UNIVERSITY OF CALIFORNIA, SAN DIEGO

Compiler and Hardware Predicated Dependency Analysis and Scheduling.

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science and Engineering

by

Lorinda Carter

Committee in charge:

Professor Bradley Calder, Chairperson
Professor J. Lawrence Carter
Professor Jeanne Ferrante
Professor Alon Orlitsky
Professor Alex Veidenbaum

2002

Copyright

Lorinda Carter, 2002

All rights reserved.

The dissertation of Lorinda Carter is approved, and it is acceptable in quality and form for publication on microfilm:

____

____

____

____

____

Chair

University of California, San Diego

2002

I dedicate this work to my family, the most important people in my life.

To my parents, George and Charlotte, who, throughout my life, have loved me and supported all I've done. Specifically, to my mom, who has been a great example to me of perseverance, dedication and strength, and in loving memory of my dad, who was convinced that I could do anything I put my mind to. To my brother, Gary, who has believed in me more than I believe in myself.

To my children, who are the greatest kids in the world. To Brad, who inspires me with his creativity and passion. To Michelle, who brings laughter to our home and demonstrates great dedication to her own schoolwork. To Janna, who has seen me as a student for half of her life, and whose sweet spirit and beautiful music provide encouragement in difficult times.

To my other parents, Edna and Percy, who I was blessed with through marriage. Thank you for endless hours of child care and many meals. Without your unending support, I would never have finished.

Finally, I dedicate this work to an amazing man, who has been at my side for over 20 years, my husband, Tom. Thank you, Tom, for introducing me to the world of computer science, and teaching me that the educational process itself is to be enjoyed, it is not just a means to an end. Thank you for always being there, offering love, support and wisdom, and for being a great father to our children.

I want to offer my great thanks to my advisor, Dr. Brad Calder, for excellent guidance, interesting discussions, and endless explanations. Thank you for being flexible with this "non-traditional" grad student. Thank you as well, to Dr. Jeanne Ferrante and Dr. Larry Carter who provided encouragement and guidance, especially in the early years.

I cannot say enough about the people I worked with for almost 6 years. First, much thanks to my friend, Glenn, who provided much wisdom, laughter and a mean tennis game. I would not be writing this without your endless encouragement. To Beth (an excellent research partner), Barbara, Chandra and Shelly, with whom I shared many wonderful times and conference hotel rooms. To John, Tim, Suleyman, Eric, Jeff and Wei for patient and insightful technical support. To Floria and Sara for their friendship. And for the many others that have added so much to my PhD experience.

Because of all these wonderful people, this has been the best of times.

Trust in the Lord with all your heart, and do not lean on your own understanding. In all your ways acknowledge Him, and He will make your paths straight

-Proverbs 3:5,6 (Holy Bible, NIV version)

TABLE OF CONTENTS

Signature Page iii

Dedication Page iv

Epigraph v

Table of Contents vi

List of Figures ix

List of Tables xii

Acknowledgments xiii

Vita, Publications, and Fields of Study xiv

Abstract xv

I Problem Statement 1

II A Brief Introduction to Predicated Execution 5
A. Predicated Execution 5
B. Instruction Set Architecture for Compiler Optimizations 9
C. Related Work 10

III Processor Pipeline Basics 12
A. In-order Execution Model 15
B. Additional Requirements for Predication 17
C. Overview of Intel Itanium 19
1. Software Pipelining 20
2. Control Speculation 21
3. ALAT Implementation 23
D. Itanium Pipeline 23

IV Predicated Static Single Assignment 26
A. Motivation for Specialized Predicate-Sensitive Data Flow Analysis 26
B. Related Work 29
C. Predicated Static Single Assignment (PSSA) 30
1. Converting to PSSA Form 33
2. Post-Optimization Clean-up 35
D. Hyperblock Scheduling Optimizations 36
1. Predicated Speculation 36
2. Control Height Reduction 38
E. Implementing PSSA in IA-64 42
F. Results 43
G. Path and Code Duplication Reduction 48
1. Renaming and Scheduling Using ϕ functions and Duplication 49
2. Full Path Predicate Creation Methodology 51
3. FPP Creation 52

4. Code Reduction . . . 57
5. A Sample Result . . . 63
H. Summary . . . 66

V IA64SimpleScalar: A Framework for Testing Hardware Optimizations . . . 69
A. EPIC Simulation Using IA64SimpleScalar . . . 70
1. ISA Implementation . . . 70
2. Predicated RAW and WAW dependences . . . 72
3. ALAT . . . 72
4. The Expand Stage . . . 73
5. Commit Stage . . . 73

VI Disjoint Path Analysis . . . 74
A. Multiple Paths Means Multiple Definitions that Result in Stalls . . . 75
1. Effect of Extraneous Definitions on the Hardware . . . 78
B. Related Work . . . 79
1. Predicated Multiple Path Compiler Analysis . . . 79
2. Hardware Solutions for Dealing with Multiple Path Definitions . . . 80
C. Disjoint Path Analysis Architecture . . . 82
1. Register Alias Table . . . 83
2. Predicate Information Table and Last Definition Table . . . 84
3. Using the PIT and RAT to Determine Actual Dependences . . . 87
4. Updating the RAT . . . 90
5. Updating the PIT . . . 90
6. Predicates Defined False . . . 91
D. Methodology . . . 91
E. In-Order Issue with Scoreboard . . . 92
1. Results . . . 95
F. Conclusions . . . 97

VII Pending Functional Units . . . 99
A. Out-of-Order Execution Model . . . 100
B. Introduction to Pending Functional Units . . . 103
C. Baseline In-order Processor . . . 103
1. Hardware Simplicity via Stop Bits . . . 104
2. Compiler Exposing ILP . . . 104
3. EPIC In-Order Pipeline . . . 106
4. Bypass Mechanism . . . 106
D. Methodology . . . 107
E. Pending Functional Units . . . 108
1. Limited Out-Of-Order Execution Via Functional Units . . . 109
2. Instruction Scheduling . . . 110
3. Managing Speculative State . . . 111
4. Summary of PFU Configurations Examined . . . 112
5. Results . . . 112
F. Conclusions . . . 116

VIII Conclusions . . . 118

IX Future Directions . . . 121

Bibliography 122

LIST OF FIGURES

II.1	Short code example showing the transformation from the original code in control flow graph form to one if-converted block.	6
II.2	Probable schedules for the traditional and if-converted code from the previous figure.	7
III.1	Basic Processor Pipeline	12
III.2	Pipelined execution with the addition of bypassing	14
III.3	A bypassing mechanism that supports predication.	17
III.4	Basic Itanium Pipeline	23
IV.1	Renamed version of previous control flow graph example prior to the join. Eliminates false dependence between 2 definitions of b (now called b1 and b2)	27
IV.2	Code is duplicated when more than one definition reaches a use. The statement y=t+r is duplicated, but it is difficult to determine on what to predicate the duplicates. Figure (c) shows that path information that would need to be represented by an appropriate guarding predicate for each duplicate	28
IV.3	Extended example of transformation from non-predicated CFG to predicated hyperblock.	31
IV.4	The PSSA dependence graph shows the flow of data and control through the PSSA-transformed code. Blocks labeled with *full-path* predicates (indicated by multiple letters) contain statements that are only executed along that path. Blocks labeled with *block* predicates (single letters) contain statements that will be executed along several paths.	32
IV.5	Extended code example after PSpec optimization has been applied. Statements (other than first statement) predicated on true have been speculated.	37
IV.6	Basic PSpec Algorithm.	37
IV.7	Extended example after PSpec and CHR optimizations have been applied. `Cmpp` instructions displayed in italics define predicates that are not used after optimization. Therefore, the statements can be removed from the final code.	40
IV.8	Dependence graph after PSpec and CHR have been applied.	41
IV.9	Basic Control Height Reduction Algorithm.	41
IV.10	Executed cycles normalized to the number of cycles to execute the original code produced by Trimaran for a 16 issue machine.	43
IV.11	Weighted average number of operations scheduled per cycle for hyperblocks when using PSSA with Predicated Speculation and Control Height Reduction.	45
IV.12	Weighted average register pressure in hyperblocks when using PSSA with Predicated Speculation and Control Height Reduction. Shown from left to right for each benchmark is the general purpose file, predicate file, branch file, and floating point file (zero utilization for some benchmarks).	46
IV.13	Static and Dynamic Code Bloat normalized to original code size.	47
IV.14	A simple example of a control flow graph and related predicated code. The included schedule follows all data and control dependence constraints.	49
IV.15	Renaming is added to the original example to allow for speculation without side-effects. Phi functions are added to multiple versions of a variable that may reach a use.	50

IV.16 Renaming is again added to enable speculation, but multiple versions of variables are reconciled by duplicating the operation for each set of operand combinations. Additional statements required for this method are shown in italics in the code. Italicized labels in the CFG are there to help show correspondence between the code and the CFG. 50
IV.17 Main code example that will be used for the rest of the chapter. This Figure depicts only 2 iterations of the 8 that are actually unrolled and predicated by Trimaran. Note that P indicates predicate registers, B indicates branch registers and r indicates that the value is being placed into an integer register. The dotted brackets in (c) represent the ranges of block predicates to be added in the process of transforming the predicated code into Full Path predicated code. 53
IV.18 Main code example showing one iteration+ that has been FPP-transformed and duplicated. This figure shows the beginning of the second iteration as well. 56
IV.19 Sample control flow graph and associated dependence graph showing how the critical path is determined. 59
IV.20 Code and schedule that could be created where the critical path is optimized. The schedule for the unoptimized code is included for comparison. 60
IV.21 Scheduling Algorithm. 62
IV.22 Structure of original code before and after speculation. Only 2 statements per iteration were speculated. 64
IV.23 Structure of FPP transformed code before and after speculation. The large number of statements at the top of (b) shows the ability of FPPs to support speculation. 65
IV.24 Schedules that were created using predicated (b) and Full Path predicated code (a). 67

VI.1 Original Control Flow Graph 75
VI.2 If Converted Version 76
VI.3 If Converted showing multiple definitions of same register 76
VI.4 If Converted showing renamed definitions of same register 80
VI.5 Augmented Register Alias Table used to implement the select-μop optimization for out-of-order processors supporting predicated execution. 81
VI.6 Three tables shown after first `cmp` statement is processed. One definition of `r5` has been entered into the RAT. As is it unguarded, there is no PIT entry associated with it. In the PIT, the bits are set at the intersections of locations 0 and 1 indicating that predicate registers `P2` and `P3` are disjoint. The LDT indicates that the current definition of predicate register 2 is found in PIT entry 0, and predicate register 3 at PIT entry 1. 83
VI.7 Maximum and average number of PIT entries required per cycle to support references in the RAT. 85
VI.8 Three tables after second `cmp` statement is processed. Two new definitions have been added to the RAT and the LDT. In the PIT, the definitions of `P4` and `P5` are guarded by `P3`, so the disjointness information for `P3` is inherited by `P4` and `P5`. 89
VI.9 Three tables after third `cmp` statement is processed. A new definition of P5 is made and given its own disjointness information. The old definition of `P5` in the LDT is replaced. Three new definitions are added to the RAT. 89
VI.10 Captures the percent of improvement seen by the path optimization verses what could have been achieved by way of perfect predicate prediction 94

VI.11 Improvement in IPC for Disjoint Path Analysis over Itanium Implementation in if-converted regions. 94
VI.12 Percent of dependences removed by Disjoint Path Analysis in if-converted regions. RAW represents Read After Write dependences while WAW represents Write After Write dependences 96
VI.13 Percent of dependences removed by Disjoint Path Analysis in if-converted regions. Results are give for Path Analysis for PITs with 4 slots. 96

VII.1 Basic Itanium Pipeline 105
VII.2 Shows how EPIC and PFU models would disperse instructions under normal conditions, and when the first `ld` instruction misses in the L1 cache (but hits in the L2), causing the final `sub` instruction to stall on the use of the loaded value. Whenever the dispersal cycle changes, a reason is provided. SB-stop bit encountered. WAW- Write after Write hazard. BUN-this is the third bundle to be considered for dispersal (instructions may only come from 2 in the EPIC model). STALL-in response to the pipeline backing up after the first load stalls. 105
VII.3 The Itanium bypassing mechanism that supports predication. 106
VII.4 Comparing the IPCs of the various PFU models and the out-of-order models with issue buffer sizes of 9 and 64 to the EPIC model. 114

LIST OF TABLES

II.1 Destination action specifiers for cmpp operations and their semantics found in the Playdoh Architecture. For the first tag character, u=unconditional, c=conditional, o=or, a=and. For the second tag character, n=normal and c=complement. An entry with - means to leave the result unchanged. 9

IV.1 Presents description of benchmarks simulated including instruction count in millions where the initialization phase of the execution ended and where the trace began recording. 44

VI.1 Presents description of benchmarks simulated including instruction count, percent of dynamic if-converted instructions and baseline IPCs for in-order and out-of-order execution. 92

VI.2 Baseline Simulation Model created to correspond the the parameters set by the Intel Itanium. 93

VII.1 Presents description of benchmarks simulated including instruction count in millions where the initialization phase of the execution ended and where the trace began recording. 108

VII.2 Baseline Simulation Model. 108

VII.3 Presents misses per 1K instructions in the various caches. Levels 2 and 3 are unified caches. The last column shows the branch prediction accuracy. 114

Acknowledgments

The text of Chapter IV is in part a reprint of the material as it appears in the proceedings of the 2000 International Journal of Parallel Programming. The dissertation author was a co-primary researcher and author (with Beth Simon), and the other co-authors listed on this publication ([18]) directed and supervised the research which forms the basis for Chapter IV.

The text of Chapter VII is in part a reprint of the material as it is to appear in the proceedings of the Fourth International Symposium on High Performance Computing, May 2002. The dissertation author was the primary researcher and author (assisted by Weihaw Chuang) and the other co-author listed on this publication ([17]) directed and supervised the research which forms the basis for Chapter VII.

VITA

June 23, 1957	Born Pasadena, CA
1975	High School Diploma, Herbert Hoover High School Glendale, CA
1979	B.A., Point Loma College San Diego, CA
1984	M.S., California State University Northridge, CA
1984-1987	Lecturer, Palomar College San Marcos, CA
1987-1996	Instructor, Point Loma Nazarene University San Diego, CA
1996-1999	Teaching Assistant, Computer Science and Engineering Department, University of California San Diego, CA
1998-2000	Senior Teaching Assistant, Computer Science and Engineering Department, University of California San Diego, CA
2002	Doctor of Philosophy, University of California San Diego, CA

PUBLICATIONS

"An EPIC Processor with Pending Functional Units". Authors: L. Carter, W. Chuang, B. Calder. To appear in the Proceedings of the Fourth International Symposium on High Performance Computing, May 2002.

"Path Analysis and Renaming for Predicated Instruction Scheduling." Authors: L. Carter, B. Simon, L. Carter, B. Calder, J. Ferrante. In the International Journal of Parallel Programming, Dec. 2000.

"Predicated Static Single Assignment." Authors: L. Carter, B. Simon, L. Carter, B. Calder, J. Ferrante. In the Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), Oct. 1999.

Field Of Study: Computer Science

ABSTRACT OF THE DISSERTATION

Compiler and Hardware Predicated Dependency Analysis and Scheduling

by

Lorinda Carter

Doctor of Philosophy in Computer Science and Engineering

University of California, San Diego, 2002

Professor Bradley Calder, Chair

The Explicitly Parallel Instruction Computing (EPIC) architecture has been put forth as a viable architecture for achieving the instruction level parallelism (ILP) needed to keep increasing future processor performance. The Itanium processor developed at Intel is an example of an EPIC architecture.

One of the new features of the EPIC architecture is its support for *predicated execution*. Predicated execution is a process that can replace branches with statements defining 2 predicate registers (one true and one false), depending on the condition in the replaced branch. Subsequent statements are then guarded by one of the predicates, depending upon whether they would have been on the taken or fall-through path of the branch. All statements begin execution, but an operation is committed only if the value of its guarding predicate is true.

An advantage of predicated execution is that it can eliminate hard-to-predict branches by combining both paths of a branch into a single path. However, data dependence analysis (for the purpose of maintaining definition-use information) is significantly more complex for the resulting code. When the two paths of a branch are combined, definitions of the same logical registers (originally from different paths) are intermingled. This makes it difficult to determine which definition a use is actually dependent on. This dissertation presents both hardware (Disjoint Path Analysis) and compiler (Predicated Static Single Assignment) solutions for improving the data dependence analysis for predicated regions of code by collecting information on predicate relationships.

Another feature of the EPIC architecture is the reduced hardware complexity. The EPIC philosophy is that the compiler should handle most of the dependence analysis and scheduling in order to simplify the processor, and at the same time the compiler has a broader view of the code. However, the compiler cannot fully anticipate run-time events such as cache misses. Consquently, it cannot always create a static schedule to mitigate the effects of the increased latency that might result. In this dissertation, we introduce Pending Functional

Units (PFU) which allow a limited amount of dynamic scheduling with minimal additional hardware overhead.

Chapter I

Problem Statement

The Explicitly Parallel Instruction Computing (EPIC) architecture has been put forth as a viable architecture for achieving the *instruction level parallelism* (ILP) needed to keep increasing future processor performance [12, 31]. The Itanium [1, 23] processor developed at Intel is an example of an EPIC architecture. An EPIC architecture issues wide instructions, similar to a VLIW architecture, where each instruction contains many operations.

One of the new features of the EPIC architecture is its support for *predicated execution* [41]. The most general form of predicated execution involves a process called *if-conversion* whereby the compiler can replace branches with statements defining 2 predicate registers (one true and one false), depending on the condition in the replaced branch. Subsequent statements are then guarded by one of the predicates, depending upon whether they would have been on the taken or fall-through path of the branch. If they would have been on neither path, they remain unguarded. All if-converted statements begin execution, but an operation is committed only if the value of its guarding predicate is true.

One advantage of predicated execution is that it can eliminate hard-to-predict branches by allowing both paths of a branch to be executed as a single path. Another advantage is that the use of predication allows several smaller basic blocks to be combined into one larger hyperblock [38]. This provides a larger pool from which to draw ILP for EPIC architectures.

The addition of predicated execution to an architecture introduces additional challenges as well, two of which will be addressed in this dissertation. First, data dependence analysis is significantly more complex in predicated regions. Additional data dependences are created when branches are replaced by predicate-defining statements. A statement guarded by

a predicate register is dependent upon the statement defining the predicate. Furthermore, when the two paths of a branch are combined, definitions of the same logical registers (originally from different paths) are intermingled. This makes it difficult to determine which definition a use is actually dependent on.

The second challenge exists because the inclusion of predicated execution in an ISA requires significant overhead in terms of analysis and logic. Consequently, the current implementations are VLIW, involving complex compilers, but allowing the hardware to remain relatively simple. This means that execution schedules are created statically without the benefit of runtime information such as cache activity. Execution, for example, cannot continue past an instruction with an outstanding dependency on a load that misses in the cache. This could create severe performance limits with regard to Instructions Per Cycle (IPC).

We address the data analysis problem first with compiler optimizations. We introduce Predicated Static Single Assignment (PSSA), a transformation based on Static Single Assignment (SSA) [25]. PSSA removes false dependences by exploiting renaming and information about the multiple control paths. We demonstrate the usefulness of PSSA for Predicated Speculation and Control Height Reduction. These two predicated code optimizations used during instruction scheduling reduce the dependence length of the critical paths through a predicated region. Our results show that using PSSA to enable speculation and control height reduction in an environment of unlimited resources reduces execution time from 10% to 58%.

Unfortunately, the unlimited application of PSSA causes a tremendous amount of code expansion. In real world applications, the resources are not unlimited. In an effort to reduce the amount of code duplication required by PSSA, we describe a sub-region formation algorithm which partitions the original predicated region and applies PSSA over these smaller regions. In addition, we define a critical-path-first scheduling algorithm which ensures that instructions along the critical path get first priority for the limited resources available. Results show that assigning full-path predicates across subregions of the hyperblock and using the critical path first scheduling algorithm we can get an improvement in IPC of up to 33% over the unoptimized code schedule.

In predicated code, actual dependences only exist between two statements that are both guarded by a predicate with the value of true. During execution, dependences will be broken if either the producer or consumer instruction is determined to be guarded by a false predicate. Consequently, it is not enough for dependency analysis to occur in the compiler. Dependency information must be calculated and maintained by the hardware as well. The

Itanium uses a scoreboard to maintain current dependency information. Predicate-sensitive dependency analysis would be useful to the hardware in addition to the compiler.

The second optimization we designed to address the data analysis problem is implemented in hardware. This system dynamically collects and analyzes predicate relationship information, providing insight into which definitions could actually be on the path to a particular use of the register. Because this system is updated at run-time, information on predicate values is available. No dependence between a producing and consuming operation must be set if the guarding predicate of either is false. The use of the Disjoint Path Analysis system on an architecture modeled after the Intel Itanium improved IPC up to 6% in predicated regions for the benchmarks we tested.

As mentioned, for EPIC architectures, the compiler is responsible for arranging the instructions into a schedule based on data dependences and instruction latencies. The instructions are executed in the order received. The pipeline must stall if the dependences of the next instruction to be executed are not resolved.

The benefit of a dynamically created schedule, such as is found on traditional out-of-order architectures, is that instructions can be scheduled to minimize the effects of occurrences such as cache misses. However, these out-of-order architectures are generally associated with complex logic and hardware, often increasing the cycle time of the machine. We introduce Pending Functional Units (PFU) which provide a small window of instructions that can be executed out of order, and require only a minimal amount of changes to the VLIW architecture. We examine 3 configurations using PFUs to determine how much of the increased performance of out-of-order architectures can be obtained without adding all of the additional complexity. Our results show that the use of PFUs can improve overall performance by up to 27% over the EPIC configuration, which is 37% of the improvement shown for complete out-of-order machines. This improvement is gained without needing register renaming or a complex out-of-order scheduling algorithm applied to a large issue window.

The rest of this dissertation is organized as follows. Chapter II introduces predicated execution, presenting the motivation behind using the technique, the actual predication process and some of the complexities associated with executing predicated code. In addition, a description of the ISA used for the compiler optimizations and of prior work in this area is included. Chapter III describes the basic background on processor pipelines, and the Itanium pipeline in particular, required to support future chapters. Chapter IV motivates the need for predicate-sensitive data flow analysis and describes Predicated Static Single Assignment

(PSSA). It includes examples of how PSSA can be used to facilitate the removal of false data dependences through renaming, instruction speculation and control height reduction. In addition, it introduces a method for reducing the substantial code expansion associated with PSSA while maintaining many of the benefits.

The majority of the second half of this dissertation deals with hardware optimizations relating to EPIC architectures. Chapter V includes a discussion of the implementation of Simplescalar IA64, the infrastructure used for the hardware experiments. Chapter VI describes the Disjoint Path Analysis Architecture which allows run-time information on predicate relationships and values to reduce the number of definitions that must be considered for each use of a register. Chapter VII examines Pending Functional Units as a way to mitigate the effects of cache misses by allowing a minimal amount of out-of-order execution on an otherwise in-order machine. The final chapters of this dissertation provide a summary of the material presented and some ideas for future research.

Chapter II

A Brief Introduction to Predicated Execution

An architecture that includes predicate registers can make use of them in many different ways. The Intel Itanium uses predicate registers to make software pipelining more efficient [15]. Combined with rotating register technology, predicates can be used to reduce the need for unrolling and separate code blocks for prolog and epilog sections. Chapter V discusses software pipelining and rotating registers in more detail. The IA64 ISA includes instructions that define both a predicate register and a floating point register. The value of the predicate register is used to give information to future instructions as to the value of the floating point register [5]. This dissertation, however, focuses on predication used for *If-Conversion* [9, 41]. In this chapter, we present an introduction to Predicated Execution, followed by a description of the Instruction Set Architecture (ISA) used for the compiler based optimizations, and finally a discussion of related work.

II.A Predicated Execution

One of the major limitations on processor performance is hard-to-predict branches. When a branch is encountered in the instruction stream, the pipeline must either stall until the target address of the branch is determined so that the next instruction can be fetched, or a prediction must be made as to what the target address will be. If the next instruction address is predicted, instruction fetching will continue and a penalty will be paid only if the prediction

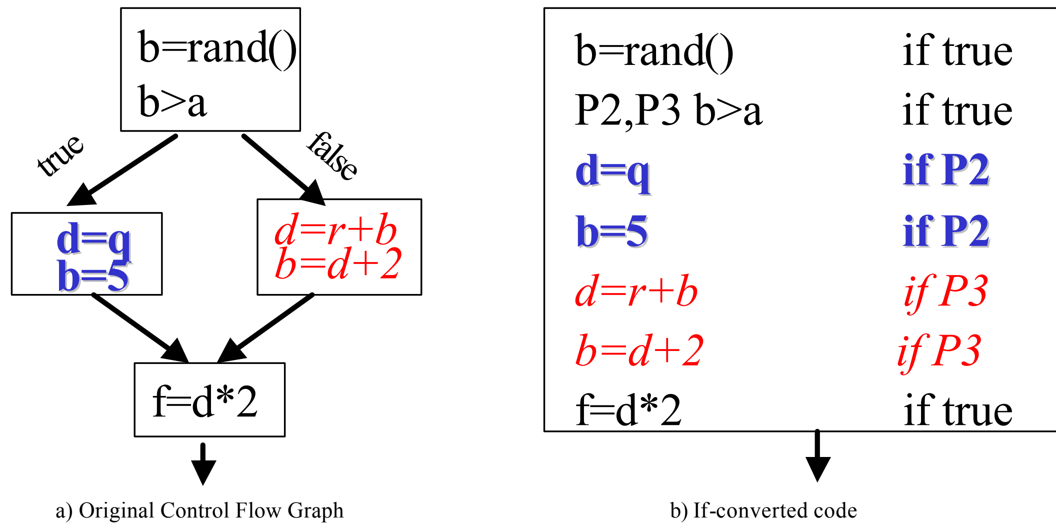
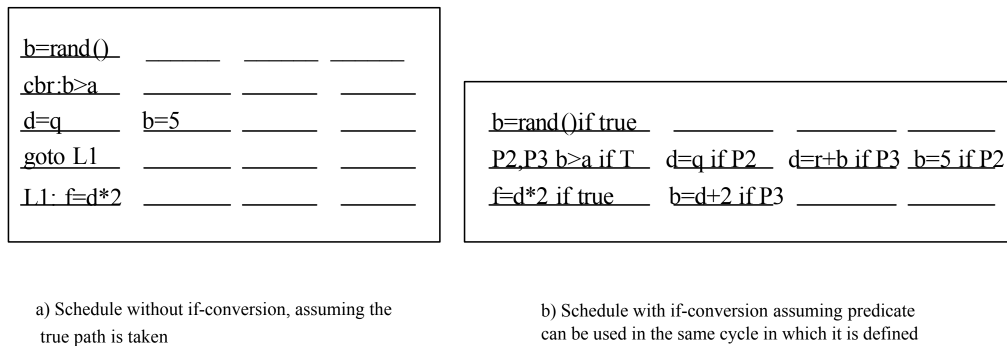


Figure II.1: Short code example showing the transformation from the original code in control flow graph form to one if-converted block.

was incorrect. Fortunately, branch prediction mechanisms are becoming quite accurate. There remain, however, branches that are highly unpredictable, and inaccurate branch prediction can have a negative affect on performance [37]. With instruction pipelines getting deeper, the penalty for mis-prediction becomes more substantial. Many instructions may have entered the pipeline between the time the branch was encountered, and the outcome of the branch determined. These instructions must be removed, and the state of the machine returned to its condition at the time of the branch. The correct next instruction must then be fetched and the pipeline restarted.

Finding sufficient *Instruction Level Parallelism* (ILP) is another limiting factor on processor performance. Multiple issue machines were created to allow many independent instructions to begin execution at once. Finding enough instructions that are independent of each other to take advantage of these architectures can be difficult.

Predicated execution is a feature designed to increase ILP and remove hard-to-predict branches. Machines with hardware to support predicated code include an additional set of registers called predicate registers. The process of if-conversion replaces branches with compare operations that set predicate registers to either true or false based on the comparison in the original branch. A simple example is presented in Figure II.1. Figure II.1(b) shows that the branch found in the original control flow graph in Figure II.1(a) has been replaced with a statement using the condition of the branch that defines predicates `P2` and `P3`. `P2` represents



a) Schedule without if-conversion, assuming the true path is taken

b) Schedule with if-conversion assuming predicate can be used in the same cycle in which it is defined

Figure II.2: Probable schedules for the traditional and if-converted code from the previous figure.

the taken path, and will be set to true if `b > a`. In this case, `P3` will then be set to false. The predicate registers will be given the opposite values if the condition is false. Predicates `P2` and `P3` are called *disjoint* predicates because they will never have the same value.

Each operation that would have been on either the taken or fall-through path is associated with the appropriate predicate register. This will be called the operation's *guarding predicate*. The statements in Figure II.1 shown in bold and guarded by `P2` are from the taken path, while the statement shown in italics and guarded by `P3` is from the fall-through path. Every operation guarded by the constant *true* will execute as usual, and every operation guarded by a predicate register will begin execution, but these operations will be committed only if their guarding predicates get assigned the value *true*. Clearly, the need to predict or wait for the calculation of the branch target is eliminated.

As can be seen from the if-converted code, removing branches can have the effect of combining many smaller basic blocks into one larger scheduling region, increasing the chances of finding independent instructions that can be scheduled in parallel. Once if-converted, both paths of a branch will be combined into one path, and included in one scheduling region. Blocks reached by both paths of the original branch can be included in the region as well. Figure II.2 (a) shows the scheduled code (from Figure II.1) before if-conversion, assuming a 4-wide issue capability and that the taken path was predicted. Only `d=q` and `b=5` could be scheduled in parallel. Many of the issue slots remained unused. Figure II.2(b) shows the schedule of the if-converted code. In the example shown, as in our compiler work, we use the philosophy that it is reasonable to schedule the operations defining predicates in the same cycle in which the predicate registers are used. This is because the predicate values are not needed until the using instructions are committed. However, our hardware work is modeled after the Itanium

implementation which requires that uses of predicate registers be scheduled in cycles after the definitions of these predicates. In either case, it is reasonable to schedule the definitions of `d` (from both paths) in the same cycle, as they are guarded by disjoint predicates. Only one of the definitions will actually be completed.

This dissertation includes optimizations for predicated code that can be accomplished in the compiler as well a optimizations meant for the hardware. The compiler work uses the notion of a hyperblock [38] for choosing which smaller blocks are to be included in the larger if-converted region. The predicated regions used for the hardware optimizations are formed in a similar manner.

A *hyperblock* is a predicated region of code consisting of a group of basic blocks with one entry point and possibly multiple branch points. Branches with both targets in the hyperblock are eliminated and converted to predicate definitions using if-conversion. All remaining branches have targets outside the hyperblock. Consequently, there are no cyclic control-flow or data-flow dependences within the hyperblock. The selection of basic blocks to be included in the hyperblock is based on program profiling which includes information such as execution frequency, basic block size, operation latencies, and other characteristics. A typical code section to include in a hyperblock is one that contains a hard-to-predict (unbiased) branch [37].

As mentioned, two of the benefits of predication are the elimination of hard to predict branches, and possibilities for increased ILP. In addition, it has been shown that increasing if-conversion reduces the static size of the program binary [20] for targets designed to support predication. As is illustrated in Figure II.2, removing branches also removes the associated labels and the higher ILP afforded by predication reduces the need to pad the wide instructions scheduled for VLIW machines with NOPs.

However, predicated code presents additional challenges as well. While the static instruction binaries are smaller, the dynamic number of instructions can be significantly larger. Instructions from both paths of a branch are sent through the pipeline. Although, as shown in Figure II.2(b), the additional instructions may not appear to add cycles to the schedule, they may have other adverse affects such as an increase in instruction cache misses. In addition, data analysis can be more difficult. This difficulty will be main topic discussed in the remainder of this dissertation.

pred input	comparison result	.un	.cn	.on	.an	.uc	.cc	.oc	.ac
0	0	0	-	-	-	0	-	-	-
0	1	0	-	-	-	0	-	-	-
1	0	0	0	-	0	1	1	1	0
1	1	1	1	1	-	0	0	-	0

Table II.1: Destination action specifiers for cmpp operations and their semantics found in the Playdoh Architecture. For the first tag character, u=unconditional, c=conditional, o=or, a=and. For the second tag character, n=normal and c=complement. An entry with - means to leave the result unchanged.

II.B Instruction Set Architecture for Compiler Optimizations

The compiler-based optimizations discussed in the next few chapters were implemented using the Trimaran System (Version 2.00) [2]. This system provides an integrated infrastructure for compilation and performance monitoring, including a parameterized processor architecture. For our experiments with limited resources, our architecture was defined as having 8 integer functional units and 2 branch units. All operations were given a latency of 1, with the exception of the MULT operation which has a latency of 3. The Trimaran System supports EPIC computing via the Playdoh ISA [34]). Examples in Chapter IV use the syntax of this ISA. In what follows, we describe three types of Playdoh operations that can be included in a hyperblock – `cmpp` operations, the predicate OR operation, and normal (non-predicate-defining) operations.

As defined in the Trimaran System, guarding predicates are assigned their values via `cmpp` operations [12]. Consider as an example the following operation:

```
B,C cmpp.un.ac a>c if A
```

The `cmpp` operation can define one or two predicates. This operation will define predicates `B` and `C`. The first tag (`.un`) applies to the definition of the first predicate `B` and the second tag (`.ac`) to `C`. The first character of a tag defines how the predicate is to be defined. The character `u` means that the predicate will unconditionally get a value, whether the guarding predicate (`A` in this case) is true or false. If `A` is false, then `B` is set to false. Otherwise, `A` is true and the value of `B` depends upon the evaluation of `a>c`.

The character `a` in the second tag (`.ac`) indicates that the full definition of the related

predicate `C` is contingent on the value of `A`, the evaluation of `a>c`, and the prior value of `C`. If `A` is false, the value of predicate `C` does not change. If `A` is true and either `C` or `a>c` evaluate to false, the new value of `C` will be false. The second character of a tag defines whether the normal (`n`) result of the condition (`a>c`) or the complement (`c`) of the condition must be true to make the related predicate true. The complete semantics for the `cmpp` operation is shown in Table II.1.

In our implementation of our optimizations, we use a new OR operation currently not defined by Trimaran. The *predicate OR* operation defines block predicates by taking the logical OR of multiple predicates. For example, consider this operation:

```
G = OR(A, B, C) if true
```

`A, B` and `C` are predicates, each defining a unique path to `G`. If any one of them has the value of true, `G` will receive a value of true, otherwise `G` will be assigned false.

When scheduling, we assume that the definition of a predicate is available for use as a source for another operation or as a guard to a subsequent `cmpp` operation in the cycle following its definition. When used as a guard for all other operations, the predicate definition is available for use in the same cycle as it is defined.

We refer to all other operations, which do not define predicates, as *normal* operations. Normal operations include assignments, arithmetic operations, branches, and memory operations.

II.C Related Work

Predicated Execution, in some form, has been in existence for many years. Many architectures support partial predication with conditional moves or select operations. The Cydra 5 [43] minisupercomputer had support for full predication prior to 1990. Mahlke [36] provides a comparison of the two approaches determining that full predication produces enough improvement in performance gain to warrant the required architectural changes.

Predicated Execution has been used for many purposes. Allen [9] shows the importance of predicated execution for removing control dependences via if-conversion. Mahlke et. al. [37] showed that if-conversion can be used to remove an average of 27% of the executed branches and 56% of the branch mispredictions. Tyson also found similar results and correlated the relationship between predication and branch prediction [55]. Park [41] provides a good explanation of the process of if-conversion, and demonstrates its use for modulo scheduling.

Gupta [30] shows the benefits of using predicated execution to enable dead code elimination.

Control speculation is another method of removing control dependences. Predication can be used in conjunction with speculation to reduce the need for recovery code on a mis-speculation. August [11] discusses how predication can be integrated with control and data speculation efficiently. Mantripragada [39] discussed the effects of different predication models on speculation.

Control height reduction is a tool that can be used to decrease branch dependence height. Schlansker [48, 47, 46] presents several techniques for reducing branch height along single paths including superblocks. In [48] he uses fully-resolved predicates to parallelize branches, allowing more flexible ordering of branches during scheduling. This method is used to reduce dependence height along critical paths.

In an effort to relieve some of the difficulties related to applying compiler techniques to predicated code, Mahlke et. al. [38] defined the hyperblock as a single-entry, multiple-exit structure to help support effective predicated compilation. The success of predicated execution can depend greatly on the region of the code selected to be included in the predicated hyperblock. August et. al. [13] relates the pitfalls and potentials of hyperblock formation heuristics that can be used to guide the inclusion or exclusion of paths in a hyperblock. Warter et. al. [57] explore the use of reverse-if-conversion for exposing scheduling opportunities in architectures lacking support for predicated execution as well as for re-forming hyperblocks to increase efficiency for predicated code [13, 57].

Chapter III

Processor Pipeline Basics

All processor pipelines must perform the same basic tasks. A general pipeline is shown in Figure III.1. In this pipeline, we have 5 stages as follows:

- Fetch - Next instruction is determined and is fetched from memory.
- Decode - Instruction is parsed to determine operands, operation and dependences. The instruction waits in this stage until the conditions are right to execute.
- Issue/Execute - When the conditions are right (values for all operands are ready and a functional unit is available), the instruction is sent to the functional unit to begin execution. Effective addresses for memory operations and branch targets for branch instructions are calculated.
- Memory and Branch Completion - Memory written to or read from for memory operations, and Program Counter updated for Branch Instructions.

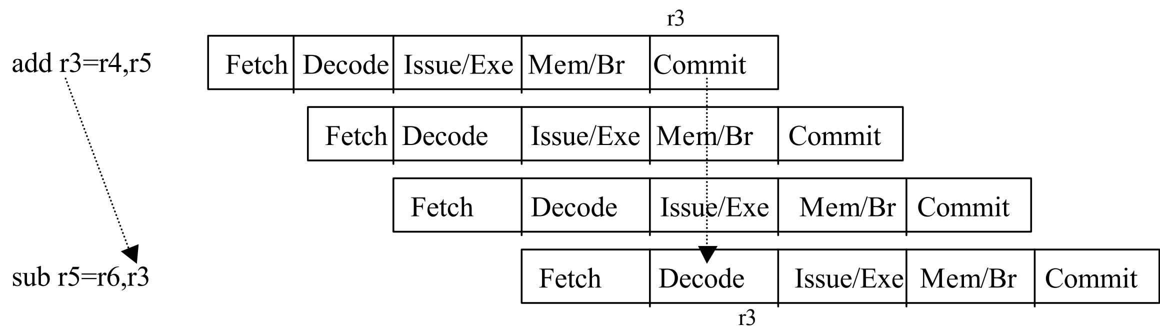


Figure III.1: Basic Processor Pipeline

- Commit - Results written to registers

When the pipeline is full, there may be a different instruction in each stage of the pipeline simultaneously. In this example, the add instruction is shown writing to register `r3` in the `commit` stage of the pipeline. The `sub` instruction is in the `decode` stage, where the value of `r3` is now available to be used for the operation.

In addition, all processor pipelines must deal with the same constraints, or hazards. Three basic categories of hazards exist: control, structural and data. Control hazards refer to instructions that cause the instruction execution order to be other than sequential. Examples of such instructions are subroutine calls and branches. When a branch instruction is encountered, for example, the outcome (taken or not taken) and the address of the next instruction to execute may not be known for several cycles after the branch has started execution. Consequently, the unoptimized pipeline must wait several cycles before it knows where to fetch the next instruction.

Structural hazards are the result of insufficient resources. For example, if there are 3 add instructions ready to execute, and only 2 adders in the processor, the third instruction will have to wait until resources are available. Similarly, if there is only one write port available, yet 2 instructions have completed and are ready to write to the register file, one instruction will have to wait, again stalling the pipeline.

Data hazards exist when the relative read/write order of instructions change due to occurrences such as pipeline stalls, or long latency instructions. The following example is used to demonstrate the kinds of data hazards that can exist. Consider this sequence of instructions scheduled for the cycles indicated:

```
cycle
  1  ld r1=[r3]
  2  sub r3=r2,r1
  3  div r1=r4,r5
  4  add r2=r1,r3
```

Although the `add` instruction should use the value of `r1` created by the `divide` instruction, divide instructions tend to have long latencies. In other words, the `divide` instruction might not finish before the `add` instruction is ready for the value. If the `add` instruction were allowed to just take the value that was currently in `r1`, incorrect program execution would result.

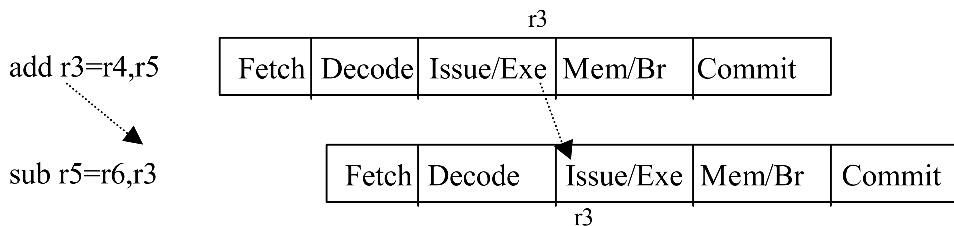


Figure III.2: Pipelined execution with the addition of bypassing

Somehow, this RAW (read after write) dependence between the `divide` and `add` instructions must be recognized and accommodated. In the case of the in-order pipeline described below, there are two alternatives. Either the static scheduler must schedule the `add` to begin execution enough cycles after the `divide` that the result of the `divide` is certain to be the one the `add` reads from `r1`, or the hardware must recognize the dependence and stall the pipeline, keeping the `add` from beginning execution until after the `divide` has completed execution. A third alternative will be discussed when out-of-order pipelines are described in Chapter V.

There are 2 other kinds of data dependences. One is a Write After Read (WAR) dependence. In our example, the `load` must read the value of `r3` before `r3` is written to by the subtract instruction. Finally, there is the possibility of the Write After Write (WAW) dependence. This dependence exists between the `load` and `divide` instructions. The `divide` must write to `r1` after the load instruction does.

Potential data hazards can be decreased when the pipeline includes the hardware and logic to implement bypassing. Our implementation of the base out-of-order model also includes a *bypassing* mechanism in the pipeline. Bypassing recognizes the fact that results are available to instructions before the results are written, and not needed by a subsequent instruction until that instruction is ready to execute.

Figure III.1 shows how the `add` and `sub` instructions would have to be scheduled without bypassing. Without bypassing capabilities, assuming a register can be written to in the first part of a cycle and read from in the second part and that the `add` takes one cycle to execute, the `add` instruction and `sub` instruction would have to be scheduled 3 cycles apart. The `sub` instruction can only get the value of `r3` from the register file, so it must wait until the result of the `add` is written to the register file in Commit.

Figure III.2 shows the possible schedule with bypassing capabilities. The value that will eventually be written to `r3` is available right after the `add` operation completes. With some extra hardware to store the values after execution and make them available to future

instructions, the subtract can proceed to the execution stage assuming that the value of `rs` will be ready at the beginning of it's execution.

The sections that follow present the essentials of execution and hazard detection. First, we describe the in-order execution model in general. Next, we discuss how the presence of predication affects this model. Finally we provide an overview of a specific in-order architecture, the Intel Itanium.

III.A In-order Execution Model

In the case of the pure in-order model, the compiler dictates the order in which the instructions execute. This model is based on the concept that the compiler has more time for analysis and a larger window of instructions to consider when choosing the best one to schedule next. The instructions are fed to the architecture arranged such that all dependences are taken into consideration. In the case of the RAW dependence between the `divide` and `add` in our earlier example, the compiler would either find enough independent instructions from other parts of the program to insert between the two dependent instructions to make the timing correct, or simply insert NOPs to force the `add` to wait until the correct operand value was ready. If the hardware is equipped with a scoreboarding mechanism, explicit NOPs are unnecessary. Instead, the hardware can cause the processor to stall until the definition required by the use has been produced. There are however, several situations that can be troublesome for static schedulers.

- **Cache miss on a load instruction** - Occasionally, a load instruction will take longer to process than expected. Ideally, the value to be loaded is located in level 1 cache. Therefore, the compiler will usually schedule for a cache *hit*. However, if the load misses in the cache instead, the correct value may not be ready for any dependent instructions. In other words, the *latency* (time until the result is ready) of a load may not be predictable.

  For the pure in-order model, a stall must occur on a cache miss. There are two possibilities in this situation. Either the whole pipeline must stall, or the front end of the pipeline must stall if an instruction has a dependency on a load that hasn't finished executing. When a stall occurs, a series of NOP instructions are sent through the pipeline until the correct value is retrieved from memory. Once the load has been serviced, the pipeline continues execution. Depending on the level of memory where the correct value is found, this stall can be very costly.

- **Memory Disambiguation** - Another problem with memory instructions is that it may be difficult to determine statically (at compile time) which memory operations are truly dependent on one another. Consider this sequence of code:

```
mov r3=r4
st [r3]=r1
mov r2=5
ld r1=[r2]
```

  It is not clear from this example that the `load` instruction is dependent on the `store` instruction. However, it could be. If `r4` had the value of 5, the `store` would be writing to the very location that the `load` was reading from. This makes it hard for the compiler to schedule loads above stores without the help of a structure such as the Advanced Load Address Table (ALAT) included in the Itanium and described in Section III.C.

- **Branch Prediction** - The direction that branches will take is unknown to the compiler. The compiler can make some static predictions based on past experience or common sense (such as loop branches are generally taken) and schedule accordingly. The static scheduler must take care not to schedule things above a branch that could cause incorrect execution if the path from which these statements originated did not turn out to be the taken path.

In summary, for the pure in-order model, the order of execution is controlled by the compiler, and by issuing stalls in the architecture. Data hazards are conservatively scheduled away, except in the case of the load where a stall may occur on a cache miss. Branch hazards are dealt with using stalls or prediction and recovery.

The in-order model is characterized as using a powerful compiler, with fast, simple hardware. One drawback to this simplicity, is the penalty that must be paid for conservatism (such as with the memory ops) or for correcting assumptions made by the compiler that differ from what actually happens at run time. Another drawback is the lack of flexibility. The code scheduled by the compiler will work optimally for one model of one processor. If the instruction latencies change, for example, the schedule could possibly force incorrect results. Predication adds complexity to the hardware since the analysis done by the compiler is not available in the hardware.

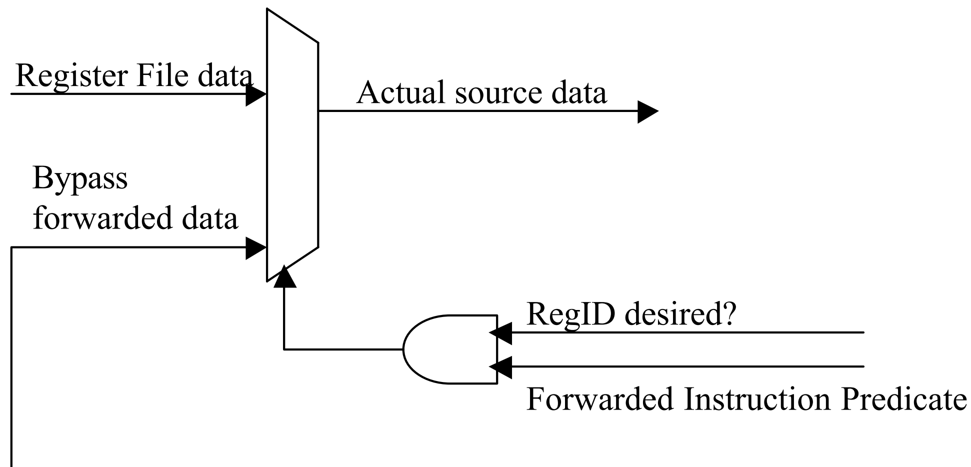


Figure III.3: A bypassing mechanism that supports predication.

III.B Additional Requirements for Predication

The presence of predicated code in the instruction stream impacts the compiler and hardware for any processor. The ISA must be expanded to include predicate defining instructions and the register set to include predicate registers. The compiler must have a method for determining regions to predicate, and have the ability to transform branch instructions into the predicate defining instructions. In addition, it must be able to associate the predicates with the appropriate conditional instructions. To create an efficient schedule the compiler must have some knowledge of predicate relationships (as discussed in Chapter IV) so that unnecessary dependences are not assigned. Predication also changes control dependences into additional data dependences [9].

For efficient resolution of dependences, the scoreboarding mechanism of a machine whose ISA includes predicated instructions must be extended to support predication. Dependences between instructions can be broken when either the instruction making the definition or the instruction using the definition of a register is determined to be guarded by a predicate register with a value of *false*. Consequently, the values of these guarding predicates must also be considered by the scoreboard when it makes a determination as to whether a data hazard exists.

The bypassing mechanism for architectures supporting predication requires additional logic and hardware. The bypassing of data must only occur for instructions that are guarded by a predicate register that evaluates to true. Figure III.3 shows an example of a mechanism

used to bypass predicated results. Note that in addition to determining if the result is intended for the register required by the operand of the consuming instruction, the bypassing mechanism must determine if the the guarding predicate of the producing instruction has been evaluated to TRUE.

Actually, dependency relationships between a producer and consumer, where at least one is predicated, cannot be determined until the predicate value has been resolved. To accomodate this in the Itanium, if the producer of the predicate register and the possible consumer of the general register produced are scheduled less than 2 cycles apart, the consumer must stall until at least the second cycle after the compare [6]. This holds true whether the predicate register has a value of true or false.

Consider the following code segment:

```
(1)      cmp P4,P5 = r8,r5    cycle 0
(2) (P4) ld r7=[r5]           cycle 1
(3) (P5) add r6=r7,3          cycle 2
```

Based on the current Itanium implementation (discussed next), these statements must execute as shown above. Statement 3 is a consumer that has to stall at least one cycle after the `cmp` defining it's guarding predicate before it knows which register definition to use.

The use of the guarding predicate of an if-converted instruction is treated differently than its other dependences. Most predicated instructions begin execution without the predicate dependency being met. This is because the predicate value is not needed until the instruction tries to forward its result, or update the machine state. As soon as the predicate register values are computed they must be made available to the guarded instructions. This generally means that the bypassing occurs later in the pipeline, and the existing mechanism cannot be used. Some additional mechanism must be included for this purpose.

As mentioned, the guarding predicate value of an instruction must be known before its result affects the state of the machine. Only instructions whose guarding predicates evaluate to true must be allowed to change this state. The commit stage must therefore have knowledge of each instruction's guarding predicate as well.

III.C Overview of Intel Itanium

The Itanium is an in-order VLIW architecture which supports predicated execution. It was designed with the goal of creating the most efficient division of labor between the compiler and the hardware. Following the EPIC philosophy [31], the designers constructed the compiler so it would find as much instruction level parallelism as possible and be able to pass this information on to the hardware [24, 50]. They included the Predicate Query System [15, 29] to provide predicate relationship information to reduce predication-related false dependences. The designers chose to implement large register files in the hardware to allow the compiler to reduce false dependences through effective register allocation. Reduction of false dependences creates more independence between instructions, providing more instructions that can be executed in parallel.

Itanium instructions are grouped in two ways by the compiler, with the intent of passing resource usage and dependence information along to the hardware [50]. First, instructions are *bundled* together in groups of three. A 5 bit template included in each bundle describes the combination of functional units that is required to execute the operations in the bundle. The use of templates simplifies the issuing of instructions to the correct instruction ports. Second, stop bits are effectively inserted at the beginning and end of a group of instructions that are independent of each other. In this way, independence found by the compiler can be communicated to the hardware.

Although the independence of instructions between two stop bits can be communicated to the hardware, independence of instructions outside of the grouping must be determined by the hardware if efficient execution of instructions is to exist in the face of hard-to-predict events such as cache misses. For this reason, the Itanium is fully scoreboarded, enforcing RAW and WAW dependences. The presence of the scoreboard enables the processor to stall only on the use of a loaded value that missed in the cache, not on the execution of the load itself [50]. In addition, the scoreboard is predicate aware, breaking dependences between producer and consumer instructions if it is determined that either is guarded by a predicate register with a value of false.

Many new techniques are employed by the Itanium to improve the chances of finding instruction level parallelism. Predication is used for more than the if-conversion previously discussed. In this architecture, predication is used to implement efficient software pipelining and control speculation. In addition, as mentioned earlier, the ALAT mechanism is included to effectively support memory disambiguation via data speculation.

III.C.1 Software Pipelining

It is common for computer programs to perform a certain sequence of operations repeatedly. These sequences are included in loop constructs in the code. Software pipelining is the process of symbolically unrolling several iterations of a loop, then selecting instructions from each iteration to run in the same cycle [42]. This is analagous to a hardware pipeline. Each instruction in a loop iteration can be thought of as a stage of the pipeline. In a hardware pipeline, in a given cycle, each stage can have a different instruction in it. For a software pipeline, each stage has a different iteration in it. Consider the following loop adapted from [5]:

```
lp:ld4 r3 = [r2], 4
  :add r4 = r3, r5
  :st4 [r6] = r4, 4
  :br.cloop lp
```

The stages of the associated software pipeline would be:

```
stage 1:  (p16) ld4 r3 = [r2], 4
stage 2:  (p17) empty
stage 3:  (p18) add r4 = r3, r5
stage 4:  (p19) st4 [r6] = r4, 4
```

Below, we show 4 iterations as we might imagine pipelining them:

cycle	0	1	2	3	4	5	6
iter 1:	ld4		add	**st4 [r6]=r4,4**			
iter 2:		ld4		**add r4=r3,r5**	st4		
iter 3:			ld4		add	st4	
iter 4:				**ld4 r3=[r2],4**		add	st4

Notice that cycle 3 contains all of the instructions that would occur in one iteration. However, these particular instructions come from various iterations (1, 2 and 4 respectively). This complete group of instructions is called the kernel. Cycles containing only partial iterations

are called the *prolog* and *epilog*. In this example, cycles 0-2 contain the prolog and cycles 4-6 the epilog.

For traditional architectures, the overhead of software pipelining can be immense. Notice, for example, that the `load` in iteration 2 will be issued before the `add` using the value loaded in iteration one. In some architectures, the kernel would have to be unrolled and the registers renamed. Software renaming requires the use of a large number of registers. The code expansion due to unrolling can increase the number of cache misses, reducing the IPC. Instructions must be added to manage the loop count, as well as prolog and epilog code, adding to the code expansion.

The IA-64 architects have added several features to significantly reduce the overhead of software pipelining for certain types of loops. Perhaps the most notable of these is *rotating registers*. These registers provide for a kind of automatic renaming. For each rotation (in general, a kernel iteration), the data in register x appears to move to register x+1 [5]. The code does not have to be expanded to accommodate explicit renaming. The number of general registers that can rotate is programmable, while the number of floating point and predicate registers that are rotatable is fixed.

For appropriate software pipelined loops in the IA-64 ISA, only the kernel instructions are required. Each instruction in the kernel is guarded by a rotating predicate register. These predicate registers, shown in the example on the previous page, are called *stage registers*, indicating that each controls a stage of the software pipeline. The value of the predicate controls whether the stage is executed. Prolog and epilog code can be executed in this manner.

Special branch instructions (`br.ctop, br.cexit, br.wtop, and br.wexit`) were included in the ISA to control software pipelined loops. These instructions automatically control and test loop counters. In addition, they decrement the base registers used by rotating registers, effectively controlling renaming.

III.C.2 Control Speculation

The Itanium provides support for *data* and *control speculation* [33] to allow the compiler to schedule loads as early as possible to avoid stalling on load misses. In the case of control speculation, a load (and possibly a chain of uses) is moved above a branch on which it is dependent. A check statement is placed after the branch in the block where the load originated. It is possible that an exception may occur with the execution of the speculated load. The exception will not be immediately serviced, but rather *deferred* and a reference to its

existence stored with the target register of the load. If the check statement is executed, it will check the target register for deferred exceptions, and if present, branch to compiler-generated recovery code, possibly re-executing the speculated load. If the check is not executed, none of the results on the speculated chain will be written to memory.

In some situations, predication can allow control speculation for any instruction without a check statement or possible recovery code. The Itanium typically implements branches in two steps. First, a compare statement is executed which assigns a predicate based on the branch condition. The branch statement itself is guarded by that predicate. Consider the following example that branches if `r2` is equal to `r3`:

```
  :cmp.eq p3,p0=r2,r3
  :(p3) br B1
  :add r4 = r2, r5
  :br B2
B1:ld4 r3 = [r2], 4
  :add r4 = r3, r5
```

If the predicate (`p3`) evaluates to *true* the jump is made to the target `B1` indicated by the branch. Otherwise, the next sequential instruction is executed. In the case of traditional code, any instruction scheduled above a branch requires the execution of recovery code if the outcome of the branch indicates that the instruction should not have been executed. In the case of the Itanium implementation of branches, if the predicate defined by the compare is available, the speculated instruction (the `ld`) is predicated as follows:

```
  :cmp.eq p3,p0=r2,r3
  :(p3) ld4 r3 = [r2], 4
  :(p3) br B1
  :add r4 = r2, r5
  :br B2
B1:add r4 = r3, r5
```

The instruction will then be executed only if the outcome of the branch would have required its execution. No recovery code is necessary.

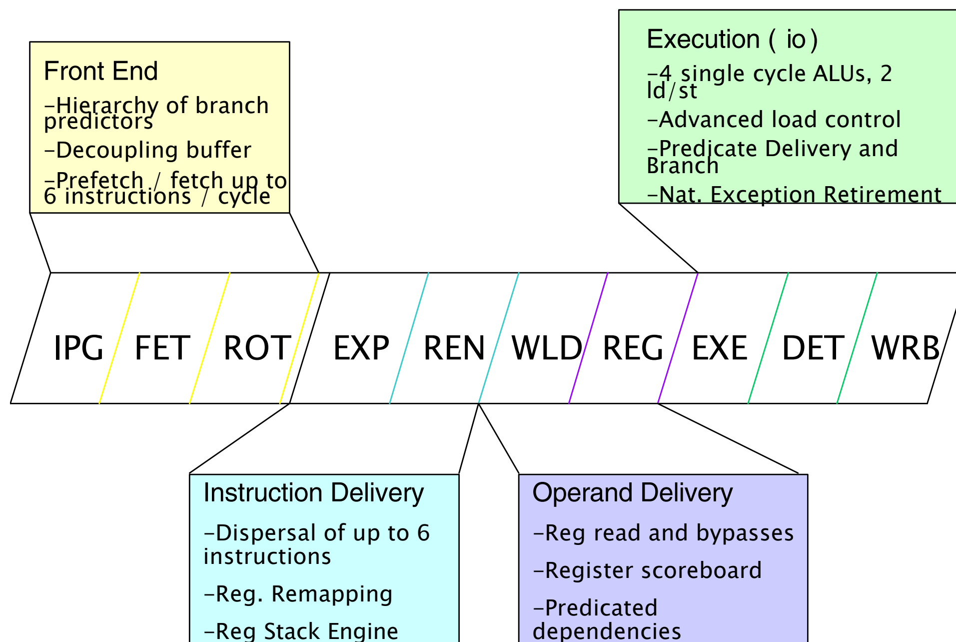


Figure III.4: Basic Itanium Pipeline

III.C.3 ALAT Implementation

The Itanium deals with the problem of memory disambiguation by implementing data speculation and recovery. Data speculation refers to moving a load above a previous store. A load check statement is put in place of the speculated load. The load executes speculatively and makes an entry including the accessed address in the *Advanced Load Address Table* (ALAT) [5]. Every store until the check accesses the table and if an entry matching the address of the store is found, that entry is removed. When the load check is executed, it attempts to find the entry for the corresponding speculated load in the table. If it does, that means that no store accessed the same address and the load can stand. Otherwise, the load is re-issued.

III.D Itanium Pipeline

The pipeline for the Itanium, as presented in [50], is shown in Figure III.4. The remainder of this section is used to describe the 10 stages of this pipeline, assuming 3 branch units, 2 integer units, 2 memory units and 2 floating point units available to the processor.

- **IPG (Instruction Pointer Generation)** - The location of the next instruction to be fetched is determined based on dynamic branch prediction and branch hint directives

generated by the compiler.

- **FET (instruction fetch)** - The fetch unit fetches 2 bundles per cycle. A decoupled fetch buffer holding 8 bundles allows fetching to continue when the rest of the pipeline has stalled.
- **ROT (instruction rotation)** - Initiates prefetch from prefetch hints in explicit branch prediction instructions. Accommodates decoupling buffer.
- **EXP (expand)** - Disperse up to 6 instructions (2 bundles) to 9 ports subject to 2 constraints:
  - *independence* -dispersal ends when a stop bit is encountered indicating that the next instruction may not be independent from the previous.
  - *oversubscription* - dispersal ends when 2 bundles have been dispersed of hardware determines that the required functional unit is unavailable. Bundles provide information (via templates) as to which instructions use which functional units.
- **REN (rename)** - The Itanium has 2 features that allow a type of dynamic renaming.
  - *register stacking* - for procedure calls, a new register frame is stacked on top of existing frames in the register file without an explicit save of the caller's registers.
  - *register rotation* - used specifically for renaming in software pipelining, registers are indirectly accessed using an index based on iteration count.
- **WLD (word line decode)** - Observation suggests that register dependency analysis occurs here. Information is recorded in a scoreboarding mechanism. As no general renaming occurs, all RAW and WAW dependences must be considered except between instructions that are in the same stop bit grouping.
- **REG (register read)** - Read registers if no data hazards exist according to scoreboard.
- **EXE (execute)** - If a hazard was detected in the previous stage, the instruction and all others dispersed at the same time are stalled in this stage. Since instructions at this stage don't have access to the register file, operands are received via the bypass mechanism.
- **DET (exception detect)**
  - up to 3 conditional branches can be executed

  - exceptions generated by speculated instructions are serviced here
  - the ALAT is accessed

- **WRB (Writeback)** - Instructions are committed.

Note that predicate value information is used in multiple stages. It is used in the REG stage to nullify all dependences for instructions guarded by false and in the EXE stage to nullify dependences between instructions guarded on false that define a register and any subsequent uses. Of course, it is also required to determine which register values to bypass and commit. To enable the earlier pipeline stages to receive the predicate values as soon as possible, the Itanium architects have taken advantage of the fact that predicate generation has a deterministic latency [50]. Instead of using the traditional bypassing mechanisms, the predicate values are written to and read from a speculative predicate register file.

Chapter IV

Predicated Static Single Assignment

In the first part of this chapter, we will demonstrate that the ability to find independent instructions in predicated code is improved by understanding predicate relationships. Once there is an understanding of the disjointness and predecessor/successor (path) relationships between the predicates, actual dependences between variables can be determined. Then, variables can be renamed to further remove false dependences, and the renamed values can be propagated down the appropriate paths of the multi-path hyperblock.

This chapter discusses some compiler-based optimizations designed to make the execution of predicated code more efficient. First, the chapter motivates the need for specialized predicate-sensitive data flow analysis required for the optimizations. Next, it discusses previous work in path and data dependence analysis. Finally, it presents Predicated Static Single Assignment and the related optimizations based on our predicate-sensitive data flow analysis.

IV.A Motivation for Specialized Predicate-Sensitive Data Flow Analysis

A major task for the scheduler of a multi-issue machine is to find independent instructions. Unfortunately, predication introduces additional dependences that traditional code doesn't have to consider. Look again at the example in Figure II.1(b). There is now a dependence between the definition of the guarding predicate `P2` and its use in the statement `d=q if`

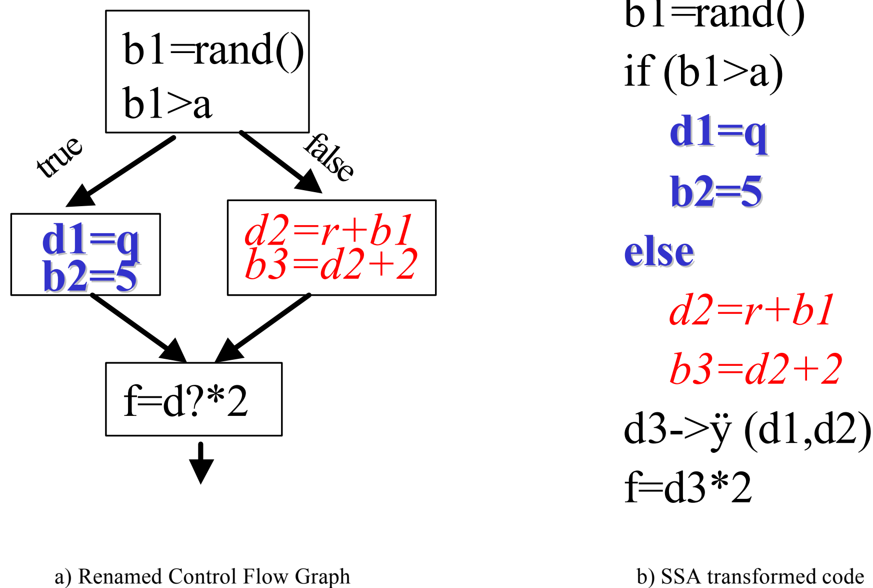


Figure IV.1: Renamed version of previous control flow graph example prior to the join. Eliminates false dependence between 2 definitions of b (now called b1 and b2)

`P2`. Also, because predication combines multiple basic blocks, it introduces false dependences between originally disjoint paths. For example, in Figure II.1(b), in the absence of further information, we would infer a dependence between the definition of `b` in `b=5 if P2` and the use of `b` in `d=r+b if P3`. However, these two statements are guarded by *disjoint* predicates. Therefore, only one of the predicates (`P2` or `P3`) can possibly be true; only one of the statements will actually be committed and no dependence does in fact exist. Clearly, it would be beneficial to create analysis that is aware of the predicate relationships and so can eliminate these additional false dependences.

Techniques such as renaming [8] and Static Single Assignment (SSA) [26, 25] have proved useful in eliminating false dependences in traditional code [8, 58]. Removing false dependences allows more flexibility in scheduling since data independent operations can move past each other during instruction scheduling. In Figure II.1(a), if the statements `b=rand()` and `b=5` were scheduled in the same cycle, erroneous execution could occur. It is not clear which assignment to `b` would actually result and be used by subsequent instructions. However, if we renamed the definitions as `b1=rand()`, and `b2=5`, we could execute the instructions simultaneously without any problem. We would then propagate, for example, value `b1` to `b1>a` and `d=r+b1` as is shown in figure IV.1(a).

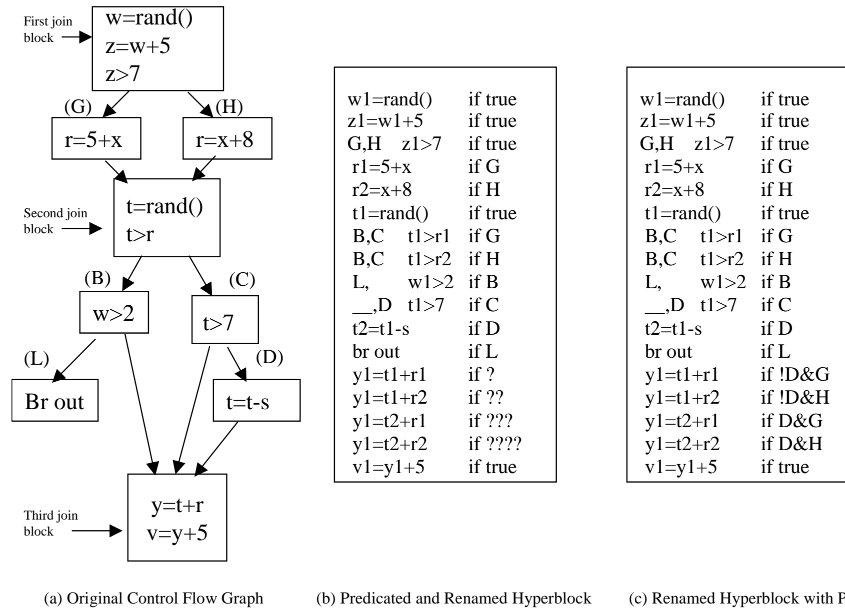


Figure IV.2: Code is duplicated when more than one definition reaches a use. The statement y=t+r is duplicated, but it is difficult to determine on what to predicate the duplicates. Figure (c) shows that path information that would need to be represented by an appropriate guarding predicate for each duplicate

Notice in this example that the variable `d` is also renamed. Which renamed version of `d` will be propagated to the use of `d` in statement `f=d*2`? Figure IV.1(b) shows that the SSA transformation would include a ϕ function prior to the use of `d`. This statement represents the assignment to a 3rd renamed version of `d` (`d3`). `d3` will be assigned the reaching definition of `d` which is to be used after the join (`d1` or `d2`).

Such renaming is important for predicated code as well as non-predicated code. However, for predicated code, it is even more important. Multiple paths containing definitions to the same registers are intermingled.

Figure IV.2 shows a more complex control flow graph and the resulting renamed, predicated code. As with the traditional code, propagating the renamed variables is fairly straightforward until multiple paths come together at a *join point*. For example, the statement `B,C t>r` has two possible definitions of `r` reaching it. One way to deal with multiple reaching definitions in predicated code is to duplicate the statement for each of the definitions, as in Figure IV.2(b). Notice that such duplication eliminates the additional dependence depth added by the are guarded by the constant TRUE because they will always be executed no matter what

path is taken. However, we cannot commit both definitions of the predicates `B` and `C`. We must guard the duplicates using disjoint predicates to ensure that only one definition will be made. In this case, we can easily choose guarding predicates. Since the definition of `r1` was guarded by `G`, if `G` is true, we want to commit the definition using `r1` (and conversely with predicate `H`).

The statement `y=t+r` in the third join block presents more of a problem. There are many reaching definitions to this statement, from many different paths. Four possible definitions of `y1` are shown in Figure IV.2(b), each representing one combination of the possible reaching definitions. The challenge is to determine what predicate should guard each of these duplicates.

To guard the execution of the first version of `y1`, (the statement `y1=t1+r1`) we need to capture where the definitions of `t1` and `r1` are coming from. `t1` reaches the third join block on paths that don't go through block `D`, and `r1` reaches on paths that go through block `G`. Therefore, as shown in Figure IV.2(c), we would need to guard this first version on a predicate that represents paths that go through block `G` AND not block `D`. Similarly, the third definition of `y1`, (the statement `y1=t2+r1`) would need to be guarded on a predicate that represents paths that go through blocks `G` AND `D`. The predicates chosen to guard these duplicates must be disjoint. No such predicates are naturally defined in the control flow graph. It is clear that we need analysis to find these predicate combinations (paths) and to determine which definitions reach subsequent operations using them. The predicate-sensitive analysis described in the next section will find these disjoint paths, determine their reaching definitions, and define precise predicates to represent them.

IV.B Related Work

The challenges of doing data-flow and control-flow analysis on hyperblocks have been previously addressed. Since hyperblocks include multiple paths of control in one block, traditional compiler techniques are often too conservative or inefficient when applied to them. Methods of predicate-sensitive analysis have been devised to make traditional optimization techniques more effective for predicated code. The *Predicate Query System* (PQS) [45] analyzes predicate defining operations to determine disjointness relationships between the predicates. The information is captured on a *Predicate Partition Graph* and used to answer questions about relationships between predicates. The PQS has been shown to be helpful in efficient register allocation for predicated code [29] and has been incorporated into the IA64 compiler for the Intel Itanium. The research presented in [18], and expanded upon in this dissertation

extends this localized predicate-sensitive analysis to complete paths through the hyperblock.

Path-sensitive analysis has previously been found useful for traditional data-flow analysis. Ammons suggests the creation and analysis of *hot paths* to improve the data flow analysis along the paths through the code that incur the highest execution cost [10]. A hot path is chosen and the effects of impossible or infrequently executed paths on the data flow analysis are discounted. Ball et. al. presents *path profiles* as a method of determining such critical paths through segment of code [14].

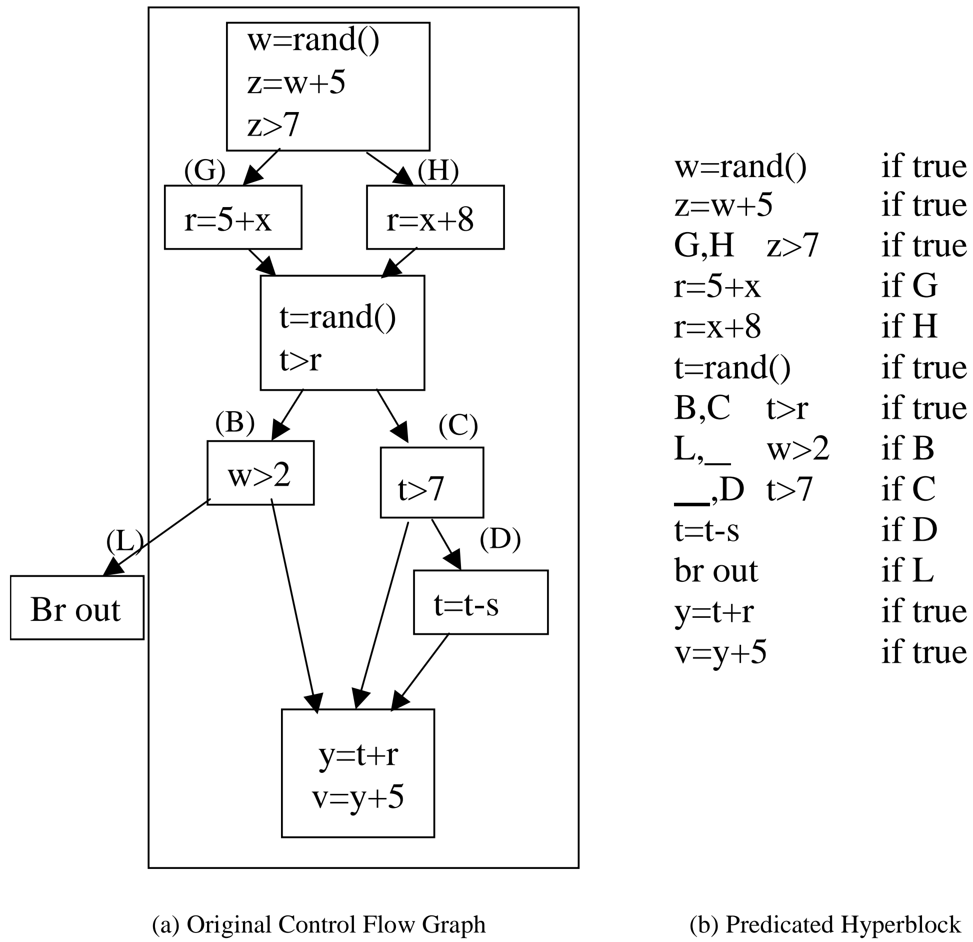
We use specialized path information to accomplish PSSA, a predicate-sensitive form of SSA [26, 25]. SSA provides an efficient representation of data dependences. SSA assigns each target of an assignment operation a unique variable. At join nodes (points in the control flow graph where path come together), a ϕ function is inserted to determine which of the multiple versions of a variable reaches the join. Code in SSA form has only true data dependences remaining as a result of this renaming process. An example of SSA transformed code was shown previously. Stoutchinin et. al. proposed a predicated form of SSA, allowing the application of current SSA-based optimizations to be extended to predicated code. In contrast, PSSA uses the renaming properties of SSA, but includes predicate-path information to give the resulting code increased scheduling flexibility for optimizations not traditionally associated with SSA.

PSSA is used to enables Predicated Speculation and Control Height Reduction for hyperblocks. Previously, these techniques been examined only in the presence of the single path of control found in superblocks [46, 47, 48]. These techniques, along with the analysis and *full-path-predicates* provided by PSSA allow scheduling at the earliest possible cycle. Moon and Ebcioglu [40] have implemented selective scheduling algorithms, which can schedule operations at their earliest possible cycle for non-predicated code. Our work extends theirs for predicated code.

IV.C Predicated Static Single Assignment (PSSA)

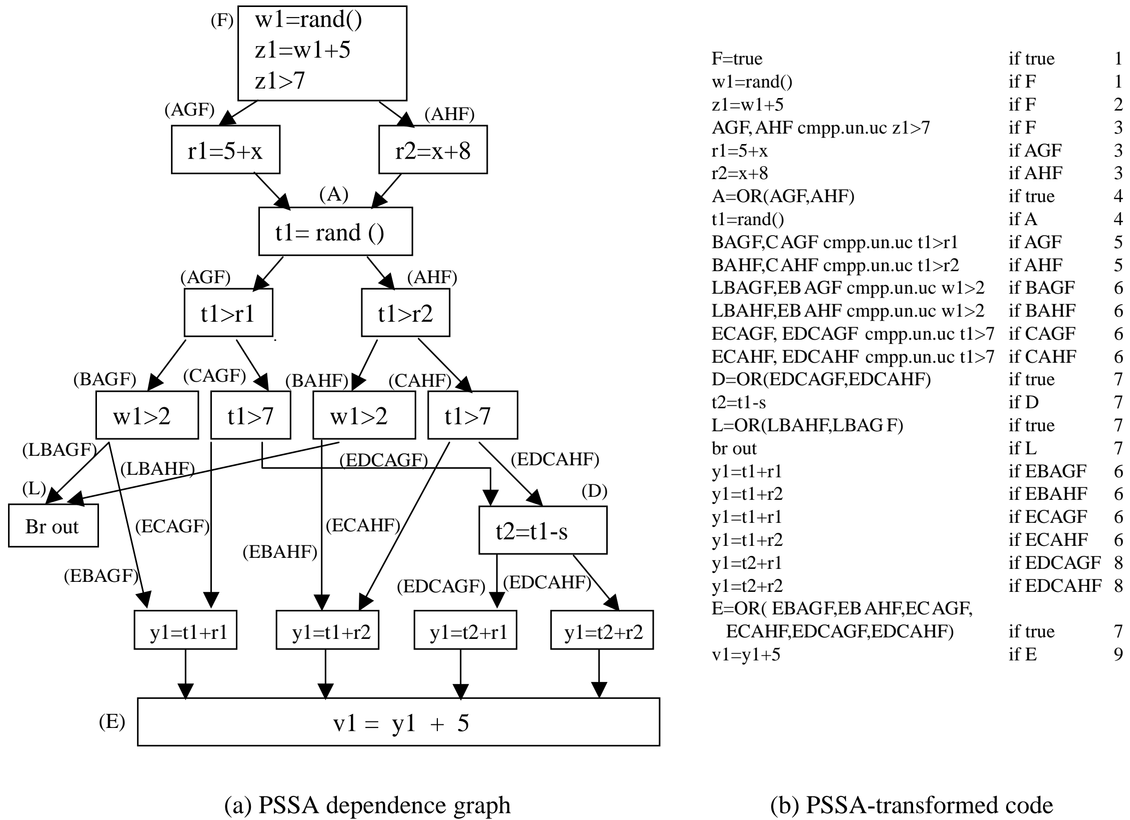
PSSA seeks to accomplish the same objectives as SSA for a predicated hyperblock. First, it must assign each target of an assignment operation in the hyperblock a unique variable. Second, at points in the hyperblock where multiple paths come together it must summarize under what conditions each of the multiple versions of a variable reaches that join. The second function is accomplished through the creation of full-path predicates and path-sensitive analysis.

Consider the sample predicated code shown in Figure IV.3 using traditional hyperblock



(a) Original Control Flow Graph (b) Predicated Hyperblock

Figure IV.3: Extended example of transformation from non-predicated CFG to predicated hyperblock.



(a) PSSA dependence graph

(b) PSSA-transformed code

Figure IV.4: The PSSA dependence graph shows the flow of data and control through the PSSA-transformed code. Blocks labeled with *full-path* predicates (indicated by multiple letters) contain statements that are only executed along that path. Blocks labeled with *block* predicates (single letters) contain statements that will be executed along several paths.

predication [38]. In this predicated example, all branches have been replaced (except the one leaving the hyperblock) with predicate-defining compare operations using if-conversion. The predicates that are defined in this example correspond to the two edges exiting each conditional branch in the CFG in Figure IV.3. In two predicate-defining operations, those defining `L` and `D` only one predicate register is defined. This occurs when the complement would lead to instructions in a join block. Figure IV.4 shows this example after PSSA has been applied and displays a graph showing the post-PSSA dependence relationships.

The PSSA transformation has 2 phases. Hyperblocks are converted to PSSA form before optimization. After optimization, PSSA inserts clean-up code onto edges leaving the hyperblock, copying renamed variables back to their original names and then removes any unused predicate definitions.

IV.C.1 Converting to PSSA Form

PSSA conversion takes two forms. *Control PSSA* is applied to predicate-defining operations, and *Normal PSSA* is applied to all other operations. When converting to PSSA form, each operation is processed in turn beginning at the top of the hyperblock and proceeding to the end.

When a normal operation is encountered, Normal PSSA is invoked. If the operation is an assignment, the variable defined is renamed. The third operation `z1=w1+5` in Figure IV.4(b) is an example. All operands are adjusted to reflect previously renamed variables (e.g. `w` becomes `w1`). If the operation is part of a join block, multiple versions of the operands may be live. The first operation (`y=t+r`) in the third join block of Figure IV.2(a) provides an example. In this situation, the operation will be duplicated for each path leading to the join and the correct operand versions for each path will be used in the duplicate statement as seen in Figure IV.4 (in the multiple definitions of `y1`). The duplicates are guarded by the full-path predicate (described in the next paragraph) associated with the path along which the operands are defined. Though there are 6 definitions of `y1` (only 4 are unique), there is only one definition of `y1` on any given path. These definitions are predicated on disjoint predicates; only one of them can possibly be true, and only one of them will be committed.

When a `cmpp` operation is processed, Control PSSA is invoked. The single `cmpp` operation that defined one or two block predicates (such as the definitions of `B` and `C` in Figure IV.3) is replaced by one or more `cmpp` operations, each associated with a particular path leading to that block. As can be seen in Figure IV.4(b) there are now two `cmpp` operations: one defining

`BAGF` and `CAGF`, and one defining `BAHF` and `CAHF`. These new predicates are called *full-path predicates* (FPPs). Each FPP definition has the appropriate operand versions for its path and each is guarded by the FPP that defined the path prior to reaching the new block. For example, the `cmpp` defining `BAGF` and `CAGF` is predicated on `AGF`.

An FPP specifies the unique path along which an operation is valid for execution, enabling PSSA to provide correct guarding predicates for the duplicate statements previously described.

In the motivational example, we pointed out that the definitions of `y1` needed to be predicated on something that captured facts about multiple predicates. The first definition of `y1` needed to be guarded by a predicate representing a path of execution going through block `G` but not block `D`. In addition, the predicate needs to reflect that the execution actually reached the block of the statement in question (`E` in this case). Register `y1` would be incorrectly modified if, for example, the branch out of the hyperblock is taken and block `E` is never reached. The new FPP `EBAGF` represents the precise information needed for correct execution.

In addition to the `cmpp` statements added to define FPPs, `cmpp` statements are included to rename join blocks whose statements were originally predicated on true. `A` and `E` and their associated FPPs are examples. The operations in Figure IV.3(b) predicated on true, are predicated on `F`, `A` and `E` in the PSSA version of the code shown in Figure IV.4. This is necessary to maintain exact path information.

It might seem more reasonable to follow the example of SSA and insert ϕ functions at join points to resolve the multiple definitions. An implementation of ϕ functions resolving `r` and `t` which are then used in the definition of `y1` could be:

(1) `r=r1 if G`
(2) `r=r2 if H`
(3) `t=t1 if true`
(4) `t=t2 if D`
(5) `y1=r+t if true`

Resolving our multiple definitions in this manner, we don't encounter the problem of the unavailability of appropriate predicates. However, we do not eliminate the need for predicate-sensitive analysis. Moreover, other side-affects that degrade performance are introduced. The insertion of ϕ functions adds data dependences. A true dependence is introduced

between the definition of `t1` and its use in the moves used to restore the renamed values to their original names. False dependences are re-introduced. An example is the output dependence between the two definitions of `t` in the moves just mentioned. In addition, ϕ functions join paths, so all paths are now forced to have the same data dependence length at the join. This puts unnecessary constraints on the scheduler. Predicate relationship information is still needed to determine the reaching definitions and associated predicates, and to determine the order of the copy operations. For example, both of the statements (3) and (4) defining `t` in the previous sequence could be committed. The literal predicate `true` is always true, and predicate `D` could be true as well. For the use of `t` in (5) to get the correct definition, statement (4) must be executed after statement (3). Thus, SSA and the usual ϕ function implementation does not give the desired scheduling flexibility.

Block predicates are also important to the PSSA transformation. PSSA uses predicate OR statements to redefine the block predicates as the union of the FPPs associated with the paths that reach the block. PSSA does not simply duplicate every path through the hyperblock. Duplication only occurs when necessary to remove false dependences. When there is only one version of all operands reaching a statement, only one version of the statement is required. This is the case with `v1=y1+5` in Figure IV.4. The variable `y1` is the only version live in node E. This statement is guarded by E, a block predicate created by taking the logical OR of EBAGF, EBAHF, ECAGF, ECAHF, EDCAGF, and EDCAHF. As long as control reaches node E, regardless of the path taken, we will execute and commit the statement `v1=y1+5`.

IV.C.2 Post-Optimization Clean-up

After optimization is applied to code in PSSA form, a clean-up phase is run to remove unnecessary code and to assure consistent code outside of the hyperblock.

The Earliest Cycle PSSA implementation described in this chapter generates `cmpp` statements for every path and block. These are entered into the PSSA data structure that maintains information about the relationships between the predicates they define, which provides maximum flexibility during optimization. However, some of these FPP definitions may not be used, and the corresponding `cmpp` operations will be discarded, reducing the code size significantly.

Finally, to assure correct execution following the hyperblock, PSSA inserts copy operations assigning the original variable names to all renamed definitions that are live out of the hyperblock. These are placed on the appropriate exits of the hyperblock. For example, the

branch out guarded by `L` in Figure IV.3 might include a move such as `t = t1 if L` if `t` was live out of the hyperblock at this exit.

IV.D Hyperblock Scheduling Optimizations

In this section, we describe how PSSA enables Predicated Speculation (PSpec) and Control Height Reduction (CHR) for aggressive instruction scheduling. PSpec allows operations to be executed before their guarding predicates are determined and CHR allows the guarding predicates to be determined as soon as possible, reducing the number of operations that need to be speculated. Used together with PSSA, we demonstrate that we can schedule the code at its earliest schedulable cycle, assuming a machine with unlimited resources.

IV.D.1 Predicated Speculation

This section describes how to perform speculation on PSSA-transformed code. In general, speculation is used to relieve constraints which control dependences place on scheduling. One can speculatively execute operations from the likely-taken path of a highly-predictable branch, by scheduling those operations before their controlling branch [35]. Similarly, Predicated Speculation (PSpec) will schedule a normal operation above the `cmpp` it is dependent upon, optimizing a hyperblock's execution time.

PSpec handles placement of the speculated predicated operation in a uniform manner. PSpec schedules a normal operation at its earliest schedulable cycle. When speculating an operation, the operation is scheduled earlier than the operation it is control dependent on, and is predicated on true. We assume that any exceptions raised by the speculated operations will be taken care of using architecture features such as poison bits [11].

Instruction Scheduling with Speculation

To demonstrate the usefulness of PSSA in enabling PSpec, Figure IV.5 shows the code from Figure IV.4 after the PSpec optimization has been applied. The assignments to `r1` and `r2` are examples of speculated operations. Notice that based on dependences, they could both be scheduled at cycle one which would have been impossible without renaming.

During predicated speculation, each operation is considered sequentially, beginning with the first instruction in the hyperblock. If it is a normal, non-store operation, PSpec compares its earliest schedulable cycle with the cycle in which its guarding predicate is currently

```
F=true                                            f true      1
w1=rand()                                         if F        1
z1=w1+5                                           if F        2
AGF, AHF                  cmpp.un.uc z1>7         if F        3
r1=5+x                                            if true     1
r2=x+8                                            if true     1
A=OR(AGF, AHF)                                    if true     4
t1=rand()                                         if true     1
BAGF,CAGF                 cmpp.un.uc  t1>r1       if AGF      4
BAHF,CAHF                 cmpp.un.uc  t1>r2       if AHF      4
LBAGF,EB AGF              cmpp.un.uc  w1>2        if BAGF     5
LBAHF,EB AHF              cmpp.un.uc  w1>2        if BAHF     5
ECAGF, EDC AGF            cmpp.un.uc   t1>7       if CAGF     5
ECAHF, EDC AHF            cmpp.un.uc   t1>7       if CAHF     5
D=OR(EDCAGF,E DCAHF)                              if true     6
t2=t1-s                                           if true     2
L=OR(LBAHF,LB AGF)                                if true     6
br out                                            if LBAGF    5
br out                                            if LBAHF    5
y1=t1+r1                                          if true     2
y2=t1+r2                                          if true     2
y3=t1+r1                                          if true     2
y4=t1+r2                                          if true     2
y5=t2+r1                                          if true     3
y6=t2+r2                                          if true     3
E= EBAGF,EBAHF, ECAGF,
  ECAHF,EDC AGF,EDC AHF)                          if true     6
v1=y1+5                                           if true     3
v2=y2+5                                           if true     3
v3=y3+5                                           if true     3
v4=y4+5                                           if true     3
v5=y5+5                                           if true     4
v6=y6+5                                           if true     4
```

Figure IV.5: Extended code example after PSpec optimization has been applied. Statements (other than first statement) predicated on true have been speculated.

```
PSpec(normal_op)
{
  if (normal_op.guarding_predicate not defined by
      normal_op.earliest_schedulable_cycle)
  {
      if (multiple defs of normal_op.target exist
      {
          rename(normal_op.target);
          update_uses(normal_op.target);
      }
      normal_op.schedule(earliest_schedulable_cycle);
      normal_op.set_predicate(true);
  }
  else
  {
      normal_op.schedule(earliest_schedulable_cycle);
  }
}
```

Figure IV.6: Basic PSpec Algorithm.

defined. If the operation can be scheduled earlier than its guarding predicate, the operation is predicated on true and scheduled at its earliest schedulable cycle.

Recall that PSSA has not performed full renaming, so further renaming may be required by PSpec. An example is the definition of `y1` in Figure IV.4. If we speculate any of the definitions of `y1` by predicating them on true without renaming, incorrect code can result. Consequently, we must rename the operations being speculated. The results of applying this to the 6 definitions of `y1` (now `y1`, `y2`, `y3`, `y4`, `y5`,and `y6`) appear in Figure IV.5. Speculation and renaming may require the duplication of operations using the definition being speculated, since there may now be multiple reaching definitions. When speculating `y1`, the operation `v1=y1+5` had to be duplicated and guarded on the appropriate FPP as shown in Figure IV.5. This is possible because PSSA already created all the necessary FPPs and path information.

If the guarding predicate has been defined by the operation's earliest schedulable cycle, we do not apply PSpec. It is again scheduled at the cycle equal to its earliest schedulable cycle, but guarded by the guarding predicate assigned by PSSA. The instruction `z1=w1+5` displays this characteristic. The algorithm for PSpec instruction scheduling is shown in Figure IV.6.

Using PSpec, the hyperblock can now be scheduled in 6 cycles as compared to 9 cycles in Figure IV.4. Since PSpec is applied whenever the definition of the operation's guarding predicate occurs later than the earliest schedulable cycle of the operation, we could reduce the number of operations that need to be speculated by moving the definition of the guarding predicates earlier. The goal of the next optimization, Control Height Reduction, is to allow predicates to be defined as early as possible.

IV.D.2 Control Height Reduction

Control Height Reduction (CHR) eases control constraints between multiple control statements. CHR allows successive control operations on the control path to be scheduled in the same cycle, effectively reducing control dependence height. For example, in the code in Figure IV.5, the control comparisons for `z1>7` and `t1>r1` are scheduled in cycles 3 and 4, respectively. However, the second comparison is only waiting for the definition of its guarding predicate `AGF`.

To schedule it earlier, consider the PSSA dependence graph in Figure IV.4. The definition of `BAGF` (defined by the condition `t1>r1`), is control dependent on the definition of `AGF` (defined by the condition `z1>7`). We could also define `BAGF` directly as the logical AND of the conditions `z1>7` and `t1>r1` removing the dependence on the definition of `AGF`. This AND

expression could also be scheduled in cycle 3.

Control Height Reduction was proposed in [47]. It was successfully used to reduce the height of control recurrences found in loops when applied to superblocks. A *superblock* is a selected trace of basic blocks through the control flow graph containing only one path of control [46]. The path defining aspects of PSSA allow our algorithm to efficiently apply CHR to predicated hyperblocks, since the full-path predicates expose all of the original separate paths throughout the hyperblock.

Schlansker et. al. [48] expanded on their previous research, applying speculation prior to attempting height reduction. Speculation can remove dependences between the branch conditions that need to be combined to accomplish the reduction. However, in that work, speculation was limited to operations that would not overwrite a live register or memory value if speculated, since they did not use renaming. In Figure IV.4, the `cmpp` operation defining `BAGF` and `CAGF` is shown scheduled at cycle 5 due to dependences on `t1` and `r1`. PSSA allows us to apply PSpec and schedule these definitions in cycle 1, making the `cmpp` available for CHR as shown in Figure IV.7.

Instruction Scheduling with PSpec and CHR

During instruction scheduling, PSpec is performed as described in Section IV.D.1. For each *control* operation (`cmpp`), CHR is performed if possible.

Recall that the operations in Figure IV.4 are scheduled in the order given in the PSSA hyperblock. Like PSpec, CHR compares when the operation could be scheduled based on its earliest schedulable cycle with when it must be scheduled if it waited for its guarding predicate to be defined. If it does not need to wait on the definition of its guarding predicate, it is simply scheduled at its earliest schedulable cycle. Without Pspec, the definition of `BAGF` was waiting on the definition of `t1` and `r1`. With Pspec, it is only waiting on the definition of its guarding predicate. Therefore, it is beneficial to control height reduce.

By ANDing the condition of the current definition with the condition that defined its guarding predicate, we can schedule this definition earlier. If the definition of the guarding predicate involved conditions that were ANDed as well, all of the conditions must be included, so the number of `cmpp` statements needed to define the current operation increases. The `.a` tag on each of these cmpp statements indicates that all of them are required for the final definition.

Consider the operations `z1>7`, `t1>r1` and `t1>7` in Figure IV.4. We control height reduce these operations in Figure IV.7, since they are all schedulable in cycle 3 based on our

```
F=true                                          if true      1
w1=rand()                                       if F         1
z1=w1+5                                         if F         2
AGF,A HF              cmpp.un.uc z1>7           if F         3
r1=5+x                                          if true      1
r2=x+8                                          if true      1
A=OR(AGF|AHF)                                   if true      4
t1=rand()                                       if true      1
BAGF, CAGF            cmpp.an.an z1>7           if true      3
BAGF, CAGF            cmpp.an.ac t1>r1          if true      3
BAHF, CAHF            cmpp.ac.ac z1>7           if true      3
BAHF, CAHF            cmpp.an.ac t1>r2          if true      3
LBAGF,EB AGF          cmpp.an.an z1>7           if true      3
LBAGF,EB AGF          cmpp.an.an t1>r1          if true      3
LBAGF,EB AGF          cmpp.an.ac w1>2           if true      3
LBAHF,EB AHF          cmpp.ac.ac z1>7           if true      3
LBAHF,EB AHF          cmpp.an.an t1>r2          if true      3
LBAHF,EB AHF          cmpp.an.ac w1>2           if true      3
ECAGF,EDC AGF         cmpp.an.an z1>7           if true      3
ECAGF,EDC AGF         cmpp.ac.ac t1>r1          if true      3
ECAGF, EDC AGF        cmpp.an.ac t1>7           if true      3
ECAHF,EDC AHF         cmpp.ac.ac z1>7           if true      3
ECAHF,EDC AHF         cmpp.ac.ac t1>r2          if true      3
ECAHF, EDC AHF        cmpp.an.ac t1>7           if true      3
D=OR(EDCAGF,EDCAHF)                             if true      4
t2=t1-s                                         if true      2
L=OR(LBAHF,LBAGF )                              if true      4
br out                                          if LBAGF     3
br out                                          if LBAHF     3
y1=t1+r1                                        if true      2
y2=t1+r2                                        if true      2
y3=t1+r1                                        if true      2
y4=t1+r2                                        if true      2
y5=t2+r1                                        if true      3
y6=t2+r2                                        if true      3
E=OR(EBAGF,E BAHF,ECA GF,
  ECAHF,E DCAGF,E DCAHF)                        if true      4
v1=y1+5                                         if EBAGF     3
v1=y2+5                                         if EBAHF     3
v1=y3+5                                         if ECAGF     3
v1=y4+5                                         if ECAHF     3
v1=y5+5                                         if EDCAGF    4
v1=y6+5                                         if EDCAHF    4
```

Figure IV.7: Extended example after PSpec and CHR optimizations have been applied. `Cmpp` instructions displayed in italics define predicates that are not used after optimization. Therefore, the statements can be removed from the final code.

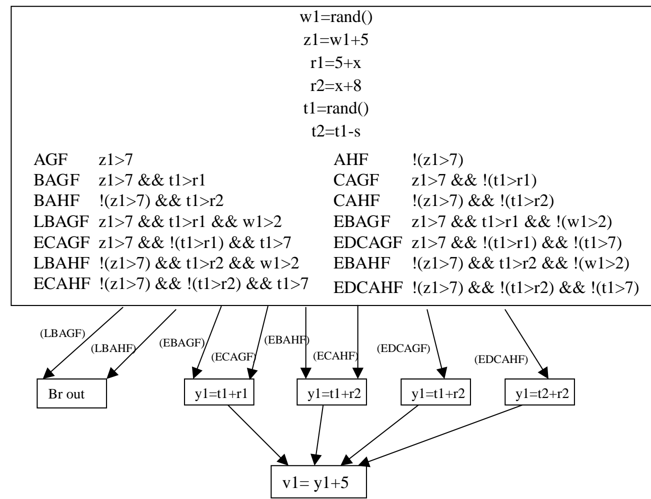


Figure IV.8: Dependence graph after PSpec and CHR have been applied.

```
CHR(cmpp_op)
{
    if (cmpp_op.guarding_pred defined
        by cmpp_op.earliest_schedulable_cycle)
    {
        cmpp_op.schedule(cmpp_op.earliest_schedulable_cycle)
    {
/* Apply Control Height Reduction */
    else
    {
        while (more_stmts_defining(cmpp_op.guarding_pred))
        {
            next_def=next_defining_stmt(cmpp_op.guarding_pred)
            copy=duplicate(next_def)
            copy.schedule(next_def.get_scheduling_time())
            copy.predicate_on(next_def.get_guarding_pred())
            copy.set_define(cmpp_op.get_pred_defined())
            copy.set_tag_to(a)
        }
        cmpp_op.schedule(next_def.get_scheduling_time())
        cmpp_op.predicate_on(next_def.get_guarding_pred())
        cmpp_op.set_tag_to(a)
    }
}
```

Figure IV.9: Basic Control Height Reduction Algorithm.

scheduling constraints. The new dependence graph resulting from PSpec and CHR is shown in Figure IV.8. The definition of ECAGF now describes the combination of `z1>7` being true AND `t1>r1` having a value of false AND `t1>7` having a value of true. We implement this logical AND, using the `.ac` and `.an` qualifiers. The definition of `ECAGF` requires that the conditions `z1>7` and `t1>7` and the condition `!(t1>r1)` evaluate to true for the FPP to get a value of true. If any one of the requirements are not met, the FPP will be set to false. The compares can architecturally be performed in the same cycle [34] allowing multiple links in a control path to be defined simultaneously. The algorithm for CHR is found in Figure IV.9.

Using PSpec and CHR on PSSA-transformed code results in the 4 cycle schedule shown in Figure IV.7. Note that this last version of the code has fewer operations than the previous version in Figure IV.5 and the operations shown in italics can be removed in a post-pass because these operations define predicates that are never used. Using predicated speculation and control height reduction together on PSSA-transformed code allows every operation to be scheduled at its earliest schedulable cycle.

IV.E Implementing PSSA in IA-64

Implementing PSSA using the IA-64 ISA [4] would be straightforward with the exception of the predicate OR statement we introduced. The OR instruction can be implemented by transferring the predicate register file into a general register using the move from predicate instruction in IA-64. The general purpose masking instruction would then be used to mask all but the bits corresponding to the sources of the predicate OR instruction. A result of zero evaluates to false, and anything else evaluates to true.

IA-64 places limits on compare instructions not found in the Playdoh ISA. For example, conditions that are included in logical AND compare statements can only compare a variable to zero. Specifically, the statement `LBAGF,EBAGF cmpp.an.an t1>r1` in Figure IV.7 would not be permitted. In implementing CHR, we would have to transform the prior expression into the following 2 statements:

```
temp = t1-r1;
LBAGF,EBAGF cmpp.an.an temp>0;
```

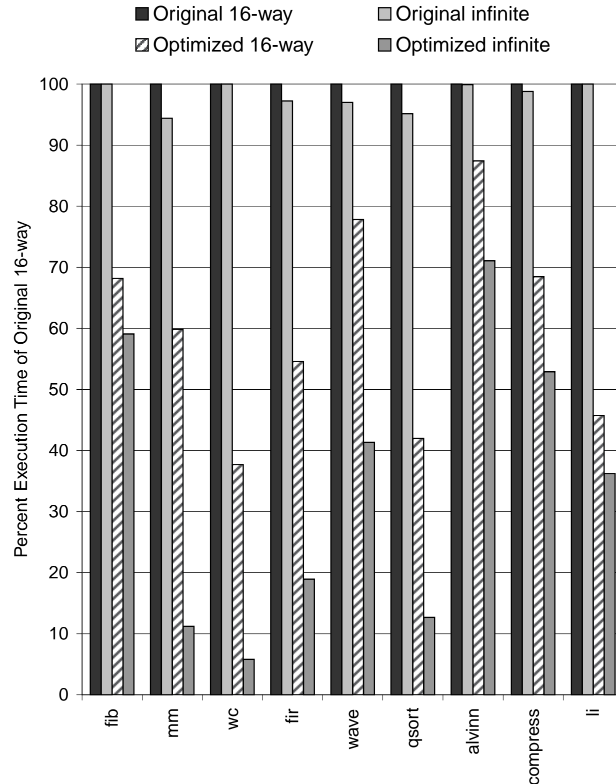


Figure IV.10: Executed cycles normalized to the number of cycles to execute the original code produced by Trimaran for a 16 issue machine.

IV.F Results

We have implemented algorithms to perform PSSA, CHR and PSpec on hyperblocks in the Trimaran System (Version 2.00). We collect profile-based execution weights for operations in the codes and schedule operations with an assumed one-cycle latency in order to calculate execution time.

Figure IV.10 shows normalized execution time when applying our optimizations for several benchmarks: `fib`, `mm`, `wc`, `fir`, `wave`, `qsort`, `alvinn`, `compress`, and `li`. These codes are described in table IV.1. The original execution times are created from the default Trimaran settings, with the exception that the architecture issue rate is set to 16. Execution time is estimated by summing together the frequency of execution of each hyperblock multiplied by the number of cycles it takes to execute the hyperblock, and a perfect memory system is assumed. The results are normalized to the original schedule generated by Trimaran for a 16 issue machine. The infinite results show the normalized execution time assuming an infinite

benchmark	suite	description
fib	Trimaran	calculates the Fibbonacci sequence
mm	Trimaran	matrix multiplication
wc	Trimaran	word count
fir	Trimaran	Finite Impluse Response
wave	Trimaran	the Wavefront computation
qsort	Trimaran	quicksort sorting algorithm
alvinn	SPECFP92	trains a neural network using back propagation
compress	SPECINT95	compression algorithm
li	SPECINT95	Language interpreter

Table IV.1: Presents description of benchmarks simulated including instruction count in millions where the initialization phase of the execution ended and where the trace began recording.

issue architecture. The optimized results show the performance after applying PSSA, PSpec, and CHR. The results show that using PSSA with PSpec and CHR results in a significant reduction in executed cycles.

Figure IV.11 shows the average number of operations executed per cycle for the configurations examined in Figure IV.10. In comparing the two graphs for the 16-way results, 3 to 4 times as many instructions are issued per cycle after applying PSSA, PSpec, and CHR, and this resulted in a reduction in execution time ranging from 12% to 62%. Since PSpec and CHR as applied to PSSAed code have the effect of removing the restrictions of control dependence, the optimized infinite results provide a picture of "best case" instruction level parallelism. Inspection of the optimized infinite results of `alvinn`, `compress`, and `li` show that, given current hyperblock formation, peak IPC is somewhat limited.

The renaming required by PSSA and PSpec also significantly increases register pressure. Trimaran's ISA (Playdoh) supports 4 register files: general purpose, floating point, branch, and predicate [2] [34]. Figure IV.12 shows the average number of live registers for the original code and the optimized code using PSSA, PSpec and CHR. The average live register results are weighted by the frequency of hyperblock execution. For example, `matrix multiply` has on average 17 live general purpose registers in the original code, and 54 live general purpose registers after optimization. Though the increase in utilization of all these register files is notable, the weighted average utilization mostly still remains well within the reported IA-64

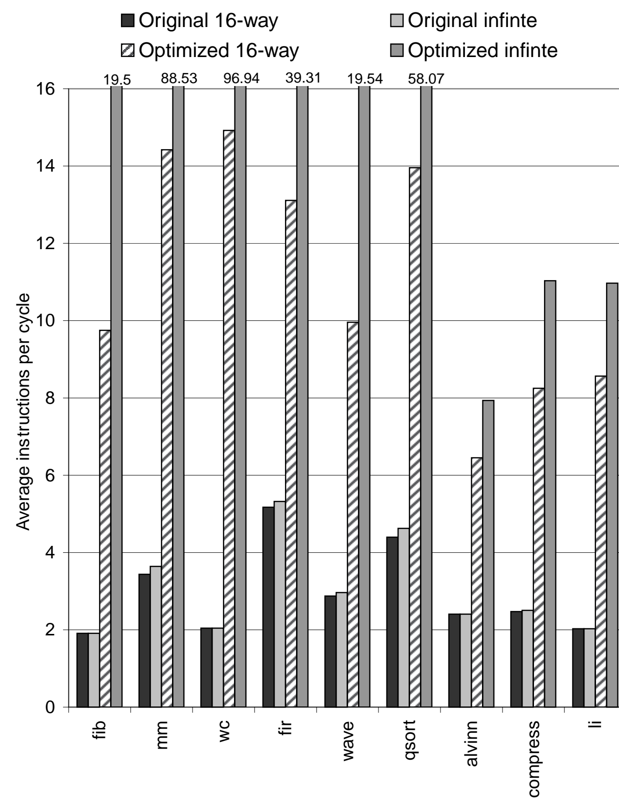


Figure IV.11: Weighted average number of operations scheduled per cycle for hyperblocks when using PSSA with Predicated Speculation and Control Height Reduction.

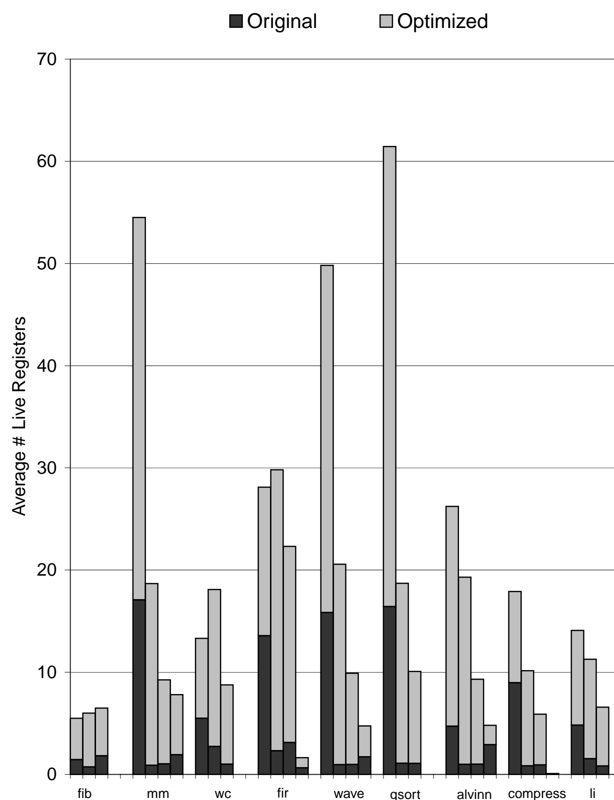


Figure IV.12: Weighted average register pressure in hyperblocks when using PSSA with Predicated Speculation and Control Height Reduction. Shown from left to right for each benchmark is the general purpose file, predicate file, branch file, and floating point file (zero utilization for some benchmarks).

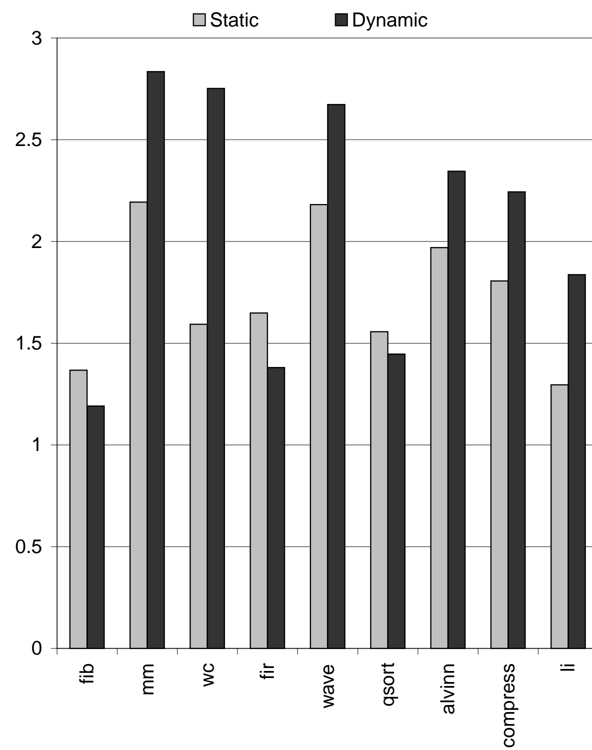


Figure IV.13: Static and Dynamic Code Bloat normalized to original code size.

register file sizes (128 general purpose, 128 floating point, 8 branch, and 64 predicate) [4].

Additionally, PSSA combined with aggressive PSpec and CHR significantly increases code size – both static and dynamic. Aggressive and resource insensitive application of CHR and PSpec aims to reduce cycles required to schedule at the cost of duplicated code specialized for particular paths (in the case of PSpec) or duplicated code for faster computation of predicates (in the case of CHR). Figure IV.13 shows both the static and dynamic code bloat of the PSSA, PSpec, and CHR optimized code over the original code. We plan to address the issue of code bloat in future work.

IV.G Path and Code Duplication Reduction

Previously in this chapter, we demonstrated the benefits of employing renaming for removing false dependences. We described two possible methods for resolving the renamed values at join nodes. These two techniques will be discussed in more detail in this section. The first uses separate names on separate paths, and then resolves the correct value by a copy placed on each incoming edge to a join node. This is a straightforward implementation of phi functions used in SSA form. The second technique duplicates code along each full path to a join, with each duplicate using the unique variable names that reach along that given path.

The first technique increases the data dependence length by one for each variable at each of its join nodes; the second technique does not add to the data dependence length, and so is theoretically superior. We showed in the previous sections that, assuming unlimited computational resources, the use of duplication using Full Path Predicates, when combined with control height reduction [48, 18, 46], reduced the lengths of schedules by 12-60%. However, the additional instructions added increase both the static code size and the number of instructions executed, potentially exponentially, and so could swamp limited resources and have great disadvantages in practice.

In this section, we re-visit these two techniques, seeking to combine the advantages of both to enable instructions to be scheduled as early as possible while limiting the amount of code duplication introduced. We present a Region Formation algorithm that carefully chooses regions of code over which to apply our techniques, and a Critical-Path-First (CPF) scheduling algorithm for predicated code. Using our method, we get a 33% reduction in cycle time on the example code in this chapter while keeping code expansion under control.

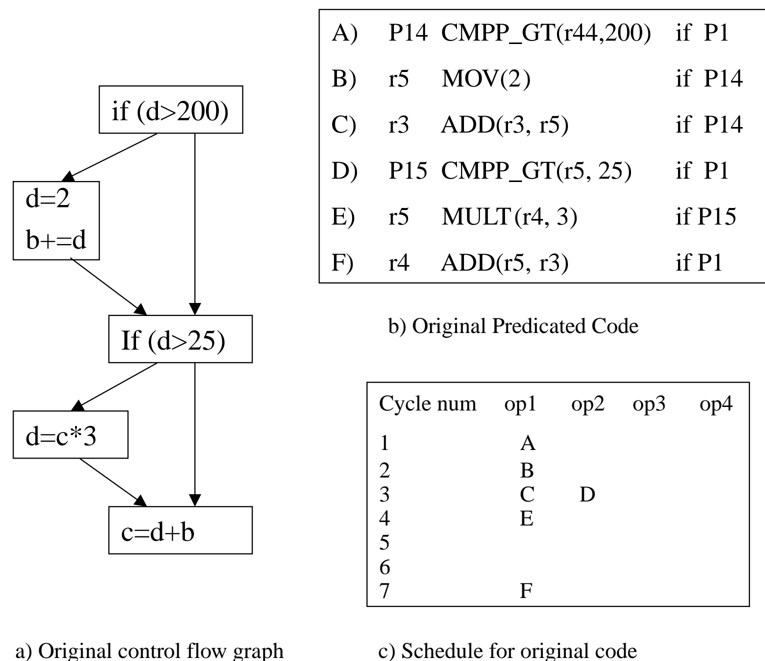


Figure IV.14: A simple example of a control flow graph and related predicated code. The included schedule follows all data and control dependence constraints.

IV.G.1 Renaming and Scheduling Using ϕ functions and Duplication

A simple example of scheduling predicated code is found in Figure IV.14. Figure IV.14a shows the original code in Control Flow Graph form. Figure IV.14b shows the predicated assembly code for the example. Figure IV.14c shows how the code could be scheduled without renaming and speculation for a 4-wide VLIW architecture assuming a latency of 3 for the multiplication operation and 1 for all other operations.

Figures IV.15 and IV.16 reflect versions of the predicated code and associated schedules where definitions of the variable `d` are renamed and speculated. Speculation is accomplished by changing the guarding predicate of an operation to P1. Since P1 is predefined, the calculation of the guarding predicate is no longer a determining factor as to when the operation can be scheduled. Figure IV.15 is an example where phi functions are added to compensate for the renaming. Phi functions are implemented as MOV operations, moving the renamed value into the original register and guarding the operation on the guarding predicate of the original speculated operation. Figure IV.15b shows that speculation improves the length of the schedule even with the data dependences added by the phi functions. This is because the longer-latency `MULT` operation can be initiated earlier. Two additional operations are required for the phi

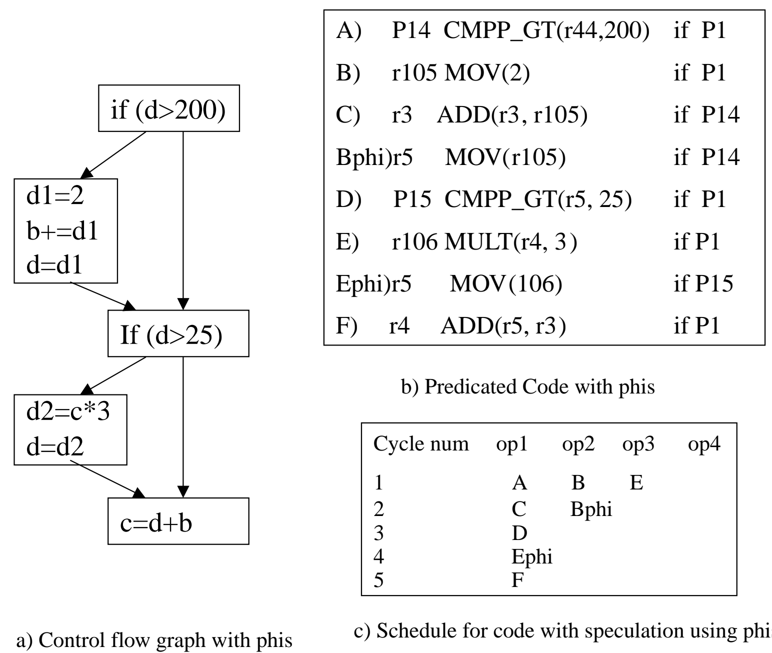


Figure IV.15: Renaming is added to the original example to allow for speculation without side-effects. Phi functions are added to multiple versions of a variable that may reach a use.

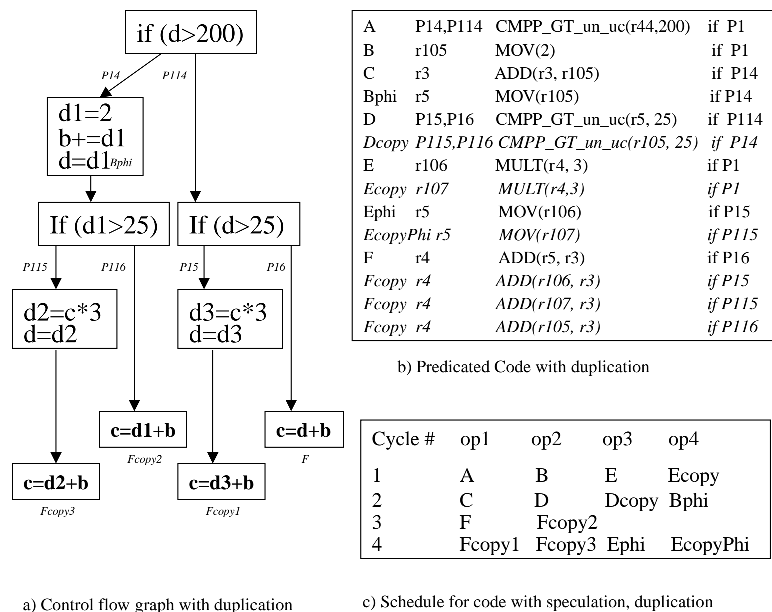


Figure IV.16: Renaming is again added to enable speculation, but multiple versions of variables are reconciled by duplicating the operation for each set of operand combinations. Additional statements required for this method are shown in italics in the code. Italicized labels in the CFG are there to help show correspondence between the code and the CFG.

functions.

Figure IV.16 is an example where renamed values are propagated to their uses. Recall that this technique necessitates duplicating the code for each version of the operand that reaches. Versions `d` and `d1` reach the operand `d` in the original condition `d>25`. For the original statement `c=d+b`, 4 versions of `d` reach. In Figure IV.16b definitions of full-path-predicates (FPP) P114 (!14), P15 (!14 && !15), P16 (!14 && !15), P115 (14 && 15), and P116 (14 && !15) are added and used to guard the duplicate statements. Phi functions are included to assure correctness leaving the region, assuming `d` is live out. Figure IV.16c shows that a very efficient schedule can be created using propagation and duplication.

The potential drawback to propagation and duplication is, of course, the amount of code expansion that results. The more paths that exist, the more FPPs that have to be created and the greater the duplication. Recall that for predicated code, all paths are executed to some extent. If FPPs were added indiscriminately to a large code segment with complex control, the schedule could not efficiently accommodate the extensive duplication, and the ultimate affect would be to expand the schedule rather than to reduce it. Consequently, it is desirable to be able to control the creation of FPPs. The next sections discuss the creation of FPPs, and suggest ways to do this in a manner that preserves the benefits of FPPs.

IV.G.2 Full Path Predicate Creation Methodology

Our Full Path Predicate examples were generated by our PSSA System. Information is retained by the this system regarding the relationships between all FPPs created. The creation of FPPs follows the following algorithm:

- Assign predicates to all statements
- Add `cmpp` statements to define these additional predicates
- Choose region over which to define FPPs
  - any region between 2 whole joins
  - a small enough region to keep duplication in check
- Define and propagate FPPs

The next sections describe this algorithm in detail.

IV.G.3 FPP Creation

Full Path Predicates provide benefits beyond the enabling of propagation and duplication suggested in the previous paragraphs. In the process of defining FPPs we create a data structure that enables us to easily maintain control flow information across multiple code transformations. Other control information is no longer necessary. The paragraphs that follow explain the two step process of FPP creation, beginning with a motivation for creating them in this manner.

Figure IV.17 shows the main example we will be using throughout the remainder of this chapter. Figure IV.17a contains the original program which was assembled and predicated. It is widely accepted that predication is not applied to the entire program. As mentioned earlier, likely candidates for predication are segments of code within a program that are frequently executed and contain hard-to-predict branches [37]. Often, the body of a loop falls into this category. The loop body is generally unrolled several times prior to predication. This is the case with the program shown in Figure IV.17a. The loop was unrolled 8 times. Figure IV.17b shows 2 iterations of the code produced by Trimaran and Figure IV.17c shows the basic structure of these 2 iterations. Each iteration ends with a branch out of the segment. Statement numbers `17` and `18` are guarded by P14, `20` is guarded by P15, `52` and `53` by P18 and `55` by P19. All other statements represent operations that are not part of a conditional and are therefore always executed. They are guarded by P1 or the literal *true*.

Notice that the conditional branch operations (BRCT) take a slightly different form. Although they are guarded on true, they have two operands. One represents the target address of the branch calculated by the *Prepare to Branch* (PBRR) instruction [34]. This value is placed into a branch register (B). The other is a predicate register (P) that holds the true or false value of the condition on which the branch is to be taken.

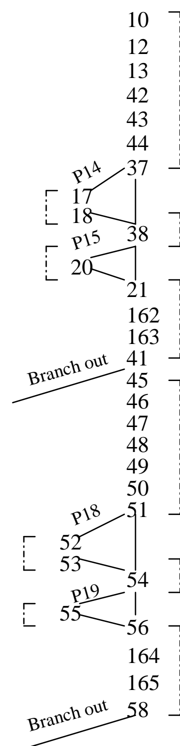
Consider again the option of speculation using renaming and phi functions. If we desired to speculate operation 47, scheduling it before the branch at operation 41, and it defined a register that was live out on that branch, we would want to rename the destination of operation 47. Subsequently, we would want to insert a phi function after the branch so later uses of the value calculated by operation 47 would get the correct value. Notice that it is not sufficient to simply propagate the new name to the uses of 47. This operation defines `r5`, and so does operation 52. Both definitions reach operation 54. If the renamed register appeared as the operand for 54, and P18 was evaluated as true, causing the definition made by 52 to be the real definition of the operand, we could get incorrect results.

```
#include <stdio.h>
int main()
{int i, a, b, c, d;
  a=b=c=0;
  for (i=0; i<200; i++)
  {  d=c*d;
      a+=d;
      d=a+b;
      if ((d*10)>200){
          d=2;
          b+=d; }
       if (d<25) {
          b+=c; }
       c=d+2;
   }
 printf (“a:%d b:%d c:%d\n”, a,b,c);
 exit(0);
}
```

a) Code used for example

10 r41	MULT(r4,r5)	if P1
12 r2	ADD(r2, r41)	if P1
13 r5	ADD(r2, r3)	if P1
42 r42	SHIFT(r5,I3)	if P1
43 r43	SHIFT(r5,1)	if P1
44 r44	ADD(r42, r43)	if P1
37 P14	CMPP_GT_UN(r44,200)	if P1
17 r5	MOV(2)	if P14
18 r3	ADD(r3,2)	if P14
38 P15	CMPP_LT_UN(r5,25)	if P1
20 r3	ADD(r3,r4)	if P15
21 r4	ADD(r5,2)	if P1
162 B298	PBRR(r8,0)	if P1
163 P299	CMPP_GEQ_UN(r139,200)	if P1
41	BRCT(B298,P299)	if P1
45 r109	MULT(r4,r5)	if P1
46 r2	ADD(r2, r109)	if P1
47 r5	ADD(r2, r3)	if P1
48 r110	SHIFT(r5,3)	if P1
49 r111	SHIFT(r5,1)	if P1
50 r112	ADD(r110, r111)	if P1
51 P18	CMPP_GT_UN(r112,200)	if P1
52 r5	MOV(2)	if P18
53 r3	ADD(r3,2)	if P18
54 P19	CMPP_LT_UN(r5,25)	if P1
55 r3	ADD(r3,r4)	if P19
56 r4	ADD(r5,2)	if P1
164 B300	PBRR(r8,0)	if P1
165 P301	CMPP_GEQ_UN(r139,200)	if P1
58	BRCT(B300,P301)	if P1

b) 2 iterations of predicated code produced by Trimaran



c) Basic form of 2 iterations of loop

Figure IV.17: Main code example that will be used for the rest of the chapter. This Figure depicts only 2 iterations of the 8 that are actually unrolled and predicated by Trimaran. Note that P indicates predicate registers, B indicates branch registers and r indicates that the value is being placed into an integer register. The dotted brackets in (c) represent the ranges of block predicates to be added in the process of transforming the predicated code into Full Path predicated code.

In our example in Figure IV.15, the phi functions were guaranteed to be scheduled in the correct position because they were guarded by the same guarding predicate that the original speculated statement was guarded by. The dependence on the definition of this guarding predicate (called a *predicate dependence*) had to be obeyed, effectually scheduling the phi function after the branch. In the example in Figure IV.17, however, the original operation was guarded on *true*. To guarantee the correct placement of the phi function, we must have the additional information that it must be placed in the schedule after the evaluation of the branch_out statement. This additional information is not based strictly on the dependence information, either data or predicate dependence. It must be obtained from the original control flow graph (CFG). As well, it is important that it has been updated, taking into account any transformations due to optimizations.

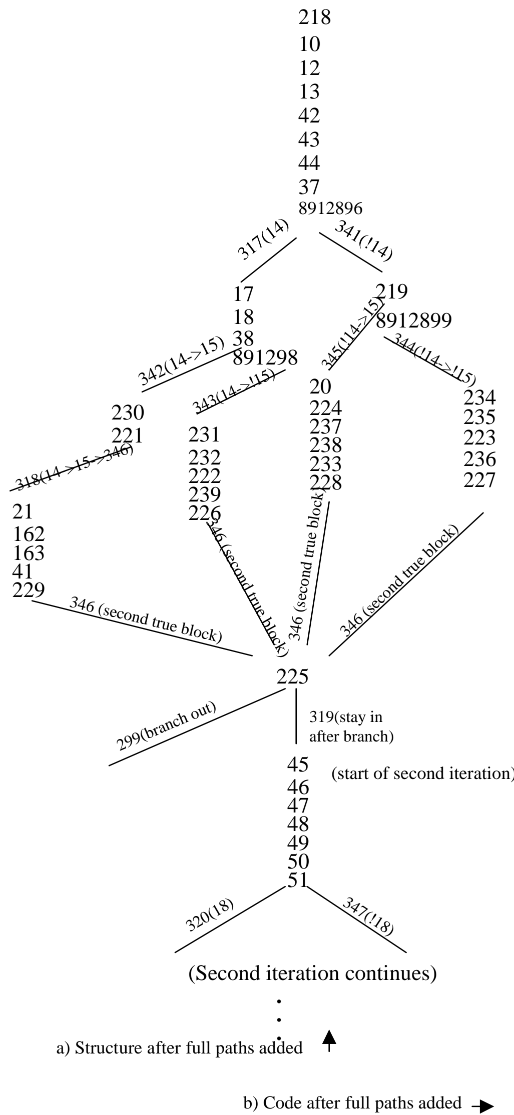
The first step in the creation of FPPs assures that we will never need the additional CFG information for proper phi placement. Because FPPs need to maintain precise predicate and path relationships, no segment of code in an FPP transformed predicated region is guarded by the constant *true*. These unconditional statements will now be guarded by *block predicates*. Only operations that are control equivalent share the same block predicate, defined as a guarding predicate that represents a certain set of control and ordering conditions. That means that statements between and including 10 and 37 will share a block predicate, statements 21-41 will have a different block predicate and 45 - 51 still another. At every place where control changes, the first step of the FPP transformation adds a `cmpp` statement defining the new block predicate (unless one has already been defined) and the block predicate is propagated to statements meeting the current control conditions. An example of a `cmpp` statement added to define a block predicate can be seen in Figure IV.18(b). The first `cmpp` statement in this code defines new block predicate `P316` which is subsequently used to guard statements between 10 and 37. The dotted brackets shown in Figure IV.17c depict the ranges of the block predicates that we will add. Block predicates replacing the predicate register P1 will always be evaluated as true, but carry with them additional control and ordering information. Statements such as 17 and 18 already have a unique predicate defined and will retain their original block predicates. Now phi functions can be precisely placed requiring only dependence information.

Once block predicates are in place, the actual FPPs can be defined. The first step is to decide the region over which we want full path information. FPPs can be created between any two *whole joins*. A whole join is a place in the control flow where all paths come together. The guarding predicate of a statement located at a whole join will always be evaluated to true. The

statement will be executed whichever path was taken. In our example, we decided to create full path information across a single iteration. Figure IV.18 shows the form of our code after full path analysis is completed and FPPs are constructed for the first iteration. Subsequent iterations will have the same pattern. For simplicity, we decided to duplicate the code along all paths as we defined the full paths. We are assuming that we will want to speculate and rename in this region. We will discuss these decisions more in a later section. Note that, in our example, an iteration starts either at the beginning of the code or at a BRCT instruction, and ends with a BRCT instruction.

FPP creation involves replacing block predicate definitions between the two whole joins chosen with full path predicate definitions and then rejoining the paths at the end of the chosen region. As we progress through the region, each `cmpp` statement defining a block predicate is replaced by FPPs doubling the number of current paths leading to the block predicate definition. This allows us to describe both the taken and fall through paths in conjunction with all the combinations that preceded. Figure 5a shows only one path leading to the definition of P14. We replace the definition of P14 with the definition of P317 representing P14, and the definition of P341 representing !P14. The definition of P15 is replaced by the definition of 4 FPPs. P318 represents 14&& 15, P343 represents 14&&!15 , P344 represents !14&&!15 and P345 represents !14&&15. Since this is the extent of the region over which we are building full paths, P346 is defined along all paths, effectively joining the paths back together.

The choice of regions over which to form FPPs is critical. Note that if we had chosen to include one more block predicate definition in our region, we would have increased the number of paths to 8. Code duplication would have increased substantially after that point. On the other hand, when we choose to re-join, we have to insert phi functions to handle data dependences across regions since we do not have duplication capabilities across regions. Recall that a region ends with bringing all the paths back together so the subsequent region has only one path entering it. There must be only one version of each variable live at the entrance to the region, as there do not exist multiple path predicates to describe multiple variable combinations. Iteration boundaries have appeared to be good break points for regions because observation has shown that there are more dependences that are restricted to the loop body, not as many that are loop-carried. Moreover, the obvious loop-carried dependence (the iteration counter) may not even be used in the body and therefore need not be considered for renaming and speculation. These characteristics contribute to a reduction in the number of phi functions that must be added.




```
218 P316      CMPP_EQ_UN                     if P1
10  r41       MULT(r4,r5)                    if P316
12  r2        ADD(r2, r41)                   if P316
13  r5        ADD(r2, r3)                    if P316
42  r42       SHIFT(r5,3)                    if P316
43  r43       SHIFT(r5,1)                    if P316
44  r44       ADD(r42, r43)                  if P316
37  P317      CMPP_GT_UN(r44,200)            if P316
8912896 P341 CMPP_GT_UC(r44,200)             if P316
17  r5        MOV(2)                         if P317
18  r3        ADD(r3,2)                      if P317
219 P345      CMPP_LT_UN(r5,25)              if P341
38  P342      CMPP_LT_UN(r5,25)              if P317
8912899  P344 CMPP_LT_UC(r5,25)              if P341
8912898  P343  CMPP_LT_UC(r5,25)             if P317
20  r3        ADD(r3,r4)                     if P345
230 r3        ADD(r3,r4)                     if P345
221 P318      CMPP_LT_UN(r5,25)              if P342
21  r4        ADD(r5,2)                      if P318
231 r4        ADD(r5,2)                      if P343
237  r4       ADD(r5,2)                      if P345
234  r4       ADD(r5,2)                      if P344
162 B298      PBRR(r8,0)                     if P318
232 B298      PBRR(r8,0)                     if P342
235 B298      PBRR(r8,0)                     if P345
238 B298      PBRR(r8,0)                     if P344
163 P299      CMPP_GEQ_UN(r39,200)           if P318
222 P299      CMPP_GEQ_UN(r39,200)           if P343
224 P299      CMPP_GEQ_UN(r39,200)           if P345
223 P299      CMPP_GEQ_UN(r39,200)           if P344
41            BRCT(B298,P299)                if P318
239           BRCT(B298,P299)                if P343
233           BRCT(B298,P299)                if P345
236           BRCT(B298,P299)                if P344
229 P346      CMPP_EQ_UN(1, 1)               if P318
226 P346      CMPP_EQ_UN(1, 1)               if P343
228 P346      CMPP_EQ_UN(1, 1)               if P345
227 P346      CMPP_EQ_UN(1, 1)               if P344
225 P299,319 CMPP_GEQ_UN_UC(r39,200)         if P346
45  r109      MULT(r4,r5)                    if P319
46  r2        ADD(r2, r109)                  if P319
47  r5        ADD(r2, r3)                    if P319
48  r110      SHIFT(r5,3)                    if P319
49  r111      SHIFT(r5,1)                    if P319
50  r112      ADD(r110, r111)                if P319
51  P320      CMPP_GT_UN(r112,200)           if P319
```

Figure IV.18: Main code example showing one iteration+ that has been FPP-transformed and duplicated. This figure shows the beginning of the second iteration as well.

IV.G.4 Code Reduction

The transformation depicted between Figures IV.17 and IV.18 produced a 140% increase in operation count. Since the original schedule was sparse using 10-wide VLIW instructions, (assuming 8 integer units and 2 branch units) the schedule could absorb the increase without substantial impact to the length of the schedule. This is not always the case, however, and attention must be paid to control code expansion while maintaining as many of the benefits of using Full Path Predicates as possible. Code reduction techniques range from simple peephole optimizations applied in a post past after the FPPs have been added to completely different strategies in the use and creation of FPPs. First we will discuss ways that we can further reduce code in our region formation approach to adding FPPs. Then we will look at some options for limiting the creation of FPPs and code duplication using strategies other than region formation.

Further Code Reduction Using Region Formation Strategy

Perhaps the simplest reduction technique to apply to our previous example is the combination of `cmpp` statements that have the same guarding predicate and use the same condition to calculate the value of the predicate defined. Consider the example in Figure IV.18. The statements defining Predicate Registers 317 and 341 are both guarded by P316. The comparison used for both is `(r44 > 200)`. The statements only differ in the Predicate Registers they define, and whether they are defined by the normal evaluation of the condition, or the complement of the evaluation. By definition, `cmpp` operations have the capability of defining two predicates in one statement, using different evaluation criteria for each. Therefore these two statements could be combined into one. The new statement would be `P317, P341 = CMPP_GT_UN_UC (r44, 200) if P316`. This statement would evaluate whether the contents of register 44 is greater than the integer literal 200. P317 would be assigned the normal evaluation, and P341 the complement of the evaluation. In our example, adding a post past to accomplish this task would reduce our code expansion by 10%.

For simplicity, as we added FPPs, we made copies for each path of the code that would appear along that path. The assumption was that most instructions would eventually be duplicated. The reality is, however, that the only statements that are even candidates for duplication are those who have at least one operand defined in the region. If this is not the case, there is only one version of each operand currently live and there is no reason to duplicate the statement. We could attempt to make this decision during the initial transformation, or,

once again make the improvement using a post pass. During a post pass, we would look for statements with the same opcode and operands with guarding predicates all created from the same block predicate. We would remove all copies but one. That copy would be guarded by the appropriate block predicate. If this statement defines a variable, that variable name would be propagated to the uses of any of the original copies. In our example, applying this technique alone would reduce the code expansion by 30would reduce the expansion an additional 10%.

Region formation itself can be applied differently. We chose to use an iteration as a region since the code under consideration contained an unrolled loop. Another approach is to simply limit the number of block predicate definitions to include within any given region.

It is not necessary to create FPPs throughout the entire predicated segment. With limited resources, there may not be enough room in various parts of the schedule to allow for code duplication. One possibility is to monitor the schedule and when the schedule indicates a capacity under a given threshold to include FPP definitions. Alternatively, static analysis might show, for example, that it would be beneficial to include FPP definitions for every other iteration.

Optimizing Only the Critical Path

Another approach to using FPPs while keeping code expansion in check is to create FPPs only along the critical path. The critical path can be defined as the longest chain of dependences, either data or predicate. The critical path can be determined statically beginning with the leaves of the data dependence graph. Each of these statements is given a dependence depth of 1. Edges exiting the leaves are followed to a parent node. The dependence depth of the parent node considers all child nodes and is calculated as MAX(child dependence depth + execution latency between child and parent)

Figure IV.19 shows a sample control flow graph labeled with guarding predicates that could be assigned and the associated dependence graph as is described in [28]. Included in the dependence graph are the dependence depths (in parenthesis) and latencies (labels on arrows) between each pair of statements. Statement `J` is a leaf and has a dependence depth of 1. `H` is a parent node to `J` with a latency of 3 between the two statements. Therefore `H` has a dependence depth of 4. Statements `H` and `I` are both children of (control dependent on) `G`, but the longest chain so far comes from `H`, so the dependence depth of `G` is `4 + 1 = 5`. The critical path includes statements `J, H, G, E, B,` and `A`. Consequently, the blocks included are all three join blocks, and the blocks that are labeled with P20 and P30.

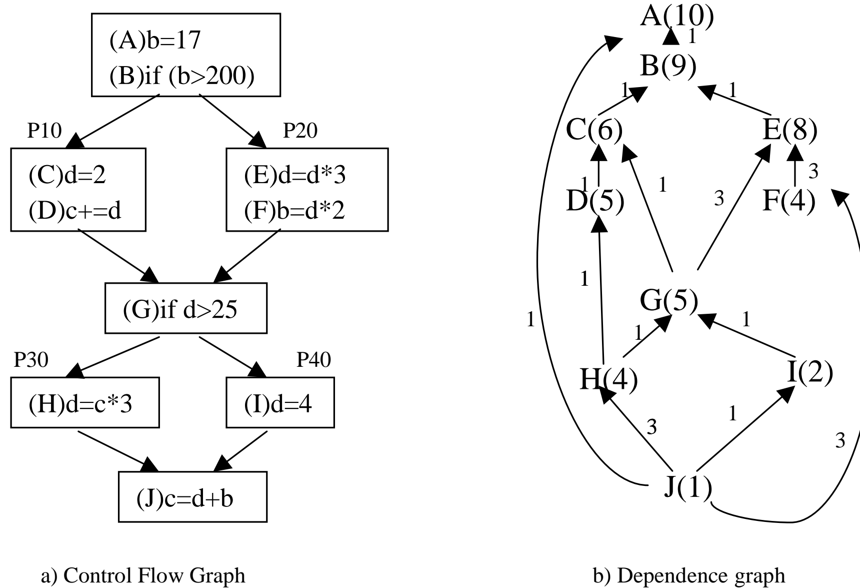


Figure IV.19: Sample control flow graph and associated dependence graph showing how the critical path is determined.

Using the critical path method, FPPs are only defined for and code is only duplicated for the critical path. The code in Figure IV.20a shows the additions in italics. If a statement along the critical path is an assignment, the destination is renamed and propagated to the critical path copy. If the statement is a `cmpp`, it is transformed into an FPP and its destination is used to guard statements until another FPP is created. The critical path may now be optimized and scheduled completely independently from the rest of the code. Figure IV.20b shows the code as it would be scheduled with the critical path optimization, and Figure IV.20c as it would be scheduled without the optimization. We are again assuming a latency of 3 for multiplication operations.

The optimized schedule is better, but does not yield as significant an improvement as one might expect. Notice that in Figure IV.20b, the critical path is scheduled in 7 cycles. However, the original versions of the critical path statements must be scheduled for the use of the remaining paths. Due to these non-critical paths, the original statement `J` cannot be scheduled until cycle 9.

Sometimes the speculation done for the critical path can help the other paths, but often it cannot. Statement `E` is an example where speculation along the critical path can help. Statement `E` can have only one possible version of operands. Therefore, the versions used for

(A)	b=7	if P1
(B) P10,P20	CMPP_UN_UC_GT(b,200)	if P1
(Bcrit) P25	*CMPP_UC_GT(b,200)*	*if P1*
(C)	d=2	if P10
(D)	c+=d	if P10
(E)	d=d*3	if P20
(Ecrit)	*d1=d*3*	*if P1*
(EcritPhi)	*d=d1*	*if P25*
(F)	b=d*2	if P20
(Fcrit)	*b1=d1*3*	*if P25*
(G) P30,P40	CMPP_UN_UC_GT(d,25)	if P1
(Gcrit) P35	*CMPP_UN_GT(d1,25)*	*if P25*
(H)	d=c*3	if P30
(Hcrit)	*d2=c*3*	*if P1*
(HcritPhi)	*d=d2*	*if P35*
(I)	d=4	if P40
(J)	c=d+b	if P1
(Jcrit)	*c1=d2+b1*	*if P35*
JcritPhi	*c=c1*	*if P35*

a) Predicated code with Critical Path Optimization included

Cycle	op1	op2	op3	op4	op5
1	A	Ecrit	Hcrit		
2	B	Bcrit			
3	C	E			
4	EcritPhi	Fcrit	Gcrit	HcritPhi	D
5	G				
6	H	I	F		
7	Jcrit				
8					
9	J				

b) Schedule with Critical Path Optimization

Cycle	op1	op2	op3	op4
1	A			
2	B			
3	C	E		
4	D			
5				
6	F	G		
7	H	I		
8				
9				
10	J			

c) Schedule without Critical Path Optimization

Figure IV.20: Code and schedule that could be created where the critical path is optimized. The schedule for the unoptimized code is included for comparison.

the critical path are the same as for all other paths and the speculated value will work for all paths. The renamed version of statement `E` can be scheduled in cycle one, with a phi function implemented in cycle 4, making this value of the variable `d` available to the critical path in cycle 4, but to all others in cycle 5. This is an improvement of 1 cycle to non-critical paths.

Statement `H` is an example where the speculation along the critical path cannot help the rest of the paths. The value of the variable `c` could have come from statement `D` or from some code previously executed. The critical path calculation assumes the later. The normal calculation of `H` must wait on the possible evaluation of `c` from statement `D`, while the critical path version did not. It would be incorrect to use the `''HcritPhi''` in this case.

It is clear that there are complications to the critical path approach. Often there is not a clear critical path to begin with. As all paths in a section of predicated code must be executed to some extent, Mahlke et.al. [38] make the case that only paths of similar lengths should be included in a section of code to be predicated. Another consideration is that once the original critical path has been optimized, it may no longer the critical path. The critical path, therefore, can change as the code is scheduled.

Creating Partial Path Predicates

It may be that we can define predicates that give us more information than block predicates, but not as much information as FPPs. They would give us "partial path information". Using the example found in Figure 1, there are 4 paths leading to the last statement (`F`). Assume that statement `F` is altered to be `c=5+b`. Using FPPs, we would define predicates for these four paths and duplicate statement `F` four times with appropriate versions of `b`. However, we don't really need to know exactly which path we came through. All we need to know is that we didn't go through the block guarded by P14. This approach would appear to reduce the amount of predicate registers we need to define. However, it seems that the analysis involved would be substantial and if not done correctly, might lead to the creation of more paths than necessary.

Our scheduling algorithm achieves good results using speculation and scheduling the critical path first. Since all paths through a section of predicated code must be scheduled, the critical path as described in the previous section dictates the ultimate length of the schedule. However, because we apply our FPP optimizations to all paths in a particular region, we are not forced to stick with a statically determined critical path. The critical path is re-evaluated as each operation is scheduled. Our algorithm uses a dependence graph that contains edges between operations representing control and data dependences. Each node on the graph also contains the dependence depth of the operation as described in the previous section and information as to when the operation is scheduled. Both of these values are updated when an operation is scheduled. The basic scheduling algorithm can be found in Figure IV.21. A discussion key points of the algorithm follows.

Find Ordered Candidates - Candidates are those operations which have not yet been scheduled and whose data dependences have all been met if they are a normal data operation. If the candidate is a branch or a `cmpp`, the control dependences must also have been met. If it is a store or load, all the stores that occur before it (in control flow order) along its path must have been scheduled.

Once chosen, the candidates are then ordered by dependence depth according to the dependence graph, longest dependence depth first. For each cycle, the candidates are considered for scheduling in this order. In other words, the operations that are currently on the critical path have first priority.

If Functional Unit is Available - Our schedule is a linked list of records, each representing a cycle of the schedule. Each record contains an array for each type of functional

```
Create_Optimal_Schedule
{
   for each cycle:
   {
      find ordered candidates
      for each candidate
      {
         if there is an available functional unit
         {           mark operation as scheduled in this cycle
            set edges to be removed
            if the operation was speculated
            {
               rename the destination
               propagate renamed values to uses within region
               if there are uses beyond the region
               {
                  insert phi function
                  make phi function dependent on original operation
                  make uses outside the region dependent on the phi function
               }
            }
            else if this operation has false dependences or moved above branchout
            {
               rename destination
               propagate renamed values to uses within region
               if there are uses beyond the region
               {
                  insert phi function
                  make phi function dependent on original operation
                  make uses outside the region dependent on the phi function
               }
            }
         }
      }
   }
}
```

Figure IV.21: Scheduling Algorithm.

unit, the size of the array determined by the number of functional units available for the architecture. When an operation is scheduled, a reference to it is placed in the array location for the appropriate cycle. If the operation has a latency of greater than one, the same location in additional records (number equal to the latency of the operation) is filled in with the reference to the operation. When trying to schedule an operation in a particular cycle, we need to determine if there is an empty slot in the array representing the appropriate type of functional unit in the record for the given cycle.

Mark operation as scheduled in this cycle - Indicate in the dependence graph when this operation was scheduled. It can no longer be considered a candidate for scheduling.

Set edges to be removed - Indicate on each edge between the scheduled operation and it's uses which cycle the operation's value will be available. This is the cycle the edge can be removed - the dependency will no longer exist. This cycle is the current cycle plus the latency of the scheduled operation.

If the operation was speculated - An operation is speculated if it was scheduled for execution before the value of the its guarding predicate would be known. Branch and `cmpp` statements cannot be speculated.

Rename the destination, propagating to uses within the region and inserting phi functions for uses outside the region - The destination is renamed to eliminate the incorrect commitment of values for instructions that should not have been executed to completion. Operations within a region are candidates for duplication, so the duplicates need the correct version for the path. Operations outside of the region cannot handle duplication, so must rely on a phi function. Phi functions are dependent on the definition of the guarding predicate that guarded the speculated statement.

IV.G.5 A Sample Result

Our algorithm was able to schedule our transformed code example in 60 cycles. In comparison, the Trimaran scheduled achieved a 89 cycle schedule for the original code. Two iterations of both are shown scheduled in Figure IV.17.

Figure IV.22a shows the structure of the code produced by Trimaran before speculation, with Figure IV.22b showing the structure after speculation. Statements 162 and 163 were speculated from the first iteration, and statements 164 and 165 from the second. Statements 162 and 164 were `PBRR` operations that assigned branch registers holding the target address of associated branches. Statements 163 and 165 assigned predicate registers as true or false

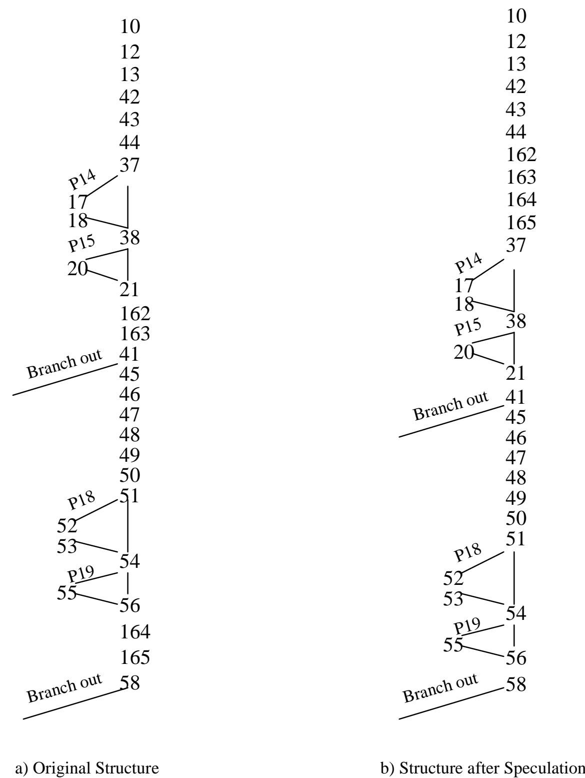


Figure IV.22: Structure of original code before and after speculation. Only 2 statements per iteration were speculated.

depending on the evaluation of the condition of the same branches. These instructions are easy to speculate because each operation only has one use, and these are the only possible definitions for each use. Therefore, no renaming is required. These same instructions were speculated for all eight iterations of the unrolled loop.

Figure IV.23 gives the same information for 1+ iterations of the the FPP-transformed code. Notice that many more operations are speculated, some replaced by phi functions in their original positions. Similar to the Trimaran scheduler, we were able to speculate statements 162 and 163. In addition, we speculated 17, 18, 20, 21, 45, 46 and 47. The additional instructions shown are duplicates of some of these.

As indicated, we speculated statements 17 and 18. Trimaran could have done this as well, but would have had to rename and add a phi function to reconcile the renaming. All uses would be dependent on the phi, and this would not have resulted in any improvement. Since we have duplication capability, we were able to schedule operation 20 and its duplicate 230 immediately after the speculated operations. No intervening phi function was necessary.

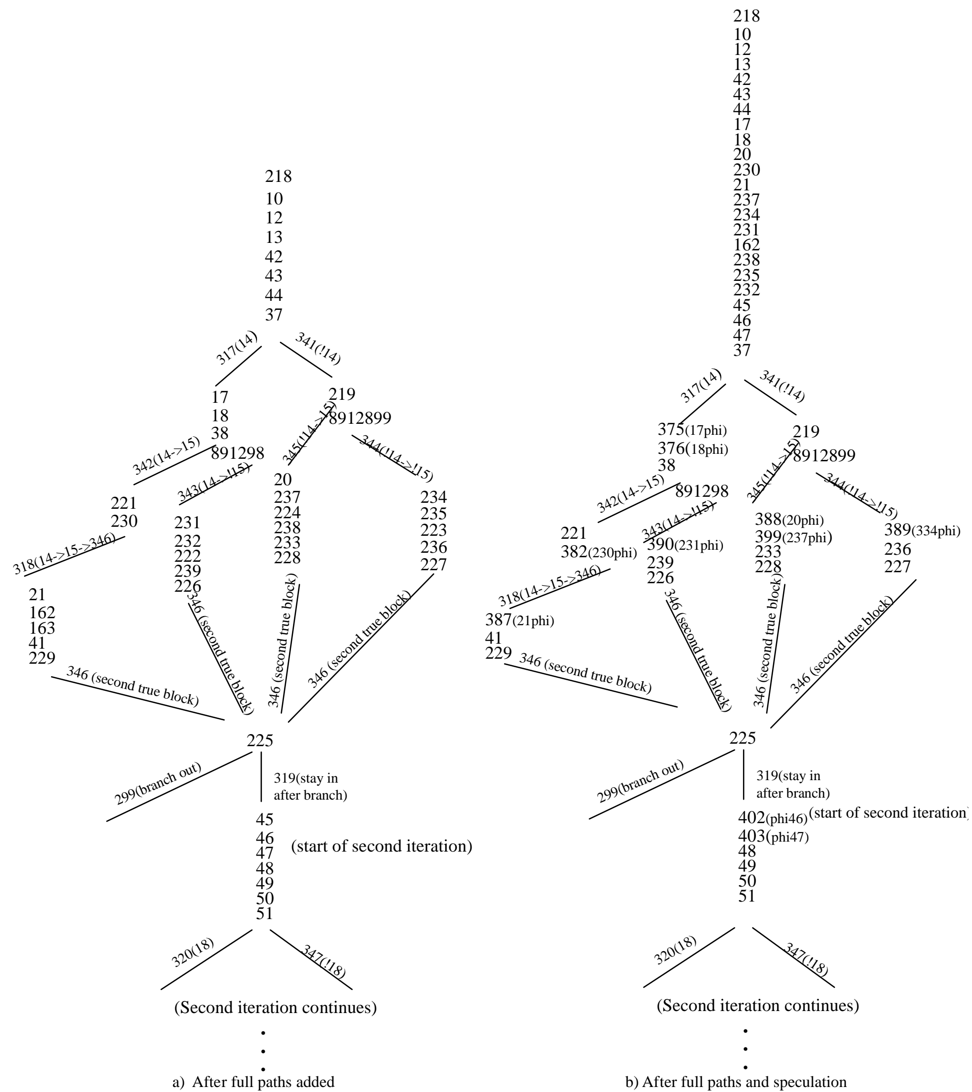


Figure IV.23: Structure of FPP transformed code before and after speculation. The large number of statements at the top of (b) shows the ability of FPPs to support speculation.

Statement 21 and duplicate 237 were scheduled early for the same reason.

We were able to speculate statement 45 without the use of a phi function because its only uses were within the same full-path region. It would have been beneficial for Trimaran to have speculated this operation since it is a higher latency operation. However, Trimaran was unable to speculate statement 21, and statement 45 has a data dependence on 21. Even if 45 could have been speculated, renaming would have been necessary if it was scheduled before the branch out. Control information would need to be available for correct placement of the phi function as described previously.

Figure IV.24 shows the schedules possible for the FPP-transformed code and the original Trimaran generated code through statement 51 for an architecture with 8 integer functional units and 2 branch units. We were able to schedule the code in 15 cycles as opposed to the 19 cycles taken by the Trimaran scheduler. This is a decrease of 21%. The full results (not shown) were even better. Operations from later iterations were able to be speculated to some of the earlier cycles, overlapping execution with earlier iterations. In addition, we were careful to always schedule the operations currently on the critical path first. The FPP-transformed code was scheduled in 60 cycles, and the original code in 89. This yielded a decrease of almost 33%.

One of the reasons that we were able to do so well is that the original schedule is very sparse. There was room in the schedule to support duplication along 4 paths. Still, it was important to be able to choose wisely the operations that would wait when the instruction for a particular cycle was full. Our continually updated knowledge of critical path assured that the appropriate operations would be chosen.

IV.H Summary

This chapter described PSSA, a renaming technique enabled by predicate-sensitive path information. It motivated the need for renaming and for predicate analysis that extends across all paths of the hyperblock. It demonstrated how Predicated Static Single Assignment (PSSA), a predicate-sensitive implementation of SSA that implements renaming using full-path predicates, can be used to eliminate false dependences for predicated code. We showed the benefit of using PSSA to enable Predicated Speculation (PSpec) and Control Height Reduction (CHR) during scheduling. Predicated Speculation allows operations to be executed at the cycle of their earliest schedulable cycle, even before their guarding predicates are determined. Control

Cycle_num	I1	I2	I3	I4	I5	I6	I7	I8	B1	B2
1	218	10	17	18						
2	20	230	162	238						
3	235	232								
4	12									
5	13									
6	42	43	21	237	234	231				
7	44	45								
8	37									
9	375	38	8912899	376	219					
10	221	227	226	228	388	389	390			
11	229	387	399	382	46					
12	225	47								
13	402	403	48	49					41	239
14	50								236	233
15	51									

a) Schedule possible with full paths and duplication

Cycle_num	I1	I2	I3	I4	I5	I6	I7	I8	B1	B2
1	10	162	163							
2										
3										
4	12									
5	13									
6	42	43								
7	44									
8	37									
9	17									
10	38									
11	20	21							41	
12	45									
13										
14										
15	46									
16	47									
17	48	49								
18	50									
19	51									

b) Schedule as generated by Trimaran scheduler

Figure IV.24: Schedules that were created using predicated (b) and Full Path predicated code (a).

Height Reduction allows guarding predicates to be defined as soon as possible, reducing the amount of speculation needed.

By maintaining information about each of the control paths that exist in a hyperblock, PSSA can provide information that allows precise placement of renamed and speculated code, and allows the correct, renamed values to be propagated to subsequent operations. The renaming used by PSSA allows more aggressive speculation, as overwriting live values is no longer a concern. In addition, PSSA supports Control Height Reduction along every control path using full-path predicates, reducing control dependence depth throughout the hyperblock.

Our experiments show that PSSA is an effective tool for optimizing predicated code. Using PSSA with PSpec and CHR results in a reduction in executed cycles ranging from 12% to 62% for a 16 issue machine.

We also suggested methods for reducing the amount of code expansion created by the PSSA transformation. These techniques included reducing the region over which the full-path-predicates created by PSSA were assigned, and a critical path first scheduling algorithm. We showed that the amount of code expansion could be decreased by 40% over that created by the original PSSA algorithm for the example we tested.

Chapter V

IA64SimpleScalar: A Framework for Testing Hardware Optimizations

Because architectures supporting complete predication follow the EPIC philosophy that the compiler should perform as many tasks as possible, keeping the hardware relatively simple, most optimizations involving predication have been directed towards the compiler. However, current EPIC implementations such as the Intel Itanium (discussed in in Chapter III) are fully scoreboarded, thus requiring data analysis to occur in the hardware as well as in the compiler stage. It is important, then, for the analysis done in the hardware to be predicate aware just as it was for the compiler analysis shown previously. In the next chapter, we present a method of extracting and using predicate relationship information at runtime.

A limitation that current VLIW implementations of predicated architectures have is the ability to adjust their schedules to react to largely unpredictable occurrences such as cache misses. As mentioned, the Itanium is scoreboarded. This allows it to stall only on a use of a load instruction that misses in the cache. However, the lack of dynamic scheduling ability prohibits the architecture from taking steps to mask the unexpected latency by scheduling instructions that are ready to execute while the load use waits. Chapter VII presents a method for adding a small amount of dynamic scheduling capability to the pipeline while trying to minimize the added complexity.

Before these hardware optimizations are presented, the rest of this chapter presents the IA64 simulator created and used to simulate the different architectures to which our experiments are directed.

V.A EPIC Simulation Using IA64SimpleScalar

To support this experimental research we needed a IA64-based functional simulator with open source that was robust and flexible. Finding that none existed, we undertook the task of creating one ourselves. We decided to build the simulator using as much existing infrastructure as was available. We built our simulator by modifying the SimpleScalar Tool Set [16]. The next few sections describe the simulator that was built based on the first implementation of the IA64 ISA, the Intel Itanium, the process we went through to build it, and some of the choices that were made.

We created a simulator that modeled a current implementation of an IA64 architecture. This meant creating a simulator that knew how to handle predicated instructions and such features as *rotating registers*. Because only statements guarded by a value of true are committed, it was important to be able to determine the predicate register values as we simulated the code. It addition, since rotating registers are incremented based upon a set of system registers, we had to be able to access those registers. It was important to know the current values of the rotating registers to get dependences right. The simulator had to be able to parse the IA64 ISA, extracting bundle, stop bit, and resource usage information from the template. Of course, we needed to be aware of effective addresses for memory instructions to simulate cache access correctly. In other words, we needed a fully functioning simulator that could maintain and report machine state and understood completely the IA64 ISA.

V.A.1 ISA Implementation

IA64SimpleScalar uses architectural event traces instead of emulating the application state of a processor. Our traces are generated on IA64 machines running Linux through the ptrace system interface [27] that allows a parent program to single-step a spawned child. Through this we record the instruction pointer (IP), the predicate register values, effective-address for memory instructions, the previous function state (PFS) for return, and the current frame marker (CFM). This data is recorded into a file for each dynamic instruction executed. IA64SimpleScalar then reads in the trace file to simulate the program's execution.

IA64SimpleScalar decodes the traces using an opcode library containing a record for each IA64 instruction, and a library that interprets each instruction. This was built from the GNU opcode library that contains opcode masks to match the instruction, operand descriptions, mnemonic, and type classification. We enhanced this by adding a unique instruction identifier, quantity of register writers, and target Itanium functional unit.

Unique Features of the IA64 ISA

Several unique features of the IA64 ISA needed to be considered. It is interesting to note that not all bundles include 3 instructions. The ISA includes a category of instructions called *extended instructions* [5]. One of these instructions occupies 2 execution slots in the bundle. When calculating the default next instruction address, this information had to be taken into account.

Several instructions write registers without an explicit encoding to this effect. As mentioned earlier in the discussion of software pipelining, IA64 implements a category of *loop-type* branches. The instructions associated with these `bt.ctop, br.cexit, br.wtop, br.wexit` may read or write *LC* (loop count), *EC* (eplilog count), the *rrbs* (rotating register base), and predicate register 63 [5]. Consequently, register dependences are created that are not explicit.

The *post increment or base update* memory operations are another example of instructions causing unexpected dependences. In this form of a load or store instruction, the base register used to calculate the effective address is automatically incremented (by an amount designated in the instruction) after the memory operation is executed. Generally, the base register is described as a *use* when calculating dependences, but in this case the instruction also redefines the base register.

A special challenge in determining dependences is the existence of rotating registers themselves. A register coded in the instruction as r42 may actually point to another register if it has been designated to be a rotating register. As mentioned, the range of general registers is programmable. Consequently, to determine dependences between definitions and uses of these registers, we must first determine if it is a rotating register, and then adjust it based on the value of the *rotating register base* (`rrb`) for the cycle the use or definition occurred.

The `rrb` is the index which is added to the number representing the explicitly named register to determine the actual register to be accessed. There is a different `rrb` for each type of register file that can rotate. Hence the CFM contains the value of the *rrb.gr* for general registers, the *rrb.pr* for predicate registers, and the *rrb.fr* for floating point registers. As mentioned earlier,

only a certain range of registers in each file can be used as rotating registers. For the general registers, this is programmable in multiples of 8 beginning with `r32`. The size of the range for general registers is included in the CFM. Floating point registers `f32 - f127` and predicate registers `p16 - p 63` are included in the range of rotating registers.

A copy of the CFM for the cycle the instruction began execution is stored in the trace with the instruction. When dependences are set for a particular instruction, the registers referenced in the instruction are first adjusted by the index in the appropriate `rrb` found in the CFM. Then, they are checked against the registers of definers that have been adjusted in the same manner.

In addition to giving SimpleScalar the ability to recognize and deal with a new ISA, some significant additions had to be made to the SimpleScalar base model. SimpleScalar is primarily a functional simulator of non-predicated out-of-order architectures. However, our immediate need was to be able to simulate the in-order, predication-capable, Itanium architecture.

V.A.2 Predicated RAW and WAW dependences

Because in-order processors in general, and the Itanium in particular, do not implement complete register renaming, WAW [42] dependences must be maintained to ensure that instructions can indeed be executed in parallel. In addition, The evaluation of dependences in a predicated environment is significantly different, as described previously. As noted in Wang et.al [56], the scoreboarding mechanism used to manage dependences must be a matrix with predicates involved instead of the typical vector. Dependences can be removed as predicate values become known.

V.A.3 ALAT

As mentioned in Chapter III, the Itanium implements Data Speculation using the Advanced Load Table (ALAT). Consequently, the existence of such a table had to be simulated. There are 2 kinds of advanced loads. The first situation involves speculating only the load above a store. In this case, a `ld.c` instruction is placed in the original location of the speculated load. When the `ld.c` statement is reached, a check is made to see if the store wrote to the location that the load accessed. If so, the load is re-executed. No other action is taken, and the penalty incurred is a 10-cycle pipeline flush. The second situation involves speculating the load and some of its uses above the store. In this case, a `chk.a` instruction replaces the load. If a store

has written to the location of the original load, the `chk.a` jumps to the location containing recovery code to re-execute the load and the instructions that were speculated along with it. The cost of a `chk.a` that misses in the ALAT is estimated to be at least 50 cycles [6].

V.A.4 The Expand Stage

The Itanium pipeline has an Expand stage that is unlike any simulated by SimpleScalar. This stage was added. The stage expands the bundle information, determining from the templates which functional units will be used. As long as the appropriate functional units are available, the pipeline assigns functional units to instructions, with no attention paid to possible outstanding dependences. Logic had to be added to SimpleScalar to extract the unit information from the templates and other pipeline stages had to be adjusted to reflect the fact that functional units were assigned prior to dependences being satisfied.

V.A.5 Commit Stage

Only instructions with a guarding predicate whose value is true will be committed or produce dependences. If the instruction to be committed was a predicate defining statement, dependences must be broken for all instructions guarded by this predicate. If the instruction was a store, the cache access that might have occurred in this stage should not occur if the store has a false guarding predicate.

Chapter VI

Disjoint Path Analysis

One of the features of the Explicitly Parallel Instruction Computing (EPIC) architecture is its support for predicated execution. Predicated Execution, in the form of if-conversion [9, 41], removes hard-to-predict branches by combining both paths of a branch into a single path. In doing so, definitions of the same logical registers (originally from different paths) are intermingled. This makes data dependence analysis significantly harder. Without the appropriate predicate sensitive analysis, dependency assignments must be very conservative, ultimately including dependences that are not required.

The EPIC philosophy is that the compiler should handle most of the dependence analysis and scheduling in order to simplify the processor, and at the same time the compiler has a broader view of the code [33]. In the case of the Intel Itanium (the first implementation of the IA64 ISA), a scoreboard is used by the hardware to make decisions on instruction dependences. While some of the independence information can be encoded by the complier into the VLIW instruction grouping, or *bundle*, and passed on to the architecture, much of it will have to be re-calculated by the hardware without the benefit of predicate relationship information.

In this chapter, we describe a Disjoint Path Analysis Architecture that allows us to re-create predicate relationship information in hardware. We show that this architecture can be used to decrease the number of data dependences that are conservatively enforced for the current Itanium IA64 implementation. If the predicate relationship information indicates that a definition's path is disjoint from the use, the data dependence is not assigned. This means that the definition was on a disjoint path.

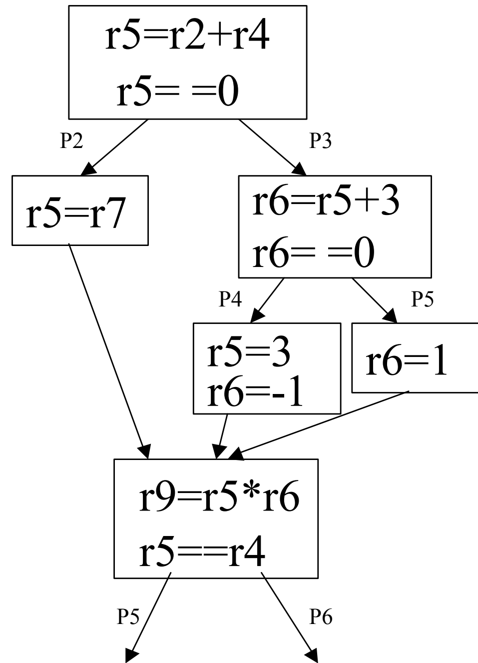


Figure VI.1: Original Control Flow Graph

Our results show that up to 14% of the dependences in if-converted regions can be removed for the Itanium model, yielding an improvement in IPC of up to 6% for these regions.

This chapter is organized as follows. Section VI.A describes in more detail the data dependence problem caused by predication. Section VI.B reviews previous work in the area of path analysis of predicated code. Section VI.C describes the Disjoint Path Analysis Architecture, including the use and implementation of the data structures involved. Section VI.D presents information on the methodology and benchmarks used to produce our results. Section VI.E explains the model we use to determine the effects of the Dynamic Path Analysis on an in-order architecture and provides the results of those experiments.

VI.A Multiple Paths Means Multiple Definitions that Result in Stalls

Predicated execution is a feature designed to increase ILP and remove hard-to-predict branches. Machines such as the Intel Itanium, with hardware to support predicated code, include an additional set of registers called predicate registers. The process of predication replaces branches with compare operations that set predicate registers to either true or false

```
      add r5=r2,r4
      cmp P2,P3 =r5,0
(P2) mov r5=r7
(P3) add r6=r5,3
(P3) cmp P4,P5 = r6,0
(P4) mov r5=3
(P4) mov r6=-1
(P5) mov r6=0
      mult r9=r5,r6
      cmp P5,P6=r4,r5
```

Figure VI.2: If Converted Version

```
[1]       add r5=r2,r4
[2]       cmp P2,P3 =r5,0
[3] (P2) mov r5=r7
[4] (P3) add r6=(r5[1]or r5[3]),3
[5] (P3) cmp P4,P5 = r6,0
[6] (P4) mov r5=3
[7] (P4) mov r6=-1
[8] (P5) mov r6=0
[9]       mult r9=(r5[1]or r5[3]or r5[6]),(r6[4] or r6[7] or r6[8])
[10]      cmp P5,P6 = r4, (r5[1] or r5[3] or r5[6])
```

Figure VI.3: If Converted showing multiple definitions of same register

based on the comparison in the original branch. Each operation is then associated with one of these predicate registers (the operations *guarding predicate*). In general, the operation will be committed only if its guarding predicate is true. This process of replacing branches with compare operations is called *if conversion* [9, 41].

An example of if-converting a set of basic blocks into a predicated region can be seen in Figures VI.1 and VI.2. Figure VI.1 shows the original control flow graph with 3 possible paths to the final block shown in the region. Figure VI.2 shows the if-converted code with the branches replaced and the 3 paths effectively combined into one.

As already mentioned, the benefits of if-conversion include the removal of hard to predict branches, and increased possibilities of finding *instruction level parallelism* (ILP). Normally, when a branch is encountered, the processor predicts the next address from which to fetch instructions and continue execution. There can be a large penalty if the prediction is incorrect. Predicated execution allows for a third possibility, executing both paths of the branch. Predication can also increase ILP because the predicated region can provide a larger pool from which to find independent instructions.

However, the process of combining multiple paths into one makes data dependence analysis significantly harder. Multiple paths containing definitions of the same architectural registers are intermingled. In the control flow graph in Figure VI.1, there are 3 definitions of `r5`, each occurring in a different basic block. However, in the if-converted code, the 3 definitions are in the same predicated scheduling region. When the compiler or hardware (in the case of an out-of-order execution model) tries to set up dependences, it is not just a matter of the last definition being the reaching definition for a use.

It is clear from the control flow graph in Figure VI.1, that the definition of `r5` made by the `mov` instruction `r5=r7` could not be the definition used by the instruction `r6=r5+3`. The definition and use in this case are on completely separate (*disjoint*) paths. However, in the if-converted code (Figure VI.2), the `mov` provides the most immediate prior definition. If the hardware has a dependence detection method telling it that `P2` and `P3` are disjoint, a use of `r5` will only have to wait on the correct definition. Otherwise, both of the 2 previous definitions of `r5` in the if-converted region must be considered as possible definitions. It would not be until the code is executed and it was determined that the guarding predicate of one of the defining statements is false that the dependence could be broken.

The `mult` instruction has multiple possible reaching definitions of `r5` that are all valid as shown in Figure VI.3. It could not be determined statically which of the definitions reached.

It would depend on the particular execution. If predicate registers `P3` and `P5` were defined as true, the very first definition of `r5` (`r5[1]`) would be the definition that reaches the use of `r5` in the `mult`. This is because it would be the only definition guarded by true, or the only definition on the taken path. However, if `P2` is true, there would be 2 definitions (from instructions 1 and 3) along the way guarded by true. Both of these are valid definitions. However, only the definition that is closest to the use and along the correct path provides the dependence. The problem in data dependence analysis is determining which is the closest and on the same path.

VI.A.1 Effect of Extraneous Definitions on the Hardware

In the baseline Itanium model, dependency relationships between a producer and consumer register, where at least one is predicated, cannot be handled by the hardware until the predicate value has been resolved. To accommodate this in Itanium, the producer of the predicate register and a potential consumer of a general purpose register guarded on that predicate must be scheduled 2 cycles apart from each other [6]. This holds true whether the predicate register has a value of true or false.

Consider the following code segment:

```
(1)       cmp P4,P5 = r8,r5    cycle 0
(2) (P4)  ld r7=[r5]           cycle 1
(3) (P5)  add r6=r7,3          cycle 2
```

Based on the current Itanium implementation, these statements must execute as shown above in their corresponding cycles. Statement 3 is a potential consumer of statement 2, so the architecture enforces a potential dependency between the two instructions. The dependency will be broken by the scoreboard when it is determined that either `P4` or `P5` has a value of false (which of course it will since these are disjoint predicates). However, as described in [6], the producer of the predicates and the potential consumer of the general register must be scheduled at least 2 cycles apart to allow time for this determination to be made. Under certain circumstances this latency is greater that 2 cycles.

In the above example, if the hardware can accurately determine that the predicates guarding instructions 2 and 3 are disjoint, then it can allow those two instructions to be scheduled and executed in the same cycle. This potential savings is the benefit exploited by the Disjoint Path Analysis architecture presented in this chapter.

The Itanium is fully *scoreboarded* to allow for real-time decisions to be made about instruction execution [33, 42]. The main function of the scoreboard is to determine when all dependences are resolved and to enforce WAW hazards. As mentioned, the mechanism is capable of breaking dependences when either the producer or consumer guarding predicate is evaluated to be false. For the scoreboard, many dependences have to be recalculated by the hardware, and we examine incorporating our Disjoint Path Analysis architecture into the Itanium scoreboard to eliminate false dependences (as described above) before the predicate definitions are resolved.

VI.B Related Work

Both compiler and hardware approaches have been proposed for handling multiple definitions.

VI.B.1 Predicated Multiple Path Compiler Analysis

Gillies et.al. and Schlansker et. al. [29, 45] presented the use of the Predicate Query System(PQS). This system uses a predicate partition graph to statically describe disjointness. A definition and a use that originated from disjoint paths cannot be dependent on one another. For a pure in-order execution model, the compiler would use the disjointness information and schedule the code accordingly. Two definitions of the same register guarded by disjoint predicates could safely be scheduled in the same cycle because only one definition would be guarded by true and ultimately be written. A definition and use on disjoint paths could also be scheduled in the same cycle for the same reason. PQS has been included in the later phases of the Intel IA-64 Compiler Code Generator [15] in the form of a relational database from which information on predicate disjointness, dominance, postdominance and predicate promotion [38] can be obtained.

In [18, 19], we presented the need for complete path analysis for predicated regions. This work extended the PQS research by maintaining information not only on predicate disjointness, but on the predicate predecessor/successor relationships that re-create a path through a predicated region. We presented the idea of Full Path Predicates (FPPs) to create predicates that represent a path through the predicated region. This allows statements to be predicated on the path that was taken to reach the statement, rather than only with a particular basic block as in predicates created by if-conversion. This allowed greater flexibility in instruction

```
     add r5=r2,r4
     cmp P2,P3 =r5,0
(P2) mov r5í=r7
(P3) add r6=(r5or r5í),3
(P3) cmp P4,P5 = r6,0
(P4) mov r5íí=3
(P4) mov r6í=-1
(P5) mov r6íí=0
     mult r9=(r5or r5íor r5íí),(r6 or r6í or r6íí)
     cmp P5,P6 = r4, (r5 or r5í or r5íí)
```

Figure VI.4: If Converted showing renamed definitions of same register

scheduling, speculation, and control height reduction.

Both the PQS and the analysis designed for creating FPPs were compile-time solutions to filling the need for specialized dependency analysis of predicated code for instruction scheduling. However, when the instructions are executed in hardware, the hardware must now deal with resolving these multiple definitions. The motivation for research presented in this chapter is to design a way to perform the analysis described above in the context of a real-time hardware environment, to eliminate the stalls described in section VI.A.1.

VI.B.2 Hardware Solutions for Dealing with Multiple Path Definitions

Wang et.al. [56] recognized that multiple definitions would be a problem in the renaming stage of an out-of-order implementation of an architecture supporting predicated code. The renaming stage is used to give each definition of an architectural register a unique physical name (removing WAW and WAR dependences). In the presence of predication it is possible to have multiple instructions, guarded by different predicate registers, write to the same architectural register. When a use of this architectural register is encountered in the rename stage, the values of the predicates may be required to determine which physical register to map to the architectural register. If the predicate values are not yet available, a stall must occur.

Figure VI.4 provides a renamed version of the code in Figure VI.2 and illustrates the

virtual register	V	slot 0 renamed reg pred	slot 1 renamed reg pred	slot 2 renamed reg pred	slot 3 renamed reg pred
0					
Ö					
5	ï	r5 p0	r5í p2	r5íí p4	
6	1	r6 p3	r6í p4	r6íí p5	
7					
8					
9	ï	r9 p0			
.					
.					
n	.				

Figure VI.5: Augmented Register Alias Table used to implement the select-μop optimization for out-of-order processors supporting predicated execution.

problem that can occur in the renaming stage. The mappings of the first few statements are unambiguous. The physical register `r5` defined by the first `add` can be mapped to the first use in the `cmp`. However, it is unclear which renamed version of `r5` should be mapped to the use of `r5` in the second `add`. It is even more ambiguous which definition will reach the use in the `mult`. This ambiguity cannot be removed until the values of predicate registers `P2` and `P4` are known.

In an effort to remove as many unnecessary stalls as possible, Wang et. al. proposed the use of the select-μop instruction for an IA64 out-of-order execution model. The new select-μop instruction was based on the phi-node used by static-single-assignment (SSA) [26]. It allows the resolution of multiple definitions to be postponed to later stages of the pipeline, providing more chance that a stall would not have to occur. To form the select-μop instructions, the possible definitions need to be kept track of. To this end they presented an augmented *Register Alias Table* (RAT), and use this to create the select-op instructions. The RAT used for this optimization is shown in Figure VI.5. Each set represents the current logical definitions for a given register. Each block contains the renamed definition and the Guarding Predicate under which it was defined. The most recent definition (this would have the highest priority) guarded by a true predicate would be the correct definition. Once this definition was encountered, any further dependences left to be reconciled could be eliminated from consideration.

The entries are ordered in the table with the most recent definition in the highest

numbered slot. Each set has four entries. If there is no available slot when a new definition must be entered, a select-μop is created, replacing all four entries. A select-μop is also created and inserted into the instruction stream when any use of a register with multiple definitions is encountered in the rename stage. For example, the first few instructions from Figure VI.4 with the select-μop included would be:

```
    add r5=r2,r4
    cmp P2,P3=r5,0
(P2)mov r5'=r7
    select r5=r5,r5'
(P3)add r6=r5,3
```

This removes any ambiguity as to which renamed version of `r5` should be mapped to the use in the final `add` instruction, avoiding the need to stall in the rename stage. A stall might be necessary if the guarding predicates of the definitions of `r5` are not determined by the time the select-μop executes.

The select-μop is not designed to be helpful for in-order processors. It is an optimization for the renaming stage, which is not a part of the in-order pipeline. In addition, it inserts another instruction and layer of dependency into the instruction stream. We use a modified version of the RAT for Disjoint Path Analysis architecture described in the next section. We do not store renamed physical registers in the RAT, since we concentrating on an in-order VLIW architecture.

VI.C Disjoint Path Analysis Architecture

In this section, we describe the Disjoint Path Analysis Architecture that allows the same predicate sensitive path analysis done in the compiler to be accomplished in the hardware. To support this analysis, we add the Path Information Table (PIT) and the Last Definition Table (LDT) to the Itanium Implementation. In addition, we replace the register status table (the mechanism for determining if there is an outstanding write of each of the logical registers) with an extended Register Alias Table (RAT). These tables can be seen in Figures VI.6, VI.8 and VI.9 in various stages as we progress through processing the code in Figure VI.3. Register Alias Tables are common to out-of-order processors to facilitate renaming. We use a modified

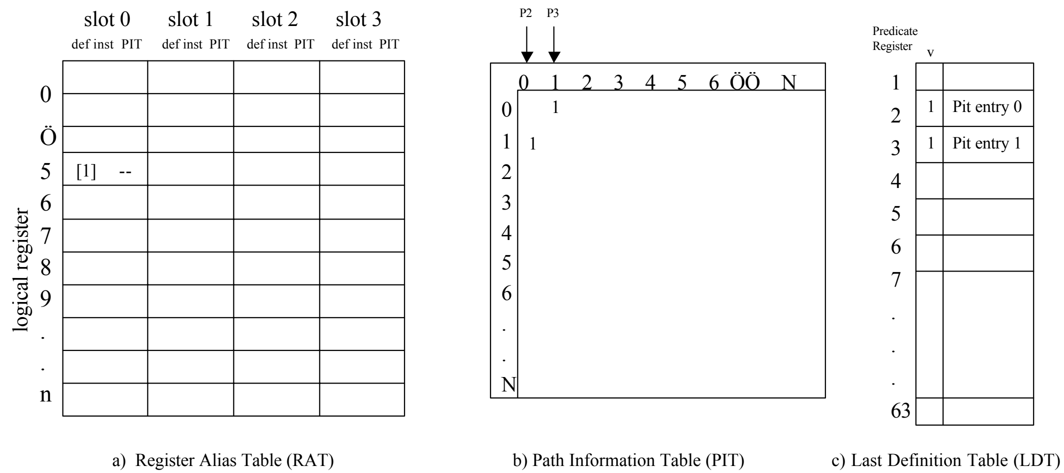


Figure VI.6: Three tables shown after first `cmp` statement is processed. One definition of `r5` has been entered into the RAT. As is it unguarded, there is no PIT entry associated with it. In the PIT, the bits are set at the intersections of locations 0 and 1 indicating that predicate registers `P2` and `P3` are disjoint. The LDT indicates that the current definition of predicate register 2 is found in PIT entry 0, and predicate register 3 at PIT entry 1.

version to maintain information about definitions that define the same logical register from the different paths combined during if-conversion. These registers are not renamed, as in the out-of-order use of the RAT [56], since we are modeling an in-order VLIW processor. The other structures are unique to the Disjoint Path Analysis Architecture. The PIT is used to maintain disjointness information about predicates that guard register definitions and uses. The LDT is used to provide information to the RAT about the latest reference in the PIT to a particular predicate definition.

The rest of this section describes these structures in more detail. In particular, we will discuss how these structures are utilized and updated in the Disjoint Path Analysis Architecture to provide information critical to eliminating non-essential register dependences.

VI.C.1 Register Alias Table

In an EPIC architecture, there can be multiple possible register definitions for a given operand as described previously. As shown in Section VI.A, extraneous definitions could cause unnecessary stalls in the pipeline.

To facilitate the process of determining the correct definition, an extended *Register Alias Table (RAT)* is used, adapted for our purposes from [56]. The RAT implementation is

used to maintain a list of the possible definitions for the use of a logical register. Since we are concentrating on an in-order model, the multiple definitions are not due to renaming. Instead, they are the result of the same logical register definition along different disjoint paths merged together through if-conversion. Consequently, each entry in a set for a logical non-predicate register contains not a new physical register, but a reference to the instruction that created the particular definition.

The second part of the slot entry is not just the predicate on which the definition was guarded, but a reference to the location in our Path Information Table where disjointness information associated with the definition's guarding predicate can be found. The Path Information Table entry comes from the LDT. If the entry in the LDT corresponding to the guarding predicate of the current instruction is valid, this information is recorded in the RAT. If the entry is invalid, no reference to disjointness information is made. This will be discussed further in Section VI.C.2.

The RAT is updated in the decode stage for each register definition with one exception. If there is not an empty slot when a new definition is encountered, the defining instruction stalls the whole front-end of the processor until a slot has been vacated. We modeled the RAT using various numbers of slots ranging from 4 to 16. With 16 slots the pipeline never had to stall due to lack of available slots. With four slots, we did have to stall occasionally, but the effects on the IPC were relatively small as we will show in Section VI.E.

VI.C.2 Predicate Information Table and Last Definition Table

The Predicate Information Table (PIT) is updated to maintain information on disjointness between predicates. The table is then used to answer the question:

- **Can two given instructions possibly be on the same path?**

This is critical information because a dependence cannot exist between two instructions that are not on the same path. If we can answer this question about the second `add` instruction and the first `mov` instruction in Figure VI.2, we can know that the use of `r5` in the `add` cannot be dependent on the definition of `r5` in the `mov`.

As in the example in Figure VI.6 depicts, the PIT is a matrix NxN representing the last N definitions of predicate registers. A given logical predicate definition can be represented multiple times in the table. We refer to these definitions as predicate definition *instances* because multiple vectors may reflect information about the same logical predicate register,

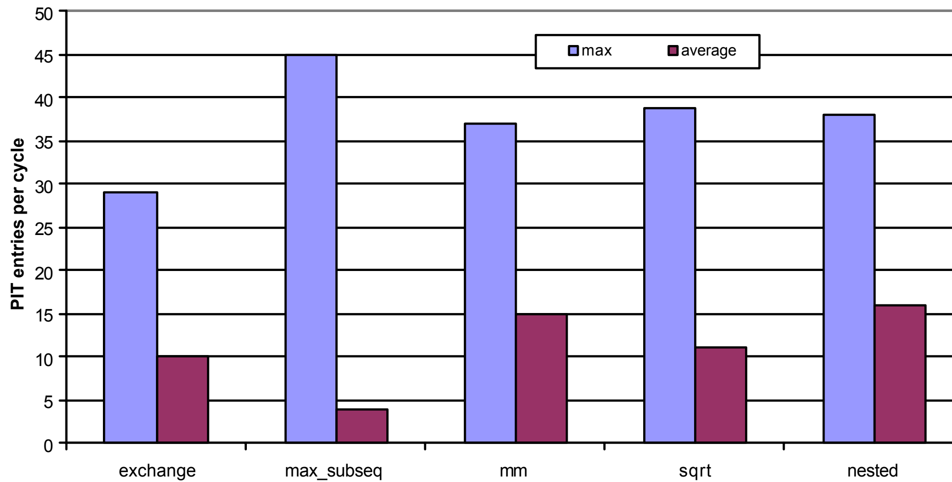


Figure VI.7: Maximum and average number of PIT entries required per cycle to support references in the RAT.

only a different instance of its definition. For example, Figure VI.2 shows `P5` being defined in two different places in the code. Each of these definitions will have different disjointness information associated with them and therefore must be represented separately. The rows in the PIT represent the predicate register instances with which the given predicate definition (represented by the column) is disjoint.

Each of these definition instances must remain *live* until it is no longer possible that an instruction guarded by it remains in the pipeline. Consider the following definitions:

```
(1)      cmp P4,P5 = r8,r5
(2) (P4) mov r7=r5
(3) (P5) cmp P9,P4 = r9,r5
(4) (P4) add r6=r7,3
```

When the dependences for the final `add` are calculated, it is important to have the disjointness information for the prior definition of `r7` available. The `add` and the `mov` are guarded by the same logical predicate register, but are not necessarily on the same path. To maintain correct disjointness information for the `mov`'s definition, we cannot allow the latest definition of `P4` to replace the previous definition in the PIT.

The first PIT definition for `P4` can be freed once the second definition of `P4` in our example is committed. Since we are executing instructions in order, at this point we are

guaranteed that there are no instructions guarded by this definition left in the pipeline. In addition, all of the predicates represented in the PIT are unconditional predicate defines, so they are guaranteed to define their two predicates even if their guarding predicate is false. For example, in statement 3, `P9` and `P4` will be defined even if `P5` is false.

A disjoint representation in the PIT, means that those two predicates are guaranteed to be disjoint. If two predicates are not represented as disjoint in the PIT, then no disjoint information is know about those two predicates. They either may or may not be disjoint. The PIT only represents disjointness information between two predicates that it can guarantee to be disjoint.

Any PIT column entry can be allocated to a given predicate definition during execution. Each predicate definition that occurs during execution is allocated the first free PIT vector. We maintain a queue of pointers to free pit vectors. The deallocation of PIT entries is handled by the LDT and described below. If there is not a free PIT entry (the free list is empty), no disjointness information will be recorded about the predicate definition instance. We used a 45x45 matrix in this work and never encountered a lack of available PIT entries. Figure VI.7 shows the maximum and average PIT entries that the RAT referenced in a given cycle. The maximum number of entries required ranged from 29-45 for the benchmarks we tested.

The matrix is initialized so that every entry (corresponding to a column) is a bit vector set to 0. To determine which predicate definition instances are disjoint from another predicate instance `x`, the column of `x` is read. This produces a bit vector representing the disjointness of predicate `x` in relationship to all of the other predicate definitions represented in the PIT. A bit set in the `n`th location of the vector indicates that Predicate definition instance `n` is disjoint from instance `x`.

The disjointness information is accumulated in the PIT from 2 sources. First, if the predicate defining statement is guarded by a predicate, the disjointness information is inherited from its guarding predicate. In Figure VI.2 `P3` guards the definitions of `P4` and `P5`, so `P4` and `P5` are successors of `P3`. Both `P4` and `P5` inherit the disjointness information from `P3` in the PIT. If the predicate defining statement is not guarded, or guarded by P0 (the constant *true*), the predicate is initially disjoint from no other predicate definitions. In our example, the definitions of `P6` and `P5` (second instance) are unguarded, so they do not inherit disjointness information. Second, disjointness information is added to the PIT stating that the newly defined predicates (`P5` and `P6`) are disjoint.

The Last Definition Table is shown in Figures VI.6, VI.8 and VI.9 at various stages as entries are made. It contains two entries for each logical predicate register. One entry is a pointer to the Column of the PIT representing the last definition instance of a given predicate register. For example, LDT[4] will contain a pointer to the last unconditional definition of `P4`, and LDT[5] to the last definition of `P5`. The other field in the LDT entry indicates if the last definition of the logical register contains a valid PIT entry. It may not be valid if either of the following are true:

- There was not an available PIT vector when the latest definition was encountered.
- The last definition of the predicate was conditional. This could mean that it was a type of `or` or `and` predicate definition [5]. We do not keep disjoint information for these definitions in the PIT. For the benchmarks we tested, these types of definitions comprised an average of .4% of the predicate definitions.

When a new entry is made into the LDT, the current PIT reference for the predicate begin defined is carried by the `cmp` instruction making the new entry. When this `cmp` instruction commits, it causes the deallocation of the PIT entry associated with the prior definition. The reason for this policy was discussed previously in this section. An entry is made in the PIT free list with the pointer (carried by the committing `cmp`) to the deallocated vector. This scheme is similar to register deallocation for out-of-order processors.

The LDT information is used by the RAT when recording the pointer to the PIT location of the guarding predicate of the definition being entered. This will be explained in more detail in Section VI.C.5.

On a branch prediction, our implementation will checkpoint both the PIT and the LDT. If a misprediction is determined, the PIT and LDT will be restored to their pre-branch condition. It is not necessary to checkpoint the RAT, as all instructions can continue through the pipeline, committing pre-branch definitions and squashing mis-predicted ones. If we wish to avoid the cost of checkpointing, an alternative is to clear the LDT and PIT on a mis-prediction and assume no disjointness information is available on recent predicate definitions.

VI.C.3 Using the PIT and RAT to Determine Actual Dependences

The PIT is accessed at most once per instruction when dependences are being set. The RAT will be accessed N times where N is the number of operands in the current instruction. These tables are used as follows:

- **Current instruction is Guarded.** When processing an instruction, the vector associated with the guarding predicate of that instruction is read from the PIT. To determine which PIT entry to read, the PIT entry corresponding with the last definition of the predicate we are guarding is looked up in the LDT. Therefore, the LDT and PIT vector read for processing an instruction is done serially. For each operand in the current instruction, their corresponding RAT entries are read in parallel. These entries can be read in parallel with the LDT and PIT lookups. Each RAT entry contains a pointer to the dynamic instruction producing the value for that definition, along with a pointer to the PIT entry that represents that defining instruction's guarding predicate. This PIT pointer is used to compute the disjointness of the RAT entry definition from the use, for the instruction we are processing. The PIT pointer of each definition is looked up in the PIT vector associated with the guarding predicate of the current instruction. If the bit for the pointer is set in the vector, the definition associated with that pointer cannot create a dependency for the current instruction. If not, or there is no PIT pointer in the RAT entry, a dependency *may* exist and the architecture will treat that definition as a potential input dependency.

  Figure VI.8 shows the PIT after the second `cmp` instruction in Figure VI.3 was considered. Entries have been made for definition instances of `P2`, `P3`, `P4` and `P5`. Let us consider what happened when the instruction `add r6=r5,3`, prior to the `cmp` was processed. First, the dependences must be set. The `add` is guarded by `P3` so we read the PIT vector associated with `P3`. This would be PIT 1. Next, we read the possible definitions for the `add`'s only use, `r5` from the RAT. At this time, the only definitions in the RAT for `r5` are `r5[1]` and `r5[3]`. We compare the PIT pointers associated with each RAT entry to the bits set in PIT vector 1. We see that the bit for the location associated with `P2` (Row 0) is set. This means that the guarding predicate instance of definition `r5[3]` is disjoint from the instance of the guarding predicate of the `add`. Consequently `r5[3]` should not be considered as a possible definition, and will not set a dependence. However, `r5[1]` will set a dependence since it is unguarded and cannot possibly be disjoint.

- **Current instruction not guarded or Guarded by P0.** The PIT is not accessed. Each possible definition for an operand in the RAT is a possible dependency to the use, and we cannot narrow the dependences down with our disjointness information.

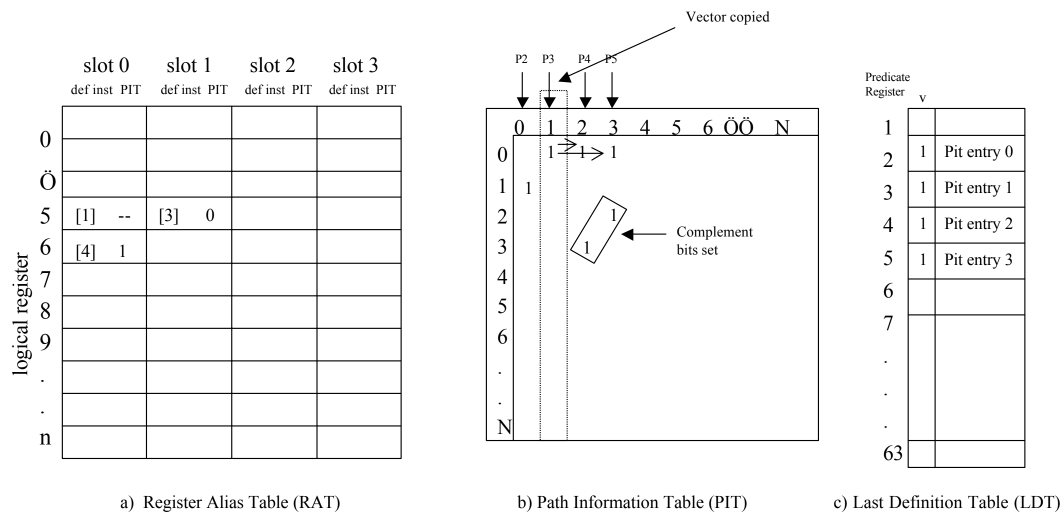


Figure VI.8: Three tables after second `cmp` statement is processed. Two new definitions have been added to the RAT and the LDT. In the PIT, the definitions of `P4` and `P5` are guarded by `P3`, so the disjointness information for `P3` is inherited by `P4` and `P5`.

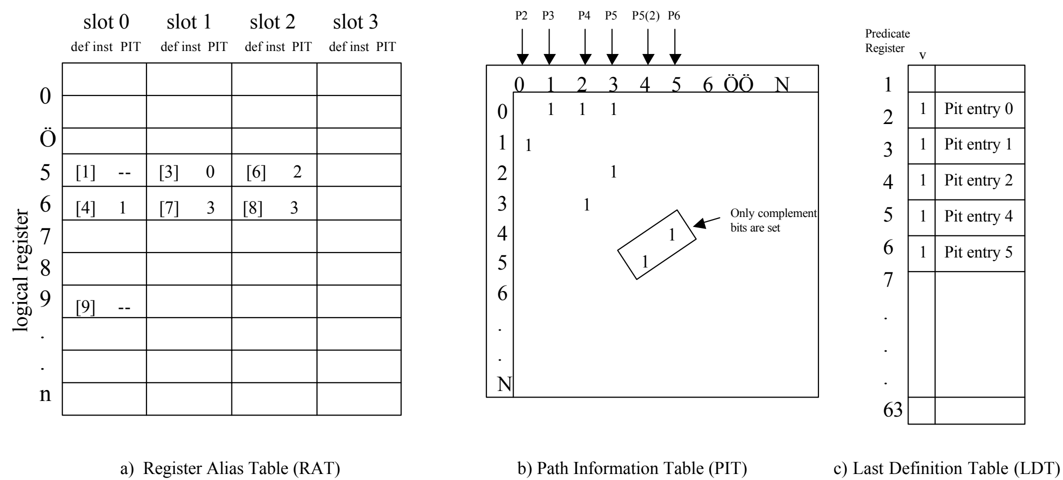


Figure VI.9: Three tables after third `cmp` statement is processed. A new definition of P5 is made and given its own disjointness information. The old definition of `P5` in the LDT is replaced. Three new definitions are added to the RAT.

VI.C.4 Updating the RAT

The RAT is updated in the decode and writeback stages except as mentioned earlier. When a defining instruction is processed, the register defined is entered into the RAT for the set corresponding to the logical register. It will be assigned a new available slot without replacing another with the following exceptions:

- An entire set in the RAT will be cleared for a logical register if the next entry to be made is un-guarded, or guarded by `P0` (the constant TRUE).
- A single slot will be replaced when a logical register is re-defined guarded by the same predicate instance.

RAT entries are removed during writeback when the defining instruction is committed or invalidated due to a guarding predicate with the value of false.

VI.C.5 Updating the PIT

After the operands are processed for an instruction, the definitions are examined. If we have an unconditional `cmp` statement (used henceforth to represent all predicate defining statements), the instruction will update the PIT for the predicates defined. The next two free entries in the PIT will be cleared and allocated to the two new predicate definitions. The LDT will be updated to reflect the new most current definitions of the predicate registers defined. If we have a conditional predicate definition, the LDT valid bit will be set to invalid and no pit entries will be made. Two writes are performed into the PIT vectors, according to the rules below:

- **Inherit Disjointness Information**
  - **A Guarded cmp**. If the `cmp` is guarded, the PIT entry corresponding to the guarding predict is used to initialize the two vector PIT entries for the two new definitions. In doing this, we capture the inherited disjoint set information. Figure VI.8 shows the effect created when the second `cmp` statement is processed. The `cmp` statement was guarded, so the entries in the column of the guarding predicate instance of `P3` are copied into the the columns of the newly defined instances of `P4` and `P5`. The vector allocated to guarding predicate `P3` was determined from LDT[3]. The complement bits are set as described below.

  - **An Unguarded cmp**. If the `cmp` is not guarded, or guarded by `P0` (the constant TRUE), the vectors of the newly defined predicates remain cleared. Figure VI.9 shows the PIT after the third `cmp` statement is encountered. Only the complement bits are set according to the next rule. Notice that the second instance of `P5` has different disjointness information than the first instance and that the entry in the LDT for `P5` pointer to the last definition.

- **Set Bit for Complement Predicate**. The two predicates defined in an unconditional `cmp` statement are always disjoint. We set the location of the complementary predicate in each of the vectors to indicate this.

  Figure VI.9 shows the complement bits set for the third `cmp`. `P5(2)` and `P6` are complementary predicates. `P5(2)` is allocated vector 4, so this location is set in vector 5 allocated to `P6`.

VI.C.6 Predicates Defined False

Statements guarded on false predicates are by definition not on the executed path. Consequently, they never create dependences and can be considered to be disjoint from every valid path. Their relationships to other invalid paths are inconsequential. The PIT can reflect this information. When a false predicate definition is encountered, the associated row and column in the PIT will be set.

VI.D Methodology

For the work in this chapter we used a number of small benchmarks, chosen for structure which would produce predication. Current production compilers produce minimal code guarded on predication. Two of which were created for use with the Trimaran System [2]. These include `mm` and `sqrt`. In addition, we included a program that completes an exchange sort, a program that computes the maximum subsequence found in a list of numbers and test program called `nested` created in an effort to find a program that produced more if-conversion. Table VI.1 provides a description of each benchmark, the number of instructions in the trace, the percent of if-converted instructions produced, and the initial IPC without the path optimziation applied. All benchmarks were compiled using the Intel IA64 C++ Compiler using the -O3 compilation option with profiling.

benchmark	trace size	% if-convert	IPC	description
exchange	506969	10.2	.8083	Exchange sort routine
max_subseq	1421327	3.3	.7954	Finds maximum subsequence in a list
mm	937964	15	1.0373	matrix multiplication and summation
sqrt	560019	11.2	.8169	Newton-Raphson - saves partial results in array
nested	4635904	16.9	.9726	Nested loops and if-then-elses

Table VI.1: Presents description of benchmarks simulated including instruction count, percent of dynamic if-converted instructions and baseline IPCs for in-order and out-of-order execution.

Table VI.2 shows the parameters used with IA64SimpleScalar (described in Chapter V) for this study. The simulated microarchitecture was modeled after the Itanium. The functional unit distribution includes 2 integer units, 2 memory units (able to execute some IALU instructions as well), 2 floating point units and 3 branch functional units. The latencies used varied by instruction, and were derived from the Itanium Processor Microarchitecture Reference [6]. The memory hierarchy is modeled after the Itanium. The L1 data and instruction caches are 4-way associative, sized at 16K with 32 byte blocks. The L2 cache is unified, with a capacity of 96K, 6-way set-associative with base latency of 6 cycles. Floating point loads bypass the L1 cache and incur an extra 3 cycle latency in the L2 cache (totaling 9). The L2 cache will allow misses on as many as 8 outstanding cache lines at once [6]. Loads made from an address to which a value was stored within the last 3 cycles will bypass the L1 cache, seeking its value from the L2 cache. The L3 cache is unified, located off chip in the Itanium implementation. The base latency is 21 cycles with floating point loads requiring 24. Although the Itanium implements two levels of DTLB, we modeled the DTLB similar to the ITLB as fully associative, with 64 entries, 4k page size and a 15 cycle latency. Memory access is assumed to require 80 cycles. All models issue up to 6 instructions per cycle. Branch misprediction penalty is a minimum of 8 cycles.

VI.E In-Order Issue with Scoreboard

We apply Disjoint Path Analysis Architecture to our Intel Itanium in-order model derived from [33, 5, 15, 50].

The IA64 compiler *bundles* instructions into groups of three. A template included with the bundle is used to describe to the hardware the combination of functional units required

L1 I Cache	16k 4-way set-associative, 32 byte blocks, 2 pipeline cycles
L1 D Cache	16k 4-way set-associative, 32 byte blocks, 2 cycle latency
Unified L2 Cache	96k 6-way set-associative, 64 byte blocks, 6 cycle latency
Unified L3 Cache	2Meg direct mapped, 64 byte blocks, 21 cycle latency
Memory Disambiguation	load/store queue, loads may execute when all prior store addresses are known
Functional Units	2-integer ALU, 2-load/store units, 2-FP units, 3-branch
DTLB, ITLB	4K byte pages, fully associative, 64 entries, 15 cycle latency
Branch Predictor	meta-chooser predictor that chooses between bimodal and 2-level gshare, each table has 4096 entries
BTB	4096 entries, 4-way set-associative

Table VI.2: Baseline Simulation Model created to correspond the the parameters set by the Intel Itanium.

to execute the operations in the bundle. Stop bits are inserted in the instruction stream as part of the template to create *instruction groups*, or sets of instructions that are known to be independent of each other. Stop bits guarantee that the instructions issued together are independent of each other, but give no information on the relationships of these instructions to those outside of the group.

In the Itanium implementation, bundles are directed or *dispersed* to the functional units two at a time, subject to independence and resource constraints. The full two bundles (up to 6 instructions) are sent unless a functional unit is unavailable, or a stop bit is encountered indicating the end of an instruction group. If any of the instructions entering the functional units must stall because a data dependence is not yet satisfied, all of the instructions waiting to begin execution stall as well. This is essentially a pure in-order processor requiring a limited scoreboard to determine when the whole instruction group can start execution.

The scoreboarding mechanism is used to detect dependences between instructions outside of instruction groups. It allows dependences to be broken when either the producing or consuming instruction is found to be guarded by a predicate evaluated to false. Our Disjoint Path Analysis architecture focuses on eliminating these false dependences, allowing instructions to start executing a little earlier as described in section VI.A.1.

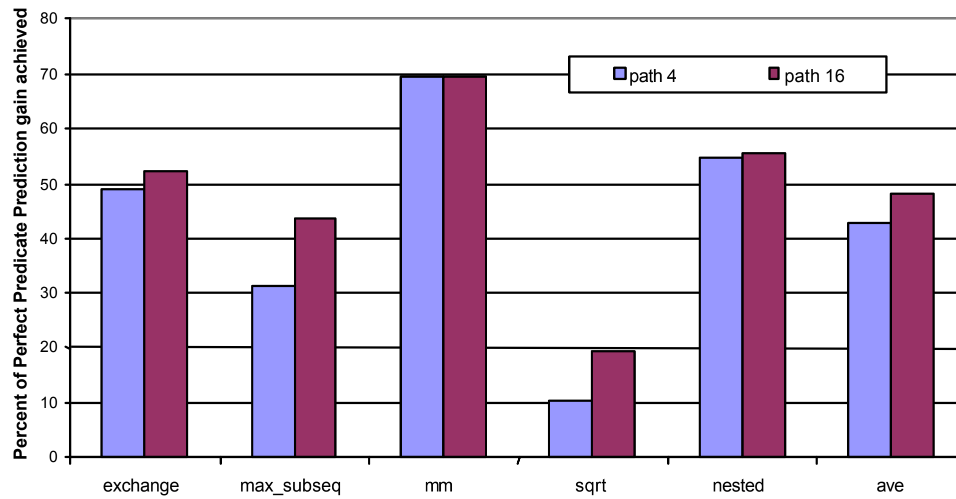


Figure VI.10: Captures the percent of improvement seen by the path optimization verses what could have been achieved by way of perfect predicate prediction

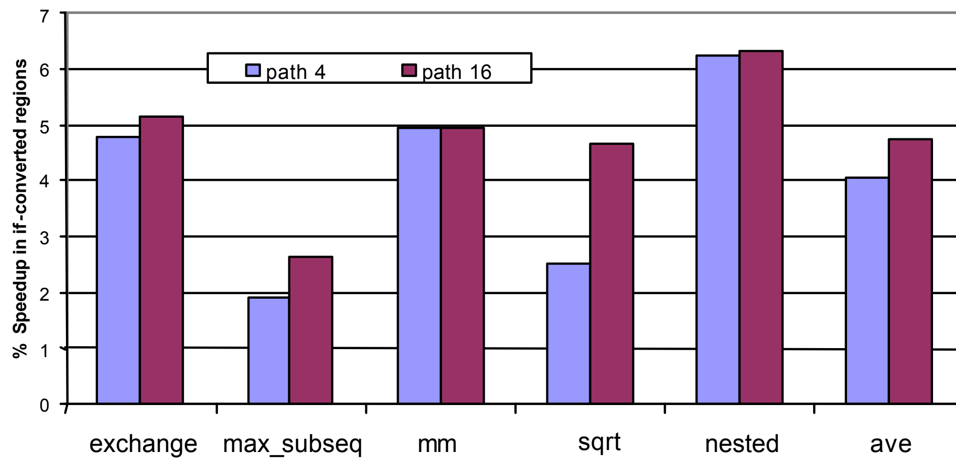


Figure VI.11: Improvement in IPC for Disjoint Path Analysis over Itanium Implementation in if-converted regions.

VI.E.1 Results

First, we compare our results with the results that could be achieved if we had perfect predicate prediction capabilities. This is important because if we had perfect knowledge of the values each predicate definition would ultimately assume, we could know exactly which dependences to set. No dependences would be set for instructions guarded by a predicate with the value of false, and the only definition to reach a use in a statement guarded by a true predicate would be the last definition guarded by a true predicate. Figure VI.10 compares the improvement in Instructions Per Cycle (IPC) for 2 path configurations against perfect predicate prediction. The Disjoint Path Architecture is represented with results for using a RAT with 4 and 16 definitions per register. In most cases, this distinction made little difference. In the cases of `sqrt` and, to a lesser extent, `max_subseq`, however, the difference was significant. On average, the gain received by the Disjoint Path Analysis was almost half of that which could be achieved given perfect predicate prediction to determine actual dependences.

Perfect predicate prediction can improve IPC for areas of the program that use predication for a variety of reasons while Disjoint Path Analysis produces improvement only in predicated regions due to if-conversion. Referring back to Table VI.1, we see that `mm` and `nested` have the highest percent of if-conversion and do the best compared to perfect predicate prediction, seeing 70% and 50% respectively of the gain achieved by perfect predicate prediction. We believe that the improvement shown by our techniques is limited due to the relatively small amount if-converted code produced by the compiler.

Next, we compare the number of dependences removed using the Path Analysis Architecture with a four entry PIT set against those removed using perfect predicate prediction. As shown in Figure VI.12, the average number of Read After Write (RAW) dependences removed was a little over 1% using the Disjoint Path Architecture while the number of Write After Write (WAW) dependences removed was less than half of that. Write After Read (WAR) dependences are not an issue due to the nature of in-order processors. However, the average number of dependences removed for the perfect-prediction version was over 16% for RAW dependences and almost 10% for WAW dependences. This is because, using perfect predicate prediction, no dependences were set between any producers or consumers guarded by false predicates. The perfect prediction version knew of all false definitions. In contrast, the Path Analysis could only determine if a producer and consumer were on the same path. A dependence could initially be set between a producer and consumer both of which had false guarding predicates. Note that these dependences will be broken by the scoreboard as soon as the producer or consumer

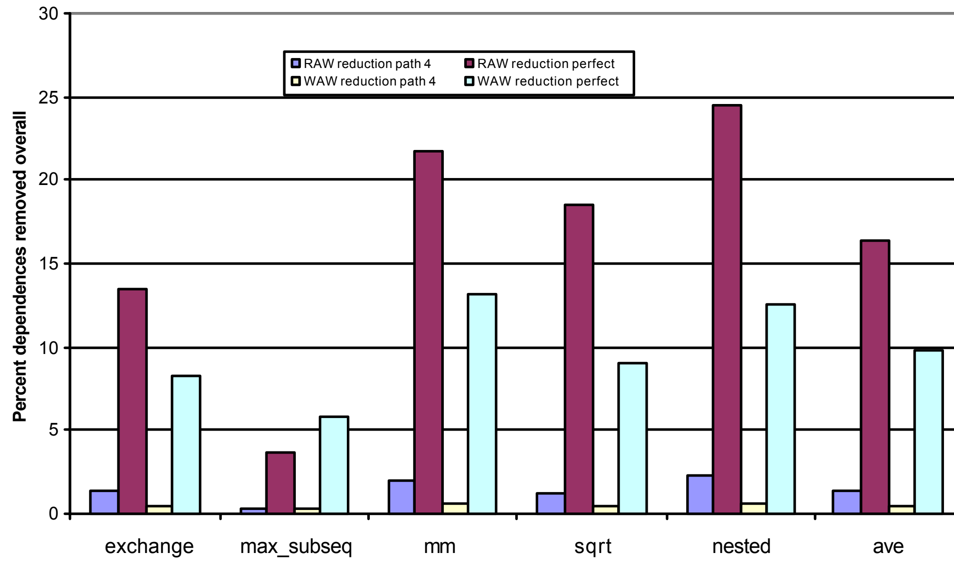


Figure VI.12: Percent of dependences removed by Disjoint Path Analysis in if-converted regions. RAW represents Read After Write dependences while WAW represents Write After Write dependences

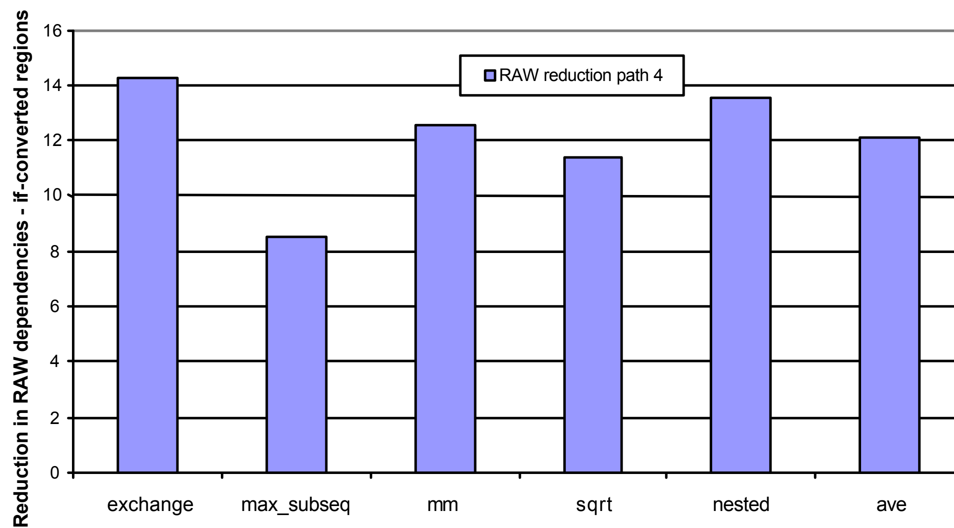


Figure VI.13: Percent of dependences removed by Disjoint Path Analysis in if-converted regions. Results are give for Path Analysis for PITs with 4 slots.

is known to be guarded by false.

The remainder of our results compare the improvement for the Disjoint Path Analysis Architecture configuration against the base Itanium configuration only for if-converted regions, since only these regions are affected by our optimization. Figure VI.13 shows the percent of RAW dependences removed in just the predicated regions for the configurations of the Disjoint Path Architecture with 4 RAT slots. The average percent of RAW dependences removed by the Path Analysis was over 12%.

It is interesting to note that there is not always a direct correlation between the percent of dependences removed for a benchmark and the improvement in IPC for that benchmark. For example, `exchange` had the largest improvement in dependency reduction at over 14%, but only the third highest improvement in IPC (shown in Figure VI.11. This is because an improvement in IPC will only be seen if a dependency with a greater latency is removed by the Path Analysis. Figure VI.11 shows that the improvement in IPC ranges from 2% (`max_subseq`) to 6% (`nested`). The average improvement found when using the Disjoint Path Analysis was between 4% and 5% for the if-converted regions, depending on the configuration.

VI.F Conclusions

In this chapter, we present an approach to dynamic path analysis used to expose erroneous data dependences in the current Itanium architecture and in an out-of-order implementation of an IA64 architecture. We present the Disjoint Path Architecture which allows us to re-create predicate relationship information at runtime in hardware. This predicate relationship information provided by the addition of the Path Information Table and related logic allows us to answer the question “can a given definition be on the same path for a use?” If the answer to this question is no, then that definition is not considered as a possible dependence. Our results showed that the number of RAW dependences set in if-converted regions could be reduced by 14% using the Disjoint Path Architecture with the current Itanium Implementation. As a result, IPC could be increased up to 6% in these regions. This improvement proved, on average, to be almost half the improvement that could be seen using perfect predicate prediction to determine actual dependences.

The benchmarks we used for this study were compiled using the Intel IA64 shrink wrapped C++ Compiler using the -O3 compilation option with profiling. Current production compilers produce minimal code guarded on predication. This is a result from having most of

the predication turned off in the compiler due to the maturity level of the compiler and some predicate performance issues in Itanium. We believe that future generations of EPIC processors and compilers will perform more predication, and this will increase the benefit one can expect from using our Disjoint Path Analysis architecture.

We presented results using our Disjoint Path Analysis architecture for a pure in-order processor. A potential application of this technique would be for an IA64 implementation employing out-of-order execution. For this model, where instructions can begin execution as soon as their operands are ready, it is even more important to eliminate false dependences, thus exposing additional instruction level parallelism. Comparing the use of this technique with the select-μop approach discussed in Section VI.B is a topic of interest for future work.

Chapter VII

Pending Functional Units

The Itanium processor, the first implementation of the IA64 Instruction Set Architecture, is based on concepts put forth by the Explicitly Parallel Instruction Computing (EPIC) community. The EPIC philosophy [49, 33] is to rely on the compiler to control the scheduling of instructions through the processor, which reduces the complexity and increases the scalability of the processor. The Itanium is an in-order processor that fetches, executes and forwards instructions to functional units in order, according to the schedule provided by the compiler.

In-order processing has severe IPC performance limits due to the inability to allow execution to continue past an instruction with an outstanding register use, where the register is being produced by a long latency instruction currently executing. In this situation the whole front-end of the processor stalls and it cannot issue any more instructions until the oldest instruction in the issue window has both of its operands ready. Out-of-order processors have the capability to allow execution to continue, but at the cost of increased hardware complexity.

In this chapter we examine how to adapt the Itanium processor model to overcome the limitations caused by unpredictable load latencies as well as long latency instructions that the compiler could not hide. The goal is to achieve this without adding a large amount of additional complexity. Rau [44] suggested the idea of small-scale reordering on VLIW processors to support object code compatibility accross a family of processors. We describe a method that makes use of features that currently exist in the Itanium model. We begin by describing a complete out-of-order implementation for purposes of comparison.

VII.A Out-of-Order Execution Model

An out-of-order pipeline tries to make use of the available run-time information and schedules instructions dynamically. In other words, the goal of an out-of-order pipeline is to execute an instruction as soon as it is ready, not according to some predetermined order that may no longer be efficient based on the run-time conditions.

Because it must ultimately determine an order of execution that is correct and hopefully efficient, the out-of-order model must detect and maintain information on all 3 types of hazards. This is typically accomplished with a *scoreboard* [42]. A scoreboard manages the issuing and completion of instructions or stalling of the pipeline based on operands and functional units being ready and dependences being met. The scoreboard was mentioned earlier when describing the in-order processor, however, the scoreboard for the out-of-order processor is much more extensive as a larger portion of the dependency analysis is placed on the hardware. As soon as the scoreboard determines that the conditions are right for an instruction to execute, it may execute. The order in which the instructions entered the pipeline are of no consequence in this regard.

Current out-of-order processors include architecturally supported *register renaming*. Register renaming removes WAW and WAR dependences. When instructions are issued, the logical register specifiers for pending operands are renamed to the names of actual physical storage locations. There are more physical locations than logical register specifiers, so different definitions of a logical register can be distinct. A particular use can then be associated with a particular definition. Consider again the example used in the discussion of processor pipelines (Chapter III):

```
cycle
  1  ld r1=[r3]
  2  sub r3=r2,r1
  3  div r1=r4,r5
  4  add r2=r1,r3
```

Recall that there was a WAW dependence between the `load` and `div` instructions, and a WAR dependence between the `load` and `sub` instructions. Below, the registers have been renamed, and the renamed results propagated to the correct uses. While the RAW dependences still exist, the WAR and WAW dependences have been removed.

cycle

```
1   ld r1=[r3]
2   sub r3'=r2,r1
3   div r1'=r4,r5
4   add r2=r1',r3'
```

Current out-of-order processors choose instructions to be scheduled for execution in a particular cycle based on when they expect that the instruction's operands will be ready. In most cases, operations have deterministic latencies, so operand readiness can be predicted based on these latencies. The initial schedules usually assume that all loads hit in the cache. However, if an instruction is dependent on a load that misses in the cache, the operand that uses the load will not be ready as expected. The pre-formed schedule will not be entirely correct and a replay needs to occur to form a legal scheduled group of instructions. Two methods that can be employed to handle this situation are:

- **Flush replay** - The Alpha 21264 Processor uses a form of scheduler recovery called *Flush Replay* [22] (also referred to as squash replay). For this method, pre-formed scheduled groups of instructions are flushed, and a new schedule is issued based on new latency information. A penalty is incurred, determined by the number of cycles between formation of the schedule and the execution stage.

- **Select replay** - The scheduler in the Pentium 4 [32] uses a replay mechanism to selectively re-issue only those instructions that are dependent on the mis-scheduled instruction. This results in a more complex scheduling algorithm than is required for flush replay.

The dynamic scheduling process described above allows the effects of long latencies to be mitigated. If a load instruction misses in the cache, future instructions dependent on the loaded value must wait, but other instructions that are ready may proceed. Memory disambiguation is often included in current out-of-order processors. Recent Alpha processors use a run-time approach, implementing memory disambiguation using *store sets* [21, 3]. A store set for a particular load is the set of stores that a load has been dependent upon. This set is compiled throughout the program execution. The hardware uses the set to predict which stores future loads will depend on. Store sets have been shown to provide near optimal prediction accuracy.

In most out-of-order models, the outcome of branches is predicted. As with the in-order model, a recovery mechanism must be included. The penalty for a mis-predicted branch may be more than with an in-order model, as the added hardware complexity generally means the pipeline is deeper.

Even though instructions are executed out-of-order, the results are committed in-order. There are 2 reasons why this is important. First, committing instructions in order ensures the correct order for writes to and reads from a register. Second, committing in-order maintains *precise exceptions*. The difficulty with exceptions is that they may occur when the execution of an instruction is part way down the pipeline. If this is the case (as with a page fault for a memory instruction) other instructions will have entered the pipeline. If the pipeline can be stopped so that the instructions just before the faulting instruction are completed and those after it can be restarted from scratch, the pipeline is said to have precise exceptions [42]. To keep track of the original order that the instructions entered the pipeline, some kind of FIFO structure is required. This structure is often called a *reorder buffer*, and is implemented as a circular queue [52].

To summarize, the order of execution for an out-of-order execution model is determined by the hardware dynamically. Run-time information can be taken into account, and accommodations can be made for difficult to predict occurrences such as cache misses. The compiler required for this model is much simpler than for the in-order model. The resulting object code is not optimized for one particular model of a processor. However, the hardware required to execute this code is significantly more complex. As mentioned above, a more complex scoreboard, reorder buffer, scheduling algorithm, and the logic to support them must be added to the hardware required for an in-order processor.

Additional Requirements for Predication

The possibility of multiple definitions existing for one use because of predication, makes the renaming stage for an out-of-order implementation supporting predication more complex. As described by Wang et.al. [56], new names of registers must be propagated to their uses. If there are 2 names for a register defined on disjoint paths, the processor must stall waiting to determine which is the correct name to propagate. This cannot be determined until the guarding predicate values for each of the definitions are known. Some mechanism must be maintained to keep track of the multiple definitions of an operand until the processor can determine the correct dependency. As predicates are determined to be false, the possible

definitions narrow to one.

VII.B Introduction to Pending Functional Units

In the Itanium, each cycle up to 6 instructions are scheduled to proceed down through the pipeline and begin executing together. If there is an outstanding dependency for a member of the scheduled group, then all of the instructions in that group stall at the functional units and wait there until all instructions in that scheduled group can start to execute at the same time. To facilitate this, the Itanium already has functional units that have bypassing logic to allow the values being produced to be directly consumed in the next cycle by another functional unit.

In this chapter, we examine *Pending Functional Units* (PFU) that expose a small window of instructions (those that have been allocated function units) to be executed out-of-order. The PFU model implements a simplified version of instruction scheduling/selection. Instructions in this new architecture are issued exactly the same as in a traditional in-order VLIW architecture (instructions cannot issue if there are WAW dependences). The main difference is that an instruction can be allocated a functional unit when its operands are not yet available. In this work we examine three PFU models. Each model increases the complexity of the processor, but also increases the amount of ILP exposed. All of these form their scheduled group of instructions in-order, and the only instruction window that is exposed for out-of-order execution are the instructions pending at the functional units.

The rest of the chapter is organized as follows. Section VII.C describes the base EPIC in-order processor we model for our experiments. Our infrastructure and benchmark selection is presented in Section VII.D. Section VII.E describes the use of Pending Functional Units, and the changes made to the pipeline. Also included in this section are results for using pending functional units. We provide a summary of the chapter in Section VII.F.

VII.C Baseline In-order Processor

Our baseline architecture used in this dissertation is an EPIC in-order processor modeled after Itanium. In this section we provide an overview of how this EPIC processor uses stop bits to communicate compiler detected Instruction Level Parallelism and to simplify its instruction dispatching, and uses load speculation to hide load latencies. In addition, we provide

a brief overview of how the EPIC pipeline and bypass mechanism functions.

VII.C.1 Hardware Simplicity via Stop Bits

In the EPIC in-order execution model [33], it is possible to have low hardware overhead with high IPC parallelism. The EPIC compiler determines much of the register dependency information and transmits this to hardware via a simple stop bit. In typical CISC or RISC architectures determining register dependences typically would take $O(n^2)$ comparisons on n instructions [50]. This comparison is eliminated in the EPIC architecture using bundles and stop-bits.

Stop bits are inserted in the instruction stream to create a group, or sets of instructions that are known to be independent of each other. Stop bits guarantee that the issued instructions are independent of each other, but give no information on the relationships of these instructions to those outside of the group. Therefore, the architecture does not have to perform any register dependency analysis between instructions within the same group, and uses a scoreboard only to maintain dependency information outside of the instruction group.

VII.C.2 Compiler Exposing ILP

The EPIC pipeline we simulate for our experiments is modeled after the Intel Itanium processor (the first IA64 ISA implementation) [6, 50]. Like the Itanium, we maintain a scoreboard which allows the pipeline to stall on a use of any outstanding value. The execution traces we use in our simulator were created for the Itanium using the Electron and SGI IA64 compilers. These compilers take advantage of a number of EPIC high ILP techniques (summarized in [15]) that are visible in the trace. In particular Itanium provides support for *data* and *control speculation* [33] (described in Section III.C.2 to allow the compiler to schedule loads as early as possible to avoid stalling on load misses.

Even with the ALAT and load speculation instructions, RAW dependences on load instructions can account for a significant amount of pipeline stalls. It is also possible that the compiler could not find good opportunities for speculation. As we will show in section VII.E.5, using pending functional units provides a small out-of-order window that can produce significant increases in ILP, allowing execution to continue past a load miss.

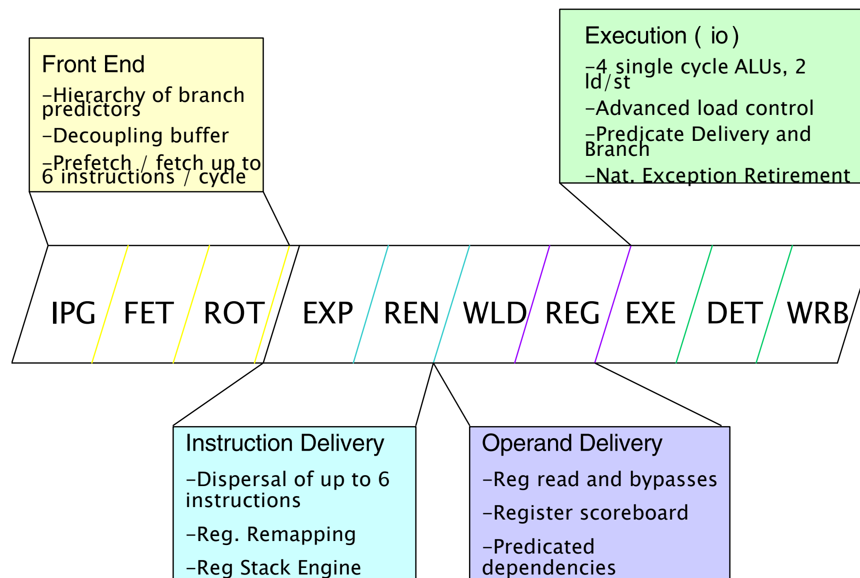


Figure VII.1: Basic Itanium Pipeline

Template or Instruction	**Cycle Dispersed**			
	EPIC	PFU	Cache Miss EPIC	Cache Miss PFU
{ .mii				
ld4 r1=r2, r3	0	0	0	0
add r4=r5, r6; ;	0	0	0	0
sub r7=r8, r9	1(SB)	0	1	0
}				
{ .mfi				
ld4 r4=[r4]	1	1 (WAW)	1	1
fadd f5=f6, f7	1	1	1	1
add r16=r18,r19	1	1	1	1
}				
{ .mmi				
sub r16=r1, r16	2 (BUN)	2 (WAW)	2	2
add r24=r26,r27; ;	2	2	2	2
add r24=r24, r59	3 (SB)	3 (WAW)	7 (STALL)	3
}				

Figure VII.2: Shows how EPIC and PFU models would disperse instructions under normal conditions, and when the first `ld` instruction misses in the L1 cache (but hits in the L2), causing the final `sub` instruction to stall on the use of the loaded value. Whenever the dispersal cycle changes, a reason is provided. SB-stop bit encountered. WAW- Write after Write hazard. BUN-this is the third bundle to be considered for dispersal (instructions may only come from 2 in the EPIC model). STALL-in response to the pipeline backing up after the first load stalls.

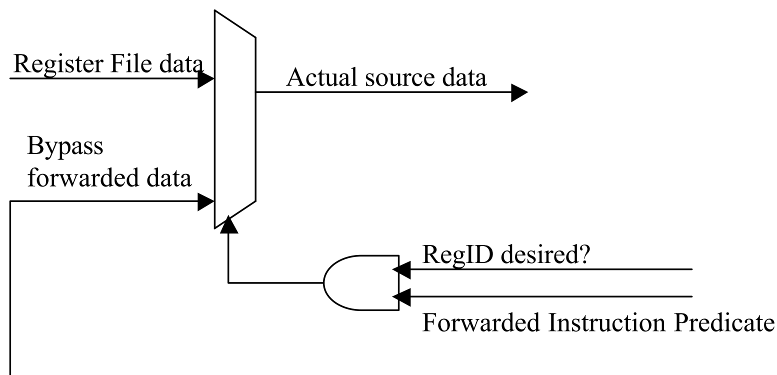


Figure VII.3: The Itanium bypassing mechanism that supports predication.

VII.C.3 EPIC In-Order Pipeline

The pipeline for the Itanium as presented in [50] is shown in Figure VII.1. The IA64 ISA instructions are *bundled* into groups of three by the compiler. A template included with the bundle is used to describe to the hardware the combination of functional units required to execute the operations in the bundle. The template will also provide information on the location of the stop bits in the bundle. Like Itanium, our *baseline in-order EPIC processor* has up to two bundles directed or *dispersed* [6] to the functional units in the EXP stage, subject to independence and resource constraints. All instructions dispersed are guaranteed to be independent of each other. We use the bundles and the stop bits to determine this independence as described previously. We call this in-order group of instructions sent to the functional units together a *scheduled group*. If any of the instructions in the scheduled group must stall because a data dependence on an instruction outside of this group is not yet satisfied, the rest of the instructions in the scheduled group will stall in the functional units as well. Figure VII.2 shows how 3 bundles would be dispersed to the functional units on the EPIC in-order processor as well as with the PFU model which will be discussed later.

VII.C.4 Bypass Mechanism

In the Itanium pipeline, data hazards are detected in the REG (register read) stage, but the instructions are allowed to continue to the EXE stage where they are stalled if the hazard still exists. However, instructions in the EXE stage no longer have read access to the register file [50]. They must get their values from the latches that precede the EXE stage. This requires the use of the bypassing mechanism. The Itanium bypassing shown in Figure VII.3

is designed to support predication. In order for a bypassed result to be accepted, it must be from the regID required, and the guarding predicate of the producing instruction must have been evaluated as TRUE. The Itanium implementation requires that a new scheduled group cannot proceed to the EXE stage until the current one has begun execution in its entirety. Consequently, the stalled instructions in the REG stage will not require the use of the bypass mechanism and this same mechanism can be used to provide the required values to the EXE stage.

VII.D Methodology

For the work in this chapter we used a number of SpecInt2000 and SpecInt95 benchmarks, four of which were compiled using the Intel IA64 electron compiler (go, ijpeg, mcf, bzip2), and two were compiled using the SGI IA64 compiler (li,perl). We used the "-O2" compilation option with static profile information. This level of optimization enables inline expansion of library functions, but confines optimizations to the procedural level. It produces a set of instructions in which an average of 24% found in the trace were predicated, with an average of 4.3% predicated for reasons other than branch implementation and software pipelining. This may include if-conversion or predicate definitions for use in floating point approximation instructions.

Each trace is built as described in Chapter V using the *ref* data set, and includes 300 million instructions collected well after the initialization phase of the execution has finished. Table VII.1 shows the end of the initialization phase and the beginning of the trace in millions of instructions. The end of the initialization phase was determined using the Basic Block Distribution Analysis tool [51].

Table VII.2 shows the parameters used for the simulated microarchitecture modeled after the Itanium. The functional unit distribution includes 2 integer units, 2 memory units (able to execute some IALU instructions as well), 2 floating point units and 3 branch functional units. The latencies used varied by instruction, and were derived from the Itanium Processor Microarchitecture Reference [6]. The memory hierarchy is modeled after that of the Itanium. The L1 data and instruction caches are 4-way associative, sized at 16K with 32 byte blocks. The L2 cache is unified, with a capacity of 96K, 6-way set-associative with base latency of 6 cycles. Floating point loads bypass the L1 cache and incur an extra 3 cycle latency in the L2 cache (totaling 9). The L2 cache will allow misses on as many as 8 outstanding cache lines at

benchmark	suite	end init	begin trace	description
go	95 C	6600	9900	Game playing; artificial intelligence
ijpeg	95 C	4900	7200	Imaging
li	95 C	1200	1800	Language interpreter
perl	2k C	4900	7200	Shell interpreter
bzip2	2k C	400	600	Compression
mcf	2k C	1300	1900	Combinatorial Optimization

Table VII.1: Presents description of benchmarks simulated including instruction count in millions where the initialization phase of the execution ended and where the trace began recording.

L1 I Cache	16k 4-way set-associative, 32 byte blocks, 2 pipeline cycles
L1 D Cache	16k 4-way set-associative, 32 byte blocks, 2 cycle latency
Unified L2 Cache	96k 6-way set-associative, 64 byte blocks, 6 cycle latency
Unified L3 Cache	2Meg direct mapped, 64 byte blocks, 21 cycle latency
Memory Disambiguation	load/store queue, loads may execute when all prior store addresses are known
Functional Units	2-integer ALU, 2-load/store units, 2-FP units, 3-branch
DTLB, ITLB	4K byte pages, fully associative, 64 entries, 15 cycle latency
Branch Predictor	meta-chooser predictor that chooses between bimodal and 2-level gshare, each table has 4096 entries
BTB	4096 entries, 4-way set-associative

Table VII.2: Baseline Simulation Model.

once [6]. Loads made from an address to which a value was stored within the last 3 cycles also will bypass the L1 cache, seeking its value from the L2 cache. The L3 cache is unified, located off chip in the Itanium implementation. The base latency is 21 cycles with floating point loads requiring 24. Although the Itanium implements two levels of DTLB, we modeled the DTLB similar to the ITLB as fully associative, with 64 entries, 4k page size and a 15 cycle latency. Memory access is assumed to require 80 cycles. All models issue up to 6 instructions per cycle. Branch misprediction penalty is a minimum of 8 cycles.

VII.E Pending Functional Units

The Itanium processor and the EPIC baseline processor we model in this chapter require in-order execution. This is enforced by the formation of a *scheduled group* of instructions

as described in Section VII.C. This can leave the functional units idle for many cycles.

VII.E.1 Limited Out-Of-Order Execution Via Functional Units

The goal of this research is to examine, in increasing complexity, the changes needed to allow the Itanium processor to execute out-of-order, for the purpose of overcoming memory and long latency functional unit delays. We provide this capability by turning the functional units of the Itanium processor into *Pending Functional Units*. These are similar to reservation stations [54], but are simpler in that no scheduling needs to be performed when the operands are ready, because the instruction already owns the functional unit it will use to execute.

The Itanium is a strict in-order implementation requiring that the scheduled group of instructions that are sent to the functional units together, *begin* execution together. In the Itanium, all of the functional units stall together and do not execute if one of the instructions in the scheduled group has an unresolved dependence. Therefore, the Itanium processor will buffer its input operands at a functional unit until all of the instructions are ready to execute.

In the PFU architecture, the instructions are sent to the functional units in-order. Once they are there, the instructions can execute individually when their operands are ready. The whole scheduled group does not have to stall. This provides limited out-of-order execution. We limit our architecture to 9 functional units in this work, since we model the PFU architecture after the Itanium processor. Figure VII.2 shows the original schedules for the EPIC model and the PFU architecture both with and without the first `ld` instruction missing in the L1 cache. For the EPIC model the final `sub` instruction must stall due to its dependence on the `ld`. Per EPIC rules, the add in the same scheduled group must stall as well. This delays the issuing of the final `add`. For the PFU model, the final `add` is allowed to proceed to the pending functional unit, and will begin execution immediately as the previous `add` was not required to stall.

In providing the PFU limited out-of-order model, we need to address how we are going to deal with the following three areas:

1. **Register Mapping.** The Itanium processor provides register renaming for rotating registers and the stack engine, but not for the remaining registers. We examine two models for dealing with these other compute registers for the PFU architecture. The first approach does not rename these registers, and instead the scheduler makes sure there are no WAW dependences before an instruction can be issued in-order to a functional unit. Note, that WAR dependences do not stall the issuing of instructions, since operands are read in-order from the register file and there can be only one outstanding write of a given

register at a time. The second approach we examine allows multiple outstanding writes to begin executing at the same time by using the traditional out-of-order hardware renaming model.

2. **Scheduling.** When forming a schedule for the baseline Itanium processor, the dispersal stage does not need to take into consideration what has been scheduled before it in terms of resource constraints. In this model, the whole scheduled group goes to the functional units and they either all stall together, or all start executing together. For the PFU architecture, the formation of the scheduled group of instructions needs to now take into consideration functional unit resource constraints, since the functional unit may have a pending instruction at it. Therefore, we add an additional scheduling pipeline stage (described in more detail below) to the PFU architecture to be used to form the scheduled groups taking into consideration these constraints, and use flush-replay when the schedule if incorrect.

3. **Speculative State.** The Itanium processor can have instructions complete execution out-of-order. To deal with speculative state, the Itanium processor does not forward/bypass its result values until they are non-speculative. For our PFU architecture we examine two techniques for dealing with speculative state. The first approach is similar to the Itanium processor, where we do not forward the values until they are known to be non-speculative. The second approach examines using speculative register state (as in a traditional out-of-order processor), and updating the non-speculative state at commit. This allows bypassing at the earliest possible moment – as soon as an instruction completes execution.

The remainder of this section describes the changes needed for scheduling and handling of speculative state for the PFU architecture.

VII.E.2 Instruction Scheduling

As stated above, for the Itanium processor the formation of the scheduled group of instructions occurred without caring about the state of the functional units. Therefore, all of the functional units are either stalled or available for execution in the next cycle. This simplifies the formation of execution groups in the dispersal pipeline stage of Itanium, because the formation of such groups can assume that all of the functional units will be available when the group progresses to them.

In our PFU architecture we add an additional pipeline stage to model the scheduling complexity of dealing with pending functional units. The scheduler we use includes instructions in-order into a scheduled group and *stops* when any of the three conditions occur:

- An instruction with an unresolved WAW dependency is encountered.
- There will be no free functional unit N cycles into the future (where N is 2 for the architecture we model) for the next instruction to be included into the group.
- The scheduled group size already includes 6 instructions, which is the maximum size of a scheduled group.

The second rule above is accomplished by keeping track of when each functional unit will be free for execution. This is the same as keeping track, for each instruction pending at a functional unit, when the operands for that instruction will be ready so that it can execute. All functional units are pipelined, so the only time a functional unit will not be free is if the instruction currently pending at the functional unit has an unresolved dependency. The only dependences that are non-deterministic are load's that result in cache misses. The schedule is formed assuming that each load will hit in the cache.

When the schedule is wrong a replay needs to occur to form a legal scheduled group of instructions. The Alpha 21264 Processor uses this form of scheduler recovery called *Flush Replay* [22] (also referred to as squash replay). The instruction scheduler forms groups of instructions to be issued together in a given cycle. If one of the instructions in a group cannot start execution because a functional unit is occupied, up to X groups of instructions scheduled directly after this scheduled group are flushed, and a new schedule is issued. The X cycle penalty is determined by the number of cycles between formation of the schedule and the execution stage. In this study we assume that the penalty is two cycles.

VII.E.3 Managing Speculative State

The out-of-order completion of instructions creates result values that are speculative and they may or may not be used depending upon the outcome of a prior unresolved branch.

The baseline EPIC processor we model after the Itanium does not forward its result values until they are non-speculative. Therefore, its pipeline up through the start of execution does not have to deal with speculative result values.

In the PFU architecture we examine two methods of dealing with speculative result values:

- Non-Speculative - The first approach is similar to the Itanium processor, where we do not forward the values until they are known to be non-speculative.
- Speculative - The second approach forwards the values in writeback right after the instruction finishes execution. This means that the register state dealt with in the earlier part of the pipeline is considered speculative register state (as in a traditional out-of-order processor), and the non-speculative register state is updated at commit. Note, multiple definitions are not an issue with speculative bypassing, since WAW dependences must be resolved prior to an instruction entering a functional unit.

VII.E.4 Summary of PFU Configurations Examined

The above variations are examined in the next section in 3 PFU configurations, which we summarize below:

- PFU - The modifications to the baseline in-order EPIC processor are to (1) add a new scheduling pipeline stage that dynamically forms a group of instructions to send to the functional units, and (2) allows these pending functional units to execute out-of-order. The scheduled group of instructions are selected in-order, and the formation stops when the criteria listed above is met. For this first configuration, only non-speculative values are bypassed to the functional units. Following the example of Thorton's simplification [53] of Tomasulo's early out-of-order execution design [54], no renaming is performed for this configuration.
- PFU Spec - Builds on PFU allowing bypassing of values as soon as they are calculated. The processor therefore needs to support speculative and non-speculative register state.
- PFU Spec Rename - Builds on PFU Spec adding traditional hardware renaming for predicates and computation registers. This eliminates having to stop forming scheduled group of instructions because of a WAW dependency.

VII.E.5 Results

In this section we discuss results for adding the PFUs to an EPIC in-order processor, and then compare the performance of this architecture to a traditional out-of-order processor.

Adding PFUs to an EPIC In-Order Processor

Figure VII.4 shows the performance benefit of allowing the in-order EPIC processor to execute out-of-order using Pending Functional Units. It shows IPC results for the baseline EPIC processor, the various PFU configurations, and a full out-of-order processor with issue buffer sizes of 9 and 64. The three PFU configurations shown in this figure are summarized above. The first configuration considered (PFU) uses PFUs with no renaming and only forwarding results and updating register state when they are known to be non-speculative. One stage was added to the pipeline for scheduling as described earlier. The improvement over the baseline EPIC processor ranged from 1% (`go`) to 27% (`bzip2`). The average improvement of the PFU model over the EPIC model was 18%.

The low IPC and limited improvement seen for `go` can be attributed to poor cache and branch prediction performance as shown in Table VII.3. This table describes for each benchmark information about cache and branch predictor activity for the baseline EPIC configuration. In particular, the second column describes the number of load misses that occurred in the L1 data cache per 1000 executed instructions. The third column is the number of instructions that missed in the L1 instruction cache per 1000 executed instructions. The next two columns refer to the L2 and L3 caches, which are unified. Hence, the numbers given indicate the number of misses to those caches encountered per 1000 executed instructions. The last column provides the branch prediction hit rate produced by the number of hits divided by the number of updates to the predictor. The predictor is updated for every non-speculative branch executed. As can be seen from this table, the branch prediction accuracy for `go` was significantly lower than that of the other benchmarks at 69%. In addition, `go` also had a high L1 instruction cache miss rate, where there was 48 misses for every 1000 executed instructions.

The second configuration considered is PFU Spec. This model builds on the prior PFU architecture, easing the bypassing constraint by allowing results to be speculatively forwarded right after execution. This approach adds an additional 2% improvement, bringing the average improvement in IPC over the EPIC model up to 20%. The largest additional improvement achieved using speculative state is seen in the `li` benchmark, at 6%.

For the final PFU configuration, we added full register renaming to the model just described. In addition to the stage added for instruction scheduling, we added another stage to accommodate the additional complexity required by PFU renaming. On average, renaming improved the performance of the previous model by 10% as compared to the baseline EPIC, bringing the total average improvement to 30%. `Li` saw the biggest improvement over EPIC

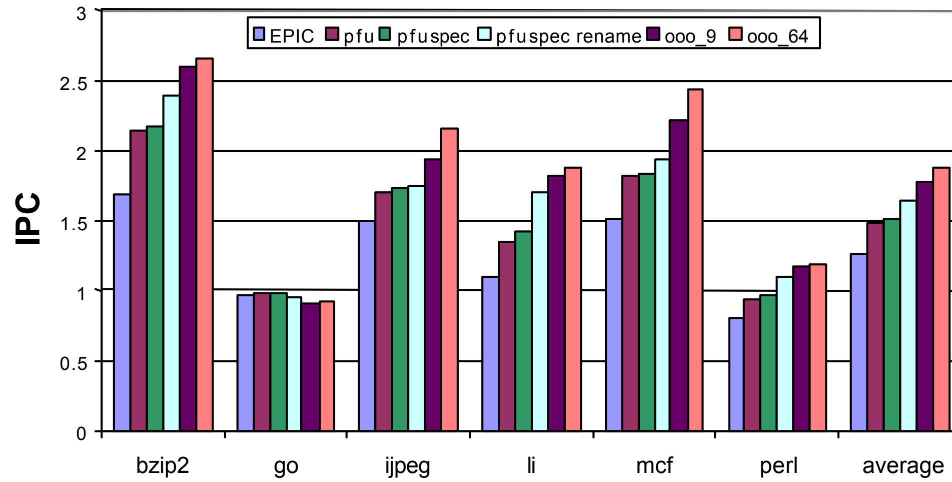


Figure VII.4: Comparing the IPCs of the various PFU models and the out-of-order models with issue buffer sizes of 9 and 64 to the EPIC model.

benchmark	dl1 miss/1K	il1 miss/1K	ul2 miss/1K	ul3 miss/1K	Br Pred Hit Rate
bzip2	.279	.005	.138	.131	91%
go	11.242	47.930	13.592	1.771	69%
ijpeg	2.629	21.193	1.058	.493	87%
li	6.336	3.451	3.066	.071	83%
perl	11.780	41.791	13.327	2.357	83%
mcf	30.949	.002	32.017	15.893	84%

Table VII.3: Presents misses per 1K instructions in the various caches. Levels 2 and 3 are unified caches. The last column shows the branch prediction accuracy.

with a gain in IPC of 56%. In this case, the IPC produced by go in this configuration was actually lower than the baseline EPIC processor. This is due to the additional pipeline stages, which we modeled in the PFU architecture, resulting in increased penalties related to branch mis-predictions. Hence, the poor prediction accuracy recorded by go translated to reduced IPC.

Comparing to a Complete Out-of-Order Processor

We now compare the PFU architecture to an out-of-order IA64 processor. The out-of-order processor we model has three major architectural changes over the baseline in-order EPIC architecture:

1. **Out of Order instruction scheduler**, selecting instructions from a window of 9 or 64 instructions to feed the functional units. Instead of using Flush Replay, modern out-of-order processors use *Selective Replay* to avoid large penalties for forming incorrect schedules often caused by load misses. The scheduler in the Pentium 4 [32] uses a replay mechanism to selectively re-issue only those instructions that are dependent on the mis-scheduled instruction. This adds more complexity in comparison to using flush replay as in the PFU architecture. We modeled flush replay for out-of-order execution, but the IPC results were the same or worse on average than the PFU architecture. Therefore, we only present out-of-order results in this study for selective reply.

2. **Register renaming** to eliminate WAW dependences

3. **Out-of-order completion** of execution, so speculative register state needs to be maintained (a reorder buffer or register pool).

An out-of-order pipeline architecture tries to make use of the available run-time information and schedules instructions dynamically. In an out-of-order environment, instructions effectively wait to execute in either an issue buffer, active list of instructions, or reservation stations. We use the notion of an issue buffer that is dynamically scheduled to supply instructions to all functional units. In this section, we examine configurations of an out-of-order architecture with issue buffer sizes of 9 and 64. We chose 9 in order to make a comparison to the PFU model with 9 pending functional units.

Our out-of-order version of the EPIC pipeline varies from the traditional out-of-order architecture only as was required to support EPIC features such as predication and speculation.

For example, in the renaming process, each definition of a register is given a unique name, and then that name is propagated to each use. However, predication allows for multiple possible reaching definitions. Consequently, this process must be extended to include knowledge of guarding predicates in conjunction with definitions. We implement out-of-order renaming in a manner similar to that described in [56]. As in that study, we add 4 stages to the baseline EPIC pipeline, 2 for renaming, and 2 for scheduling, to model the additional complexity for out-of-order execution.

Figure VII.4, provides IPC results for the 2 different configurations that were evaluated for the out-of-order model. As mentioned, these varied by the size of the schedulable window of instructions. In the out-of-order architecture, up to six instructions can be scheduled/issued to potentially 9 functional units. The results show that an issue window of 64 instructions produces an additional speedup of 31% over the base PFU configuration. For `bzip2`, the baseline PFU architecture experienced half of the improvement seen by the full out-of-order model. `Go` is again an interesting case. The out-of-order models perform worse than all other configurations. This is attributed to the high penalties due to a deeper pipeline and poor branch prediction.

For several of the programs, the out-of-order model with an issue window of 9 performs only slightly worse than the out-of-order model with 64. For the benchmarks `bzip2`, `go`, `li`, and `perl`, the difference in IPC was under 3%. The average difference in IPC between the two out-of-order configurations was only 5%. For `bzip2`, `ijpeg` and `perl`, the baseline PFU architecture achieves half the performance improvement that is seen by the aggressive OOO model with a window of 9 instructions.

VII.F Conclusions

In-order processors are advantageous because of their simple and scalable processor pipeline design. Performance of the program is heavily dependent upon the compiler for scheduling, load speculation, and predication to expose ILP. Even with an aggressive compiler, the amount of ILP will still be constrained for many applications.

To address this problem we explored adding a small out-of-order instruction window to the in-order processor. We examined the Pending Functional Unit (PFU) architecture, where instructions are allocated to the functional units in order (as in the Itanium), but they can execute out-of-order as their operands become ready. Augmenting the EPIC in-order processor with PFUs provides an average 18% speedup over the EPIC in-order processor. The additional

cost in complexity to achieve this speedup is the dynamic scheduling of the functional units and the implementation of flush replay for when the schedule is incorrect.

When comparing the base PFU architecture to a full out-of-order processor, our results show that the PFU architecture yields 44% of the speedup, on average, of an out-of-order processor with an issue window of 9 instructions and renaming. The out-of-order processor with a window of 9 instructions achieved results within 5% of having a window of 64 instrutions.

Executing instructions in a small out-of-order window is ideal for hiding L1 cache misses, and long latency functional unit delays. Another way to attack these stalls would be to increase the amount of prefetching performed by the processor, and reduce the latency of the L1 cache and functional units. The plans for the next generation of Itanium processors, the McKinley, include increased bandwidth and larger caches [7] to address this problem. While latency reduction via prefetching and memory optimizations will provide benefits, a small out-of-order window will most likely still be advantageous, since not all L1 misses will be able to be eliminated. Performing this comparison is part of future work.

Chapter VIII

Conclusions

One of the new features of the EPIC architecture is its support for *predicated execution*. For the *if-conversion* form of predication, the compiler can replace branches with statements defining 2 predicate registers (one true and one false), depending on the condition in the replaced branch. Subsequent statements are then guarded by one of the predicates, depending upon whether they would have been on the taken or fall-through path of the branch. All if-converted statements begin execution, but an operation is committed only if the value of its guarding predicate is true.

Including predicated code in an instruction stream makes data analysis different for 2 reasons. First, control dependences are changed into data dependences when branches are if-converted. Second, when multiple paths are combined through this process, it becomes more difficult to determine which of the intermingled definitions actually reach a use.

This dissertation addresses the data analysis problem, suggesting both compiler-based and hardware-based solutions. The key to determining true data dependences with either approach is determining and understanding the relationships between predicates and how these relationships can indicate if a definition and use can possibly be on the same path.

From the compiler side, this dissertation presents Predicated Static Single Assignment (PSSA), a renaming technique inspired by Static Single Assignement (SSA) and enabled by predicate-sensitive path information. This work demonstrates how PSSA uses *full-path predicates*, to eliminate false dependences for predicated code. We show the benefit of using PSSA to enable Predicated Speculation (PSpec) and Control Height Reduction (CHR) during scheduling. Predicated Speculation allows operations to be executed at the cycle of their ear-

liest schedulable cycle, even before their guarding predicates are determined. Control Height Reduction allows guarding predicates to be defined as soon as possible, reducing the amount of speculation needed.

By maintaining information about each of the control paths that exist in a predicated region, PSSA can provide information that allows precise placement of renamed and speculated code, and allows the correct, renamed values to be propagated to subsequent operations. The renaming used by PSSA allows more aggressive speculation, as overwriting live values is no longer a concern. In addition, PSSA supports Control Height Reduction along every control path using full-path predicates, reducing control dependence depth throughout the hyperblock.

Our experiments show that PSSA is an effective tool for optimizing predicated code. Using PSSA with PSpec and CHR results in a reduction in executed cycles ranging from 12% to 62% for a 16 issue machine.

We also suggest methods for reducing the amount of code expansion created by the PSSA transformation. These techniques included reducing the region over which the full-path-predicates created by PSSA were assigned, and a critical path first scheduling algorithm. We show that the amount of code expansion could be decreased by 40% over that created by the original PSSA algorithm for the example we tested.

We would like to produce similar predicate-sensitive data dependence analysis in the hardware. The Disjoint Path Architecture allows us to re-create predicate relationship information at runtime. This is important because much of the analysis completed in the compiler cannot be communicated to the hardware. In the case of the current implementation of an EPIC architecture, the Itanium, a scoreboard is used to determine instruction readiness so dependences must be re-calculated. The Disjoint Path Analysis Architecture helps ensure that only definitions and uses that can possibly be on the same path produce dependences. Our results show that the number of RAW dependences set in if-converted regions could be reduced by 14% using the Disjoint Path Architecture with the current Itanium Implementation. As a result, IPC could be increased up to 6% in these regions. This improvement proved, on average, to be almost half the improvement that could be seen using perfect predicate prediction to determine actual dependences. We believe that future generations of EPIC compilers will support more if-conversion, increasing the need for this vital analysis.

Another feature of the EPIC architecture is the relatively simple hardware design allowed by performing all of the scheduling in the compiler. For EPIC machines, the schedule is provided to the hardware by the compiler and instructions are executed strictly in the order

in which they enter the pipeline. The drawback of this design is that even with an aggressive compiler, the amount of ILP will still be constrained for many applications by instructions that use the result of a load that misses in the cache, especially loads that have to go all the way to main memory.

To help alleviate this problem we explore adding a small out-of-order instruction window to the in-order processor. We examine the Pending Functional Unit (PFU) architecture where instructions are allocated to the functional units in order (as in the Itanium), but they can execute out-of-order as their operands become ready. Augmenting the EPIC in-order processor with PFUs provides an average 18% speedup over the EPIC in-order processor. The additional cost in complexity to achieve this speedup is the dynamic scheduling of the functional units and the implementation of flush replay for when the schedule is incorrect. Allowing the speculative register state, the PFU configuration yielded an average speedup up 20%.

We also compare the PFU architecture to a full out-of-order processor that has the additional features of register renaming and dynamic scheduling over a large window of available instructions. Our results show that the PFU architecture yields 44% of the speedup of an out-of-order processor with an issue buffer of size 9 and renaming, and 37% of the speedup of an out-of-order configuration with an issue buffer of size 64. This benefit is achieved without needing register renaming or a complex out-of-order scheduling algorithm.

Chapter IX

Future Directions

We recognize that there are other approaches to performance improvement not addressed in this dissertation, both in terms of data dependence analysis and hardware optimizations. Aside from re-creating the data dependence analysis in the hardware, it would be interesting to pursue additional methods for communicating the compiler analysis to the hardware. In the Itanium, some dependence information is currently sent to the hardware from the compiler via stop bits. However, especially in the presence of predication, it would be helpful to send the predicate-sensitive analysis that can determine actual dependences that exist across the combined paths.

The plans for the next generation of Itanium processors, the McKinley, include increased bandwidth and larger caches [7]. Prefetching is a current topic of interest for improved performance. All of these techniques are aimed at reducing some latency in the processor. While latency reduction is an important issue, there comes a point where these approaches will plateau. We expect that architects will still have to return to techniques which expose greater instruction level parallelism. However, we believe that a study aimed at determining whether the best use of increased hardware is for bandwidth, caches, prefetching or increased ILP would be worth pursuing in the future.

Bibliography

[1] Intel Press Release. Merced processor and IA-64 architecture., 1998. http: //developer.intel.com/ design/processor/future/iaa64.htm, 1998.

[2] Trimaran, An Infrastructure for Research in Instruction Level Parallelism, 1998. http: //www.trimaran.org.

[3] Alpha and IA64., 1999. http: //www.compaq.com/hpc/ref/ref_alpha_ia64.pdf.

[4] IA-64 Application Instruction Set Architecture Guide, Revision 1.0, 1999.

[5] *Intel IA-64 Architecture Software Developer's Manual.* Intel, 2000.

[6] Itanium Processor Microarchitecture Reference:for Software Optimization., 2000. http: //www.developer.intel.com/design/ia64/itanium.htm.

[7] 2001 - a processor odyssey: the first ever McKinley processor is demonstrated by hp and Intel at Intel's developer forum, February 2001. http: //www.hp.com/products1/itanium/news_events/archives/NIL0014KJ.html.

[8] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers : Principles, Techniques and Tools.* Addison-Wesley, 1986.

[9] 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*, December 1994.

[10] G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. *ACM SIGPLAN Notices*, 33(5):72–84, May 1998.

[11] 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 predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th Intl. Symp. on Computer Architecture*, July 1998.

[12] D. I. August, K. M. Crozier, J. W. Sias, P. R. Eaton, Q. B. Olaniran, D. A. Connors, and W. W. Hwu. The IMPACT EPIC 1.0 Architecture and Instruction Set reference manual. Technical Report IMPACT-98-04, IMPACT, University of Illinois, Feb 1998.

[13] D. I. August, W. Hwu, and S. A. Mahlke. A framework for balancing control flow and predication. In *30th Annual Intl. Symp. on Microarchitecture*, December 1997.

[14] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 46–57, December 2–4, 1996.

[15] J. Bharadwaj, W. Chen, W. Chuang, G. Hoflehner, K. Menezes, K. Muthukumar, and J. Pierce. The Intel IA-64 Compiler Code Generator. *IEEE Micro*, 20(5):44–52, September 2000.

[16] D.C. Burger and T.M. Austin. The Simplescalar Tool Set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, Jun 1997.

[17] L. Carter, W. Chuang, and B. Calder. An epic processor with pending functional units. In *Proceedings of the Fourth International Symposium on High Performance Computing*, May 2002.

[18] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Predicated static single assignment. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, October 1999.

[19] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Path analysis and renaming for predicated instruction scheduling. In *International Journal of Parallel Programming*, December 2000.

[20] Y. Choi, A. Knies, L. Gerke, and T. Ngai. The impact of if-conversion and branch prediction on program execution on the intel itanium processor. In *Proceedings of the 34th Annual Intl. Symp. on Microarchitecture*, December 2001.

[21] G. Chrysos and J. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual Intl. Symp. on Computer Architecture*, pages 142–153, June 1998.

[22] COMPAQ Computer Corp. Alpha 21264 microprocessor hardware reference manual, July 1999.

[23] J. Crawford. Introducing the Itanium Processors. *IEEE Micro*, 20(5):9–11, September 2000.

[24] J. Crawford and J. Huck. Motivations and Deisgn Approach for the IA-64 64-Bit Instruction Set Architecture, October 1997. http://www.intel.com/pressroom/archive/speeches/mpf1097c.htm.

[25] R. Cytron, J. Ferrante, B. K. Rosen, M. K. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, 1989.

[26] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[27] S. Eranian and D. Mosberger. The Linux/IA64 Project: Kernel Design and Status Update. Technical Report HPL-2000-85, HP Labs, June 2000.

[28] J. Ferrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[29] 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 Annual Intl. Symp. on Microarchitecture*, pages 114–125, December 1996.

[30] R. Gupta, D. A. Berson, and J. Z. Fang. Path profile guided partial dead code elimation using predication. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, pages 102–113, November 10–14, 1997.

[31] L. Gwennap. Intel, HP make EPIC disclosure. *Microprocessor Report*, 11(14):1–9, October 1997.

[32] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the pentium 4 processor. *Intel Technology Journal Q1*, 2001.

[33] J. Huck, D. Morris, J. Ross, A. Knies, H. Mulder, and R. Zahir. Introducing the IA-64 Architecture. *IEEE Micro*, 20(5):12–22, September 2000.

[34] V. Kathail, M. S. Schlansker, and B. R. Rau. HPL PlayDoh architecture specification: Version 1.0. Technical Report HPL-93-80, HP Labs, Feb 1994.

[35] 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*, 7(1-2):51–142, May 1993.

[36] S. Mahlke, R. Hand, J. McCormick, D. August, and W. Hwu. A comparison of full and partial predicated execution support. In *Proceedings of the 22nd Annual Intl. Symp. on Computer Architecture*, June 1995.

[37] 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 Annual Intl. Symp. on Microarchitecture*, pages 217–227, December 1994.

[38] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual Intl. Symp. on Microarchitecture*, pages 45–54, December 1992.

[39] S. Mantripragada and A. Nicolau. The effects of predicated execution on architectures supporting dynamic speculation. In *International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, October 1998.

[40] S. Moon and K. Ebcioğlu. Parallelizing nonnumerical code with selective scheduling and software pipelining. *ACM Transactions on Programming Languages and Systems*, 19(6):853–898, November 1997.

[41] J. C. H. Park and M. Schlansker. On Predicated Execution. Technical Report HPL-91-58, HP Labs, May 1991.

[42] D. Patterson and J. Hennessy. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann, 1996.

[43] B. Rau, D. Yen, W. Yen, and R. Towle. The Cydra 5 departmental supercomputer. *IEEE Computer*, 22:12–35, January 1989.

[44] B. R. Rau. Dynamically scheduled vliw processors. In *Proceedings of the 26th Annual Intl. Symp. on Microarchitecture*, pages 80–92, December 1993.

[45] M. Schlansker and R. Johnson. Analysis techniques for predicated code. In *Proceedings of the 29th Annual Intl. Symp. on Microarchitecture*, pages 100–113, December 1996.

[46] M. Schlansker and V. Kathail. Critical path reduction for scalar programs. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 57–69, November 29–December 1, 1995.

[47] M. Schlansker, V. Kathail, and S. Anik. Height reduction of control recurrences for ILP processors. In *Proceedings of the 27th Annual Intl. Symp. on Microarchitecture*, pages 40–51, December 1994.

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

[49] M. S. Schlansker and B. Rau. EPIC: Explicitly Parallel Instruction Computing. *IEEE Computer*, 33(2):37–45, February 2000.

[50] H. Sharangpani and K. Arora. Itanium processor microarchitecture. In *IEEE MICRO*, pages 24–43, 2000.

[51] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[52] J. E. Smith and G. S. Sohi. The microarchitecture of superscalar processors. In *Proceedings of the IEEE*, December 1995.

[53] J. E. Thornton. Design of a Computer, the Control Data 6600, 1970. Scott, Foresman, Glenview, Ill.

[54] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[55] G. S. Tyson. The effects of predicated execution on branch prediction. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 196–206, November 30–December 2, 1994.

[56] P. H. Wang, H. Wang, R. M. Kling, K. Ramakrishnan, and J. P. Shen. Register renaming for dynamic execution of predicated code. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, February 2001.

[57] N. J. Warter, S. A. Mahlke, W. W. Hwu, and B. R. Rau. Reverse if-conversion. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 290–299, June 1993.

[58] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley, Redwood City, CA, 1996.