purged from the block is only different in the "then" and "else" blocks but not in the control equivalent split or join blocks. Once the scheduler has completely scheduled the "then" and "else" parts, no further scheduling is necessary since the remaining schedules are likely to be very similar. The only differences may be dangling latencies or other small differences in the available resources at the boundary. To be more accurate, the schedules can continue until they become identical, which is likely to occur at some point, though is not guaranteed to occur in all cases. An additional use

for the detection of this point is code size reduction. This point is a logical location to branch from the ejected block back into the original hyperblock.

With all of the above schedule reuse and reduction techniques, it can be shown that the number of times an instruction is scheduled is usually $1+2d$, where d is that instruction's depth in its hammock. In the predication domain, this depth is the number of predicates defined in the chain used to compute that instruction's guarding predicate.

If the cost of scheduling is still high, estimates may be used instead. There are many types of scheduling estimates which have been proposed and can be found in the literature [58]. It is possible to create a hybrid scheduler/estimator which may balance good estimates with compile time cost. As mentioned previously, the schedule height of the two paths in the hammock must be obtained. Instead of purely scheduling both paths, which may be costly, or just estimating both paths, which may be inaccurate, a part-schedule and part-estimate approach may obtain more accurate results with lower cost. In the context of a list scheduler, one solution is the following: The scheduler could schedule an initial set of operations and estimate the schedule on those remaining. Accurate results will be obtained for the scheduled portion, in addition, the estimate may be able to benefit from information obtained from the schedule, as the characteristics of the scheduled code may be likely to match the characteristics of the code to be estimated. In the experiments presented in the next section, actual schedules are used in the decision to reverse if-convert because the additional compile time was acceptable.

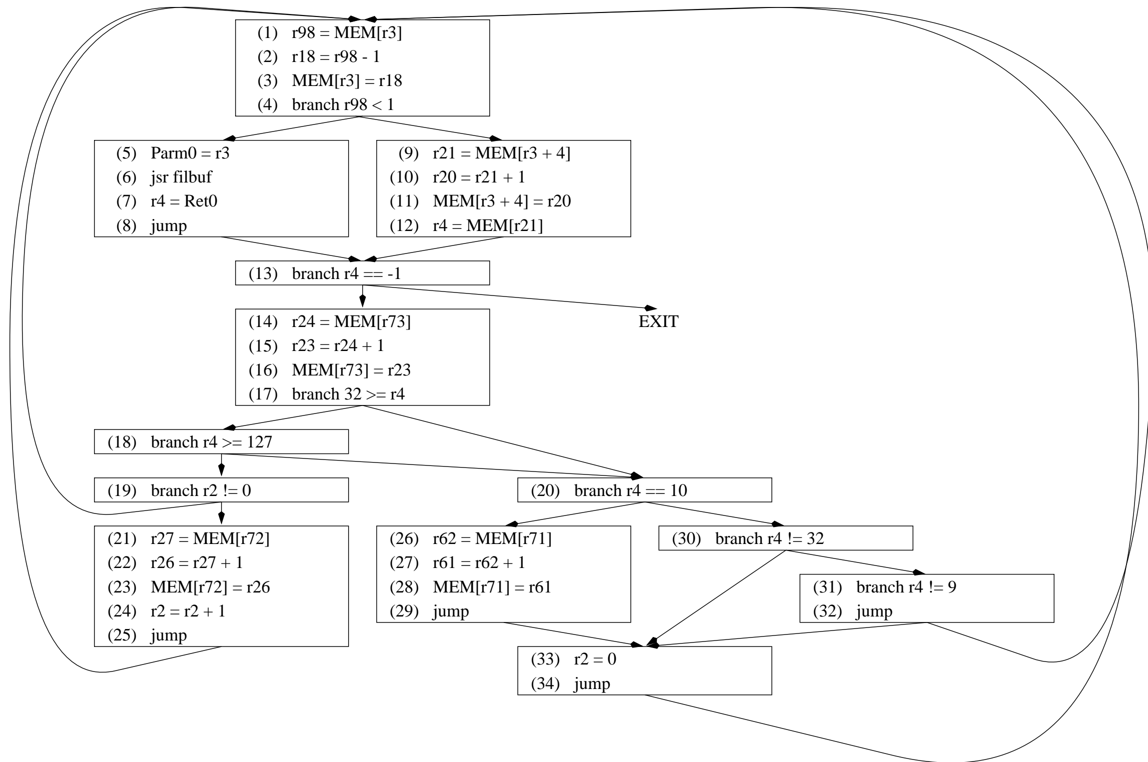


Figure 7.5: The control flow graph from the Unix utility *wc*.

7.1.4 Code example

The examples presented up to this point in this section have been artificially crafted to illustrate the application of partial reverse if-conversion. It is, however, useful to examine the operation of the framework in a more realistic setting. The inner loop from the smallest benchmark considered, the Unix utility *wc*, is chosen for this purpose. Figure 7.5 presents the original control flow graph for this code segment. Instructions are numbered 1 through 34 for reference. For this code segment, there are 22 paths of execution. The hyperblock formation heuristics must consider all of the possible paths to identify those profitable for inclusion in a hyperblock. The interaction between the paths is also important because they share many instructions.

The hyperblock formation heuristics select all instructions except 5-8 to combine into a single hyperblock. The resultant schedule of the hyperblock for an example three-issue processor is shown in Figure 7.6(a). The processor utilized for this example is assumed to consist of one arithmetic unit, one memory unit, and one branch unit. All instructions have a latency of one cycle except loads, which have a latency of two cycles. Using branch profile information, the estimated execution cycle count for the hyperblock is 2.0M cycles. Unfortunately, the original code in which each basic block is scheduled separately requires only 1.7M cycles to execute. The hyperblock formation heuristics were too aggressive, leading to an oversubscription of the processor resources.

The partial reverse if-conversion framework is used to overcome this performance loss. The main problem with the original hyperblock is that the arithmetic unit is over-saturated with instructions. Therefore, the scheduler needs to choose one or more paths to eject to reduce the pressure on the arithmetic unit. The resultant schedule with partial reverse if-conversion is shown in Figure 7.6(b). Two reverse if-conversions are performed. First, predicate $p3$ is reverse if-converted by inserting a new jump instruction ($18'$). The jump conditionally branches to a new hyperblock, labeled B, when $p3$ is TRUE. The new jump is inserted into the schedule of the original hyperblock at the earliest cycle when $p3$ is available, cycle 8. All unscheduled instructions are put into one of three categories: required when $p3$ is TRUE, required when $p3$ is FALSE, or required for either value of $p3$. Instructions 19, 21, 22, 23, and 24 are only needed when $p3$ is TRUE; thus, they are ejected from the original hyperblock. Instructions 15 and 16 are needed for either value of $p3$; thus, they must be replicated in both the original and the new hyperblock. The remainder of the instructions are required when $p3$ is FALSE, so they remain in

A: 1	(1) r98 = MEM[r3]		
2	(9) r21 = MEM[r3 + 4]		
3	(2) r18 = r98 - 1		
4	(3) MEM[r3] = r18	(4) branch r98 < 1	(10) r20 = r21 + 1
5	(11) MEM[r3 + 4] = r20		
6	(12) r4 = MEM[r21]		
7	(14) r24 = MEM[r73]		
8	(13) branch r4 == -1, EXIT	(17) p5(ot), p2(uf) = 32 >= r4	(21) r27 = MEM[r72]
9	(18) p5(ot), p3(uf) = r4 >= 127 <p2>	(26) r62 = MEM[r71]	
10	(19) p4 = 0 == r2 <p3>		
11	(20') p7(ot), p6(uf) = r4 == 10 <p5>		
12	(30) p7(ot), p8(uf) = r4 == 32 <p6>		
13	(15) r23 = r24 + 1		
14	(20) p9 = r4 == 10 <p5>	(16) MEM[r73] = r23	
15	(22) r26 = r27 + 1		
16	(24) r2 = r2 + 1 <p4>	(23) MEM[r72] = r26 <p4>	
17	(27) r61 = r62 + 1		
18	(31) p7(ot) = r4 == 9 <p8>	(28) MEM[r71] = r61 <p9>	
19	(33) r2 = 0 <p7>	jump A	

(a)

A: 1	(1) r98 = MEM[r3]		
2	(9) r21 = MEM[r3 + 4]		
3	(2) r18 = r98 - 1		
4	(12) r4 = MEM[r21]	(10) r20 = r21 + 1	
5	(3) MEM[r3] = r18	(4) branch r98 < 1	
6	(11) MEM[r3 + 4] = r20	(17) p5(ot), p2(uf) = 32 >= r4	(13) branch r4 == -1, EXIT
7	(14) r24 = MEM[r73]	(18) p5(ot), p3(uf) = r4 >= 127 <p2>	
8	(20) p9 = r4 == 10 <p5>	(26) r62 = MEM[r71]	(18') jump B <p3>
9	(20') p7(ot), p6(uf) = r4 == 10 <p5>		
10	(30) p7(ot), p8(uf) = r4 == 32 <p6>		
11	(31) p7(ot) = r4 == 9 <p8>		
12	(15) r23 = r24 + 1		
13	(27) r61 = r62 + 1 <p9>	(16) MEM[r73] = r23	
14	(33) r2 = 0 <p7>	(28) MEM[r71] = r61 <p9>	jump A

B: 1	(19) p7(ot) = r4 == 9		
2	(19') jump C <p4>	(15) r23 = r24 + 1	
3	(16) MEM[r73] = r23	jump A	

C: 1	(21) r27 = MEM[r72]	(15) r23 = r24 + 1	
2	(24) r2 = r2 + 1	(16) MEM[r73] = r23	
3	(22) r26 = r27 + 1		
4	(23) MEM[r72] = r26	jump A	

(b)

Figure 7.6: Schedule for *wc* after hyperblock formation and optimization (a), and schedule after partial reverse if-conversion of that hyperblock (b).

the original hyperblock. The remainder of the original hyperblock is scheduled without further reverse if-conversions.

A second reverse if-conversion is applied when hyperblock B is scheduled. Predicate $p4$ is reverse if-converted by introducing a new jump instruction ($19'$). The jump conditionally branches to the new hyperblock, labeled C, when $p4$ is TRUE. As a result of the transformation, instructions 21, 22, 23, and 24 are ejected from hyperblock B. Instructions 15 and 16 are again replicated. The scheduling process is complete when hyperblock C is scheduled.

The overall result is that execution of the code with partial reverse if-conversion is reduced to 1.3M cycles. This compares favorably with 2.0M cycles for the original hyperblock code and 1.7M cycles for the original basic block code. The primary reason for the improvement in this example was the ability to eject instructions to reduce the contention for the arithmetic unit.

7.2 Experimental Evaluation

The partial reverse if-conversion framework described in this chapter has been implemented in the second generation instruction scheduler of the IMPACT compiler version 971004-R. This section presents an experimental evaluation of this framework.

7.2.1 Methodology

The IMPACT compiler utilizes a machine description file to generate code for a parameterized superscalar processor. To measure the effectiveness of the partial reverse

if-conversion technique, a machine model similar to many current processors was selected. The machine modeled is a four-issue, in-order superscalar processor that contains two integer arithmetic-logic units, two memory ports, one floating point ALU, and one branch unit. The instruction latencies assumed match those of the HP PA-7100 microprocessor [32]. The instruction set contains nontrapping versions of all potentially excepting instructions, with the exception of branch and store instructions, to support aggressive speculative execution. The instruction set also contains support for predication similar to that provided in the PlayDoh architecture [20].

The execution time for each benchmark is derived from the static code schedule weighted by dynamic execution frequencies obtained from profiling. Static branch prediction based on profiling is also utilized. Benchmark performance ignores dynamic stall cycles associated with the memory system including instruction and data cache misses. Previous experience with this method of run time estimation has demonstrated that it accurately estimates simulations of an equivalent machine with perfect caches.

The benchmarks used in this experiment consist of 14 non-numeric programs: the six SPEC CINT92 benchmarks, *008.espresso*, *022.li*, *023.eqntott*, *026.compress*, *072.sc*, and *085.cc1*; two SPEC CINT95 benchmarks, *132.ijpeg* and *134.perl*; and six Unix utilities *cccp*, *cmp*, *eqn*, *grep*, *wc*, and *yacc*.

7.2.2 Results

For the experiments, the performance of the traditional hyperblock and the new partial reverse if-conversion frameworks are compared. The hyperblocks formed in these experiments represent those formed by the IMPACT compiler's hyperblock formation

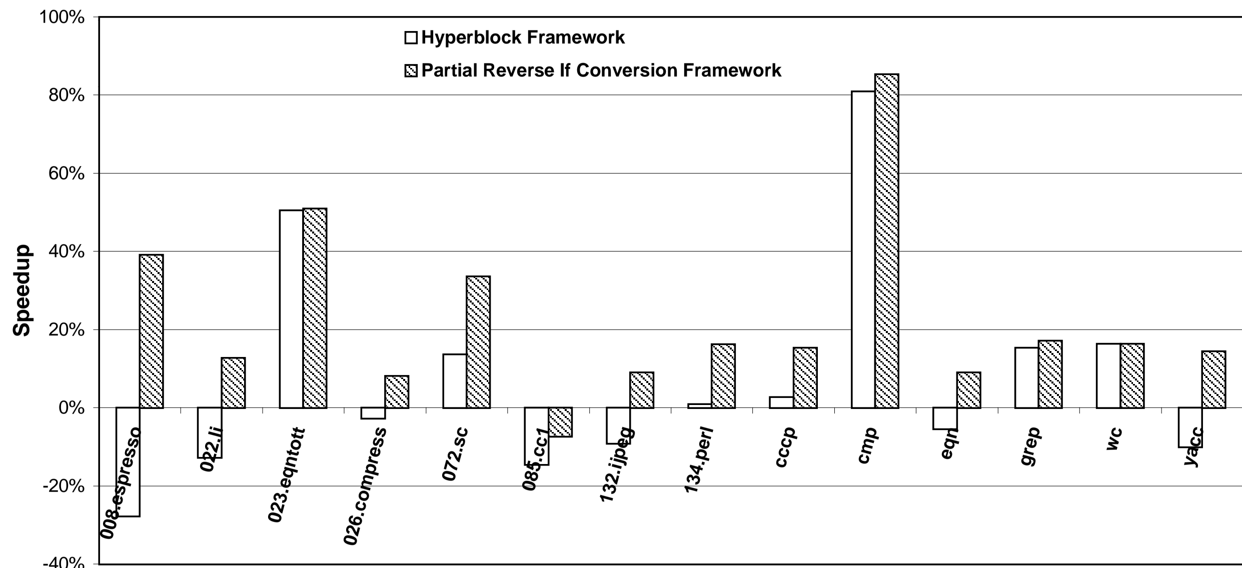


Figure 7.7: Performance increase over superblock exhibited by the hyperblock and partial reverse if-conversion frameworks with no misprediction penalty.

heuristic for the target machine [17]. For the traditional hyperblock framework, these hyperblocks are directly scheduled for the target machine. These same hyperblocks were also used as input to the partial reverse if-conversion framework. In this case, the hyperblocks are scheduled and reverse if-converted as appropriate for the target machine. The performance of both frameworks is presented relative to the results obtained with superblock compilation for the same target machine. Superblock compilation performance is selected as the base because it represents the best possible performance obtainable by the IMPACT Compiler without predication [60].

Overall performance. Figures 7.7 and 7.8 compare the overall benchmark performance of the hyperblock and partial reverse if-conversion frameworks. Performance is reported as the speedup in execution cycles versus superblock compilation.

Figure 7.7 shows the performance of the hyperblock and partial reverse if-conversion frameworks assuming no branch misprediction penalty. Since branch mispredictions are

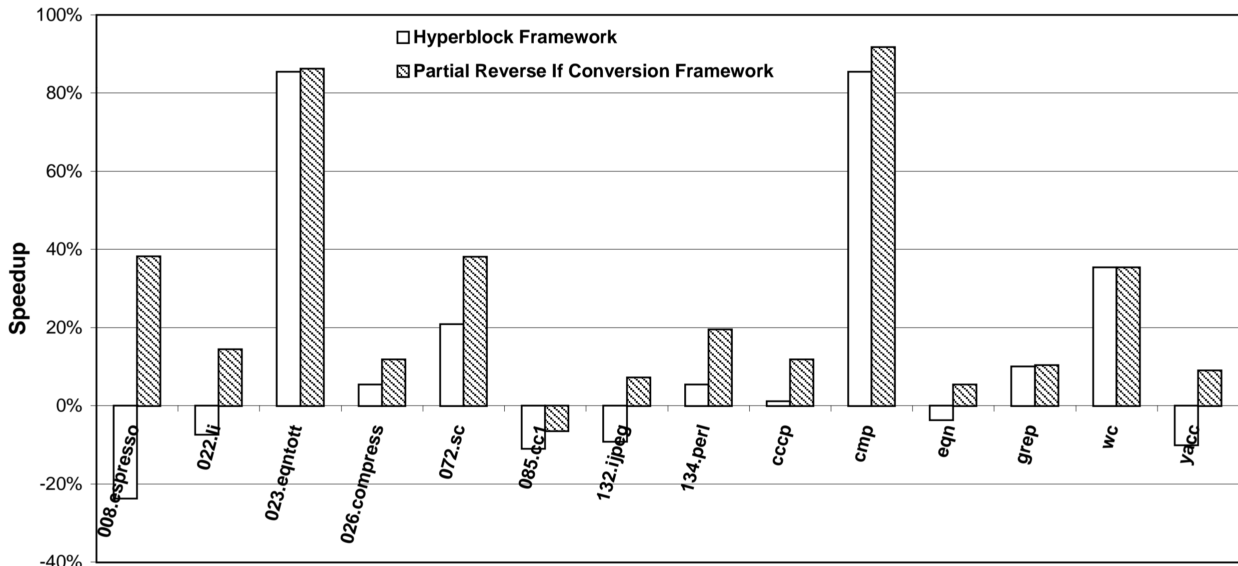


Figure 7.8: Performance increase over superblock exhibited by the hyperblock and partial reverse if-conversion frameworks with a four cycle misprediction penalty.

not factored in, benchmarks exhibiting performance improvement in this graph show that predication has performed well as a compilation model. In particular, the compiler has successfully overlapped the execution of multiple paths of control to increase ILP. Hyperblock compilation achieves some speedup for half of the benchmarks, most notably for *023.eqntott*, *cmp*, *072.sc*, *grep*, and *wc*. For these programs, the hyperblock techniques were able to utilize processor resources better than superblock techniques. On the other hand, hyperblock compilation results in a performance loss for the other half of the benchmarks. This dichotomy is a common problem experienced with hyperblocks: hyperblocks can do well, but performance often falls victim to poor hyperblock selection.

In all cases, partial reverse if-conversion improved upon or matched the performance of the hyperblock code. For six of the benchmarks, partial reverse if-conversion was able to change a loss in performance by hyperblock compilation into a gain. This is most evident for *008.espresso* where a 28% loss was converted into a 39% gain. For *072.sc*,

134.perl, and *cccp*, partial reverse if-conversion significantly magnified relatively small gains achieved by hyperblock compilation. These results indicate that the partial reverse if-converter was successful at undoing many of the poor hyperblock formation decisions while capitalizing on the effective ones. For the four benchmarks where hyperblock techniques were highly effective, *023.eqntott*, *cmp*, *grep*, and *wc*, partial reverse if-conversion did not have a large opportunity to increase performance since the hyperblock formation heuristics worked well in deciding what to if-convert.

It is useful to examine the performance of the worst performing benchmark, *085.cc1*, more closely. For this benchmark, both frameworks result in a performance loss with respect to superblock compilation. Partial reverse if-conversion was not completely successful in undoing the bad hyperblock formation decisions. This failure is due to the policy that requires the list scheduler to decide the location of the reverse if-converting branch by its placement of the predicate define instruction. Unfortunately, the list scheduler may delay this instruction as it may not be on the critical path and is often deemed to have a low scheduling priority. Delaying the reverse if-conversion point can have a negative effect on code performance. To some extent this problem occurs in all benchmarks, but is most evident in *085.cc1*.

Figure 7.8 shows the performance of the benchmarks in the same manner as Figure 7.7 except with a branch misprediction penalty of four cycles. In general, the relative performance of hyperblock code is increased the most when mispredicts are considered because it has the fewest mispredictions. The relative performance of the partial reverse if-conversion code is also increased because it has fewer mispredictions than the

superblock code. Partial reverse if-conversion, however, inserts new branches to accomplish its transformation, so this code contains more mispredictions than the hyperblock code. For several of the benchmarks, the number of mispredictions was actually larger for hyperblock and partial reverse if-conversion than that of superblock. When applying control flow transformations in the predicated representation, the compiler will actually create branches with much higher misprediction rates than those removed. Additionally, the branches created by partial reverse if-conversion may be more unbiased than the combination of branches in the corresponding superblock.

Function-level performance. It is illustrative to examine performance at a finer granularity for a better understanding of the results. Figure 7.9 compares the performance of selected functions from the benchmarks in the same manner as Figure 7.7. The figure assumes no branch misprediction penalty for the results. The functions were selected based on two criteria: contributing a high fraction of the overall benchmark execution time and benefiting a large amount from partial reverse if-conversion. The figure shows the potential of partial reverse if-conversion to increase performance dramatically. In particular, two functions in *132.ijpeg* achieve greater than 250% gain with partial reverse if-conversion. Hyperblock compilation is relatively ineffective for these two functions. The ability to adjust the control structure of the code during scheduling, however, allows the execution time of these same hyperblocks to be reduced dramatically. Even the worst overall benchmark, *085.cc1*, contains an important function that achieves 100% gain with partial reverse if-conversion.

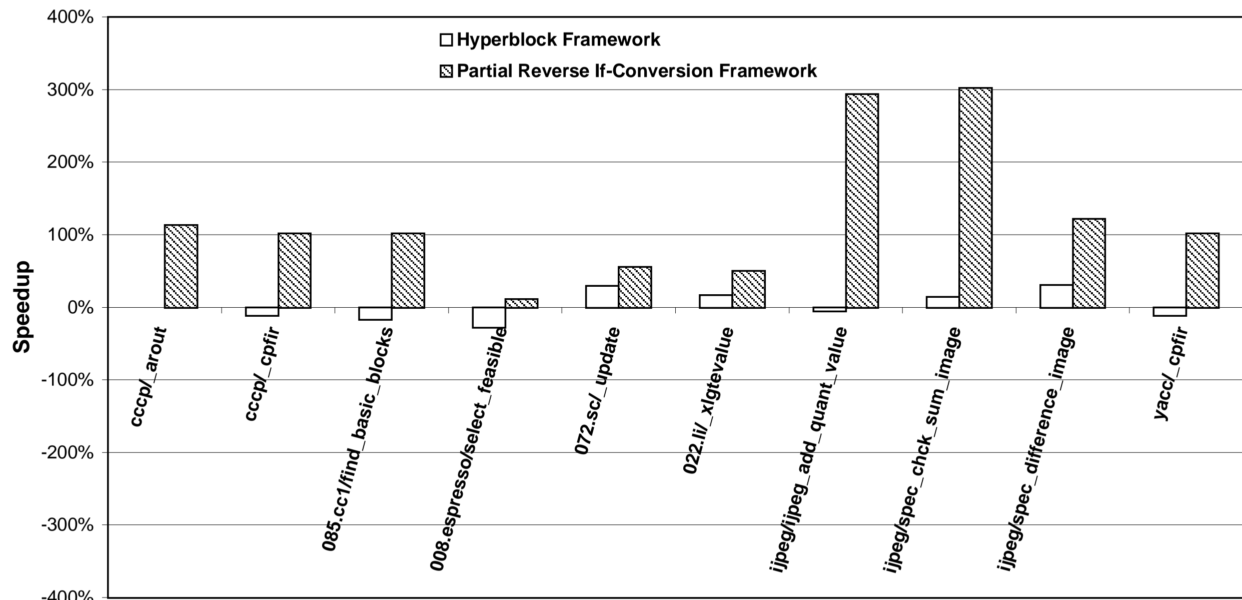


Figure 7.9: Selected function level performance increase over superblock exhibited by the hyperblock and partial reverse if-conversion frameworks with no misprediction penalty.

One of the benchmarks that performed best overall was *072.sc*. For this program, hyperblock compilation increased performance by a fair margin, but partial reverse if-conversion increased this gain substantially. Most of *072.sc*'s performance gain was achieved by transforming a single function, *_update*. This function with superblock compilation executes in 25.6 million cycles, with a rather sparse schedule due to a large number of data and control dependences. Hyperblock compilation increases the available ILP by eliminating a large fraction of the branches and overlapping the execution of multiple paths of control. This brings the execution time down to 19.7 million cycles. While the hyperblock code is much better than the superblock code, it has excess resource consumption on some paths which penalizes other paths. The partial reverse if-converter was able to adjust the amount of if-conversion to match the available resources, efficiently utilizing the processor. As a result, the execution time for the *_update* function is reduced

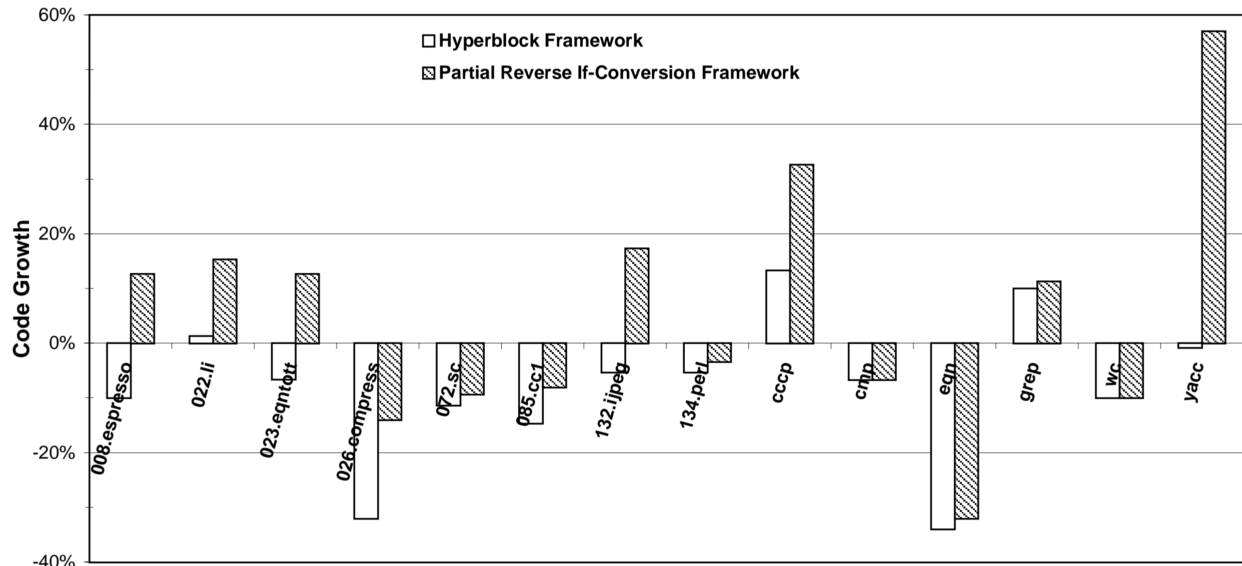


Figure 7.10: Relative static code size exhibited by the hyperblock and partial reverse if-conversion frameworks compared with superblock.

to 16.8 million cycles with partial reverse if-conversion, a 52% performance improvement over the superblock code.

Code size. The static size of code generated using the hyperblock and partial reverse if-conversion frameworks is compared to the size of superblock code in Figure 7.10. From the figure, the use of predicated execution by the compiler has varying effects on the code size. The reason for this behavior is a trade-off between increased code size caused by if-conversion with the decreased code size due to less tail duplication. With superblocks, tail duplication is performed extensively to customize individual execution paths. With predication, multiple paths are overlapped via if-conversion, so less tail duplication is required. The figure also shows that the code produced with the partial reverse if-conversion framework is consistently larger than the hyperblock framework. On average, the partial reverse if-conversion code is 14% larger than the hyperblock code, with the largest growth occurring for *yacc*.

Common to all the benchmarks which exhibit a large code growth was a failure of the simple code size reduction mechanism presented Section 7.1. Inspection of the resulting code indicates that many instructions are shared in the lower portion of the tail duplications created by the partial reverse if-converter. For this reason, one can expect these benchmarks to respond well to a more sophisticated code size reduction scheme.

Application statistics. Finally, the frequency of partial reverse if-conversions that were performed to generate the performance data is presented in Table 7.1. The "Reverse If-Conversions" column specifies the actual number of reverse if-conversions that occurred across the entire benchmark. The "Opportunities" column specifies the number of reverse if-conversions that could potentially have occurred. The number of opportunities is equivalent to the number of unique predicate definitions in the application, since each predicate define can be reverse if-converted exactly once. All data in Table 7.1 are static counts. The table shows that the number of reverse if-conversions that occur is a small fraction of the opportunities. This behavior is desirable as the reverse if-converter should try to minimize the number of branches it inserts to achieve the desired removal of instructions from a hyperblock. In addition, the reverse if-converter should only be invoked when a performance problem exists. In cases where the performance of the original hyperblock cannot be improved, no reverse if-conversions need to be performed. The table also shows the expected correlation between large numbers of reverse if-conversions and larger code size increases of partial reverse if-conversion over hyperblock (Figure 7.10).

Table 7.1: Application frequency of partial reverse if-conversion.

Benchmark	Reverse If-Conversions	Opportunities
008.espresso	204	1552
022.li	50	393
023.eqntott	43	443
026.compress	11	56
072.sc	33	724
085.cc1	479	3827
132.ijpeg	134	1021
134.perl	42	401
cccp	77	1046
cmp	4	49
eqn	33	326
grep	3	103
wc	0	88
yacc	247	1976

7.3 Significance

The partial reverse if-converter, a mechanism which allows the compiler after aggressive if-conversion and predicate domain optimization to, at schedule time, balance predication and control flow ideally for a target architecture. A first generation partial reverse if-converter was implemented and the effectiveness of the framework was measured for this chapter. The framework was able to capitalize on the benefits of predication without being subject to the sometimes negative side effects of overaggressive hyperblock formation. Furthermore, additional opportunities for performance improvement were exploited by the framework, such as partial path if-conversion. These points were demonstrated by the hyperblock performance losses which were converted into performance gains, and by moderate gains which were further magnified. In addition, the framework provides

an important mechanism for eliminating the negative effects of overly aggressive transformations at schedule time. With such a backup mechanism, unique opportunities are introduced for the aggressive use and transformation of the predicate representation early in the compilation process.

8. SYSTEMATIC COMPILATION FOR PREDICATED EXECUTION

8.1 System Overview

The techniques presented in Chapters 4 through 7 are combined together in the IMPACT compiler to create a mechanism that can systematically compile for architectures with predicated execution support. A block diagram of the new backend compilation path is shown in Figure 8.1. After Lcode generation, classical optimizations are performed on the traditional, nonpredicated code. This provides the if-converter with relatively clean code upon which to work. The if-converter then aggressively forms predicated code by converting large amounts of control flow to the predicated representation. Iteratively, ILP optimizations and predicate optimizations are performed. Iterative application of optimizations is essential because optimizations expose opportunities for other operations in an unordered manner. Optimizations include loop unrolling, register renaming, and induction variable expansion. The Predicate Decision Logic Optimizer presented in this work is included among the predicate optimizations.

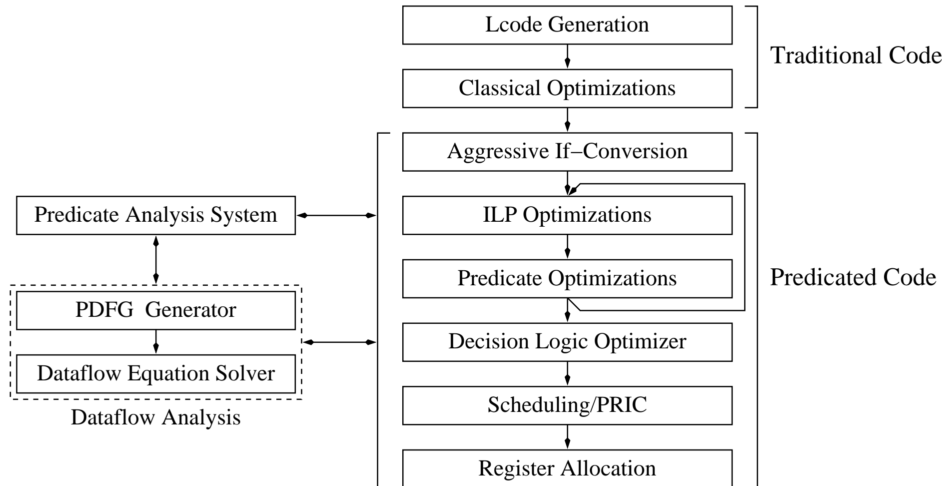


Figure 8.1: Block diagram of the backend compilation path with hyperblocks.

After the code has been sufficiently optimized, usually to a locally optimal state for the given set of optimizations, the scheduler performs partial reverse if-conversion during the prepass scheduling phase. The Partial Reverse If-Converter (PRIC) reinserts control flow in order to provide a balance of predication and control flow ideal for the modeled target machine. The PRIC need only reintroduce control flow to achieve the appropriate balance because the code it is operating on has already been if-converted aggressively with respect to the target machine.

The PRIC is run after the Predicate Decision Logic Optimizer (PDLO) for two reasons. First, in certain hyperblocks, the PDLO may favor flattening over subexpression reuse, creating a large number of partially redundant parallel predicate defines. In some cases, the number of predicate defines in the optimized code exceeds the resources of the machine. Here, the PRIC can rectify the situation by dividing the oversubscribing instructions between two blocks, each with a portion of the total predicate define network. Second, the reverse if-converting branch cannot be placed before the definition of its

guard predicate. Because the effectiveness of a reverse if-conversion is tied to how early the reverse if-converting branch can be placed, flattening the predicate define network is desirable before placing it. In this way, the PDLO may influence the PRIC to perform a now profitable reverse if-conversion which it would previously have ignored.

During processing of predicated code in the compiler, information regarding the flow of data and the relationships among predicates is consumed by the optimization, scheduling, and register allocation routines. To provide essential, accurate information, the PAS analyzes and stores in a database the relationships among predicates. These predicate relationships are provided directly to the optimizer, scheduler, and register allocator. The PDLO ties heavily into the PAS's database in order to perform its functions, which are only possible due to the perfect accuracy of the PAS. Additionally, the PAS assists in the creation of the predicate dataflow graph (PDFG). The PDFG represents the flow of data through the predicated code matrix in such a way that traditional dataflow analysis solvers can operate on it directly, as they do on traditional control flow graphs.

8.2 System Example

To illustrate the synergistic manner in which the PRIC Framework, the PAS, the PDFG, and the PDLO operate, a running code example is used in this section. Figure 8.2(a) shows the control flow graph of a segment of code crafted for this purpose. This code is characteristic of the code seen after Lcode generation in the phase ordering shown in Figure 8.1. (Typically, real codes are larger and more complex, making them unsuitable for the purposes of illustration.) This code segment, a loop, consists of seven internal branches and a loop exit branch. These internal branches form the decision

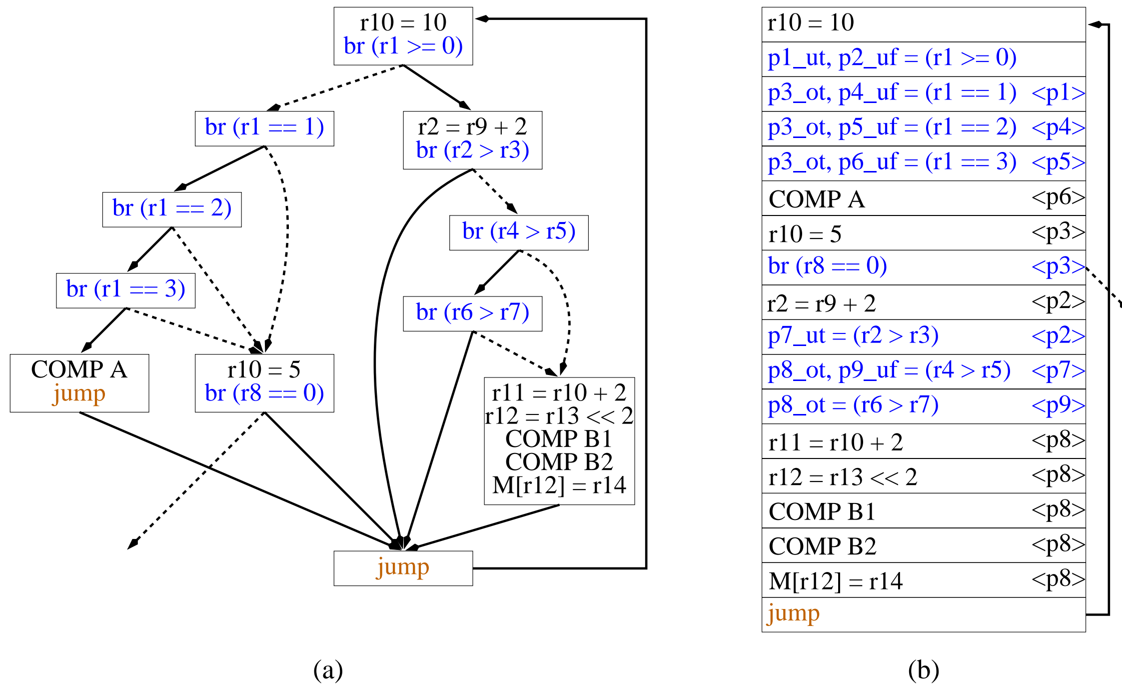


Figure 8.2: Control flow graph for a loop (a) and its corresponding hyperblock (b).

component of this loop. The dashed lines emanating from each branch correspond to the "true" or "taken path," and the solid lines represent the "false" or "fall through path." The computation in the loop consists of nine instructions arranged in five basic blocks. For simplicity of illustration, some computation is simply shown as **COMP** followed by an identifier. The three-instruction computation utilizing register $r10$ illustrates optimization in the predicated domain, while the register $r2$ illustrates an interaction on the computation component/decision component boundary.

Figure 8.2(b) shows the code, now a hyperblock, in the CFG after aggressive if-conversion. Aggressive if-conversion is performed in such a way that target machine resources will likely be over-saturated. Unlike the Hyperblock Compilation Framework, the Partial Reverse If-Conversion Framework does not require carefully crafted heuristics at this phase of the compilation. In the complete if-conversion of this loop, each internal

C_0	(r1>=0)
C_1	(r1==1)
C_2	(r1==2)
C_3	(r1==3)
C_4	(r2>r3)
C_5	(r4>r5)
C_6	(r6>r7)

(a)

$p1$	C_0
$p2$	$\overline{C}_0$
$p4$	$C_0\overline{C}_1$
$p5$	$C_0\overline{C}_1\overline{C}_2$
$p6$	$C_0\overline{C}_1\overline{C}_2\overline{C}_3$
$p3$	$C_0C_1+C_0\overline{C}_1C_2+C_0\overline{C}_1\overline{C}_2C_3$
$p7$	$\overline{C}_0C_4$
$p9$	$\overline{C}_0C_4\overline{C}_5$
$p8$	$\overline{C}_0C_4C_5+\overline{C}_0C_4\overline{C}_5C_6$

(b)

Figure 8.3: The conditions (a) and predicate expressions (b) for the code segment.

conditional branch is replaced by a definition of one or more predicates. The network of predicate computation generates nine predicates, one for each conditionally executed basic block in the original control flow graph. The computation in each of these basic blocks is guarded by a predicate source operand with the same execution condition as the original code. Control flow is removed since the only remaining conditional branch is the loop exit condition.

Before optimization can proceed in the predicated code domain, the PAS must compute the relationships among all predicates. This information is analogous to that provided by control flow analysis, such as dominator sets, except with the added accuracy provided by consideration of the branch/predicate define conditions. The conditions extracted from the predicated code in Figure 8.2(b) are shown in Figure 8.3(a). For example, the first predicate defining instruction performs comparison C_0, which is the register constant inequality $(r1 >= 0)$. These conditions are the constituent variables for the expressions characterizing each predicate. For example, the first predicate defining instruction uses the result of the C_0 comparison to define $p1$. As shown in Figure 8.3(b),

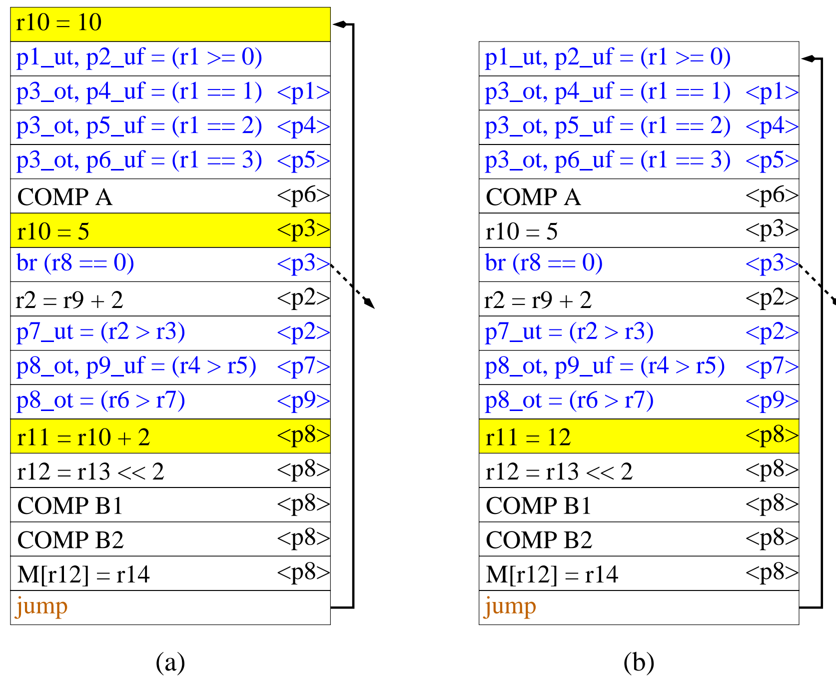


Figure 8.4: The hyperblock before (a) and after (b) an application of the constant propagation optimization.

$p1 = C_0$. The information summarized in Figure 8.3 is stored by the PAS in a manner that allows relationship among predicates to be determined easily for use in code manipulation.

In Figure 8.4(a), three references to register $r10$ are made. In the first two references, $r10$ is assigned a constant while in the third reference, $r10$ is used to define $r11$. In a non-predicated environment this situation seems ideal for a constant propagation optimization. In the predicated representation, however, the execution conditions of these instructions must be considered. The first definition is unconditional; $r10$ always assumes the value 10 at the beginning of the hyperblock. Depending on the value of predicate $p3$, $r10$ maybe be redefined to 5. Because the third instruction executes only under $p8$, a constant propagation to this instruction need only be valid when $p8$ is 1. The question

which must be considered by the constant propagation optimization is: What constant value, if any, does $r10$ hold when $p8$ is 1? To answer this question, the relationship of $p8$ and $p3$ must be considered. The PAS is queried and indicates that $p3$ and $p8$ do not intersect; $p3 \cap p8 = \emptyset$. This can be confirmed by considering the expression for these predicates when C_0 is 1 and when C_0 is 0. This result indicates that for each iteration of this loop $r10 = 10$ whenever $p8$ is 1. The constant propagation can proceed as shown in Figure 8.4(b).

As shown in Figure 8.1, predicate optimizations are performed iteratively with ILP optimizations. However, PDLO is applied only once. before scheduling. The PDLO simplifies the program's decision component, extracted by the PAS and represented by Boolean expressions, and reformulates it into a more efficient form. The decision component consists of two types of predicates: essential and nonessential. Essential predicates are those which guard the execution of computation in the program, and as such are the only predicates which must be reformulated by the PDLO. Nonessential predicates are predicates used solely as intermediates to compute essential predicates.

Prior to the determination of the essential predicates, another predicate specific optimization called promotion should be performed. Promotion is the predicate domain's equivalent to speculation by code motion. With promotion, an instruction's guard predicate is weakened. Under the weaker predicate, the instruction will execute more frequently. However, the promotion has broken a data dependence between the definition of the original predicate and the promoted instruction. Breaking data dependences in this way provides the scheduler more freedom in extracting ILP. Since promotion changes which predicates guard computation, it also determines which predicates are essential.

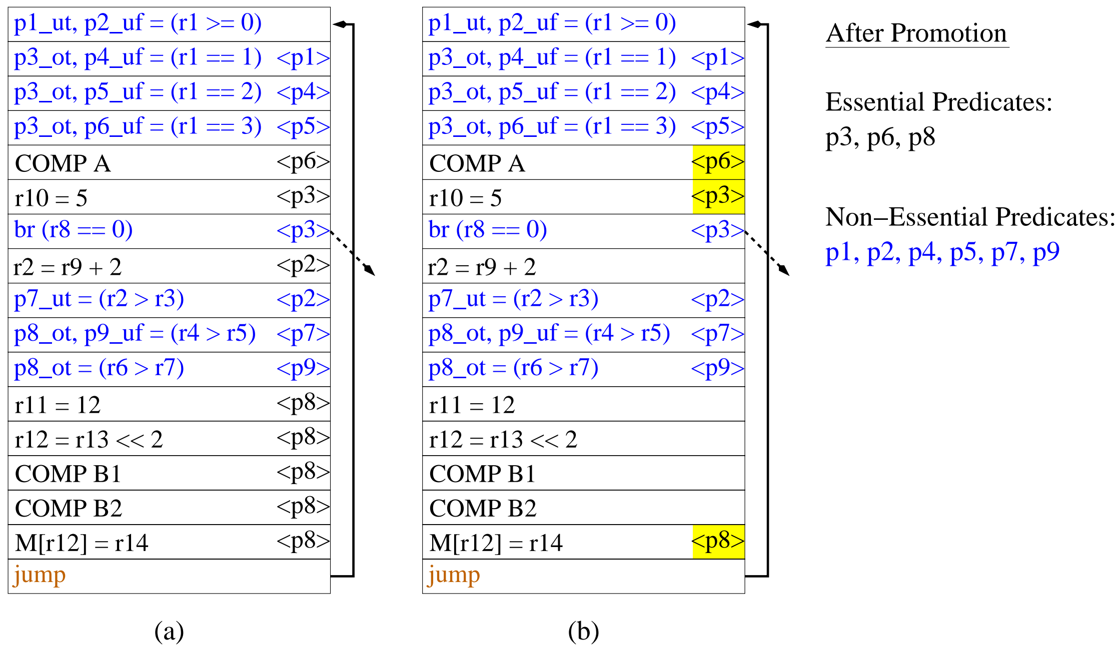


Figure 8.5: The hyperblock before (a) and after (b) promotion is performed.

Figure 8.5 shows the hyperblock before and after promotion. Before promotion, predicates $p2$, $p3$, $p6$, and $p8$ guard computation. Promotion is able to remove most uses of $p8$ and the use of $p2$, leaving $p2$ nonessential. The essential predicates, those which must be computed in any reformulated decision logic, are $p3$, $p6$, and $p8$.

After promotion, the PDLO minimizes the program decision logic Boolean expressions. The logic expressions for $p3$ and $p6$ before minimization are shown as gates in Figure 8.6(a). Intermediate predicates $p1$, $p4$, and $p5$ are lost after the optimization, as shown in 8.6(b). In the reformulated code, there is often a one-to-one relationship between level of gate delay in these circuits and dependence height. The relationship between the number of gates and the number of predicate defining instructions is less reliable, but typically fewer is better. In this example, the levels of gate delay height has

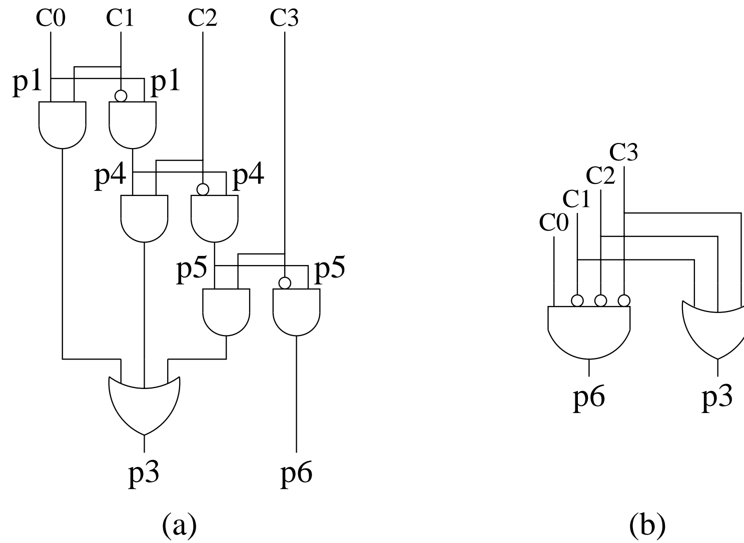


Figure 8.6: The logic network for the computation of predicates $p3$ and $p6$ before (a) and after (b) optimization.

dropped from four to one, and the number of gates has been significantly reduced from seven to two.

In Figure 8.7, the effect of these reductions can been seen. The code in the running example is scheduled on a general four-issue machine. Figure 8.7(a) shows the decision component as consisting of seven predicate defines with a dependence height of four. This dependence height delays the execution of the four highlighted instructions to cycle five. Assuming the path weighting shown in the figure, the average execution time for each invocation of this hyperblock is 5.8 cycles. The promoted instruction $r2 = r9 + 2$ can be executed in cycle one rather than a later cycle because of the broken data dependence on $p2$. The movement of this definition of $r2$ one cycle earlier has the even more desirable effect of allowing the computation of $p7$, which uses the value of $r2$, to proceed a cycle earlier. While this allows $M[r12] = r14$ to proceed one cycle earlier, this does not affect the hyperblock's performance in this case.

p1_ut, p2_uf = (r1 >= 0)	r2 = r9 + 2	r11 = 12	r12 = r13 << 2
p3_ot, p4_uf = (r1 == 1) <p1>	p7_ut = (r2 > r3) <p2>	COMP B1	COMP B2
p3_ot, p5_uf = (r1 == 2) <p4>	p8_ot, p9_uf = (r4 > r5) <p7>		
p3_ot, p6_uf = (r1 == 3) <p5>	p8_ot = (r6 > r7) <p9>		
COMP A <p6>	r10 = 5 <p3>	M[r12] = r14 <p8>	br (r8 == 0) <p3>
			jump

(r1 < 0) path [30%]: 6 cycles
(r1 >= 0, r8 == 0) path [20%]: 5 cycles
(r1 >= 0, r8 != 0) path [50%]: 6 cycles

Total: 5.8 cycles

(a)

p3_ot, p6_af = (r1 == 1)	p3_ot, p6_af = (r1 == 2)	p3_ot, p6_af = (r1 == 3)	r11 = 12
p1_ut, p2_uf = (r1 >= 0)	r10 = 5 <p3>	r12 = r13 << 2	br (r8 == 0) <p3>
p7_ot = (r4 > r5) <p2>	p7_ot = (r6 > r7) <p2>	COMP B1	r2 = r9 + 2
p8_ut = (r2 > r3) <p7>		COMP B2	COMP A <p6>
		M[r12] = r14 <p8>	jump

(r1 < 0) path [30%]: 5 cycles 17%
(r1 >= 0, r8 == 0) path [20%]: 2 cycles 60%
(r1 >= 0, r8 != 0) path [50%]: 5 cycles 17%

Total: 4.4 cycles 24%

(b)

Figure 8.7: Hyperblock before (a) and after (b) Program Decision Logic Optimization.

After the decision logic has been minimized, PRIC balances control flow during prepass scheduling. PRIC must decide which predicates, if any, to reverse if-convert. To do this it must ascertain the profitability of each predicate. A significant factor in an if-conversion's profitability is the schedule it enables in the reverse if-conversions taken and fall-through paths. To determine these schedules, the location of each instruction (taken path, fall through path, or both) must be determined. Figure 8.8 shows the predicate flow graph after partial dead code elimination. All possible paths are enumerated in this figure for illustration. In practice the number of paths could become excessive, so active range computations are done, as described in Chapter 7, to reduce the space and time required to compute partial dead code. Each path has a set of true and false predicates.

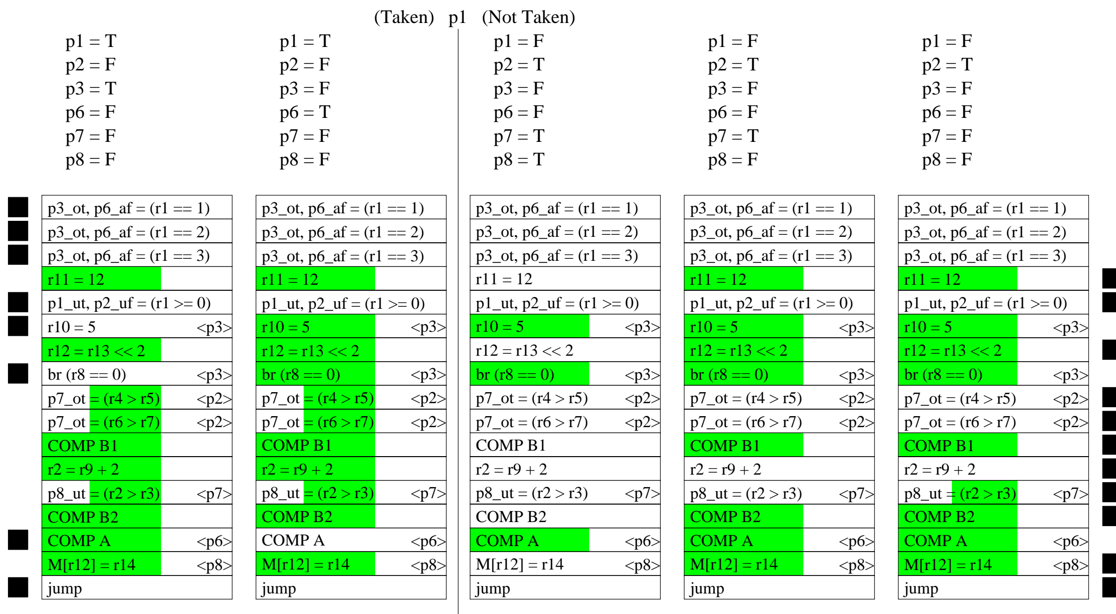


Figure 8.8: Partial Dead Code Elimination after Program Decision Logic Optimization.

The first step is to eliminate all instructions in a path predicated on a false predicate. For example, the instruction "$M[r12] = r14$" in the first path is based on predicate $p8$ which is false. The following step, run iteratively until there is no change, is to run live variable analysis on the code and to remove all instructions from paths in which its destination is not live out. For example, $r12$ in "$r12 = r13 << 2$" of the first column is no longer live out since "$M[r12] = r14$", its consumer, has been marked dead in the same column. The process iterates and the resulting graph is shown in the figure. The shaded regions of code are are the portion of the code which is dead. Notice that predicate uses are never marked as dead because a predicate is used even if it is false, unlike the rest of the instruction. For the purposes of partial reverse if-conversion, another partial dead code elimination step is taken which cannot be done for other tasks such as scheduling. This step is the removal of predicate source operands for instructions in

which the predicate is always false. Consider the predicate flow graph as a complete reverse if-conversion. In a path, instructions which are never executed can be removed from that path. If an instruction does not exist in a path, its source predicate is no longer necessary to prevent its execution in that path. With the removal of source predicates, other instructions, such as predicate defines, can also be deleted. One result of this step is the deletion of the predicate defines of $p7$ and $p8$ in the first column.

At some point, the partial reverse if-conversion selection process considers predicate $p1$. To determine where instructions fall with respect to this predicate, the predicate flow graph in Figure 8.8 is divided in two parts, a "$p1$ is true" or "taken" part and a "$p1$ is false" or "fall through" part. For each part, each instruction is considered for inclusion. If an instruction is live in either path of the bisection, it exists in that part. An instruction can exist in one part, both parts, or neither part in the case of dead code. The result of this process is shown in the figure as the colored squares at each side of the figure. These squares indicate that the instruction in the corresponding row exists in this part. For example, "$M[r12] = r14$" will only exist on the path where $p1$ is false.

Once the location of each instruction has been determined, the partial reverse if-converter estimates the execution time by considering the schedule before the partial reverse if-conversion, the schedule in the taken path and the schedule in the fall through path. In this example, after measuring the cost of inserting $p1$, it determines that a $p1$ if-conversion is profitable.

The result of this reverse if-conversion is shown in Figure 8.9. The PDLO optimized code shown in Figure 8.9(a) has an average execution time for each invocation of this hyperblock of 4.4 cycles, assuming the path weighting shown in the figure, on a four-issue

machine. After flattening of the predicate network, the code becomes resource limited. Looked at another way, the code is not operating at its full potential because the issue rate of the machine is only four instructions per cycle. To reduce the issue requirements, the reverse if-converter split the code by inserting the the reverse if-converting jump on predicate $p1$. The shaded code in this example corresponds to the instructions that must exist on the taken path and is a subset of the instruction marked in Figure 8.8. These instructions can be removed from the block and placed in the taken path. In this case, three of the six instructions were removed from the block. Three predicate defines remain in the original block since they were scheduled before the reverse if-converting branch. The back edge jump must exist on both paths since it is effective regardless of the value of $p1$. The definition of $p1$ remains in the original block before the reverse if-conversion. The back edge jump must be scheduled after all the code in the block due to control dependences; therefore, it is duplicated so that it may exist on both the fall through and taken paths. The resulting average execution time for each invocation of this region containing two hyperblocks is 3.3 cycles, down from 4.4 cycles, with the same assumptions.

In this example, the synergy between the PDLO and the PRIC was illustrated. The PDLO code reduced the dependence height but over-saturated the assumed machine. The PRIC recognized this fact and undid the decision to if-convert on $p1$ early in the compilation process. The code is better for having if-converted $p1$ early because it allowed some instructions from the $p1$ block, namely the instructions defining predicates $p3$ and $p6$, to migrate to the dominating header block. In effect, global scheduling was performed by the local scheduler with reverse if-conversion capability.

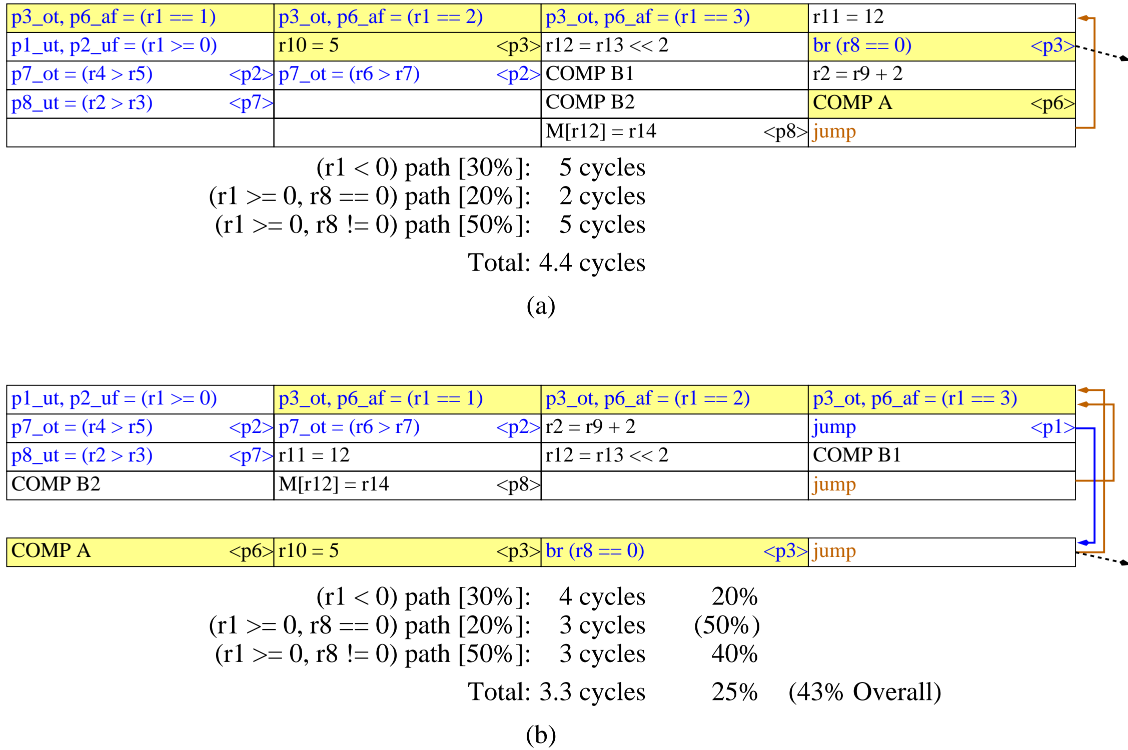


Figure 8.9: Hyperblock before (a) and after (b) partial reverse if-conversion.

8.3 Experimental Evaluation

The effectiveness of predicated execution using the proposed compilation techniques is evaluated in this chapter.

8.3.1 Experimental methodology

For the evaluation of the entire compilation framework presented in this work, IMPACT version 991224-I was used. The PAS, PDFG, PDLO, and PRIC were all implemented together in an integrated fashion within the IMPACT compiler framework.

A block diagram of the IMPACT compiler is presented in Figure 8.10. The compiler is divided into three distinct parts based on the level of intermediate representation (IR) used. The highest level IR, *Pcode*, is a parallel C code representation with loop constructs

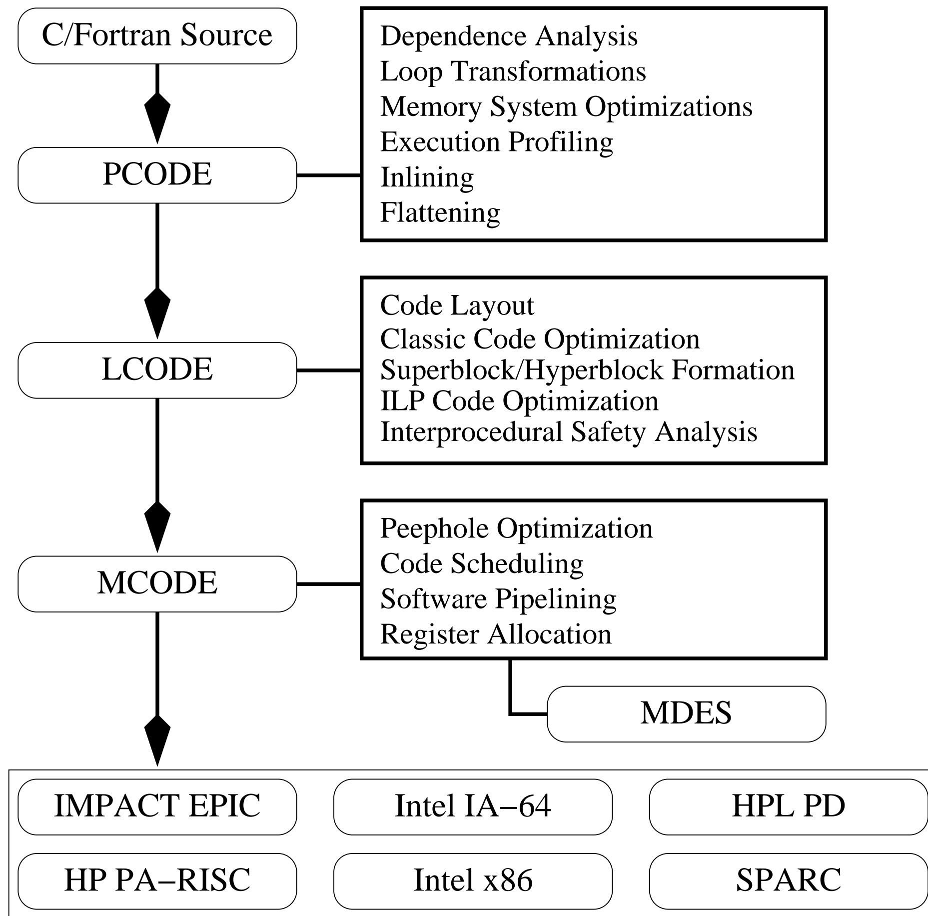


Figure 8.10: The IMPACT compiler.

intact. In Pcode, memory dependence analysis [61], [62], loop-level transformations [63], execution profiling, function inlining, and memory system optimizations [64], [65] are performed.

The next level of IR in the IMPACT compiler is referred to as *Lcode*, which is a generalized register transfer language similar in structure to most load/store processor assembly instruction sets. Lcode is designed to be a machine-independent IR. The machine-specific IR is called *Mcode*. The data structures for the Lcode and Mcode are identical. The difference is that Mcode is broken down such that there is a one-to-one

mapping between Mcode instructions to the target machine's assembly language. Therefore, to convert Lcode to Mcode, the code generator breaks up Lcode instructions into one or more instructions which directly map to the target architecture. Lcode instructions are broken up for a variety of reasons including limited addressing modes, limited opcode availability, ability to specify a literal operand, and field width of literal operands.

At the Lcode level, all machine independent classic optimizations are applied [66]. These include constant propagation, forward copy propagation, backward copy propagation, common subexpression elimination, redundant load elimination, redundant store elimination, strength reduction, constant folding, constant combining, operation folding, operation cancellation, code reordering, dead code removal, jump optimization, unreachable code elimination, loop invariant code removal, loop global variable migration, loop induction variable strength reduction, loop induction variable elimination, and loop induction variable reassociation. Additionally at the Lcode level, interprocedural safety analysis is performed [57]. This includes identifying safe instructions for speculation and function calls that do not modify memory (side-effect free).

Predication is introduced through if-conversion performed in the Lcode IR and the Predicate Hierarchy Graph (PHG) is used to analysis the predicate relationships. Predicate dataflow analysis is performed in a conservative manner.

All code generation in the IMPACT compiler is performed at the Mcode level. The two largest components of code generation are the instruction scheduler and register allocator. Scheduling is performed via either acyclic global scheduling [57], [67] or software pipelining using modulo scheduling [12]. For acyclic global scheduling, code scheduling is applied both before register allocation (prepass scheduling) and after register allocation

(postpass scheduling) to generate an efficient schedule. For software pipelining, loops targeted for pipelining are identified at the Pcode level and marked for pipelining. These loops are then scheduled using software pipelining, and the remaining code is scheduled using the acyclic global scheduler. In addition to control speculation, both scheduling techniques are capable of exploiting architectural support for data speculation to achieve more aggressive schedules [62], [68], [69].

Graph coloring based register allocation is utilized for all target architectures [70]. The register allocator employs execution profile information if it is available to make more intelligent decisions. For each target architecture, a set of specially tailored peephole optimizations are performed. These peephole optimizations are designed to remove inefficiencies during Lcode to Mcode conversion, to take advantage of specialized opcodes available in the architecture, and to remove inefficient code inserted by the register allocator.

A detailed machine description database, *Mdes*, is also available to all Lcode compilation modules [71]. The Mdes contains a large set of information about target architectures to assist with optimization, scheduling, register allocation, and code generation. Information such as the number and type of available function units, size and width of register files, instruction latencies, instruction input/output constraints, addressing modes, and pipeline constraints is provided by the Mdes. The Mdes is queried by the optimization phases to make intelligent decisions regarding the applicability of transformations. The scheduler and register allocator rely more heavily on the Mdes to generate efficient as well as correct code.

Table 8.1: Machine models used in the evaluation.

Machine	Issue	Integer	Memory	Branch	Floating Point
6G_1BL	6	6	6	1	6
6L_1BL	6	4	2	1	2
8G_1BL	8	8	8	1	8
8L_1BL	8	5	4	1	3
8L_2BL	8	5	4	2	3

Many architectures are actively supported by the IMPACT compiler. Among them are SPARC [72], HP PA-RISC, and Intel x86. Three other supported architectures, IMPACT EPIC, HPL PD [20], and IA-64 [22], are ILP architectures which support predicated execution. For this thesis, all experiments target the IMPACT EPIC architecture as discussed in Chapter 2.

The systematic compilation framework proposed in this work is implemented at the Lcode and Mcode levels. The PAS completely replaces the Predicate Hierarchy Graph for all transformation done after if-conversion. The Predicate Dataflow Graph is also used to perform predicate-aware dataflow analysis for all these transformations as well. The Predicate Decision Logic Optimizer is performed on Lcode using information from the scheduling framework [37]. The Partial Reverse If-Converter is done at prepass scheduling time. To see how these techniques relate to one another refer to Figure 8.1.

8.3.2 Derivation of results

To measure the effectiveness of the proposed compilation model, five in-order EPIC machine models were selected. Table 8.1 shows the IMPACT machine model name and its corresponding functional unit mix. With the exception of memory latencies which

were 3 cycles, the instruction latencies assumed match those of the HP PA-7100 microprocessor [32]. The instruction set contains non-trapping versions of all potentially excepting instructions, with the exception of branch and store instructions, to support aggressive speculative execution.

The benchmarks used in this experiment consist of 17 non-numeric programs: the six SPEC CINT92 benchmarks, *008.espresso*, *022.li*, *023.eqntott*, *026.compress*, *056.ear*, and *072.sc*; five SPEC CINT95 benchmarks, *124.m88ksim*, *129.compress*, *130.li*, *132.ijpeg*, and *134.perl*; and six Unix utilities *cccp*, *cmp*, *grep*, *lex*, *wc*, and *yacc*.

8.3.3 Results

Overall performance. The execution time for each benchmark is derived from the static code schedule weighted by dynamic execution frequencies obtained from profiling. Benchmark performance ignores dynamic stall cycles associated with the memory system including instruction and data cache misses. Previous experience with this method of run time estimation has demonstrated that it accurately estimates simulations of an equivalent machine with perfect caches.

Figures 8.11 through 8.15 compare the overall benchmark performance of the Hyperblock Framework, the PDLO, and the PRIC Framework. All models were generated using the PAS and Predicate Dataflow Graph. Performance is reported as the speedup in execution cycles versus the Superblock Compilation Framework.

These figures show the performance of the of the system assuming no branch misprediction penalty. Since branch mispredictions are not factored in, benchmarks exhibiting performance improvement in these graphs show that predication has performed well as a

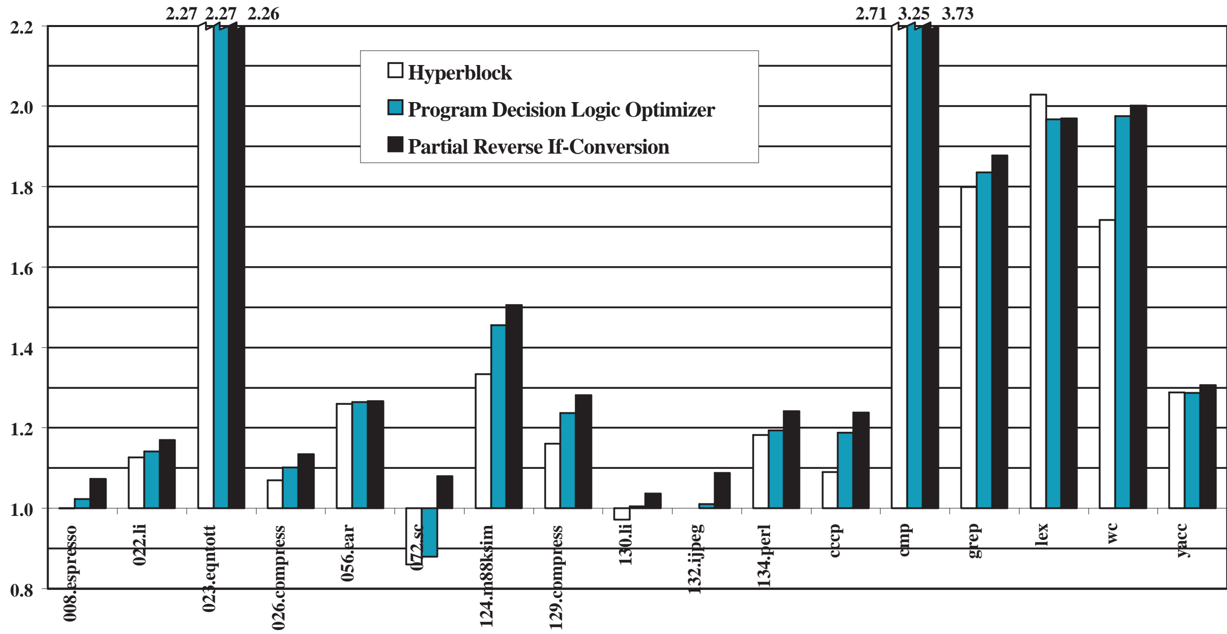


Figure 8.11: System performance on machine 6G_1BL relative to superblock.

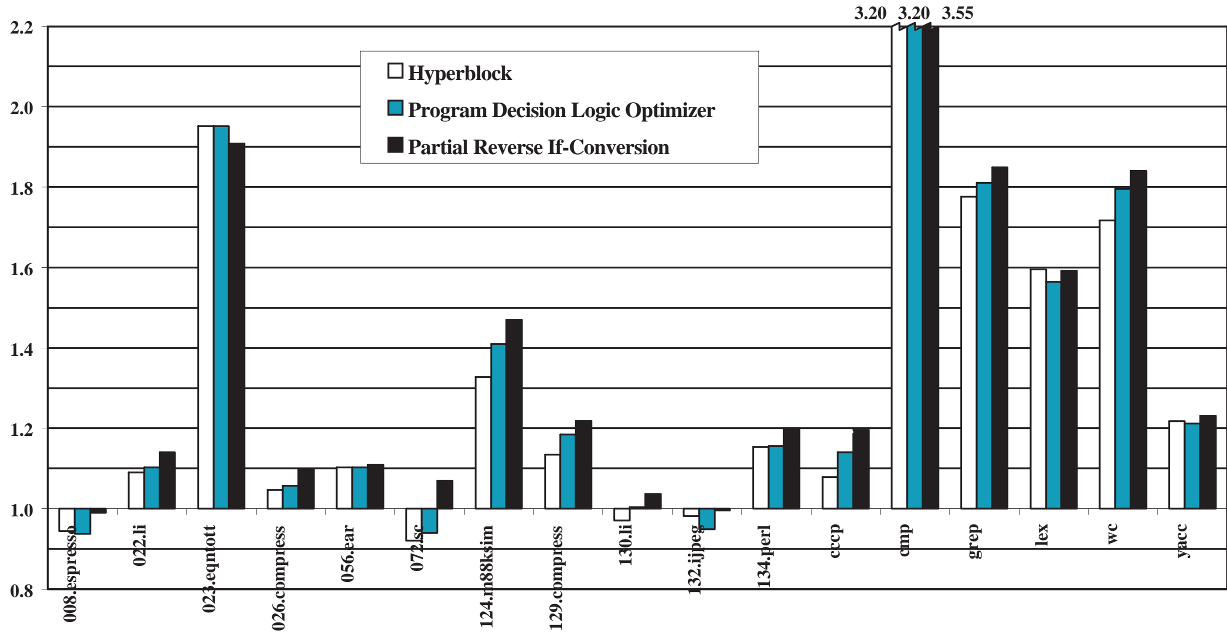


Figure 8.12: System performance on machine 6L_1BL relative to superblock.

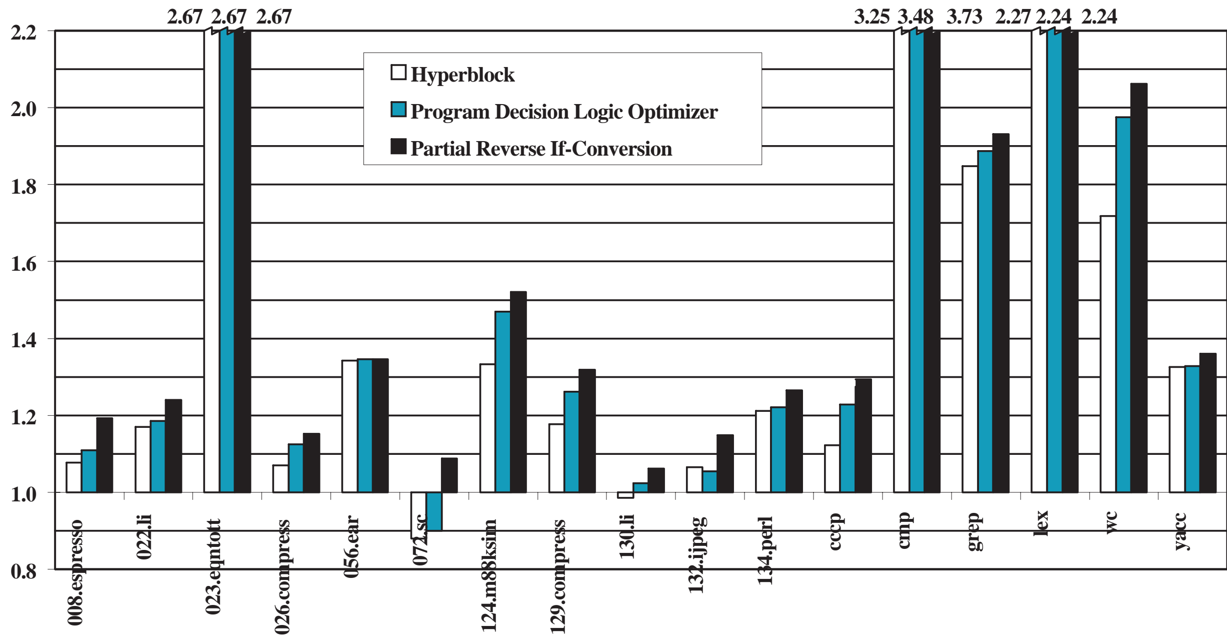


Figure 8.13: System performance on machine 8G_1BL relative to superblock.

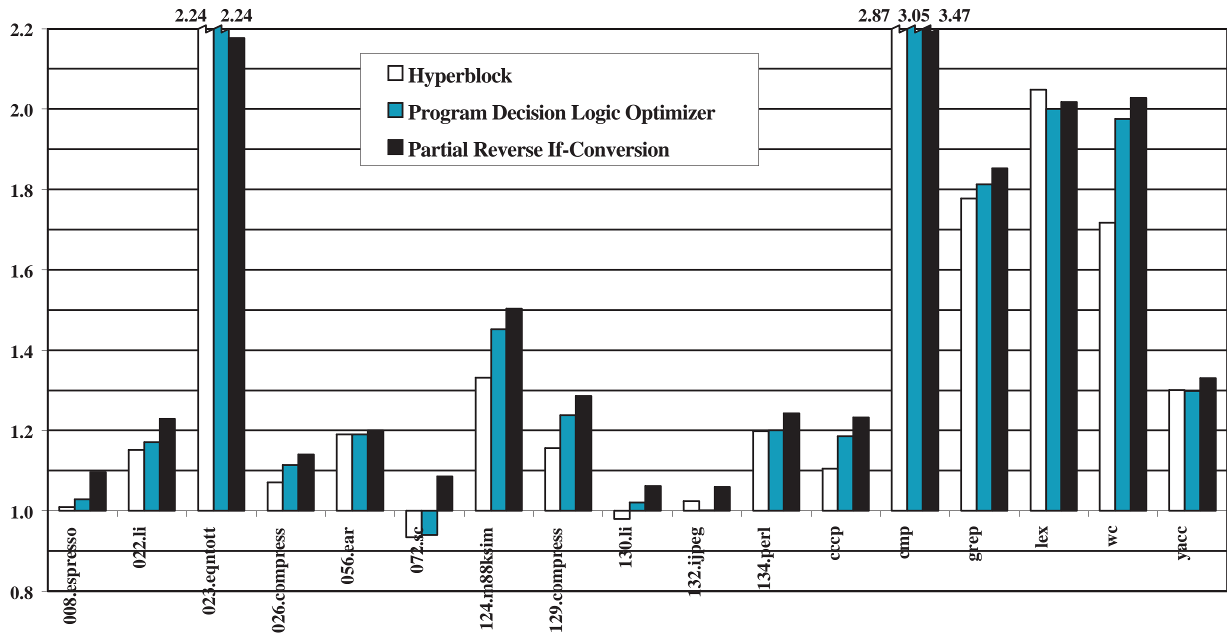


Figure 8.14: System performance on machine 8L_1BL relative to superblock.

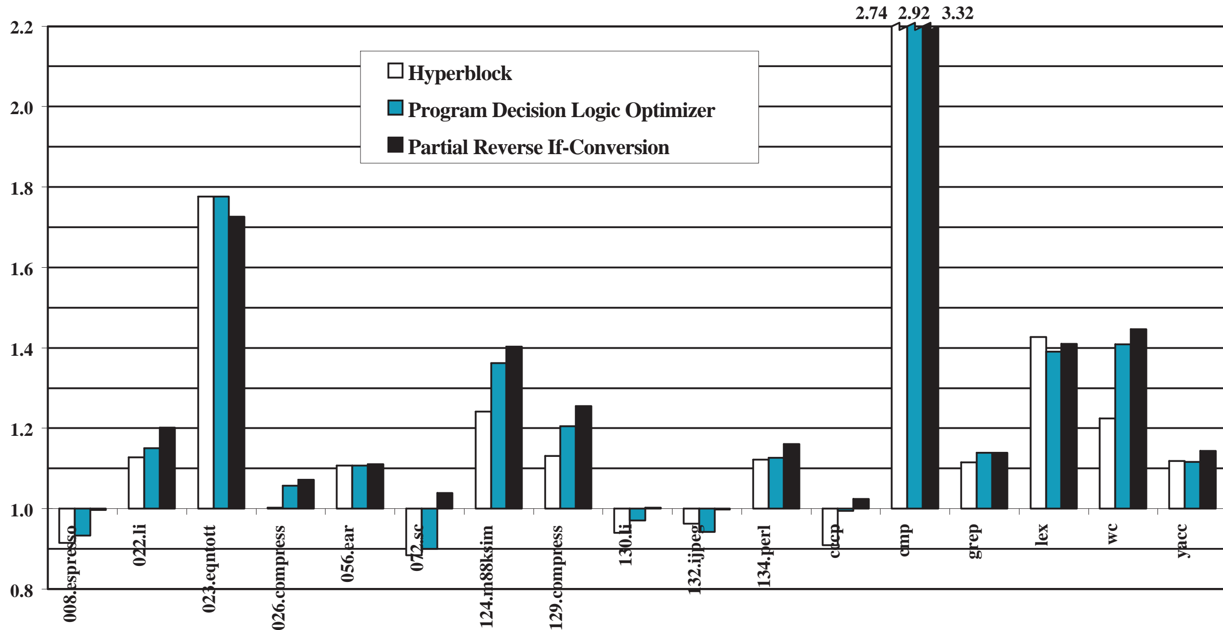


Figure 8.15: System performance on machine 8L_2BL relative to superblock.

compilation model. In particular, the compiler has successfully overlapped the execution of multiple paths of control to increase ILP. Hyperblock compilation achieves speedup for most of the benchmarks. However, for some benchmarks, the Hyperblock Compilation Framework performed worse than the Superblock Compilation Framework, most notably in *072.sc* and *130.li* which behaved this way in all machine models. In all cases, decision logic optimization combined with partial reverse if-conversion beat or matched the performance of the superblock code. For five of the benchmarks, the framework was able to change a loss in performance by hyperblock compilation into a gain. In all but two benchmarks the PDLO, matched or improved on superblock program performance. In these cases, *132.ijpeg* and *lex*, the optimizer inappropriately chose flattening over factorization, causing an oversaturation of resources.

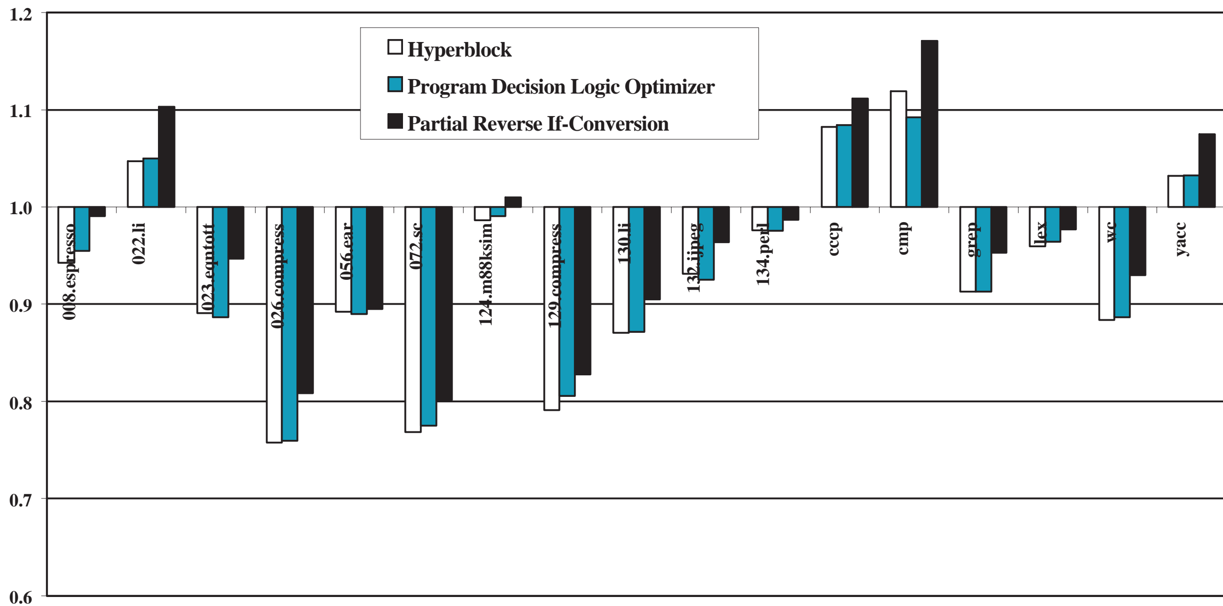


Figure 8.16: Code size on machine 8G1BL relative to superblock.

Code size. The static size of code generated using these techniques is compared to the size of superblock code in Figure 8.16. From the figure, the use of predicated execution by the compiler has dramatically reduced the size of the code than the superblock model due to performing less tail duplication. With superblocks, tail duplication is performed extensively to customize individual execution paths. With predication, multiple paths are overlapped via if-conversion, so less tail duplication is required. Further, predicate optimizations, such as operation merging, remove redundant code which exists in the unoptimized code. The figure also shows that the code produced with the partial reverse if-conversion framework is consistently larger than the hyperblock framework, by 4%, with the largest growth occurring for *026.compress* at 6%.

With consistently good performance at a low code size overhead, the compilation system proposed in this work demonstrates that early predication has very desirable

properties. As the a prototype predication compilation framework, it has demonstrated that predication, used correctly, is extremely desirable, and that given a reliable substrate predicate optimizations can be consistently effective. The framework has also exposed the next set of obstacles to ever higher levels of ILP and has revealed hints on how to overcome them.

9. CONCLUSION

9.1 Summary

Predicated execution has been shown to be a powerful tool for enhancing ILP by reducing or eliminating the detrimental effects of branching control flow. Effective application of this technology, however, requires the compiler to fully understand the new paradigm on which it is based. This dissertation presented a set of four related subsystems which, when combined, provide a systematic method of compilation for producing predicated code with consistent high performance. Three techniques, the Partial Reverse If-Conversion (PRIC) Framework, the Predicate Analysis System (PAS), and the Predicate Dataflow Graph (PDFG), provide improvement over previous methods by eliminating inaccuracy and the need for predictive heuristics. One technique, the Predicate Decision Logic Optimizer (PDLO), performs a new type of optimization made possible only by the increased accuracy of the PAS.

The PRIC Framework allows early aggressive if-conversion to be performed early in the compilation process while delaying the delicate balancing of control flow and

predication until schedule time when final code characteristics are evident. Early if-conversion is necessary to allow optimizations access to predicated code and to be more effective.

Effective optimization, scheduling, and register allocation of code in the predicated representation require the compiler to understand the logical relationships among predicates since execution conditions are no longer solely determined by branches. The PAS extracts and presents information on these relationships in an exact manner. The accuracy of this relationship database makes possible a new mechanism called the PDLO.

Dataflow analysis requires information about both predicate flow and branch control flow in order to generate accurate results. A new structure called the Predicate Dataflow Graph accurately represents both predicate and branch execution possible outcomes while allowing traditional iterative dataflow equation solvers to be employed.

PDLO extracts the semantics of the predicate decision logic and represents it as a set of Boolean expressions. The predicate network is then optimized using standard Boolean factorization and minimization techniques. The PDLO reformulates the program decision logic back as more efficient predicate define networks.

The proposed techniques work in a synergistic manner. More aggressive if-conversion made possible by the PRIC makes predicate optimizations, such as the PDLO, more effective. The PAS, necessary to accurately support the high level of predication also makes PDLO possible by providing the Boolean expressions which completely describe the decision logic. The PDLO restructures the predicate definition network which is then, in some cases, converted into control flow by PRIC. This new control flow often does not resemble the inefficient control created by the programmer, but it is a new,

more efficient layout for the given target architecture. Predicate dataflow analysis makes efficient reverse if-conversion possible by eliminating unnecessary code duplication when a branch is reinserted. The predicate dataflow analysis also saves compiler components a loss in effectiveness which would occur as a result of aggressively if-converting code in a conservative analysis environment.

The presented set of tightly coupled systematic compilation techniques for predicated execution allows a large benefit from predication to be extracted without much risk of performance penalty originating from poor compilation decisions.

9.2 Future Research

The work presented in this dissertation motivates several promising opportunities for future research.

The PAS and PDLO only operate on predicated codes. However, both of these techniques could be useful in the context of branch-only code. The PAS could be made to extract basic block relationships directly into represented Boolean expressions. This information could then be used to perform decision logic minimization and regenerate new branch code. Extracting the expressions directly from the branches would allow the PDLO to act as a sophisticated if-converter when predicate defines are reformulated in place of branches.

As an alternative to instruction execution control, predication is quite effective at reducing the ill effects of branches. However, predication provides relief only in acyclic execution control, and loop iteration control must, at some level, still be performed by branch control flow. The application of predicate-like control to represent instruction

iteration information may be useful in enhancing scheduling and optimization in the presence of loops, as well as in the presence of acyclic branch structures. As a limitation of this work and a research opportunity, extensions along this line may facilitate the extraction of ILP by enabling scheduling and optimization techniques to be applied on a much larger scale within a program.

A system enabling the application of the PDLO and other predicate optimizations to branch code could be implemented within a complete if-converting/reverse if-converting framework. This framework would be similar to the partial reverse if-converting framework in that it would eliminate the need for control flow optimizations by converting all control flow to data flow and applying the appropriate dataflow optimizations. Combined with a mechanism to represent cyclic regions, complex control and loop transformations could be performed efficiently with more desirable results.

REFERENCES

[1] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 272–282.

[2] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on multiple instruction issue," in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 290–302.

[3] M. A. Schuette and J. P. Shen, "An instruction-level performance analysis of the Multiflow TRACE 14/300," in *Proceedings of the 24th International Workshop on Microprogramming and Microarchitecture*, November 1991, pp. 2–11.

[4] T. M. Conte, K. Menezes, P. Mills, and B. Patel, "Optimization of instruction fetch mechanisms for high issue rates," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995, pp. 333–344.

[5] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: A low latency approach to high bandwidth instruction fetching," in *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996, pp. 24–34.

[6] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th International Symposium on Computer Architecture*, May 1981, pp. 135–148.

[7] T. Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, November 1991, pp. 51–61.

[8] P. Y. Hsu and E. S. Davidson, "Highly concurrent scalar processing," in *Proceedings of the 13th International Symposium on Computer Architecture*, June 1986, pp. 386–395.

[9] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 departmental supercomputer," *IEEE Computer*, vol. 22, pp. 12–35, January 1989.

[10] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, January 1983, pp. 177–189.

[11] J. C. Park and M. S. Schlansker, "On predicated execution," Hewlett Packard Laboratories, Palo Alto, CA, Tech. Rep. HPL-91-58, May 1991.

[12] N. J. Warter, "Modulo scheduling with isomorphic control transformations," Ph.D. dissertation, University of Illinois, Urbana, IL, 1993.

[13] D. N. Pnevmatikatos and G. S. Sohi, "Guarded execution and branch prediction in dynamic ILP processors," in *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994, pp. 120–129.

[14] 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.

[15] G. S. Tyson, "The effects of predicated execution on branch prediction," in *Proceedings of the 27th International Symposium on Microarchitecture*, December 1994, pp. 196–206.

[16] 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.

[17] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, R. A. Bringmann, and W. W. Hwu, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th International Symposium on Microarchitecture*, December 1992, pp. 45–54.

[18] A. Klauster, T. Austin, D. Grunwald, and B. Calder, "Dynamic hammock predication for non-predicated instruction set architectures," in *Proceedings of the 18th Annual International Conference on Parallel Architectures and Compilation Techniques*, October 1998, pp. 278–285.

[19] D. M. Gillies, D. R. Ju, R. Johnson, and M. Schlansker, "Global predicate analysis and its application to register allocation," in *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996, pp. 114–125.

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

[21] 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.

[22] *IA-64 Application Developer's Architecture Guide*, Intel Corporation, May 1999.

[23] D. I. August, J. W. Sias, J. Puiatti, S. A. Mahlke, D. A. Connors, K. M. Crozier, and W. W. Hwu, "The program decision logic approach to predicated execution," in *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999, pp. 208–219.

[24] G. R. Beck, D. W. Yen, and T. L. Anderson, "The Cydra 5 minisupercomputer: Architecture and implementation," *The Journal of Supercomputing*, vol. 7, pp. 143–180, January 1993.

[25] J. C. Dehnert and R. A. Towle, "Compiling for the Cydra 5," *The Journal of Supercomputing*, vol. 7, pp. 181–227, January 1993.

[26] J. C. Dehnert, P. Y. Hsu, and J. P. Bratt, "Overlapped loop support in the Cydra 5," in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 26–38.

[27] A. V. Someren and C. Atack, *The ARM RISC Chip, A Programmer's Guide*. Reading, MA: Addison-Wesley Publishing Company, 1994.

[28] D. L. Weaver and T. Germond, *The SPARC Architecture Manual*, SPARC International, Inc., 1994.

[29] Digital Equipment Corporation, *Alpha Architecture Handbook*. Maynard, MA: Digital Equipment Corporation, 1992.

[30] *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, Intel Corporation, 1997.

[31] D. S. Blickstein, P. W. Craig, C. S. Davidson, R. N. Faiman, K. D. Glossop, R. B. Grove, S. O. Hobbs, and W. B. Noyce, "The GEM optimizing compiler system," *Digital Technical Journal*, vol. 4, no. 4, pp. 121–136, 1992.

[32] *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, Hewlett-Packard Company, 1990.

[33] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg, "The Multiflow Trace scheduling compiler," *The Journal of Supercomputing*, vol. 7, pp. 51–142, January 1993.

[34] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*. Reading, MA: Addison-Wesley Publishing Company, 1991.

[35] S. A. Mahlke, "Exploiting instruction level parallelism in the presence of conditional branches," Ph.D. dissertation, University of Illinois, Urbana, IL, 1995.

[36] 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 22th International Symposium on Computer Architecture*, June 1995, pp. 138–150.

[37] J. C. Gyllenhaal, "An efficient framework for performing execution-constraint-sensitive transformations that increase instruction-level parallelism," Ph.D. dissertation, University of Illinois, Urbana, IL, 1997.

[38] D. I. August, W. W. Hwu, and S. A. Mahlke, "A framework for balancing control flow and predication," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*, December 1997, pp. 92–103.

[39] N. J. Warter, S. A. Mahlke, W. W. Hwu, and B. R. Rau, "Reverse if-conversion," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, June 1993, pp. 290–299.

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

[41] R. Johnson and M. Schlansker, "Analysis techniques for predicated code," in *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996, pp. 100–113.

[42] M. S. Schlansker, S. A. Mahlke, and R. Johnson, "Control CPR: A branch height reduction optimization for EPIC architectures," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, May 1999, pp. 155–168.

[43] S. B. Akers, "Binary decision diagrams," *IEEE Transaction on Computers*, vol. C-27, pp. 509–516, June 1978.

[44] R. E. Bryant, "Symbolic Boolean manipulation with ordered binary decision diagrams," School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU-CS-92-160, October 1992.

[45] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transaction on Computers*, vol. C-35, pp. 677–691, August 1986.

[46] K. S. Brace, R. R. Rudell, and R. E. Bryant, "Efficent implementation of a BDD package," in *Proc. of the 27th ACM/IEEE Design Automation Conference*, January 1990, pp. 40–45.

[47] F. Somenzi, "CUDD: CU Decision Diagram package, release 2.30," University of Colorado at Boulder, 1998, http://vlsi.colorado.edu/~fabio/CUDD/.

[48] J. W. Sias, "Condition awareness support for predicate analysis and optimization," M.S. thesis, University of Illinois, Urbana, IL, 1999.

[49] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu, "Integrated predication and speculative execution in the IMPACT EPIC architecture," in *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998, pp. 227–237.

[50] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 319–349, July 1987.

[51] J. Knoop, O. Rüthing, and B. Steffen, "Partial dead code elimination," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementaton*, June 1994, pp. 147–158.

[52] D. J. Kuck, *The Structure of Computers and Computations.* New York, NY: John Wiley and Sons, 1978.

[53] M. Schlansker and V. Kathail, "Acceleration of first and higher order recurrences on processors with instruction level parallelism," in *Proceedings of Languages and Compilers for Parallel Computing, 6th International Workshop*, August 1993.

[54] R. Bodik, R. Gupta, and M. L. Soffa, "Interprocedural conditional branch elimination," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, June 1997, pp. 146–158.

[55] F. Mueller and D. B. Whalley, "Avoiding conditional branches by code replication," in *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995, pp. 55–66.

[56] M. Yang, G.-R. Uh, and D. B. Whalley, "Improving performance by branch reordering," in *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, June 1998, pp. 130–141.

[57] R. A. Bringmann, "Compiler-controlled speculation," Ph.D. dissertation, University of Illinois, Urbana, IL, 1995.

[58] B. L. Deitrich and W. W. Hwu, "Speculative hedge: Regulating compile-time speccu-lation against profile variations," in *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996, pp. 70–79.

[59] D. I. August, D. A. Connors, J. C. Gyllenhaal, and W. W. Hwu, "Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results," in *The 3rd International Symposium on High-Performance Computer Architecture*, February 1997, pp. 84–93.

[60] 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," *The Journal of Supercomputing*, vol. 7, pp. 229–248, January 1993.

[61] G. E. Haab, “Data dependence analysis for fortran programs in the IMPACT compiler,” M.S. thesis, University of Illinois, Urbana, IL, 1995.

[62] D. M. Gallagher, “Memory disambiguation to facilitate instruction-level parallelism compilation,” Ph.D. dissertation, University of Illinois, Urbana, IL, 1995.

[63] K. Subramanian, “Loop transformations for parallel compilers,” M.S. thesis, University of Illinois, Urbana, IL, 1993.

[64] S. Anik, “Architectural and software support for executing numerical applications on high performance computers,” Ph.D. dissertation, University of Illinois, Urbana, IL, 1993.

[65] Y. Yamada, “Data relocation and prefetching for programs with large data sets,” Ph.D. dissertation, University of Illinois, Urbana, IL, 1995.

[66] S. A. Mahlke, “Design and implementation of a portable global code optimizer,” M.S. thesis, University of Illinois, Urbana, IL, 1991.

[67] 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, pp. 353–370, March 1995.

[68] W. Y. Chen, “Data preload for superscalar and VLIW processors,” Ph.D. dissertation, University of Illinois, Urbana, IL, 1993.

[69] 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 Architectual Support for Programming Languages and Operating Systems*, October 1994, pp. 183–193.

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

[71] J. C. Gyllenhaal, “A machine description language for compilation,” M.S. thesis, University of Illinois, Urbana, IL, 1994.

[72] R. G. Ouellette, “Compiler support for SPARC architecture processors,” M.S. thesis, University of Illinois, Urbana, IL, 1994.

VITA

David Isaac August was born on November 27, 1970, in Troy, New York. After growing up in the beautiful state of New Jersey, he returned to Troy, New York, to pursue his undergraduate studies at Rensselaer Polytechnic Institute, where he received the bachelor of science degree in electrical engineering in May 1993. In the fall of 1993, he began his graduate studies at the University of Illinois. During his graduate tenure, he was a member of the Center for Reliable and High-Performance Computing and the IMPACT project directed by Professor Wen-mei Hwu. He completed the M.S. degree in electrical engineering in 1996. After completing the Ph.D. work in 1999, he joined the faculty at Princeton University in Princeton, New Jersey. At Princeton, he holds an appointment from the Department of Computer Science and is affiliated with the Department of Electrical Engineering.