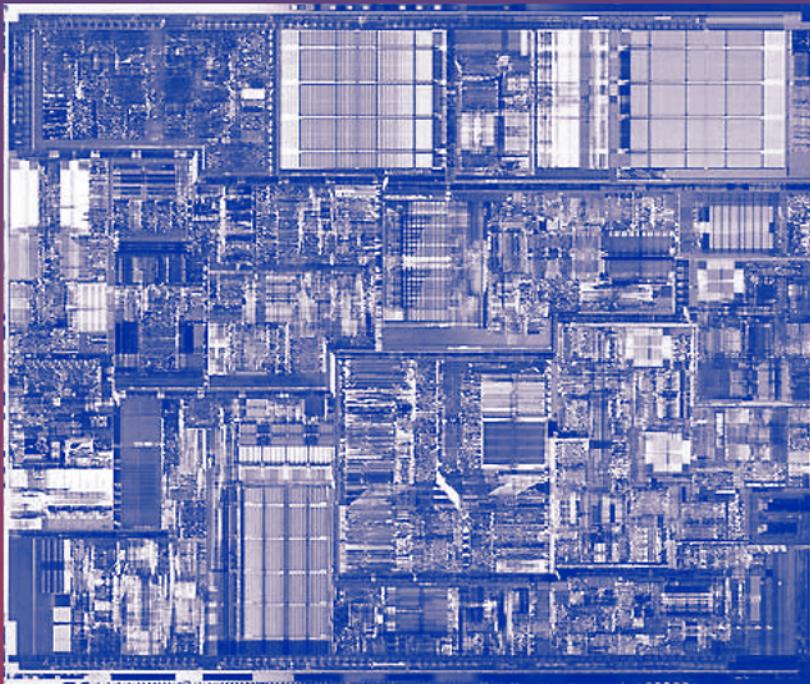


CAPMAN & HALL/CRC COMPUTER and INFORMATION SCIENCE SERIES

Speculative Execution in High Performance Computer Architectures



Edited by
David Kaeli
and Pen-Chung Yew

 Chapman & Hall/CRC
Taylor & Francis Group

CHAPMAN & HALL/CRC COMPUTER and INFORMATION SCIENCE SERIES

Speculative Execution in High Performance Computer Architectures

CHAPMAN & HALL/CRC COMPUTER and INFORMATION SCIENCE SERIES

Series Editor: Sartaj Sahni

PUBLISHED TITLES

HANDBOOK OF SCHEDULING: ALGORITHMS, MODELS, AND PERFORMANCE ANALYSIS
Joseph Y-T. Leung

THE PRACTICAL HANDBOOK OF INTERNET COMPUTING
Munindar P. Singh

HANDBOOK OF DATA STRUCTURES AND APPLICATIONS
Dinesh P. Mehta and Sartaj Sahni

DISTRIBUTED SENSOR NETWORKS
S. Sitharama Iyengar and Richard R. Brooks

SPECULATIVE EXECUTION IN HIGH PERFORMANCE COMPUTER ARCHITECTURES
David Kaeli and Pen-Chung Yew

CHAPMAN & HALL/CRC COMPUTER and INFORMATION SCIENCE SERIES

Speculative Execution in High Performance Computer Architectures

Edited by

David Kaeli

Northeastern University
Boston, MA

and **Pen-Chung Yew**

University of Minnesota
Minneapolis, MN



Chapman & Hall/CRC

Taylor & Francis Group

Boca Raton London New York Singapore

Published in 2005 by
CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2005 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group

No claim to original U.S. Government works
Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number-10: 1-58488-447-9 (Hardcover)
International Standard Book Number-13: 978-1-58488-447-7 (Hardcover)
Library of Congress Card Number 2005041310

This book contains information obtained from authentic and highly regarded sources. Reprinted material is quoted with permission, and sources are indicated. A wide variety of references are listed. Reasonable efforts have been made to publish reliable data and information, but the author and the publisher cannot assume responsibility for the validity of all materials or for the consequences of their use.

No part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (<http://www.copyright.com/>) or contact the Copyright Clearance Center, Inc. (CCC) 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Library of Congress Cataloging-in-Publication Data

Kaeli, David R.
Speculative execution in high performance computer architectures / David Kaeli and Pen Yew.
p. cm. -- (Chapman & Hall/CRC computer and information science series)
Includes bibliographical references and index.
ISBN 1-58488-447-9 (alk. paper)
1. Computer architecture. I. Yew, Pen-Chung, 1950-. II. Title. III. Series.

QA76.9.A73K32 2005
004.35--dc22 2005041310



Taylor & Francis Group
is the Academic Division of T&F Informa plc.

Visit the Taylor & Francis Web site at
<http://www.taylorandfrancis.com>

and the CRC Press Web site at
<http://www.crcpress.com>

About the Editors

David Kaeli received his B.S. in Electrical Engineering from Rutgers University, his M.S. in Computer Engineering from Syracuse University, and his Ph.D. in Electrical Engineering from Rutgers University. He is currently an Associate Professor on the faculty of the Department of Electrical and Computer Engineering at Northeastern University. Prior to 1993, he spent 12 years at IBM, the last 7 at IBM T.J. Watson Research in Yorktown Heights, N.Y. In 1996 he received the NSF CAREER Award. He currently directs the Northeastern University Computer Architecture Research Laboratory (NUCAR). Dr. Kaeli's research interests include computer architecture and organization, compiler optimization, VLSI design, trace-driven simulation and workload characterization. He is an editor for the *Journal of Instruction Level Parallelism*, the *IEEE Computer Architecture Letters*, and a past editor for *IEEE Transactions on Computers*. He is a member of the IEEE and ACM. URL: www.ece.neu.edu/info/architecture/nucar.html

Pen-Chung Yew is a professor of the Department of Computer Science and Engineering, University of Minnesota. Previously, he was an Associate Director of the Center for Supercomputing Research and Development (CSR) at the University of Illinois at Urbana-Champaign. From 1991 to 1992, he served as the Program Director of the Microelectronic Systems Architecture Program in the Division of Microelectronic Information Processing Systems at the National Science Foundation, Washington, D.C. Pen-Chung Yew is an IEEE Fellow. He served as the Editor-in-Chief of the IEEE Trans. on Parallel and Distributed Systems between 2002-2005. He has served on the program committee of major conferences. He also served as a co-chair of the 1990 International Conference on Parallel Processing, a general co-chair of the 1994 International Symposium on Computer Architecture, the program chair of the 1996 International Conference on Supercomputing, and a program co-chair of the 2002 International Conference on High Performance Computer Architecture. He has also served on the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems* from 1992 to 1996, and *Journal of Parallel and Distributed Computing* from 1989 to 1995. He received his Ph.D. from the University of Illinois at Urbana-Champaign, M.S. from University of Massachusetts at Amherst, and B.S. from National Taiwan University.

Acknowledgments

Professors Kaeli and Yew would like to thank their students for their help and patience in the preparation of the chapters of this text. They would also like to thank their families for their support on this project.

Contents

1	Introduction	1
<i>David R. Kaeli¹ and Pen-Chung Yew² Northeastern University,¹ University of Minnesota²</i>		
2	Instruction Cache Prefetching	9
<i>Glenn Reinman UCLA Computer Science Department</i>		
3	Branch Prediction	29
<i>Philip G. Emma IBM T.J. Watson Research Laboratory</i>		
4	Trace Caches	87
<i>Eric Rotenberg North Carolina State University</i>		
5	Branch Predication	109
<i>David August Princeton University</i>		
6	Multipath Execution	135
<i>Augustus K. Uht University of Rhode Island</i>		
7	Data Cache Prefetching	161
<i>Yan Solihin¹ and Donald Yeung² North Carolina State University;¹ University of Maryland at College Park²</i>		
8	Address Prediction	187
<i>Avi Mendelson Intel Mobil Micro-Processor Architect</i>		
9	Data Speculation	215
<i>Yiannakis Sazeides,¹ Pedro Marcuello,² James E. Smith,³ and Antonio González^{2,4} ¹University of Cyprus; ²Intel-UPC Barcelona Research Center; ³University of Wisconsin-Madison; ⁴Universitat Politècnica de Catalunya</i>		
10	Instruction Precomputation	245
<i>Joshua J. Yi,¹ Resit Sendag,² and David J. Lilja³ ¹Freescale Semiconductor Inc.; ²University of Rhode Island; ³University of Minnesota at Twin Cities</i>		

11 Profile-Based Speculation	269
<i>Youfeng Wu and Jesse Fang</i> Intel Microprocessor Technology Labs	
12 Compilation and Speculation	301
<i>Jin Lin, Wei-Chung Hsu, and Pen-Chung Yew</i> University of Minnesota	
13 Multithreading and Speculation	333
<i>Pedro Marcuello, Jesus Sanchez, and Antonio González</i> Intel-UPC Barcelona Research Center; Intel Labs; Universitat Politecnica de Catalunya; Barcelona (Spain)	
14 Exploiting Load/Store Parallelism via Memory Dependence Prediction	355
<i>Andreas Moshovos</i> University of Toronto	
15 Resource Flow Microarchitectures	393
<i>David A. Morano,¹ David R. Kaeli,¹ and Augustus K. Uht²</i>	
¹ Northeastern University; ² University of Rhode Island	
Index	421

List of Tables

4.1	Growth in instruction-level parallelism (ILP).	87
5.1	IMPACT EPIC predicate deposit types.	113
5.2	Five comparison type combinations for IA-64	124
6.1	Multipath taxonomy and machine/model characteristics	139
10.1	Characteristics of the 2048 most frequent unique computations	249
10.2	Number of unique computations	250
10.3	Key performance parameters	254
10.4	Selected SPEC CPU 2000 benchmarks and input sets	254
10.5	Processor performance bottlenecks	261
12.1	Using control and data speculation to hide memory latency. .	302
12.2	Example 12.2	312
12.3	Enhanced Φ insertion allows data speculation.	317
12.4	Enhanced renaming allows data speculation	318
12.5	An example of speculative load and check generation	321
12.6	Different representations used to model check instructions and recovery code	322
12.7	Two examples of multi-level speculation	323
12.8	Examples of check instructions and their recovery blocks in multi-level speculation	324
12.9	Examples of check instructions and their recovery blocks in multi-level speculation	325
12.10	Example of recovery code generation in speculative PRE . . .	328
12.11	The recovery code introduced in the speculative PRE interacts with the instruction scheduling	329
14.1	Program input data sets	376
14.2	Default configuration for superscalar timing simulations	377
14.3	Misspeculations with speculation/synchronization	387

List of Figures

2.1	Instruction delivery mechanism	10
2.2	Different instruction cache designs	12
2.3	Sample instruction address stream	15
2.4	The stream buffer in action	17
2.5	Out-of-order fetch with a lockup-free instruction cache	18
2.6	Out-of-order fetch in action	20
2.7	The fetch directed prefetching architecture.	21
2.8	The fetch directed architecture in action	22
2.9	Integrated prefetching architecture	25
3.1	Von Neumann's IAS machine	32
3.2	The pipelining concept	38
3.3	Instruction prefetching	41
3.4	An RS-style pipeline with various approaches to branch prediction	44
3.5	Simple state-machines for branch prediction	49
3.6	Decode-time branch predictors	51
3.7	Branch target buffer	56
3.8	Subroutine call and return	59
4.1	Conventional instruction cache.	88
4.2	High-level view of trace cache operation.	90
4.3	Instruction fetch unit with trace cache.	92
4.4	Two-way interleaved instruction cache.	93
4.5	Contents of trace cache line.	96
5.1	A simple predication example	111
5.2	Example from the Cydra 5	121
5.3	Execution time of the SPEC benchmarks	125
5.4	Execution time of the SPEC benchmarks	126
5.5	The multiple definition problem	127
5.6	Serialization to solve the multidef problem.	128
6.1	Branch Trees	138
6.2	Static DEE tree [15] operating on a dynamic code stream.	143
6.3	Typical branch path ID numbering scheme.	144

7.1 Example illustrating Mowry's algorithm.	165
7.2 Example illustrating greedy pointer prefetching.	169
7.3 Example illustrating jump-pointer and prefetch-array pointer prefetching.	171
7.4 Illustration for stream buffer operation.	175
7.5 Correlation prefetching algorithms	178
7.6 Heuristics for pointer identification	179
8.1 Load distribution	195
8.2 Breakdown of address predictability	196
8.3 Predictability of addresses	197
8.4 Predictability of addresses.	198
8.5 Address predictability	199
8.6 Value prediction vs. address prediction - SPECint.	200
8.7 Value prediction vs. address prediction - SPECfp.	201
8.8 Performance gain for the APDP technique.	202
8.9 Colliding vs. conflicting loads.	203
8.10 Speedup of perfect disambiguation over no speculation.	204
8.11 Speedup of naive speculation	206
9.1 Implications of Data Value Speculation (DVS).	218
9.2 Various value predictors.	221
9.3 Data dependence speculation.	227
9.4 Store sets data dependence predictor.	230
9.5 A microarchitecture with data value speculation.	234
10.1 Frequency of unique computations	248
10.2 Percentage of dynamic instructions	249
10.3 Operation of a superscalar pipeline	251
10.4 Instruction precomputation speedup for profile A, run A . . .	255
10.5 Instruction precomputation speedup for profile B, run A . . .	256
10.6 Instruction precomputation speedup for profile AB, run A . .	258
10.7 Instruction precomputation speedup; Profile B, run A.	259
10.8 Speedup due to value reuse; Run A.	260
11.1 Pointer-chasing code with stride patterns.	282
11.2 Irregular code with multiple stride patterns.	283
11.3 Example of stride profile guided prefetching.	285
11.4 High-level algorithm flow.	287
12.1 Redundancy elimination	303
12.2 Instrumentation-based profiling process	304
12.3 Detect data dependence using shadow.	306
12.4 False dependences	307
12.5 Dependence probability	309

12.6 Speculative analyses and optimization	310
12.7 Speculative alias and dataflow analysis	313
12.8 Types of occurrence relationships	316
12.9 Speculative code motion.	320
12.10 Partial ready code motion	327
13.1 Helper thread example.	337
13.2 Speculative architectural thread example.	338
13.3 Example of execution in a speculative multithreaded processor.	343
14.1 Sending memory reads early	356
14.2 Run-time disambiguation: A static approach	359
14.3 The memory conflict buffer: A hybrid approach	360
14.4 Memory dependence speculation: A dynamic approach	362
14.5 Memory dependence speculation policies	364
14.6 Memory RAW dependence set locality	366
14.7 Dynamic synchronization.	367
14.8 Instances of the same RAW dependence	367
14.9 Synchronizing memory dependences.	372
14.10 Supporting multiple static dependences	373
14.11 Performance potential with load/store parallelism	379
14.12 Naive memory dependence speculation	380
14.13 Memory dependence speculation vs. address-based scheduling	382
14.14 Oracle disambiguation	384
14.15 Speculation/synchronization	386
15.1 High-level block diagram of the active station.	401
15.2 Block diagram of an operand block	403
15.3 Operand snooping	405
15.4 High-level view of a resource flow microarchitecture	407
15.5 High-level block diagram of a representative microarchitecture	408
15.6 The Execution Window of a distributed microarchitecture . .	411

Chapter 1

Introduction

David R. Kaeli¹ and Pen-Chung Yew²

Northeastern University,¹ University of Minnesota²

1.1	Speculation for the Instruction Stream	2
1.2	Speculation for the Data Stream	3
1.3	Compiler and Multithreading-Based Speculation	4
1.4	Speculative Microarchitectures	6
	References	6

Advances in VLSI technology will soon enable us to place more than a billion transistors on a single chip. Given this available chip real estate, new computer architectures have been proposed to take advantage of this abundance of transistors. The two main approaches that have been used successfully to improve performance are:

1. increasing the clock rate, and
2. exploiting the available instruction-level parallelism through pipelining, out-of-order execution, and multithreading.

However, to expose the potential instruction-level parallelism present in programs, control flow and data flow constraints inherent in a program must be overcome.

Speculative execution has become a mainstream technique to reduce the impact of dependencies in high performance microprocessors. Speculation can be implemented using a combination of hardware and software techniques. Compiler support is sometimes necessary to fully exploit the benefits of speculation. The key attributes associated with effective speculative techniques are:

- the ability to accurately predict the event with high accuracy,
- the ability to recover quickly when a speculative guess was incorrect,
- the ability to identify those events which can be predicted with high accuracy, and
- the ability to filter out those events which are hard to predict.

This book brings together experts from both academia and industry who are actively involved in research in the various aspects of speculative execution. The material present in this book is organized around four general themes:

1. instruction-level speculation,
2. data-level speculation,
3. compiler-level support and multithreading for speculative execution, and
4. novel speculative execution architectures.

Next we will discuss each of these topics as a brief introduction to this book.

1.1 Speculation for the Instruction Stream

Chapters 2-6 cover topics related to speculation techniques for determining the instruction stream which will be followed by the program. To be able to obtain any benefits from data speculation, we must first be executing the correct instruction stream. *Control hazards* are inherent in the execution of any program where decisions need to be made. Conditional branches can introduce a large number of stalls in the pipeline, waiting for the dependent condition to be evaluated. Once we know the direction of the conditional branch (i.e., taken or not taken), we need to compute the target address to fetch instructions from.

Chapter 2 treats the subject of instruction cache prefetching by looking at a number of mechanisms that try to load instructions into the cache before the address is requested. Cache design dominated architecture literature during the 1980's [1], and still remains a key design point for obtaining high performance. Cache prefetching is commonly implemented in most instruction caches today.

Chapter 3 presents a complete overview of issues related to speculating on the direction and target address of branch [2] and jump instructions [3]. The subject of branch prediction dominated computer architecture literature during the 1990's [4]. This chapter presents some of the early history of control flow in the architecture field, and also some of the earliest attempts to speculate beyond these dependencies [5].

Chapter 4 introduces a more recent speculative instruction delivery mechanism, the *Trace Cache* [6]. This mechanism attempt to make more efficient use of the available memory space by storing sequences of instruction blocks in an on-chip memory. Trace caches have begun to appear on high performance microprocessors [7]. Trace caches [8] supply a speculative *trace* of instructions, based on past execution frequencies. The goal is to provide the processor with instructions that may span many control flow blocks.

Chapter 5 covers a very different approach to handling control flow dependencies. Instead of attempting to decide which path a branch will take, we can use *branch predication* [9] to allow us to execute instructions down both paths. Once the branch is resolved, the instructions speculatively executed down the wrong path can be *squashed* (i.e., their results can be discarded). Predication has recently been implemented in recent microprocessor architectures [10].

Finally, in the last chapter in this section, an aggressive branch resolution approach is described. The concepts of *eager execution* [11] and *multi-path execution* are presented. The key concept here is to begin to issue and execute instructions from both legs of a branch decision. As long as there are enough functional units available, one of the speculative paths should be able to provide the correct execution. While this form of speculation may require significant hardware resources, it does guarantee some significant performance benefits [12].

Next we consider how best to overcome data dependencies that are present in our execution.

1.2 Speculation for the Data Stream

Chapters 7-10 cover topics related to using speculative execution for the purpose of reducing access time of the data in programs. As the gap between the speed of microprocessors and their supporting memory systems continues to grow, data access time to memory, especially to the second-level cache memory and beyond, has become the most dominant performance bottleneck on today's computer systems (commonly referred to as the *memory wall*).

In Chapter 7, data prefetching schemes that are used to increase the data cache hit ratio are surveyed and discussed. In general, the spatial and temporal locality present in the data stream is not as high as locality in the instruction stream. Data prefetching, if done properly, has been shown to be a very effective way to hide cache miss latency. These mechanisms load data into data cache before they are needed. These schemes are speculative in nature because the addresses used for prefetching are often speculatively generated. Hence, the prefetched data may not be exactly what are needed later in the execution, and they may replace some useful data still being used in the cache. There have been many software and hardware schemes proposed for data prefetching. Array-based data structures possessing constant strides are simple to prefetch, while structures addressed by pointers present difficulty. Schemes such as jump pointers [13], dependence-based prefetching [14] and content-based prefetching are used for such complex memory accesses.

Chapter 8 focuses on speculative schemes that can improve so-called *fetch – load – to – use delay* (i.e., the time between the memory load instruction

is fetched and the time the instruction that depends on the value of the load instruction can use it.) These schemes use techniques such as address prediction and dependency prediction [15]. The emphasis in this chapter is on out-of-order processor architectures. A taxonomy of address calculation and address prediction is presented. Address prediction is a special case of value prediction, and utilizes similar schemes [16]. Issues related to speculative memory disambiguation in out-of-order processors, as well as empirical data on the potential of such speculative disambiguation, are also presented.

Chapter 9 presents mechanisms that perform *data value* speculation. In general, *data dependence* speculation tries to speculatively ignore data dependences that are either too ambiguous or too rare to occur [17]. Data value speculation tries to predict values of the *true* data dependencies that will actually occur during program execution. The data dependent instructions can thus proceed speculatively without waiting for the results of the instructions they depend on [18]. Various value predictors and the issues related to their implementation such as their hardware complexity are presented. Issues related to data dependence speculation are described in the context of data value prediction. Four different approaches are discussed that can be used to verify the validity of a value prediction, and to recover from a value mis-prediction [19].

Chapter 10 focuses an approach that overcomes data access latency and data dependence by combining aggressive execution and value speculation. It has been observed that many computations are performed repeatedly and redundantly [20]. By dynamically removing such redundant computations, the stalls due to memory latency and data dependences for those computations can be removed. The approach itself could be implemented as a non-speculative mechanism (i.e., the block of computations could wait until all of its live-in inputs are available before it determines whether or not the block has been computed before, and thus redundant). However, using value prediction for those live-in inputs can provide a significant performance advantage and can allow us to take large speculative steps through the execution space. Empirical data is presented in this chapter which demonstrates the potential effectiveness of this scheme.

1.3 Compiler and Multithreading-Based Speculation

For most software-based speculative schemes, the compiler is a crucial and

不可或缺的，
必不可少的 indispensable component that impacts the effectiveness of the schemes. Chapters 11 and 12 present recent developments in compiler technology that support speculation. In Chapter 13, issues related to the design of multithreaded processors that support thread-level speculation are discussed.

Chapter 11 presents how to utilize profiles during compilations. These profiles include instruction-level *control flow* profiles such as the execution frequency of basic blocks, control flow graph edge and path profiles [21]; path and call graph profiles [22]; data access profiles such as memory characteristics such as affinity profiles, cache profiles, hot data stream profile, alias profile, data dependence profile and access stride profile [23, 24]; and value profiles such as top-value profile and reuse profile. These profiles can be obtained by either using software instrumentation or performance monitoring hardware. However, to feed the profiling information back to the compiler and use it effectively for compiler optimizations, we will need special support in the compiler. The remainder of the chapter describes methods to use profiling information to support compiler optimizations such as speculative code scheduling that can improve instruction-level parallelism, and instruction and data layout that can improve instruction cache and data cache performance. The profiles can also be used by the compiler to support data prefetching, thread-level speculation, speculative computation reuse and speculative pre-computation for better cache memory performance.

In Chapter 12, a compiler framework is presented that supports both data and control speculation for many common compiler optimizations using alias analysis [25] and data dependence [26]. By providing profile-generated alias and dependence information to the compiler, a number of optimizations can be applied speculatively. If profiling only depends on a particular set of inputs used during the profile run, only a subset of the program execution is profiled. This may produce incorrect assumptions if this information is used during compilation. Instead, the compiler framework described in this chapter uses speculative partial redundancy elimination (PRE) and shows how alias profiling information can be used in PRE to improve its effectiveness and performance. This chapter also touches on issues related to how well the compiler can generate recovery code to produce correct results when mis-speculations occur. The chapter also covers issues related to data dependence and alias profiling.

Chapter 13 presents an overview of some recently proposed multithreaded processors [27] that support thread-level speculation. Even though these processor architectures have multiple processing units, they are quite different from the traditional multiprocessors in the sense that their primary design goal is on improving instruction-level parallelism (ILP) in the application programs, i.e., they are trying to improve the number of instructions executed in each clock cycle. Hence, they are not concerned with issues such as scalability to hundreds or thousands of processing units as in traditional multiprocessors. They are mostly limited to a small number of processing units and are targeting general-purpose applications. Using thread-level parallelism (TLP) to enhance ILP speculatively for general-purpose applications presents an unique challenge to the architecture design. This chapter gives a detailed account on the design issues related to such machines.

1.4 Speculative Microarchitectures

In Chapters 14 and 15 we present two different approaches to instruction and memory dependence from a system perspective. Chapter 14 looks at how best to overcome data dependencies by issuing memory operations out of order [28]. Obviously, this can introduce questions of coherency and consistency. Chapter 14 presents a complete architecture that can be used to address data memory stalls.

In Chapter 15 the concept of resource flow execution is described. The basic idea is that instructions should be able to execute as soon as their dependent operands are available. If we introduce speculation into this model, we can further improve performance through aggressive execution. The authors present the basic elements of the Levo microarchitecture [29].

References

- [1] A. J. Smith. Cache Memories. *Computing Surveys*, 14(3):473–530, September 1982.
- [2] C.H. Perleberg and A.J. Smith. Branch target buffer design and optimization. *IEEE Transactions on Computers*, 42(4):396–412, Apr. 1993.
- [3] D.R. Kaeli and P.G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 34–42, May 27-30 1991.
- [4] T.Y. Yeh and Y.N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124–134, May 19-21 1992.
- [5] J. Cocke, B. Randell, H. Schorr, and E.H. Sussenguth. Apparatus and method in a digital computer for allowing improved program branching with branch anticipation, reduction of the number of branches, and reduction of branch delays. U.S. Patent #3577189, assigned to IBM Corporation, Filed Jan. 15, 1965, Issued May 4, 1971.
- [6] Q. Jacobson, E. Rotenberg, and J.E. Smith. Path-based next trace prediction. In *Proceedings of the 30th Annual International IEEE/ACM Symposium on Microarchitecture*, pages 14–23, Dec. 1-3 1997.

- [7] 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*, Feb. 2001.
- [8] A. Peleg and U. Weiser. Dynamic flow instruction cache memory organized around trace segments independent of virtual address line, Jan 1995.
- [9] P. Y. Hsu and E. S. Davidson. Highly concurrent scalar processing. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 386–395, June 1986.
- [10] Inside the Intel Itanium 2 processor. Hewlett Packard Technical White Paper, July 2002.
- [11] A. Uht and V. Sindagi. Disjoint Eager Execution: An Optimal Form of Speculative Execution. In *Proc. of the 28th International Symposium on Microarchitecture, Ann Arbor, MI*, pages 313–325, Nov./Dec. 1995.
- [12] T.F. Chen. Supporting highly speculative execution via adaptive branch trees. In *Fourth International Symposium on High-Performance Computer Architecture*, pages 185–194, Feb. 1-4 1998.
- [13] C. Luk and T. C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, October 1996.
- [14] A. Roth, A. Moshovos, and G. Sohi. Dependence Based Prefetching for Linked Data Structures. In *the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, October 1998.
- [15] B. Calder and G. Reinman. A comparative survey of load speculation architectures. *Journal of Instruction-Level Parallelism*, 1(39):2–40, January 2000.
- [16] F. Gabbay and A. Mendelson. Using value prediction to increase the power of speculative execution hardware. *ACM Transactions Computing Systems*, 16(3):234–270, 1998.
- [17] M.H. Lipasti and J.P. Shen. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 226–237, December 1996.
- [18] J. Gonzalez and A. Gonzalez. Control-Flow Speculation through Value Prediction for Superscalar Processors. In *International Conference on Parallel Architectures and Compilation Technique*, pages 57–65, September 1999.

- [19] D. Balkan, J. Kalamatianos, and D. Kaeli. The Impact of Value Misspeculation on Branch Resolution of Out-of-Order Processors. In *Proceedings of the 1st Annual Value Prediction Workshop*, pages 3–9, June 2003.
- [20] J. Yi, R. Sendag, and D. Lilja. Increasing Instruction-Level Parallelism with Instruction Precomputation. In *Proceedings of EuroPar*, pages 481–485, August 2002.
- [21] T. Ball and J. Larus. Branch prediction for free. In *Proceedings of Programming, Language and Design Implementation*, 1993.
- [22] R. Hall. Call path profiling. In *Proceedings of the 14th International Conference on Software Engineering*, Jun 1992.
- [23] Y. Wu and Y.-F. Lee. Accurate invalidation profiling for effective data speculation on epic processors. In *13th International Conference on Parallel and Distributed Computing Systems*, Aug. 2000. Vegas, Nevada.
- [24] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2003.
- [25] T. Chen, J. Lin, W. Hsu, and P.-C. Yew. An Empirical Study on the Granularity of Pointer Analysis in C Programs. In *15th Workshop on Languages and Compilers for Parallel Computing*, pages 151–160, July 2002.
- [26] T. Chen, J. Lin, X. Dai, W. Hsu, and P.-C. Yew. Data Dependence Profiling for Speculative Optimizations. In *13th International Conference on Compiler Construction*, March 2004.
- [27] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous Multithreading: A Platform for Next-Generation Processors. *IEEE Micro*, 17(5):12–19, Sept./Oct. 1997.
- [28] A. Nicolau. Run-time disambiguation: Coping with statically unpredictable dependencies. *IEEE Transactions on Computers*, 38(5):663–678, May 1989.
- [29] A.K. Uht, D. Morano, A. Khalafi, M. de Alba, and D. Kaeli. Levo – A Scalable Processor with High IPC. *Journal of Instruction Level Parallelism*, 5, Aug 2003.

Chapter 2

Instruction Cache Prefetching

Glenn Reinman

UCLA Computer Science Department

2.1	Introduction	9
2.2	Scaling Trends	9
2.3	Instruction Cache Design	10
2.4	Instruction Cache Prefetching	14
2.5	Future Challenges	26
	References	27

2.1 Introduction

Instruction delivery is a critical component of modern microprocessors. There is a fundamental producer/consumer relationship that exists between the instruction delivery mechanism and the execution core of a processor. A processor can only execute instructions as fast as the instruction delivery mechanism can supply them.

Instruction delivery is complicated by the presence of control instructions and by the latency and achievable bandwidth of instruction memory. This Chapter will focus on the latter problem, exploring techniques to hide instruction memory latency with aggressive instruction cache prefetching.

2.2 Scaling Trends

Interconnect is expected to scale poorly due to the impact of resistive parasitics and parasitic capacitance. The latency of a memory device, to a first order, is the latency to exercise the logic in the decoder, assert the wordline wire, read the memory cell logic, assert the bitline wire, and finally exercise the logic in the bitline multiplexor to select the accessed data. As the process feature size is scaled, the latency of the transistors is scaled proportional

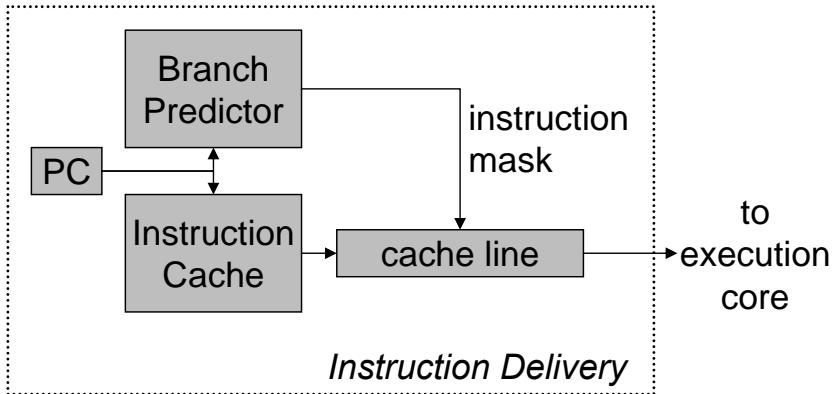


FIGURE 2.1: Instruction delivery mechanism: High level view of the structures of the front-end.

to their size, thus the latency of the logic scales linearly with feature size reductions.

The latency of the wordlines and bitlines, on the other hand, does not scale as well due to parasitic capacitance effects that occur between the closely packed wires that form these buses. As the technology is scaled to smaller feature sizes, the thickness of the wires does not scale. As a result, the parasitic capacitance formed between wires remains fixed in the new process technology (assuming wire length and spacing are scaled similarly).

Since wire delay is proportional to its capacitance, signal propagation delay over the scaled wire remains fixed even as its length and width are scaled. This effect is what creates the interconnect scaling bottleneck. Since on-chip memory tends to be very wire congested (wordlines and bitlines are run to each memory cell), the wires in the array are narrowly spaced to minimize the size of the array. As a result, these wires are subject to significant parasitic capacitance effects. Agarwal et al. [AHKB00] conclude that architectures which require larger components will scale more poorly than those with smaller components. They further conclude that larger caches may need to pay substantial delay penalties.

2.3 Instruction Cache Design

Figure 2.1 illustrates a typical instruction delivery mechanism. A coupled branch predictor and instruction cache are accessed in parallel with the address contained in the program counter (PC). The PC stores a pointer to the next instruction in memory that is to be executed. The program itself is

loaded into a portion of memory, and the PC represents the memory address where a given instruction is stored.

The instruction cache stores a subset of the instructions in memory, dynamically swapping lines of instructions in and out of the cache to match the access patterns of the application. The size of the lines that are swapped in and out of the cache depend on the *line size* of the cache. Larger line sizes can exploit more spatial locality in instruction memory references, but consume more memory bandwidth on cache misses. The line size, along with the associativity and number of sets of the cache, influences the size, latency, and energy consumption of the cache on each access.

The more instruction addresses that hit in the instruction cache, the more memory latency that can be hidden. However, architects must balance this against the latency and energy dissipation of the cache. The latency of the instruction cache access impacts the branch misprediction pipeline loop of the processor. Therefore, the processor must be designed to make the most efficient use of the available cache space to maximize latency reduction.

Unlike data cache misses, where aggressive instruction scheduling and large instruction windows can help hide memory access latency, instruction cache misses are more difficult to tolerate. Instructions are typically renamed in program order, which means that a single instruction cache miss can stall the pipeline even when subsequent accesses hit in the cache.

Typically the line size is selected based on the target bandwidth required to feed the execution core. An entire cache line is driven out, and a contiguous sequence of instructions are selected from this line based on branch prediction information and the starting PC.

Figure 2.2 demonstrates some alternatives for instruction cache design. These alternatives differ in the latency and energy efficient of the cache, and in the complexity of the implementation.

2.3.1 Direct Mapped Cache

In a direct mapped cache, any given address maps to only one location in the cache. This cache can be impacted more severely by conflict misses, but is fast to access and is energy efficient. Each address can only map to one location in the cache. On an access, the data component drives out the cache line in that one location and the tag component determines whether or not there is a match.

2.3.2 Set Associative Cache

In a set associative cache, any given address can map to more than one location in the cache. This cache has better tolerance of conflict misses, but can take longer to access than the direct mapped cache. There are two main types of set associative caches: serial and parallel access.

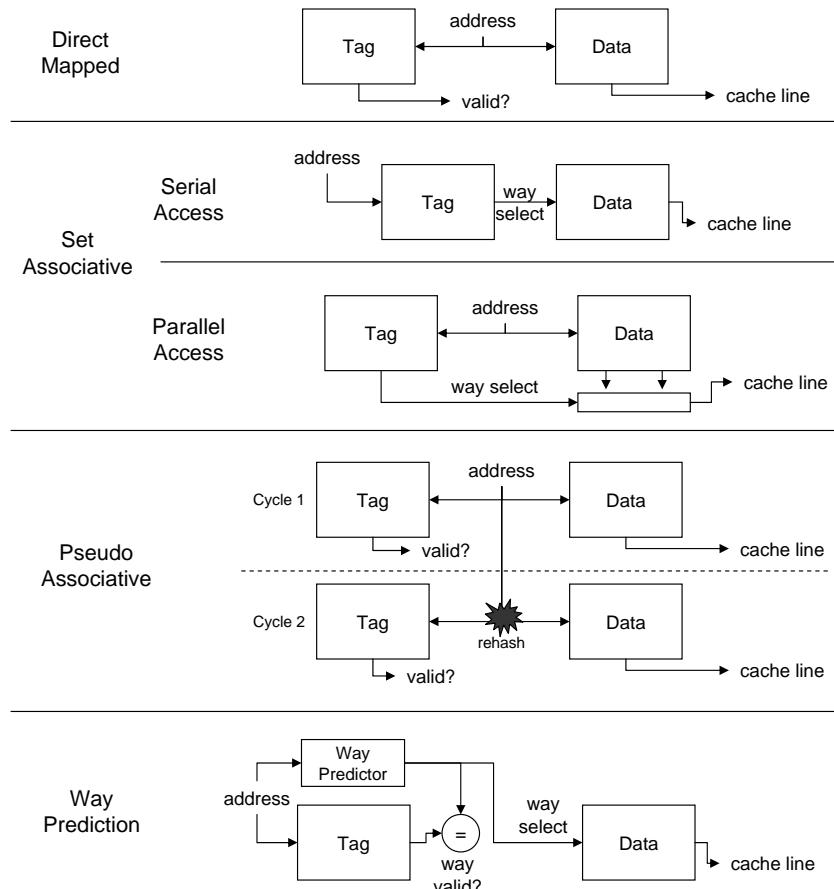


FIGURE 2.2: Different instruction cache designs, trading hit coverage, timing, area, energy, and complexity.

A serial access cache does not overlap the tag and data component accesses as in the direct mapped cache and therefore will have longer latency. But the energy efficiency is close to that of a direct mapped cache since at most one line will be driven out of the data component per access. In fact, it may even provide an energy benefit over the direct mapped cache since there must be a cache hit for a cache line to be driven out. This benefit would depend on how often the cache misses and how much of an energy impact the associative tag comparison has.

A parallel access cache approaches the latency benefits of the direct mapped cache, as the latency of this cache is the maximum of the tag and data component latency, plus some delay for the output driver. However this cache can use dramatically more energy than the direct mapped cache. On a given access to an n -way set associative cache, n different cache lines will be driven to the data output part of the cache until the tag component can select one.

2.3.3 Pseudo Associative Cache

One approach to providing set associative performance with the energy and timing benefits of a direct mapped cache is the pseudo associative cache. This direct mapped cache features multiple hash functions to index into the cache. For example, consider a cache with two hash functions. On a cache access, if the first hash function sees a cache miss, the second hash function is used to initiate a second access in the next cycle. When the first hash function hits, the access only takes a single cycle. But when the second hash function is used, the access takes two cycle and subsequent accesses must be stalled for one cycle. When the second pseudo associative access results in a hit, the cache line stored in the primary location for that address (i.e., the location indicated by the first hash function) is swapped with the cache line stored in the secondary location for that address (i.e., the location indicated by the second hash function).

One challenge of pseudo associative cache design is reducing the number of times that there is a miss on the first access. The other main challenge is performing the swap on a second access hit. However, such a cache can often compare favorably to a multicycle set associative cache. The energy impact of multiple cache accesses can often be tempered by the reduction in per access energy.

2.3.4 Way Prediction Cache

This cache approaches the problem of finding a middle ground between direct mapped timing and energy benefits with set associative performance in a different way. This approach starts with a set associative cache, but uses a small predictor called a way predictor to guess in what way the desired cache line will be found. The tag component is accessed in parallel with the way predictor to verify the guess. In order for the design to make sense, the way

predictor time must be much less than the tag component, to parallelize the tag and data component accesses as much as possible. If the way prediction is correct, only a single cache line will be driven out of the data component, providing the energy benefits of the serial access set associative cache with the timing benefits of the parallel access set associative cache. However, if the way prediction is wrong, the correct cache line will be driven in the following cycle based on the tag access. In this case, the latency of the access is no worse than the serial access set associative cache, but the energy dissipation is slightly worse since two cache lines are driven on a way misprediction. The way predictor must be designed to be fast enough to parallelize the access time to the cache components, small enough to avoid contributing to energy dissipation, but large enough to provide reasonable prediction accuracy.

2.4 Instruction Cache Prefetching

Without any form of prefetching, cache lines are only brought in on demand. These types of misses are known as *demand misses*. The challenge of prefetching is to hide the latency of a future cache miss as much as possible before the miss occurs, but without impacting demand misses. The goal is to bring the *right* line into the cache *before* it is needed. Prefetch schemes can be evaluated based on their accuracy and their timeliness. Prefetch accuracy is critical: it is a waste of memory bandwidth and cache space to bring in lines that will not be used before their eviction from the cache. Timeliness is also key: if a cache line is prefetched only a few cycles before that line is to be used, there is little latency hidden by the prefetch. If a prefetch is started too far in advance, the line may be evicted before it is even used. The following techniques have been proposed to guide instruction cache prefetch or tolerate instruction cache latency.

To better evaluate the following approaches, Figure 2.3 demonstrates a sample instruction address stream. The stream starts at address 992 and continues beyond address 800. For the stream of addresses, assume that 992 and 576 are the only addresses that begin in the instruction cache – the remaining addresses in the stream will be instruction cache misses.

2.4.1 Next Line Prefetching

The most natural instruction cache prefetching strategy is to simply fetch the next consecutive cache line from memory. For example, if the instruction cache line size is 32 bytes, and if the processor is currently fetching cache line address x from the instruction cache, then line $x+32$ would be prefetched.

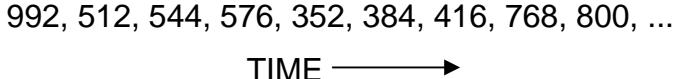


FIGURE 2.3: Sample instruction address stream: This sample stream of *cache line addresses* will be used in subsequent figures. The stream starts with instruction memory address 992.

To limit the number of prefetches, Smith [Smi82] suggested that each instruction cache line be augmented with a single bit to indicate whether or not the next consecutive line should be prefetched. On an instruction cache miss, the bit for the line that missed would be set. On a cache access, if the bit is set for the line that is read from the cache, the next sequential cache line would be prefetched from memory, and the bit for the current line would be cleared.

For the sample instruction stream shown in Figure 2.3, next line prefetching would capture the misses at 544, 384, 416, and 800.

2.4.2 Target Prefetching

Instruction memory references are not always contiguous, and control instructions can reduce the accuracy of next line prefetching. Hsu [HS98] enhanced next line prefetching with target prefetching. They used a line target prediction table to guide what cache line typically follows the current cache line. If the table predicts a next line, that line can then be prefetched while the current cache line is fetched. Target and next line prefetching can work together, either by selectively applying one technique for a given cache line or by using both approaches together on the same cache line, potentially prefetching two lines (a next and a target) for a given cache line.

As observed by [HS98], next line prefetching exploits the same spatial locality that makes larger line sizes attractive for instruction caches. Together with target prefetching, this relatively simple approach can provide high accuracy. However, recent demands on fetch bandwidth and the growing latency to memory can impact the timeliness of these approaches, as the next line may be required in the next cycle of fetch.

With correct branch prediction, Target and Next line prefetching would be able to start a prefetch for all of the misses in the sample instruction stream of Figure 2.3, but would likely start these prefetches too late to hide most of the memory latency of these accesses.

2.4.3 Stream Buffers

Jouppi [Jou90] originally proposed the use of stream buffers to improve the performance of direct mapped caches. On a cache miss, a stream of se-

quential cache lines, starting with the line that missed, are prefetched into a small, FIFO queue (the stream buffer). The prefetch requests are overlapped, so the latency of the first line that misses can hide the latency of any subsequent misses in the stream. Subsequent work has considered fully associative stream buffers [FJ94], non-unit stride prefetches of sequential cache lines [PK94], and filtering unnecessary prefetches that already exist in the instruction cache [PK94]. This latter enhancement is critical in reducing the number of prefetches generated by stream buffers. Idle cache ports or replicated tag arrays can be used to perform this filtering. A redundant prefetch (one that already exists in the instruction cache) not only represents a wasted opportunity to prefetch something useful, but also represents wasted memory bandwidth that could have been used to satisfy a demand miss from the instruction or data cache, or even a data cache prefetch. Multiple stream buffers can be coordinated together and probed in parallel with the instruction cache for cache line hits.

With a conventional stream buffer approach, contiguous addresses will be prefetched until the buffer fills. The buffer must then stall until one of the cache lines in the buffer is brought into the cache (on a stream buffer hit) or until the buffer itself is reallocated to another miss. Confidence counters associated with each buffer can be used to guide when a buffer should be reallocated. One policy might be to use one saturating two-bit counter with each stream buffer. On a stream buffer hit, the counter for that buffer is incremented. When all stream buffers miss, the counters for all buffers are decremented. On a cache miss (and when all stream buffers miss), a stream buffer is selected for replacement if the counter for that stream buffer is currently 00. If no stream buffers exist with a cleared confidence counter (set to 00), then a buffer is not allocated for that cache miss. The counters would not overflow or underflow. This policy would allow stream buffers that are successfully capturing the cache miss pattern of the incoming instruction address to continue prefetching cache lines, and would deallocate buffers that are no longer prefetching productive streams. Similar policies can also be used to guide what stream buffer should be allowed access to a shared memory or L2 port for the next prefetch request.

Figure 2.4 demonstrates the stream buffer in action on our instruction stream example. The PC, instruction cache, result fetch queue, and four stream buffer entries are shown. The *V* field in the stream buffer indicates that the entry is valid and is tracking an in-flight cache line. The *R* field in the stream buffer indicates that the entry is ready – that it has arrived from the other levels of the memory hierarchy.

Cycle k sees a successful cache access for address 992. In cycle k+1, the instruction cache misses on address 512, stalling fetch. A stream buffer is allocated for the miss. Each cycle, the next contiguous cache line in memory is brought into the cache. Assuming that the stream buffer uses the filtering approach of [PK94], address 576 will not be brought into the stream buffer since it already exists in the instruction cache. Eventually, the stream buffer

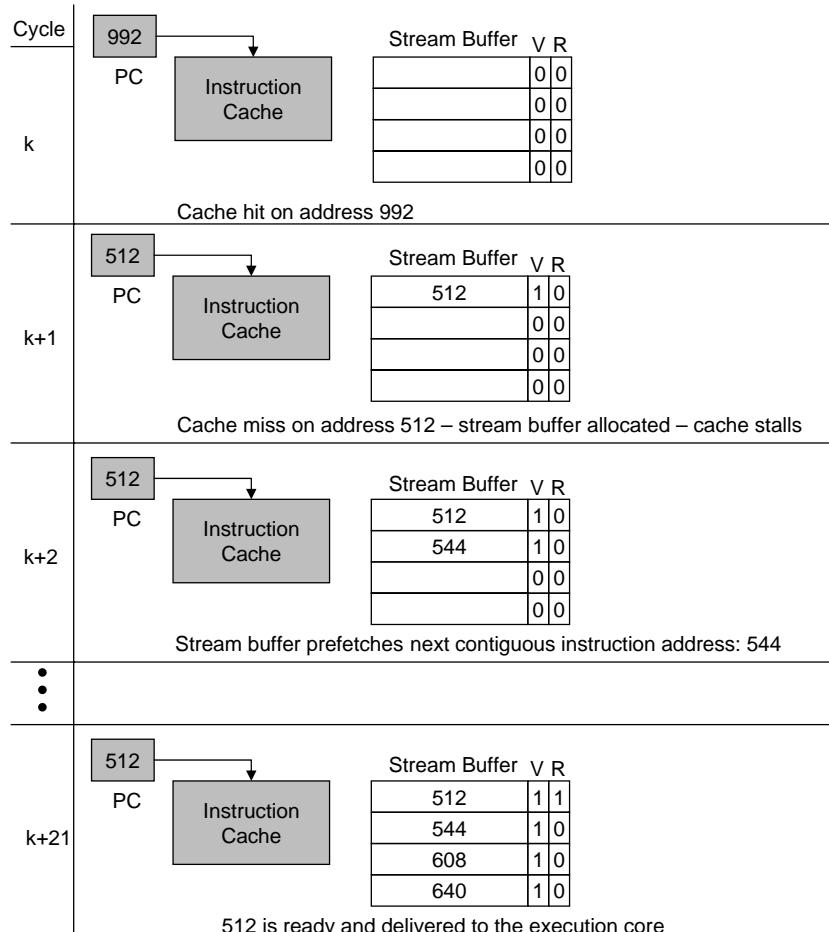


FIGURE 2.4: The stream buffer in action on the first part of the example from Figure 2.3.

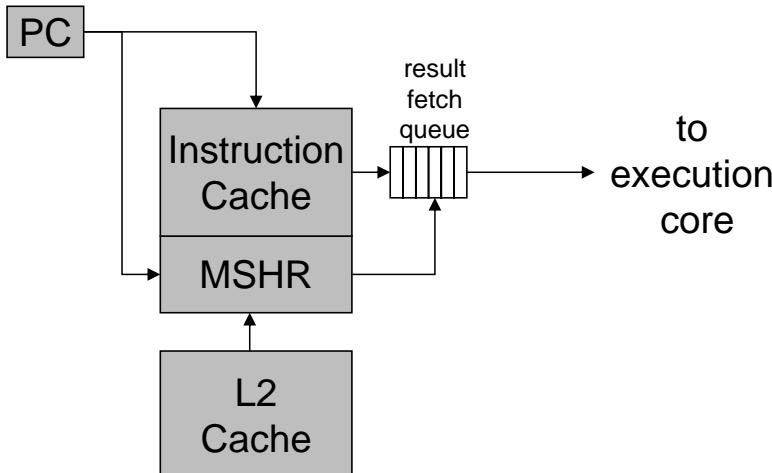


FIGURE 2.5: Out-of-order fetch architecture with a lockup-free instruction cache.

prefetches address 608, which is not referenced in the example in Figure 2.3. In this example, the stream buffer would fill with addresses that are not referenced in the near future, and assuming only a single stream buffer, this buffer would likely be reallocated to a new miss stream. However, there may be some benefit to these prefetches even if the buffer is reallocated, as they may serve to warm up the L2 cache.

2.4.4 Nonblocking Caching and Out-of-Order Fetch

In [Kro81], Kroft proposed a cache architecture that could continue to operate in the presence of a cache miss. This *lockup-free* instruction cache used a number of cache line buffers - known as miss information status history registers or *MSHRs* - to track in-flight cache misses (i.e., those that have not yet returned from the levels of the memory hierarchy). On a cache miss, an available MSHR is allocated to the address that missed and maintains this request until it can be satisfied and installed in the instruction cache. This approach left the instruction cache free to service other requests. The number of in-flight misses was limited by the number of available MSHRs. Nonblocking can help to alleviate the memory latency without some of the downsides of prefetching. It is more of a slave mechanism that can handle more cache accesses during cache misses.

Stark et al. [SRP97] extended the idea of lockup-free caches as an alternative to instruction cache prefetching called *out-of-order instruction fetch*. This architecture is illustrated in Figure 2.5. Instructions can be fetched out-of-order from the instruction cache into a result fetch queue. When the instruction cache misses, an MSHR is allocated for the missed address. Their fetch ar-

chitecture continues to supply instructions after the cache miss to the result queue. A placeholder in the queue tracks where instructions in in-flight cache lines will be placed once they have been fetched from other levels of the memory hierarchy. However, this architecture still maintains in-order semantics as instructions leave this queue to be renamed. If the next instruction in the result queue to be renamed/allocated is still in-flight, renaming/allocation will stall until the instructions return from the other levels of the memory hierarchy.

Out-of-order fetch requires more sophistication in the result fetch queue implementation to manage and update the placeholders from the MSHRs, and to stall when a placeholder is at the head of the result queue (i.e., the cache line for that placeholder is still in-flight).

Figure 2.6 illustrates our example address stream on the out-of-order fetch architecture. The PC, instruction cache, result fetch queue, and four MSHRs are shown. The *V* field in the MSHR indicates that the entry is valid and is tracking an in-flight cache line. The *R* field in the MSHR indicates that the entry is ready – that it has arrived from the other levels of the memory hierarchy.

Cycle k sees a successful cache access for address 992, and that cache line is placed in the result fetch queue. In cycle $k+1$, address 512 misses in the instruction cache and is allocated to an MSHR. A placeholder for this line is installed in the result fetch queue. If there are no other lines in the result fetch queue, stages fed by this queue must stall until the line is ready. However, instruction fetch does not stall. In the next cycle, address 544 also misses in the cache and is also allocated an MSHR. A placeholder for this address is also placed in the queue. In cycle $k+3$, address 576 hits in the cache, and the corresponding cache line is placed in the queue. The stages that consume entries from this queue (i.e., rename) must still stall since they maintain in-order semantics and must wait for the arrival of address 512. Once the cache line for address 512 arrives, it will be placed in the instruction cache and the MSHR for that address will be deallocated. In this simple example, the latency of the request for address 512 helped to hide the latency of 544, 320, and 352. More address latencies could be hidden with more MSHR entries and assuming sufficient result fetch queue entries. This approach is also heavily reliant on the branch prediction architecture to provide an accurate stream of instruction addresses.

The similarity between MSHRs and stream buffers should be noted, as both structures track in-flight cache lines. The key difference is that MSHRs track demand misses and only hold instruction addresses, while stream buffers hold speculative prefetches and hold the actual instruction memory itself.

2.4.5 Fetch Directed Instruction Prefetching

Another approach is to guide stream buffer allocation with the instruction fetch stream. Chen et al. [CLM97] used a lookahead branch predictor to guide

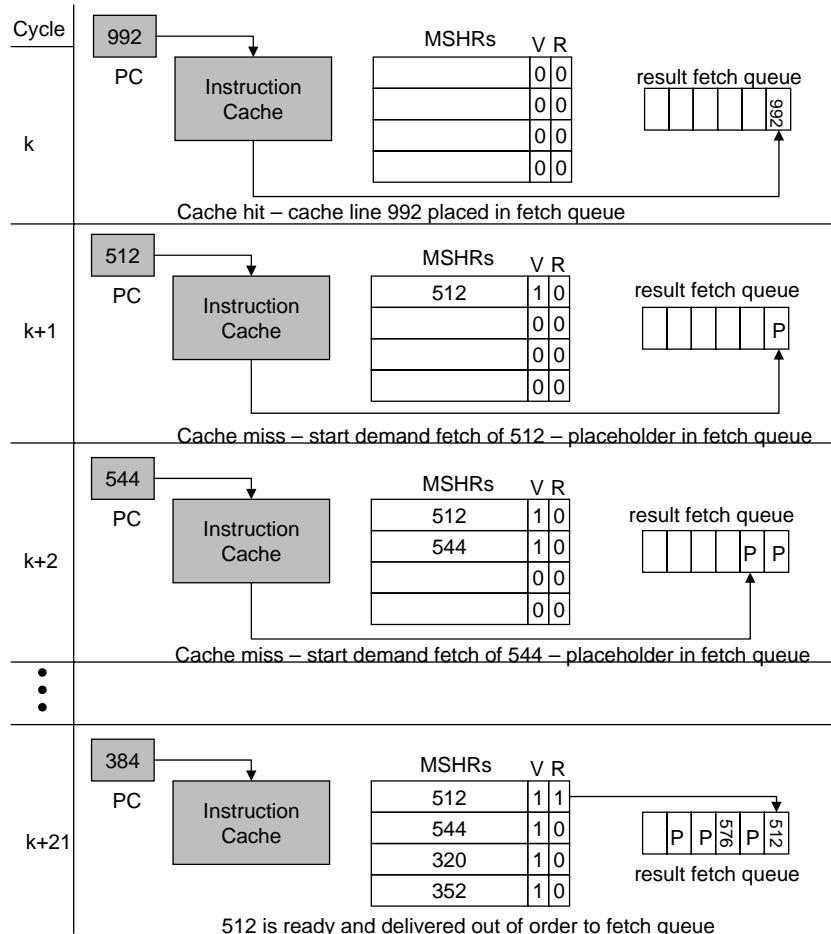


FIGURE 2.6: Out-of-order fetch in action on the first part of the example from Figure 2.3.

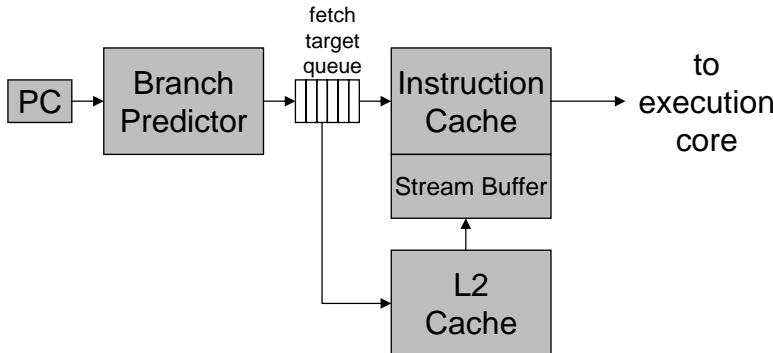


FIGURE 2.7: The fetch directed prefetching architecture.

instruction cache prefetch. Reinman et al. [RCA99] further explored this approach by decoupling the main branch predictor from the instruction cache rather than using a smaller, separate predictor. They used the stream of fetch addresses from this predictor to prefetch the instruction cache. The benefit of this approach is that the same predictor that guides instruction fetch is used to guide instruction prefetch. Aggressive branch prediction designs have improved predictor accuracy, and the benefit obtainable through prefetching is heavily reliant on the accuracy of the prefetches made. As in stream buffer prefetching, idle or replicated instruction cache tag ports are used to filter out redundant prefetches in such schemes.

Figure 2.7 illustrates the architecture of [RCA99]. The branch predictor and instruction cache are decoupled via a queue of fetch target addresses, the fetch target queue. The entries in this queue are used to guide instruction cache prefetching into a FIFO or fully associative stream buffer. The buffer is probed in parallel with the instruction cache.

Figure 2.8 demonstrates the performance of the fetch directed prefetching architecture on our example address stream. The fetch target queue supplies instruction addresses from the branch predictor to the instruction cache. The arrow under the queue indicates the cache line currently under consideration for prefetch. In cycle k, cache line 992 is fetched from the instruction cache while cache line 512 is under considering for prefetch. Address 512 is not found in the instruction cache using cache probe filtering and is prefetched. In the next cycle, the cache misses on address 512 and stalls until this miss is satisfied. In the same cycle, a prefetch is initiated for address 544. Prefetching continues guided by the fetch target queue up until the stream buffer is full. Once 512 is fetched from the next level of memory, fetch can continue.

Fetch directed prefetching and out-of-order fetch are very similar, but there are some key differences. Out-of-order fetch is limited by the number of available MSHRs, just as fetch directed prefetching is limited by the number of available stream buffer entries. But out-of-order fetch is also limited by the

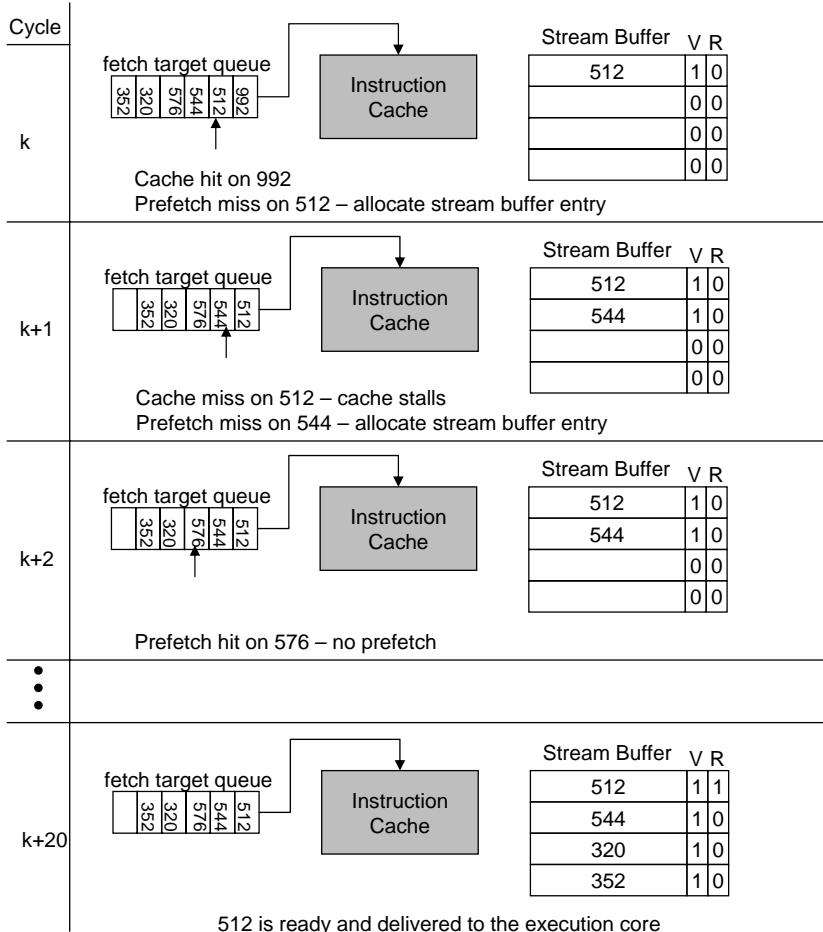


FIGURE 2.8: The fetch directed prefetching architecture in action on the first part of the example from Figure 2.3.

number of entries in the result fetch queue, a structure which stores instructions. Fetch directed prefetching is limited by the number of entries in the fetch target queue, a structure which stores fetch addresses. Occupancy in either of these queues allows these mechanisms to look further ahead at the fetch stream of the processor and tolerate more latency, but the fetch target queue uses less space for the same number of entries. Fetch directed prefetching is only limited by the bandwidth of the branch predictor. To scale the amount of prefetching, the branch predictor need only supply larger fetch blocks. To scale prefetching with accurate filtering, the tag component of the cache must have more ports or must be replicated. In order to scale the number of cache lines that can be allocated to MSHRs by out-of-order fetch in a single cycle, the branch predictor bandwidth must be increased and the number of ports on the instruction cache must increase.

One other difference is that prefetches can often start slightly before out-of-order fetches. Since prefetching is a lookahead mechanism, a prefetch can be initiated at the same time that a cache line is fetched from the instruction cache. This is illustrated in the simple example of Figure 2.8 where address 512 is prefetched one cycle earlier than in out-of-order fetch.

2.4.6 Integrated Prefetching

One drawback to both fetch directed prefetching and out-of-order fetch is the complexity of the queue structures. Fetch directed prefetching uses a queue that can initiate a prefetch from any entry of the queue. Moreover, the tag component of the instruction cache is accessed twice for each cache line address – once to determine whether or not to prefetch a given cache line, and once during the actual instruction cache access.

One alternative to this policy is to more closely integrate instruction prefetching with the design of the instruction cache, as in [RCA02]. The approach of [RCA02] is to decouple the tag and data component of the instruction cache from one another via a queue of cache line requests (the cache block queue). For every cache access, the tag component determines the cache way where a given cache line resides (if any). If it is in the cache, it places the requested address and way in the cache block queue. The data component can then consume this information from the queue and provide the requested line.

Since the tag is only checked once, additional hardware is necessary to ensure that the verified cache lines stored in the cache block queue are not evicted from the cache before they are read from the data component of the cache. This mechanism is the cache consistency table (CCT). The CCT has as many sets and as much associativity as the instruction cache, but only stores 3 bits per line in the CCT. A 64KB 4-way set associative instruction cache would only need a 768B CCT. Each time a cache line is verified by the tag component, the CCT line corresponding to that cache line is incremented. Each time a cache line is provided by the data component, the CCT line corresponding to that cache line is decremented. On a cache replacement, a line will not be

kicked out if its CCT entry is not zero. If the CCT entry saturates, the tag component will not verify any more requests for that cache line until the CCT counter is decremented for that entry. In addition to providing a consistency mechanism, the CCT also provides an intelligent replacement policy. The tag component does not stall on an instruction cache miss, and therefore it can run ahead of the data component. The CCT then reflects the near future use pattern of the instruction cache, and can help guide cache replacement. The larger the cache block queue, the further ahead the CCT can look at the incoming fetch stream. This of course requires accurate branch prediction information.

On a cache miss, the integrated prefetching mechanism allocates an entry in the stream buffer. The stream buffer can bring in the requested cache line while the tag component of the cache continues to check the incoming fetch stream. Once the missed cache line is ready, it can either be installed into the instruction cache (if the CCT can find a replacement line) or can be kept in the stream buffer. This allows the stream buffer to be used as a flexible repository of cache lines, providing extra associativity for cache sets with heavy thrashing behavior.

Figure 2.9 demonstrates the integrated prefetching architecture of [RCA02]. The stream buffer is also decoupled, and requires a CCT of its own to guide replacement. The tag components of both the stream buffer and instruction cache are accessed in parallel. There is a single shared cache block queue to maintain in-order fetch. Each entry in the cache block queue is consumed by the data component of both the instruction cache and stream buffer. However, only one of the two data components will be read depending on the result of the tag comparison.

2.4.7 Wrong-Path Prefetching

Many of the prefetching techniques described in this chapter rely on accurate branch prediction to capture the instruction fetch address stream. However, no branch predictor is perfect, and mispredictions must be handled. While the cache block queue, fetch target queue, or result fetch queue would all be flushed on a misprediction, flushing the stream buffers might throw away potentially beneficial prefetches. Pierce and Mudge [PM96] found that prefetching on incorrectly speculated branch paths can actually be beneficial in many cases. If a branch is predicted to be taken when it is actually not taken, it may be taken at a later point in time. If addresses on the taken path are prefetched and waiting in a stream buffer, these may avoid cache misses when the branch is seen to be taken later. For example, consider a frequently executed while loop. If control is mispredicted to exit the while loop, and the instructions following the loop are prefetched, then that will hide the latency seen when the while loop actually is exited. Another case where wrong path prefetches can help is when short forward branches are mispredicted. The caveat here is that the wrong path prefetch may be evicted by prefetches on

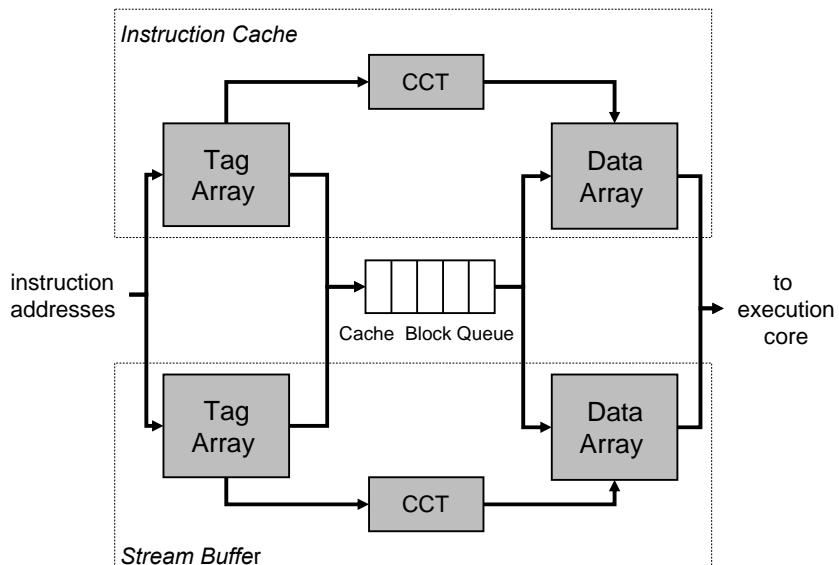


FIGURE 2.9: Integrated prefetching architecture: The cache and stream buffer tag components are decoupled from their respective data components. Cache and stream buffer replacement are both guided by the cache consistency tables (CCT) for the respective structures. If both the cache and stream buffer miss, the desired line is brought into the stream buffer from the L2 or main memory.

the correct path, and that the wrong path prefetch may itself evict useful blocks.

One way to preserve entries in the stream buffers, but still allow new prefetch requests, is to add another bit to each stream buffer entry that indicates whether or not that entry is replaceable or not. If the bit is set, that means that while the entry may be valid, it is from an incorrectly speculated path. Therefore, it may be overwritten if there is demand for more prefetches or misses. But assuming the entry is valid and ready, it should still be probed on stream buffer accesses to see if there is a wrong-path prefetch hit.

2.4.8 Compiler Strategies

Compiler directed approaches have also been proposed to help tolerate instruction cache misses. One common approach is code reordering, where instruction memory is laid out to keep the most frequently taken control path as a contiguous stream of instruction addresses [PH90, HKC97]. By keeping the most frequently taken control path as one sequential instruction address stream, simple techniques like stream buffers or next line prefetching become more effective. This of course requires some compiler heuristic or prefetching to determine the most frequently taken control path. Such work can also help to reduce the conflict misses seen in the instruction cache by restructuring the application to keep code frequently executed together in separate cache sets.

Similarly, procedure inlining [CL99] replaces the actual procedure call with the entire body of the procedure call. While this can increase the overall code size, it also reorders the code so that simple instruction prefetchers can follow a contiguous pattern of instruction stream references.

Another approach is to use explicit prefetch instructions in the ISA. Such instructions can be placed at strategic points in the code where prefetches should be initiated for good timeliness. The limitation is how accurately static techniques can predict the control flow of an application at run time.

The Itanium processor [MS03] makes use of two types of software prefetch instructions: hints and streams. A hint prefetches between 12-48 instructions depending on the type (brp.few or brp.many) and streams continue to prefetch contiguous instruction cache blocks until redirected.

We will explore more compiler strategies in Chapter 4.

2.5 Future Challenges

Instruction cache prefetching has seen improvements in both accuracy and timeliness of prefetches, but there is still room for improvement. Memory usage of applications has been growing recently, spurred on by the increasing

memory capacity projected in future processors. However, as clock rates become more aggressive, **the latency impact of instruction caches** may prevent these caches from growing large enough to meet increased application demand. This will require more intelligent and flexible instruction delivery mechanisms to take advantage of application locality. Multiple application threads may also contribute to this problem, requiring architects to devise novel approaches to warm up threads on context switches. On an SMT machine, what is the best way to use available memory bandwidth and accelerate the particular thread mix in the processor? Researchers are also exploring application specific customization of instruction caches and prefetching policies to individual program phases or compiler directed hints.

References

- [AHKB00] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus ipc: The end of the road for conventional microarchitectures. In *27th Annual International Symposium on Computer Architecture*, 2000.
- [CL99] R. Cohn and P. Lowney. Feedback directed optimization in compaq's compilation tools for alpha. In *In Proceedings of the 2nd Workshop on Feedback-Directed Optimization*, 1999.
- [CLM97] I.K. Chen, C.C. Lee, and T.N. Mudge. Instruction prefetching using branch prediction information. In *International Conference on Computer Design*, pages 593–601, October 1997.
- [FJ94] K. Farkas and N. Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *21st Annual International Symposium on Computer Architecture*, pages 211–222, April 1994.
- [HKC97] A. Hashemi, D. Kaeli, and B. Calder. Efficient procedure mapping using cache line coloring. In *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1997.
- [HS98] W. Hsu and J. Smith. A performance study of instruction cache prefetching methods. In *IEEE Transactions on Computers*, 47(5), May 1998.
- [Jou90] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers.

In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

- [Kro81] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *8th Annual International Symposium of Computer Architecture*, pages 81–87, May 1981.
- [MS03] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. In *IEEE Micro*, March/April 2003.
- [PH90] K. Pettis and R. Hansen. Profile guided code positioning. In *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1990.
- [PK94] S. Palacharla and R. Kessler. Evaluating stream buffers as secondary cache replacement. In *21st Annual International Symposium on Computer Architecture*, April 1994.
- [PM96] J. Pierce and T. Mudge. Wrong-path instruction prefetching. In *29th International Symposium on Microarchitecture*, pages 165–175, December 1996.
- [RCA99] G. Reinman, B. Calder, and T. Austin. Fetch directed instruction prefetching. In *32nd International Symposium on Microarchitecture*, November 1999.
- [RCA02] G. Reinman, B. Calder, and T. Austin. High performance and energy efficient serial prefetch architecture. In *In the Proceedings of the 4th International Symposium on High Performance Computing*, May 2002.
- [Smi82] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.
- [SRP97] J. Stark, P. Racunas, and Y. Patt. Reducing the performance impact of instruction cache misses by writing instructions into the reservation stations out-of-order. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 34–45, December 1997.

Chapter 3

Branch Prediction

Philip G. Emma

IBM T.J. Watson Research Laboratory

3.1	What Are Branch Instructions and Why Are They Important? Some History	29
3.2	Why Are Branches Important to Performance?	36
3.3	Three Dimensions of a Branch Instruction, and When to Predict Branches	42
3.4	How to Predict Whether a Branch is Taken	48
3.5	Predicting Branch Target Addresses	54
3.6	The Subroutine Return - A Branch That Changes Its Target Address	58
3.7	Putting It All Together	62
3.8	Very High ILP Environments	68
3.9	Summary	70
	References	72

3.1 What Are Branch Instructions and Why Are They Important? Some History

The branch instruction is the workhorse of modern computing. As we will see, **resolving branches** is the essence of computing.

3.1.1 The von Neumann Programming Model vs. ENIAC

According to the von Neumann model of computing, instructions are fetched, and then executed in the order in which they appear in a program. The machine that he built in Princeton at the Institute for Advanced Studies during World War II (which we will call the IAS machine) did exactly this [1] [2]. Modern processors might not actually do this (fetch and execute instructions in program order), but when they don't, they generally ensure that no outside observer (other processors, I/O devices, etc.) is able to tell the difference.

A program is a *static* sequence of instructions produced by a programmer or by a compiler. A compiler translates static sequences of high-level instructions (written by a programmer or by another compiler) into static sequences of

lower-level instructions. A machine (processor) executes a *dynamic* sequence of events based on the static sequence of instructions in the program.

Programs contain four primary types of instructions:

1. Data *movement* instructions (called *Loads* and *Stores*),
2. Data *processing* instructions (e.g., add, subtract, multiply, AND, OR, etc.),
3. *Branch* instructions, and
4. *Environmental* instructions.

The first two categories (movement and processing) is what the average person conceives of as *computing*. It is what we typically do with a pocket calculator when we balance our checkbooks. In fact, Echert and Mauchley built ENIAC (slightly before IAS) using only these instructions. ENIAC comprised Arithmetic-Logic Units (ALUs), which operated on data using ALU control information (i.e., Add, Subtract, etc.) [3]. The calculations to be done were configured ahead of time by a human operator who connected the appropriate units together at a patch-panel to create the desired dataflow.

ENIAC was used to generate *firing tables* for field artillery being produced for the War. Specifically, it was used to generate and print solutions to simple (linear) polynomials for specific sequences of inputs covering relevant operational ranges for the artillery (each cannon barrel is slightly different because of the process variations inherent to the manufacturing of things this large).

The kind of computation done by ENIAC is still done today, but today we don't normally think of these applications as *computing*. For example, digital filters do exactly this - for a sequence of input data, they produce a corresponding sequence of output data. A hardwired (or configurable) filter doesn't execute *instructions*. It merely has a dataflow which performs operations on a data stream as the stream runs *through* the filter. Although this is very useful, today we would not call this a *computer*.

3.1.2 Dataflow and Control Flow

What von Neumann featured in his programming model was the ability for a program to change what it was doing based on the results of what it was calculating, i.e., to create a dynamic sequence of events that differed from the static sequence of instructions in the program. Prior to von Neumann, a computer (like ENIAC) merely had *dataflow*. There had been no working notion of a program.

Clearly, Alan Turing had had a similar notion, since without the ability to make conditional decisions, a Turing machine would only run in one (virtual) direction. Godel is famous for a remarkable proof that there are meaningful functions that cannot be computed in a finite number of steps on a Turing machine, i.e., whether the machine will halt is not deterministic [4]. This is

called “The Halting Problem.” Any finite-length program without conditional decision points certainly would be deterministic (i.e., whether it halts could be determined).

Many claim that others - such as Charles Babbage - had had similar notions (conditional control) in the mechanical age of calculation [5].

Von Neumann conceived a *program* as comprising two orthogonal sets of operation that worked in conjunction:

1. A *dataflow* which did the physical manipulation of the data values, and
2. A *control flow*, which dynamically determined the sequence(s) in which these manipulations were done.

The syzygy of these two things allowed a much more complex (today, we might say *capricious*) set of calculations. In fact, von Neumann conceived this kind of computing with the goal of solving a set of nonlinear differential equations (from Fluid Dynamics) for the purpose of designing the detonation means for atom bombs. The War ended before the machine was fully operational.

The control flow of a program is implemented by instructions called *branch* instructions.

3.1.3 The Branch Instruction

Recall that a (static) program is a sequence of instructions. When the program is loaded into memory so that it can be executed, each instruction in the sequence is physically located at a specific address. (In the vernacular of Computer Science, *address* means *location in memory*.) Since the program is a static sequence, the addresses of its instructions are sequential as well. For example, a program that is loaded at address 100 will look like this in memory:

LOCATION	INSTRUCTION
100	Instruction #1
101	Instruction #2
102	Instruction #3

and so on. If no instruction was a branch instruction, the execution of the program (dynamic flow of instructions) would be exactly the static sequence: Instruction #1, Instruction #2, Instruction #3, and so on.

A branch instruction causes the flow of the program to divert from the next sequential instruction in the program (following the branch instruction) to a different instruction in the program. Specifically, it changes the flow of addresses that are used to fetch instructions, hence it changes the instruction flow.

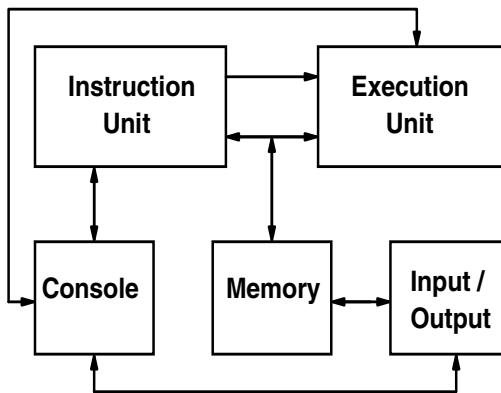


FIGURE 3.1: Von Neumann's IAS Machine.

3.1.4 The IAS Machine: A Primitive Stored-Program Architecture

Recall that von Neumann conceived two essential (and orthogonal) aspects of a computation: a dataflow, and a control flow. The IAS machine embodied these concepts in separate *engines*, which today are called the *Instruction Unit* (IU) and the *Execution Unit* (EU). Machines today can have a plurality of each of these; both real (for high instruction-level parallelism) and virtual (for multiple threads).

Conceptually and physically, the IAS machine looks like the diagram in Figure 3.1. In addition to the IU and EU, there is a memory (M) which holds the data to be operated on by the IU and EU, an Input/Output device (I/O) for loading data into M, and an operator's Console (C) that allows the operator to initiate I/O and subsequent execution.

By today's standards, this is an extremely primitive system, although it is considered to be the first *stored program computer*, so called because it is the first machine having an actual program that was stored into memory where it was operated upon. This machine is primitive in two important ways.

First, the machine and its *programming model* (the way in which a programmer conceives of its operation) draws no distinction between an *instruction* and an *operand* (a datum to be operated on). To this machine, they are both *data*. Whether the content of a memory location is interpreted as an

instruction or as an operand depends entirely on the context in which it is fetched.

When the IU issues an *instruction fetch* to the memory, the data that is returned is considered to be an instruction. The IU decodes that data, and initiates actions based on the values of the bits in the (predetermined) positions containing the *operation code* (called the *opcode field* of the instruction). When the IU issues an *operand fetch* to the memory, the data that is returned is sent to the EU, where it is subsequently processed. Since the machine draws no distinction between instruction data and operand data, instructions in a program can be (and were) operated on by the EU, and changed by the program as the program ran. Indeed, in those days, this was the programmer's concept of how to implement control flow (which would be anathema to any programmer today - in fact, this idea would probably not even occur to most students of Computer Science today).

3.1.5 Virtuality

The second important way in which the IAS machine is primitive is that there is no notion of anything virtual. While today, we have virtual systems, virtual processors, virtual processes, virtual I/O, and virtual everything else (all outside the scope of this chapter), a very basic concept appearing in commercial processors (well after IAS) was virtual memory.

The concept of virtual memory was first used in the Atlas computer [6]. The basic notion of virtual memory is that the logical addresses (as used logically by a program) did not need to be in direct correspondence with specific physical locations in the memory. Decoupling the logical from the physical enables three things:

1. The amount of logical memory (as conceived of by the program) can be much larger than the amount of real storage in the machine - providing that there is a dynamic way of mapping between the two as the program runs.
2. Programs can be written without specific knowledge of where they will reside in memory when they are run. This allows multiple coresident programs to use the same logical addresses although they are referring to different parts of the physical storage, i.e., it allows every program to *think* that it begins at address 0, and that it has all of memory to itself.
3. It allows parts of programs to be moved into and out of the memory (perhaps coming back to different physical locations) as they run, so that an operating system can share the available real memory between different users as multiple programs run concurrently.

Without virtual memory, programs written for the IAS machine could only refer to real locations in memory, i.e., a program was tightly bound to a

specific physical entity in those days. (In modern programming, the logical and the physical are deliberately decoupled so as to allow a program to run on many different machines.)

In the IAS machine, this means that the program and its data is constrained to not exceed the size of the real memory. The IAS memory system has 1024 words in total. Can you image solving nonlinear differential equations using 1024 words total? Programs written to do this were exceedingly abstruse, with instructions changing each other's contents as the program ran. This is mind-boggling by today's standards.

3.1.6 Branch Instruction Semantics

A branch instruction has the form “If $\langle\text{condition}\rangle$ Then GoTo $\langle\text{address}\rangle$,” where $\langle\text{condition}\rangle$ specifies some bits that can be set by other (temporally prior) instructions, and $\langle\text{address}\rangle$ is the address of a place in the program to which instruction sequencing is diverted if $\langle\text{condition}\rangle$ is true. This allows the sequencing of instructions in a program to be dynamically altered by data values that are computed as the program runs (e.g., “If $\langle\text{end of record}\rangle$ Then GoTo $\langle\text{program exit}\rangle$ ”).

If the specified $\langle\text{condition}\rangle$ is the constant value *true*, then the branch is always taken. This is called an *unconditional branch*, and it has the abbreviated semantics “GoTo $\langle\text{address}\rangle$.¹ Branches for which $\langle\text{condition}\rangle$ is a variable (i.e., for which $\langle\text{condition}\rangle$ may or may not be true - depending on the results of certain calculations) are called *conditional branches*. As we will see later, there is a big difference in the ability to predict conditional branches. (Hint: unconditional branches are always taken.)

As should be evident, adding branch instructions to a program increases the flexibility of the program (literally) infinitely. It enables a completely new dimension (called control flow) to be part of a calculation. More than any other thing (except perhaps virtualization), this simple concept has enabled computing machines to undertake tasks of previously unimaginable complexity - for example, financial modeling, engineering simulation and analysis (including the design of computers), weather prediction, and even beating the world's best chess player.

Without the branch instruction, computing would probably not have advanced beyond mundane calculations, like simple digital filters. With the branch instruction, we (humans) have enabled an infinitely more complex system of filters to evolve - called *data mining* - which is a proxy for *intuiting* (i.e., it has a nearly psychic overtone).

3.1.7 General Instruction-Set Architectures and Extensions

Today, user programs comprise (primarily) movement instructions, processing instructions, and branch instructions. Above, we had also mentioned *environmental instructions* as a fourth category. Today, this last category is

primarily limited to privileged operations performed by the *Operating System*, and not usually allowed by user programs. These are mainly used to facilitate aspects of virtualization - to manage and administer physical resources via the manipulation of architected state that is *invisible* to the computer user. This state includes page tables and other kinds of system status, interrupt masks, system timers and time-of-day clocks, etc.

As applications become increasingly complex, some of this state is becoming subject to manipulation by certain applications in a very controlled way. Today, this *state* corresponds to tangible entities. On the horizon, some of this state will become genuinely abstract. Four such examples of abstract state are:

1. What is the parallelism in a sub-sequence of instructions? What can be done out of order, and what must be rigorously in order? What are the predictable future actions of the program at this point? This sort of information can be used by hardware (if it exists) to get more performance out of a program.
2. What level of reliability is required for this calculation? Do we need this to be right, or do we just need it to be fast? This can influence the hardware and algorithms brought to bear on an abstract semantic construct.
3. How hot (temperature) is it? Should we run certain units differently and/or shift part of the load around the processor so as to maintain a lower ambient temperature? This targets energy efficiency and long-term product reliability.
4. Will this code cause power transients as it is run? Is there a better set of controls, or other switching that can be brought to bear to keep the voltage stable?

These last categories can be thought of as *green* instructions because they target efficient energy usage and product reliability.

For the most part, this *green* category does not yet exist in any commercial instruction set, but as systems are becoming more power limited, these things will emerge. We will not discuss environmental instructions further; instead, we will focus on the branch instructions. We will argue that branch instructions are the most important instructions from a performance perspective, because they are the primary limiters to instruction-level parallelism [7, 8, 9, 10].

3.2 Why Are Branches Important to Performance?

Computation is all about changing *state*, where state comprises the observable data within a system. A computer operates on state by fetching it (so that the computer can observe the state), and by storing it (so that the computer can alter the state). Therefore, at first blush, computing is all about fetching and storing, and there is an important duality between these two operations.

3.2.1 Memory Consistency and Observable Order

While the following illustrations sound trivial, understanding them is vital to the concept of coherency among (and within) processes.

First, imagine a computer that can only fetch, but cannot store. Such a computer can only observe state; it cannot alter it. Hence the state is static, and no computation is ever manifest - which is not distinguishable from no computation ever being done. Hence, this is not really a computer.

Next, imagine a computer that can only store, but cannot fetch. (This is the dual of the above.) This computer continually changes the state, but is unable to observe any of these changes. Since the state is not observable, the state need not change - or even exist. This is also not distinguishable from no computation being done, hence this is not really a computer either.

For a computer to do anything meaningful, it must be able to observe state (fetch) and to change state (store). What makes any particular computation unique and meaningful is the sequence in which the fetches and stores are done. In the absence of branch instructions, the order of the fetches and stores would be statically predetermined for any program - hence the result of any calculation would be predetermined, and there would be no philosophical point to doing a computation.

The branch instructions create unique and dynamic orderings of fetch and store instructions, and those orderings are based on the observation of state (via the “If <condition>” clause) as the state is changed by running processes. Ergo, computation is essentially all about resolving branches.

3.2.2 Branches and Performance

Then from first principles (see above), the maximum rate of instruction flow is gated by the ability to resolve branches. *Minsky’s Conjecture* is that the maximum parallelism in a program is $O(\log N)$ where N is the number of processing elements [11]. My fancied interpretation of this (which was not necessarily Minsky’s, although valid nonetheless) is that each unresolved branch splits the processing into speculative threads, which creates a specula-

tion tree that can be no more than $\log N$ deep, if N is the number of processing elements.

In addition to branches being the essence of computation, their frequency (static or dynamic) is dominant among all instructions [12]. For typical applications, branches comprise between one-third and one-fifth of all instructions, more-or-less independent of the specific instruction-set architecture, and of the program. In the ensuing examples, we will assume that on average, one in every four instructions is a branch. By frequency alone, it is clear that branches are crucial to performance.

In addition, branches gate performance in several other fundamental dimensions. These dimensions have to do with common performance-enhancing techniques used in microarchitecture to improve *Instruction-Level Parallelism* (ILP). ILP is usually measured (in the literature) in *Instructions Per Cycle* (IPC), although we find it much more convenient to work with its inverse: *Cycles Per Instruction* (CPI) [13].

The prevalent techniques that improve ILP are called pipelining, superscalar processing, and multithreading. Each of these is discussed below so as to make the relevance of branches clearer.

3.2.3 Pipelining

Figure 3.2 progressively illustrates the concept of pipelining. Figure 3.2.a shows the phases of instruction processing as previously discussed for a von Neumann computer. Simply, an instruction is fetched, decoded and dispatched, and executed. The machine that von Neumann built did these three things sequentially for each instruction prior to fetching a next instruction.

Pipelining is basically an assembly-line approach to instruction processing in which multiple instructions can simultaneously be in different phases of processing. For example, suppose that the instruction fetch process fetches instructions continually (without waiting for resolution signals from the other end of the processor), and that the three major blocks shown are run autonomously (with respect to each other) with queues in between them (to decouple them) as shown in Figure 3.2.b. In this figure, at least three instructions can be active simultaneously. Hence the arrangement in Figure 3.2.b has the potential of running a program three times faster (assuming that the logical phases are all equal in time).

The phases in Figure 3.2.b can be partitioned further, as shown in Figures 3.2.c and 3.2.d. The finer the partitioning, the greater the rate at which instructions can enter the pipeline, hence the greater the ILP. Note that partitioning the pipeline more finely does not alter the latency for any given instruction, but it increases the instruction throughput. (In fact, partitioning the logic requires the insertion of latches between each new partition, hence the latency actually increases with the degree of pipelining.) Therefore, the pipeline in Figure 3.2.d has a longer latency, but potentially higher throughput than the pipeline in Figure 3.2.c.

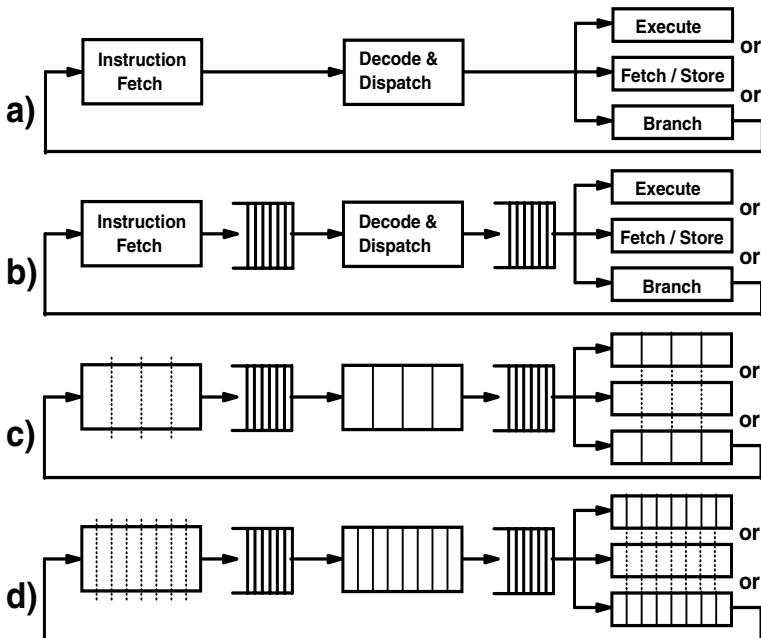


FIGURE 3.2: The pipelining concept: a) Phases of instruction processing; b) Decoupling the phases by the insertion of queues; c) Pipelining the phases; and d) Using a deeper pipeline structure.

Note that combinational logic (the Decode, Execute, and Branch blocks) can be finely partitioned by inserting latches between smaller numbers of logic levels. The number of resulting partitions is called the number of *pipeline stages*, or the *length* (also called the *depth*) of the pipeline. The memory blocks in the pipeline (Instruction Fetch, and Operand Fetch and Store) cannot have latches inserted into them because they are (basically) analog circuits. The pipeline stages in a memory path are conceptual, which is why they are denoted by dotted lines in Figures 3.2.c and 3.2.d. Hence, the memory must have enough banks (which enable concurrent accesses to be in progress) to support the flow implied by the degree of pipelining.

3.2.4 Pipeline Disruptions and Their Penalties

Given enough memory banks, and given no disruptions to the instruction flow, the pipeline can be made very deep, and can achieve very high throughput (hence, ILP). The largest inhibitor of pipeline flow is the branch instruction - recall that we had said that roughly one in four instructions is a branch. As shown in Figure 3.2.a, the branch is (nominally) resolved at the end of the pipeline, and, if taken, will redirect instruction fetching. When this occurs, the entire contents of the pipeline must be invalidated, and instruction flow must be reinitiated. If there are N pipeline stages, this costs (approximately) N cycles.

In fact, all pipeline disruptions revolve around the major functional phases of instruction processing shown in Figure 3.2.a, and are temporal - independent of the degree of pipelining. Hence, the average number of cycles required to process an instruction (called *Cycles Per Instruction*, or CPI) is linear in N , since each disrupting event causes a fixed number of stages to be invalidated [13].

In commercial programs, roughly two out of three branches is taken, hence there is a taken branch every six instructions. Therefore, there is a nominal delay (due to branches alone - before including any of the actual execution) of $N/6$ CPI for an N -stage pipeline. All high-performance processors today are pipelined. Up until this year, the trend had been to make pipelines deeper and deeper [14] to achieve high frequency (which is related to $1/N$). It appears that this trend is reversing as it is becoming much more important to make processing more energy efficient [15].

3.2.5 Superscalar Processing

Superscalar processing is another technique used to achieve ILP. *Superscalar* merely refers to processing multiple simultaneous instructions per stage in the pipeline. That is, a superscalar processor decodes (and executes) at least two instructions per cycle. Many machines try to do four per cycle, and there are advocates of as many as sixteen or even thirty-two instructions per

cycle [16, 17]. With an N-stage pipeline running M-way superscalar, there can be as many as MN active instructions in the processor.

What superscalar processing also does is that it puts interlocked events (like branches and dependencies) closer together in the pipeline so that their resulting delays appear to be longer. And with a branch occurring every four instructions, a four-way superscalar processor should expect to encounter a branch instruction every cycle. That is, using the model described by Figure 3.2, a four-way superscalar processor should expect an N-cycle delay after every decode cycle.

A sixteen-way superscalar processor should expect to see four branches per cycle. This is philosophically disturbing, since the relevance of each of these branches (i.e., their very presence) depends on the outcomes of the predecessor branches - with which they are coincident in time.

It should be obvious that branch instructions severely inhibit the potential benefits of superscalar processing.

3.2.6 Multithreading

The last pervasive ILP technique is multithreading. In multithreaded processors, multiple independent instruction streams are processed concurrently [18]. There are many variations on the specifics of how this is done, but the general idea is to increase the utilization of the processor (especially given the ample delays present in any stream), and the aggregate processing rate of instructions, although each stream may run slower. Multithreading is mentioned in this context because it is the dual of superscalar - its general effect is to spread out the distance (in pipeline stages) between interlocked events.

3.2.7 Instruction Prefetching and Autonomy

In pipelined processors, instruction fetching can be made autonomous with respect to the rest of the machine. That is, instruction fetching can be made to drive itself, and to stage what is fetched into instruction buffers - which are (autonomously) consumed by the instruction decoder(s). This is sometimes called *instruction prefetching*. (Note that the word *prefetching* is used in many different contexts in a computer. Be sure to understand what it does and does not connote in each context.) As we will see later, enabling the autonomy of instruction prefetching is a precondition of achieving high performance [19].

If there were no branch instructions, then instruction prefetching would be exceedingly simple. It would merely involve fetching sequential blocks of instructions starting at the beginning of a program. This can be done by loading a *Prefetch Address* register with a starting address, and repeatedly fetching blocks of instructions, putting them into an instruction buffer, and incrementing the Prefetch Address register. This is shown in Figure 3.3.a. (A simple interlock is also needed that stops prefetching when the instruction buffer is full, and resumes prefetching when it is not.)

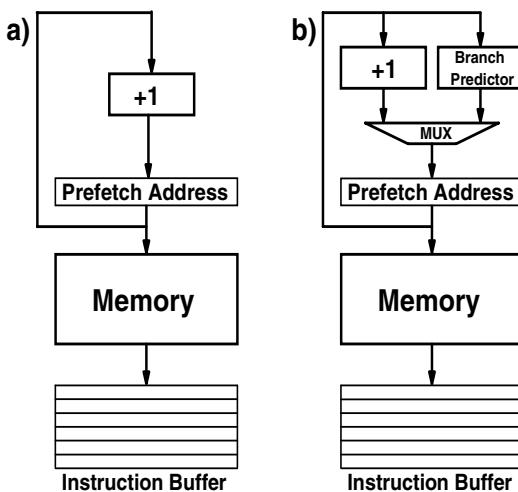


FIGURE 3.3: Instruction prefetching: a) Sequential prefetching in the absence of branches; and b) Augmenting the prefetching with a branch predictor.

Because there are branch instructions, this autonomy breaks down. When any instruction in the instruction buffer is found to be a taken branch instruction, then generally: 1) the instructions that were prefetched into the buffer following this instruction are useless, and 2) the instructions that are actually needed (the branch target instruction and its successors) have not been prefetched, and are not available for decoding following the branch. As already discussed, this causes a penalty proportional to the length of the pipeline. Thus, branches inhibit the ability to prefetch instructions.

The key to achieving high ILP is to preserve the autonomy of instruction prefetching. This requires that the branch instructions in the running program be anticipated, and be predicted correctly so that instruction prefetching can fetch the correct dynamic sequence of instructions to be executed [20, 21]. Figure 3.3.b shows the prefetching process augmented with a branch predictor which attempts to redirect the prefetching as appropriate.

In the remainder of this chapter, we will explore the dimensions and consequences of various branch prediction methods.

3.3 Three Dimensions of a Branch Instruction, and When to Predict Branches

As explained in the introduction, a branch instruction causes the instruction flow to be diverted to a non-sequential address in the program depending on the program and the system state. There are three dimensions to this diversion of flow:

1. Does the diversion occur, i.e., is the branch taken or not?
2. If the branch is taken, where does it go, i.e., what is the target address?
3. If the branch is taken, what is the point of departure, i.e., what should be the last instruction executed prior to taking the branch?

Most of the literature focuses on the first dimension - whether the branch is taken. In real machines, and depending on the *Instruction-Set Architecture* (ISA), it may be necessary to deal with the second dimension (target address) as well. This depends on *when* (in the pipeline flow) the branch is to be predicted. The predictor in Figure 3.3.b implicitly handles this dimension, since it shows the branch being predicted prior to its actually having been seen. (The branch is predicted at the same time that it is being prefetched.)

The third dimension is seldom discussed, since in all machines today, the point of departure is the branch instruction itself. Historically, there have been variations on branches in which the branch instruction is not the last instruction in the sequential sequence in which it acts. For example, the 801

processor - the first *Reduced Instruction Set Computer* (RISC) - has a *Delayed Branch* instruction [22, 12].

3.3.1 The Delayed Branch Instruction

The delayed branch instruction causes instruction flow to be diverted after the instruction that sequentially follows the delayed branch instruction. I.e., if the delayed branch is taken, then the next-sequential instruction (following the delayed branch) is executed prior to diverting the flow to the branch target instruction.

The 801 has a two-stage pipeline: Instruction Decode; and Execute. If an instruction is found that can be migrated (by the compiler) past the branch instruction, the delayed branch completely hides the delay associated with diverting the instruction fetching. What would have been an empty stage in the pipeline (following a normal branch) is filled by the execution of the migrated instruction as the branch target instruction is fetched.

In practice, the delayed branch has limited application, since statistically, most branch groups (the set of instructions following a branch and ending with the very next branch) are very short, and the ability to migrate instructions is limited. While this still provides some benefit in a two-stage pipeline, the delayed branch has fallen out of use as pipelines have become longer.

3.3.2 Branch Flow in a Pipeline: The *When* of Branch Prediction

Before discussing *how* to predict branches, it is important to understand *when* to predict them. Consider the pipeline abstraction shown in Figure 3.4.a. This is an RS-style pipeline, which is typical of IBM 360 machines [23]. In this pipeline, instructions are fetched sequentially, and put into an instruction buffer. Then they are decoded, and an address is generated by adding the contents of general-purpose registers.

For a branch instruction, the address generated is a branch-target address which is sent back to the beginning of the pipeline. For a load or store instruction, the address generated is an operand address, and an operand fetch is done.

Next, the instruction is executed. For branch instructions, execution is when the branch outcome (taken or not taken) is determined.

In this pipeline, the decode stage is the first stage to actually *see* the branch instruction. Prior to decode, instructions are merely unidentified data being moved through buffers. When the decoder first *sees* a branch instruction, what should it do? These are the various options and their consequences:

1. Assume that the branch is not taken. Leave the target address in an address buffer, but keep fetching and decoding down the sequential path. If the branch is not taken, no cycles are lost. If the branch is taken, this

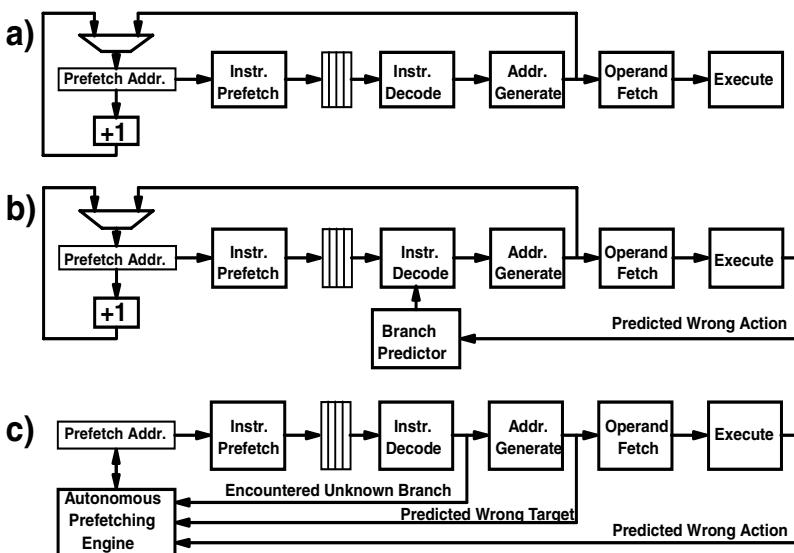


FIGURE 3.4: An RS-style pipeline with various approaches to branch prediction: a) Without branch prediction; b) With decode-time branch prediction; and c) With prefetch-time branch prediction.

is discovered at execution time, and the target instruction can be fetched subsequently, using the buffered target address. The penalty for being wrong is the entire latency of the pipeline.

2. Assume that the branch is taken. Save the sequential instructions that have already been fetched into the instruction buffer, but redirect fetching to the branch target address. If the branch is taken, then redirecting is the right thing to do, and the cost of the branch is the inherent latency of instruction fetch, decode, and address generation (i.e., the beginning of the pipeline). If the branch is not taken (which is discovered in execution), then it was a mistake to redirect the stream, and the prior sequential stream (which was saved in the instruction buffer) can resume decoding. The cost of being wrong is the latency of the pipeline not including the instruction fetch.
3. Do one of the above with *hedge fetching*. Specifically, follow one of the paths, but do some instruction-fetching down the other path to enable decoding to resume immediately if the guess is wrong [24, 25, 26, 27, 28, 29, 30]. In this case, we lose less when the guess is wrong, but we risk losing some cycles (due to sharing the fetch-bandwidth between the two paths) when the guess is right.
4. Fetch and execute down both paths [31, 32]. Again, we lose less when the guess is wrong, but we also lose performance unnecessarily when the guess is right.
5. Stop decoding. This is the simplest thing to do (because it does not require pipeline state to be subsequently invalidated). In this strategy, we concede a startup penalty (essentially, a delay until the branch executes) to every branch instruction, but the downside (the penalty for being wrong) can be smaller, and the design is simpler, since there cannot be wrong guesses.

Which of these strategies is best? It depends on:

1. The various penalties for being right, wrong, or partially right/wrong as described (e.g., for hedge fetching, following both paths, or stopping), and
2. The probability of being right or wrong in each case.

The penalties depend on the specific details of the pipeline - with larger penalties for deeper pipelines [33]. The interesting variable is that the probability of being right or wrong is not the same for all branches [34, 35, 36, 37, 38]. Many real machines have used combinations of all of the above strategies depending on the individual branch probabilities [39].

3.3.3 Predicting Branches at Decode Time

Figure 3.4.b shows a branch predictor that works in conjunction with the decoder. In this case, the *when* is at decode-time. These predictors can be very simple. A big advantage of predicting at decode time is that we know that the instruction is a branch because the decoding hardware is looking at it [40, 41, 42]. (This will not be the case for certain other predictors.) Also, generally these predictors do not have to predict a target address, since the processor will calculate the address in concert with the prediction.

All that these predictors need to do is to examine the branch instructions and guess whether they are taken. Optionally in addition, the predictors can make estimates as to the confidence in their guess for each given branch, and select one of the five courses of action listed above (calculated as a function of the penalties and the certainties of the guesses). The processor need not follow the same course of action for every predicted branch. After the branch instructions execute, their outcomes can be used to update information in the predictor.

Note that when predicting at decode time, there is always a penalty for taken branches, even when the prediction is right. This is because the target instruction has not yet been prefetched. The predictor does not know that a branch instruction is in the sequence until it is seen by the decoder. For this configuration, this is the earliest point at which there can be an inkling that instruction fetching needs to be redirected.

Therefore, when there is a taken branch, there will always be a penalty at least as large as the time required to fetch an instruction. Since the latency of a fixed-size SRAM array (where the instructions are stored) is fixed, as the pipeline granularity increases, the number of cycles for a fetch (hence the penalty for a taken branch) grows. As we said in the previous section, typical programs have taken branches every six instructions. Thus, branch delays fundamentally remain significant for decode-time predictors, even if the predictors are very accurate.

3.3.4 Predicting Branches at Instruction-Prefetch Time

To completely eliminate branch penalty requires that branch instructions be anticipated (i.e., before they are *seen* by decoding logic), and that branch target instructions be prefetched early enough to enable a seamless decoding of the instruction stream. Specifically, to eliminate all penalty requires that branches be predicted at the time that they are prefetched [20, 21]. This arrangement is shown in Figure 3.4.c.

Predicting branches at prefetch-time is much more difficult to do for two reasons. First, the instruction stream is not yet visible to the hardware, so there is no certainty that branches do or do not exist in the stream being prefetched. The prefetch mechanism must intuit their presence (or lack thereof) blindly.

Second, it is not sufficient to predict the branch action (taken or not taken). To eliminate all penalty requires that the target addresses be predicted for branches that are taken. The target address needs to be provided by the predictor so that prefetching can be redirected immediately, before the branch has actually been seen.

The increased complexity and difficulty of predicting at prefetch-time is evident in Figure 3.4.c. Note that there are now three ways in which the predictor can be wrong (shown as three feedback paths to the predictor in Figure 3.4.c) as opposed to just one way for decode-time predictors (shown as a single feedback path in Figure 3.4.b). The three ways are:

1. The predictor can be wrong about the presence of a branch, which will be discovered at decode time. That is, the predictor will not be aware of a branch that is in the stream, and the error will be discovered when the decoder encounters a branch instruction. Or, the predictor could indicate the presence of a taken branch in the stream, and the decoder could find that the instruction that was predicted to be a branch is not a branch instruction. (This will happen because of hashing collisions if full address tags are not used, and because of page overlays if the predictor uses virtual addresses.)
2. The predictor can be wrong about the branch target address for correctly predicted taken branches. This will be discovered at address-generation time if the processor generates a target address that is different than the predicted target address [43]. When this happens, it is usually because the branch instruction changes its target - as for a subroutine return. It can also be caused by hashing collisions and page overlays (as in the previous case) when the collision or overlay happens to have a taken branch in the same location as what it replaces.
3. The predictor can be wrong about whether the branch is taken - as for a decode-time predictor.

Thus for prefetch-time predictors, the number of ways of being wrong, and the number of recovery points, further fragment the various penalties involved. Note that these predictors can be wrong in more than one way, which can cause more complexity in the recovery processes. For example, the predictor could predict a branch to be taken, but predict the wrong target address. The processor can redirect the fetching following address-generation, and shortly thereafter (when the branch executes) discover that the branch isn't taken, which must trigger a second recovery action for the same branch.

The various complexities of these elements of prediction are discussed in the following sections.

3.4 How to Predict Whether a Branch is Taken

3.4.1 Static Branch Prediction

The very first predictors made *static guesses* based on the branch opcode, the condition specified, the registers used in the address calculation, and other static state [44, 45]. For example, unconditional branches are certainly taken. Looping branches (like *Branch on Count*) are usually taken, and can be predicted as such with high accuracy. These predictions were eventually augmented by static predictions made by compilers [46, 47, 48, 49]. This required more branch opcodes (to denote *likely taken* or *likely not-taken* branches). The compiler's guess was used in conjunction with a guess made by the hardware to determine the overall best guess and pipeline strategy to use for each branch instruction.

3.4.2 Dynamic Branch Prediction

Not long thereafter, branches were predicted using dynamic information of various sorts. Specifically, the past execution of branches was used in several different ways to predict the future actions of branches. The basic variations are:

1. Any branch is assumed to behave (be taken or not) in accordance with the behaviors of other recently executed branches, i.e., a *temporal average behavior* is assumed for branches.
2. Each specific branch (identified by its address) is assumed to do (to be taken or not) the same thing that it did the last time.
3. Each specific branch (identified by its address) is assumed to be taken or not based on some pattern of its own behavior exhibited in the past. This is an evolution of variation 2.
4. Each specific branch (identified by its address) is assumed to be based on its past actions, and on the actions of other recently executed branches. This is a mixture of all of the above methods.

The first variation is the simplest, since it does not require much state. It merely requires a few bits to record the actions of the last several branches via a state machine that generates predictions. The other three variations involve predicting actions for specific (by address) branches, which require tables to store past actions by branch address. The last variation requires both of these mechanisms.

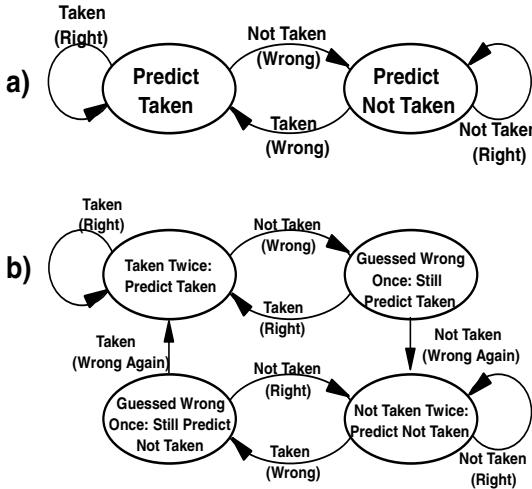


FIGURE 3.5: Simple state-machines for branch prediction: a) A 1-bit state-machine; and b) A 2-bit state-machine.

3.4.3 Branch Prediction with Counters

Figure 3.5 shows state-transition diagrams for two simple state-machines that track recent branch behavior in a program. The first (Figure 3.5.a.), is just a saturating one-bit counter. It records the action of the last branch (*Taken* or *Not Taken*), and it predicts that the next branch will do the same thing. The second (Figure 3.5.b) is only slightly more complex, and requires a two-bit up/down counter that saturates both high and low.

The state-transition diagrams for both figures are symmetric. Each state records the past history of branches (e.g., *Taken Twice*), and what the next prediction will be (e.g., *Predict Taken*). The state transitions (shown as arcs) are labeled with the next branch outcomes (*Taken* or *Not Taken*) and whether the prediction (from the originating state) is *Right* or *Wrong*.

What the state machine in Figure 3.5.b does is to stick with a particular guess (*Taken* or *Not Taken*) until it has been wrong twice in succession. This particular predictor first appeared in a CDC patent [50]. The CDC environment was predominantly a scientific computing environment in which many of the branches are used for loops. The basic idea is to not change the *Taken* prediction for a looping branch when it first falls through [51, 52, 53].

In addition, in that time period, flip-flops were very precious. Distinct branch instructions were allowed to alias to the same state-machine so as

to provide an *average group behavior* of the branches in a program. As mentioned, recording state transitions for individual branches requires tables (i.e., a record of state for each branch instruction).

3.4.4 Predicting by Profiling Branch Actions

There were a number of papers in the literature at that time that made obvious extensions to this (three-bits, four-bits, etc.). Instead of counting (which has intuitive basis), many of these devolved into what is essentially profiling [54, 55, 56, 57, 58, 59] [60, 61, 62, 63]. For example, if the state comprises ten bits, then there are 1024 possible sequences of branch actions, and the appropriate prediction for each sequence can be predetermined by profiling a trace tape. (E.g., when the sequence “1001101100” occurs in the trace, measure whether “1” or “0” occurs most often as the next branch action in the trace, and use this as the prediction.)

Each such prediction defies intuitive rationalization - it is just based on measurement. In some of these studies, the prediction was run (again) on the very same tape that was used to generate the profiles. While (quite unsurprisingly) the *guesses* get better as the number of bits is increased because some of the unusual patterns essentially identify specific branch occurrences (which certainly happens when the sequence only occurs once) in the profiling trace. What was not shown in these studies was whether the profiling statistics result in robust prediction. I.e., does it work well on a different execution trace?

Therefore going beyond two or three bits of state is probably not that useful in this particular context. More recently, it has become useful in a different context to be discussed later. There are even more exotic finite-state machines (using more bits) - including neural networks and Fourier analysis - that have been used to predict branch outcomes [64, 65, 66].

3.4.5 Group Behaviors vs. Predicting Individual Branches

Getting high predictive accuracy requires predicting the branches individually, and relying on the behavior of each individual branch rather than a group average. This is easy to understand, since some of the individual branches in a program tend to be taken almost always (e.g., looping branches), or almost never (e.g., exception handling), and the *group average* filters out this information.

Note that the worst possible predictive accuracy is 50%, because 50% indicates that there is no correlation between the prediction and the branch action. If the accuracy is <50%, and this is monitored, the predictor can be made to work at >50% accuracy by continuing to use the same algorithm, but inverting the prediction. Therefore, 50% accuracy should be a lower bound.

Maintaining state for each individual branch requires a table - with an entry for each branch. Since the state stored for a branch is small (e.g., one bit),

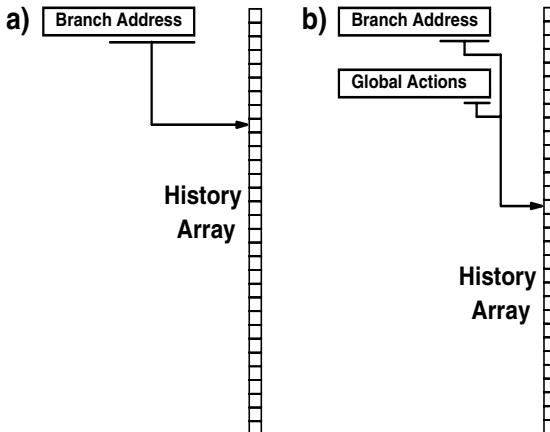


FIGURE 3.6: Decode-time branch predictors: a) Using the branch address; and b) Using the branch address together with the last global actions.

it is inefficient to make the table rigorously discriminant addresses by storing tags (the addresses). In practice, the table is implemented by performing a hash function on the branch address, and using the hash to address a bit vector. The most straightforward hash (requiring no circuitry) is truncation, the elimination of high-order bits.

3.4.6 The Decode History Table

Figure 3.6.a shows a simple predictor that is used in many real machines. This method was first disclosed as a *Decode History Table* (called a *DHT* in parts of the industry) because it worked at decode time [67, 68, 69], but has become to be known as a *Branch History Table* (BHT) in most of the literature today. In Figure 3.6.a, the low-order $\log(n)$ bits of the branch address are used to access the n -entry table that contains state information for the branches. Branches whose addresses share these low-order bits will hash to the same entry.

The smaller the table, the more hashing collisions will result, and the more potential for error because of this. Therefore, when there are a fixed number of bits to be used for prediction, it is always useful to assess whether the predictor benefits more from making the state more complex, or from having

more entries. (E.g., would you rather have n one-bit predictors, or $n/2$ two-bit predictors?) In commercial code where the working-sets are large, it is almost always better to have more entries [70, 71].

3.4.7 Discriminators

The branch address is a *discriminator* which uniquely identifies a particular branch instruction. A hash of that address is also a discriminator - with some aliasing. As the hashing becomes cruder, the discriminator does a poorer job of uniquely identifying branches [72, 73, 74, 75, 76]. (In the first predictor discussed, there was no hash function - all branches aliased to the same state machine.)

Recall that when we were discussing profiling using n -bit sequences, we had said that long unique strings were (essentially) uniquely identifying particular branch occurrences on a profiling tape. In this sense, a string of branch outcomes is also a discriminator for the next branch in the sequence, since it correlates to the path that was traversed (through the code) in reaching this branch [59]. The longer the discriminating string, the better it does at discriminating.

3.4.8 Multiple Discriminators: A Path-Based Approach

The fourth prevalent prediction technique (described in the opening paragraphs of this section) uses both discriminator types in tandem. That is, for each particular branch address (the first discriminator), the unique sequences of branch outcomes preceding this branch (the second discriminator) are recorded, and a prediction is made based on past history for this branch when it was preceded by the same sequence of global branch outcomes [77, 78, 79, 80, 81, 82, 83].

Conceptually, the rationale behind this is that some of the unique branches (by address) are not very predictable (e.g., they have accuracies closer to 50% than to 100%), although when those branches are reached by taking particular paths through the program, they are relatively predictable for each particular path. The first discriminator (branch address) identifies the branch, and the second discriminator (current sequence of global branch actions) correlates to the path being followed. When used in conjunction, the two discriminators achieve a higher accuracy than either could by itself.

3.4.9 Implementation

In principle, this could be implemented as a table of tables involving a sequence of two lookups. In real machines, the problem with cascades of lookups is that they take too much time. A branch predictor needs to provide a prediction in the cycle in which it must predict [84]. It does the machine little good to have a correct prediction that is later than this.

Thus, realizations of this kind of predictor are best made by hashing both discriminators together into a single lookup as shown in Figure 3.6.b. In this figure, the *Global Branch Actions* register records the actions of the last n branches as a sequence of ones and zeros. Whenever a branch executes, its action (taken or not) is shifted into one end of the register as a zero or one. The lookup into the history array is done by merging the low-order bits of the branch address with a portion of the Global Branch Actions register.

3.4.10 A Timing Caveat

A caveat with algorithms that use the *last branch actions* in real machines is that the *last actions* might not yet have occurred at the time that the prediction needs to be done. For example, if there is a branch every four instructions, and the pipeline between decode and execution is longer than this, then at decode-time for each branch, its immediately preceding branch can not have executed yet, hence the *last branch outcome* is unknown. A state register that tracks *last outcomes* (as in Figure 3.6.b) will be out of phase by one or more branches.

Therefore, a predictor that is designed based on studying statistics from an execution trace might not behave as well as anticipated in a real machine. In this case, the trace statistics will be based on the actual *last actions* of the branches, but the real machine will be using a different (phase-shifted) set of *last actions*. One way of dealing with this is to use the *Global Branch Predictions* (not shown, but obvious to conceive of) instead of the *Global Branch Actions*. This will align the phases correctly, although some of the bits may be wrong - which is likely academic.

3.4.11 Hybrid Predictors

Finally, there are compound (hybrid) prediction schemes. We had previously mentioned that hardware could monitor the predictive accuracy for a given branch, and change the prediction if appropriate. In that example, we had said that if the accuracy fell below 50%, we would invert the predictions from that point on.

This basic idea can be used in a more general way. Specifically, we can build multiple independent prediction mechanisms, and dynamically monitor the accuracies of each of them for each branch. We can achieve the best of all of them by using the monitored data to dynamically select the best predictor for each individual branch [35] [85] [86] [87] [88] [37] [89].

In a hybrid scenario, some of the branches are predicted by using one-bit counters, while others are predicted using two-bit counters, and others using global branch information. While this is not the most efficient use of bits, it achieves the highest accuracy - assuming that the working-set is adequately contained.

3.5 Predicting Branch Target Addresses

Branch target addresses are predicted at prefetch time so as to facilitate the redirection of instruction prefetching so as to eliminate all delay for correctly predicted taken branches. As mentioned previously, this drives additional complexity.

The simple part of target prediction is that the future target address of a branch is usually whatever the historical target address was. This must be recorded for each branch instruction.

In addition to whatever state is required to predict branch action, we now need to store a target address (e.g., thirty-two bits for a thirty-two-bit address space). Therefore, the amount of state for a branch prediction entry is much larger in this environment than what we discussed in the previous section. The point is that branch-target prediction done at prefetch-time is a much more costly proposition than branch-action prediction done at decode-time, although doing it well is essential to achieving high performance [20] [21].

3.5.1 Instruction Prefetching

Since we predict target addresses to facilitate the redirection of instruction prefetching, it is necessary to understand what we mean by *instruction prefetching*. In the first place, the quanta being fetched are not instructions. They are generally some larger fixed data-width that can be accommodated by the busses and buffers, and that provide ample instruction prefetching bandwidth.

In many machines, *instruction prefetching* is the act of prefetching quadwords (sixteen bytes) or double-quadwords (thirty-two bytes) aligned on integral boundaries (sixteen-byte boundaries for quadwords, and thirty-two-byte boundaries for double-quadwords) [90] [91] [92] [93]. For the remainder of this discussion, we will use quadwords as the unit of fetching, although the discussion easily generalizes to other widths.

The quadwords that are fetched contain multiple instructions, one or more of which can be a branch. For example, if the length of an instruction is a word (four bytes), then a quadword contains four instructions, and (as many as) four of them can be branches. For variable-length instruction-sets, a quadword need not contain an integer number of instructions, and some instructions (including branches) will span quadword boundaries.

Therefore, two halves of a single branch instruction can reside in consecutive quadwords. A taken branch can leave the middle of a quadword, and can branch into the middle or end of a different quadword. Or a branch can branch into the quadword containing itself. These realities drive further complexity.

3.5.2 The Branch History Table

Basically, a prefetch-time predictor is a table of entries, where each entry corresponds to a specific branch instruction, and contains the address of the branch-target instruction and some other state information. The table must be organized to be compatible with the instruction prefetch width (e.g., a quadword) so that it can be searched on behalf of each instruction prefetch.

The other state information in an entry includes state to predict whether the branch is taken, state having to do with the organization of the table itself, and state that enables the hit-logic to find the right branches within the quadwords.

Since we are now putting more at stake by redirecting instruction prefetching, it is more important to discriminate branches correctly. Therefore instead of merely hashing entries, branch tags are generally used. (But since the entry already needs to hold a full target address, the overhead for a tag is relatively less than it was in the previous section.) Since a quadword can contain multiple branches, the table should be set-associative so as to avoid thrashing between branches contained in the same quadword.

When originally disclosed, this was called a *Branch History Table* (BHT) [94, 95, 96, 97, 91, 98, 99, 100, 101], but has come to be called a *Branch Target Buffer* (BTB) by many [102, 103, 104]. Recall that the recent literature uses *Branch History Table* (BHT) to denote the table of prediction bits described in the previous section - which was originally called a Decode History Table (DHT). This can sometimes lead to confusion in discussions, so one must always take pains to make the context clear.

A BTB is sketched in Figure 3.7.a, and the fields within a BTB entry are depicted in Figure 3.7.b. The fields shown are:

1. Valid - A bit that indicates that this is a real entry, and not uninitiated state.
2. Action - The state bits that are used to predict whether the branch is taken. This corresponds to the state described in the previous section. (In some implementations, the Valid bit is interpreted directly as an indication that the branch is taken, and there is no explicit Action field.)
3. Tag - The high-order address bits of the branch instruction (used by the hit-logic to match an entry to the correct quadword).
4. Offset - The low-order address bits of the branch instruction (the byte address within the quadword) that is used by the hit-logic to locate the branch within the quadword.
5. Target Address - The address of the branch-target instruction.
6. Length - A bit used to indicate whether the branch instruction spills into the next quadword. If it does, then the next-sequential quadword

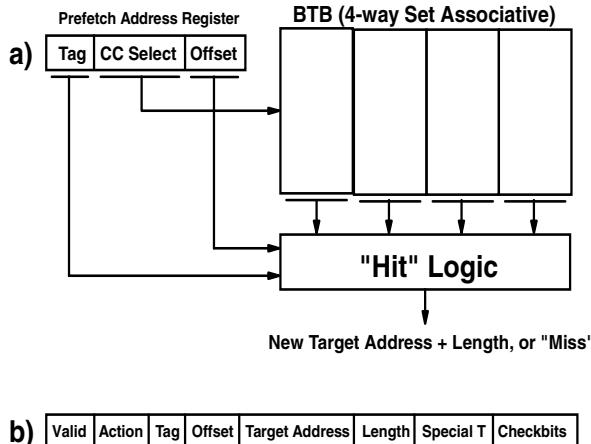


FIGURE 3.7: The branch target buffer (nee, *branch history table*): a) The BTB structure and lookup; and b) The field within a BTB entry.

needs to be fetched prior to redirecting fetching to the target address, or the processor will not be able to decode the branch instruction.

7. Special T - A field to indicate how the Target Address field is to be interpreted for this entry. In some implementations, the penalty (in complexity) can be severe for predicting the target address incorrectly. Special T can simply be used to alert the predictor not to trust the Target Address field, and to tell the processor to stop decoding when it encounters this branch, pending verification of the target address by the address-generation stage in the pipeline. Or it can be used more elaborately to create hints about branches that change their target address. This is discussed in the next section.
8. Check Bits - Parity or other check bits to detect array errors - these would generally be in any real product.

3.5.3 Operation of the BTB

The BTB works as follows. The Prefetch Address (shown in Figures 3.3 and 3.4) is used to search the BTB at the same time that the corresponding quadword of instructions is prefetched. In particular, the BTB can be the

Autonomous Prefetch Engine shown in Figure 3.4.c. Referring now to Figure 3.7.a, the low order four bits of the prefetch address (the byte within quadword) are sent to the hit-logic. The next set of bits (the log of the number of rows in the BTB) is used to select a congruence class within the BTB. In this case, the BTB is shown as a four-way set-associative table, so four candidate entries are read out. The remaining high-order bits of the prefetch address are sent to the hit-logic for a tag comparison with each of the four candidate entries.

The hit logic determines the presence of a branch within a quadword by doing the following four evaluations on the entries within the selected congruence class of the BTB:

1. If the entry does not have the Valid bit set, then it is removed from consideration.
2. If the Tag field in the entry does not match the high-order bits of the prefetch address (which comprise the Tag portion of the address shown in Figure 3.7.a), then the entry denotes a branch in a different quadword, and it is removed from consideration.
3. If the Offset field in the entry is less than the low order bits (byte within quadword) of the prefetch address (see Figure 3.7.a), then the entry denotes a branch that resides in a lower part of the quadword than the point at which instruction flow enters the quadword, and it is removed from consideration. For example, if flow enters this quadword at the end of the quadword (due to a branch), then the processor will not encounter the branches that are at the beginning of the quadword.
4. If more than one entry passes the first three tests, then we choose the first entry (the one with the smallest Offset field) that will be predicted as taken (by the Action field). The taken branch with the smallest Offset (greater than the Offset in the Prefetch Address Register - as determined in Step 3) is the first taken branch that will be encountered in the flow. This is the *hit* that is sought.

If a *hit* is encountered, then instruction prefetching will be redirected to the address specified in the Target Address field of the entry that caused the hit. Note that if the corresponding Length field indicates that the branch spills into the next quadword, then the next-sequential quadword is prefetched prior to redirecting the prefetching to the target address. (Care must be taken to suppress BTB results during the prefetching of the next-sequential quadword in this case.)

If no *hit* is encountered, then prefetching (and BTB searching) continues with the next-sequential quadword.

3.5.4 Fetch Width and Branch Mispredictions

We have stated that the BTB search must be done on the same quantum boundaries as the instruction prefetching. This is not an absolute requirement, but the BTB should (at least) keep up with the prefetching, or the prefetching cannot be redirected in a timely manner. It is equally acceptable to organize the BTB on double-quadwords if the instruction fetching is on quadwords. This may actually be preferable, because it allows sufficient BTB bandwidth to enable timely updates. This is discussed later.

As mentioned in a previous section, we can predict the target incorrectly for three reasons:

1. If full address-tags are not used, we can hit on a coincident entry from an aliased quadword which generally will have a different target address.
2. Since we are not doing address translation prior to the BTB search, we are searching with a virtual address. When the operating system performs page overlays, we may (coincidentally) have an entry for a branch at a location that also contains a branch in the new page, but the target address will likely be different.
3. Some branches really do change their target addresses. This is discussed in the next section.

3.6 The Subroutine Return - A Branch That Changes Its Target Address

3.6.1 The Subroutine Call and Return Structure

Most branches in a program always branch to the same target instruction. A canonical exception to this is the branch used for returning from a subroutine. A subroutine is a generic procedure that can be called from different places in programs. When a subroutine is called from procedure A, upon completion it should return to procedure A (typically, at the instruction immediately following the calling instruction). When it is called from procedure B, it should return to procedure B. Figure 3.8.a shows this structure.

A problem arises in prefetch-time predictors (BTBs) that treat the subroutine return as any other branch. While some *Instruction-Set Architectures* (ISAs) have explicit Call and Return instructions - and the subroutine return can be recognized by the decoding hardware (and tagged as such in the Special T field of the BTB), other ISAs implement these primitives with standard branch instructions, and the decoding hardware is oblivious to the call-return primitive.

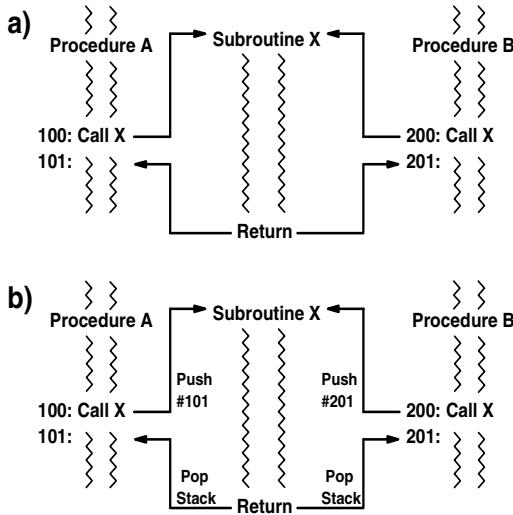


FIGURE 3.8: The subroutine call and return: a) The call-and-return structure; and b) The same structure annotated with stack operations.

When the BTB is unaware that a particular branch is a subroutine return branch, it always predicts the target address to be whatever it was last time (the historical return point). When a subroutine last returns to procedure A, it remembers the return point in A. When the subroutine is next called from procedure B, the BTB predicts the returning branch as returning to procedure A. This is an incorrect prediction, as we know that the correct return point is in B.

3.6.2 Predicting Return Addresses by Using a Stack

In this section, we explain how the branch predictor is made sensitive to the call-return primitive so as to correctly predict subroutine return branches at prefetch time [105, 106, 107, 90, 108, 109, 110, 111, 112]. First, so as to make the explanation as simple as possible, we explain how to do this in the context of an ISA having explicit Call and Return instructions, and we augment the prefetch-time predictor to include a stack.

In this scenario, whenever the decoder encounters a Call instruction, the address of the instruction that sequentially follows the Call instruction (i.e., the return point) is pushed onto the stack in the predictor hardware. When a Return instruction is first encountered, it is unknown to the BTB, and it is

mispredicted. When a BTB entry is created for the Return instruction, the *Special T* field is set to indicate that this is a Return instruction.

Now in future predictions, whenever the BTB encounters a hit in which the *Special T* field is set to indicate a subroutine return, the Target Address from the BTB entry is not used. Instead, the address on the top of the stack is used as the return point. Figure 3.8.b is annotated with the corresponding stack actions to illustrate this behavior.

3.6.3 Recognizing Subroutine Calls and Returns

Now, consider ISAs that do not have explicit Call or Return instructions. The question is how to recognize that a particular branch instruction is implementing a subroutine return? Although ISAs may not have an explicit Call instruction, they generally have a particular kind of branch that tends to be used for subroutine calls.

For example, the Branch And Link (BAL) is a branch that stores the address of its successor instruction (the return point) into a general purpose register as it branches. While being used for subroutine calls, it can also be used for other things (e.g., it performs the *load context* primitive, which enables a program to know its current location in storage).

If there is no explicit Return instruction, then we need to use clues that a branch instruction is a subroutine return instruction. There are a few such clues:

1. If the BTB predicts the target address incorrectly, the corresponding branch could be a subroutine return.
2. If the target address of a branch is specified as being the contents of the same register used by a preceding BAL instruction, then this branch could be a subroutine return.
3. If the target address of the branch happens to be an address that immediately follows the address in which a BAL is stored (which should be on top of the call-stack), then the branch could be a subroutine return.

A predictor can be made sensitive to one or all of these things. The first clue is already provided for us in the existing structure - we just need to observe that the BTB makes a wrong target prediction. The second clue requires more hardware to keep track of specific register usage. The third clue is the most interesting, because it allows us to use a trick that is an artifact of how the BTB is organized.

3.6.4 Taking Advantage of the BTB Structure

Recall that instruction prefetching is performed on either a quadword or double-quadword basis, and that the BTB is organized around this width (or

around an even larger width). When a branch target is mispredicted, and the processor recovers and redirects prefetching to the correct target address (Clue 1), the first thing that is done is that the BTB is searched to see if there are any branches at the new target address.

While the hit logic is looking for branches at or after the new target address (the entry point into the target quadword), it sees entries for *all* branches within the target quadword, i.e., it sees the branches that are located before the entry point into the quadword (unless the entry point is 0 - the beginning of the quadword).

Within this context, we can do the following things:

1. Whenever we encounter a branch that could be a subroutine call (e.g., a BAL), in addition to pushing the next instruction address onto the stack, when we create the BTB entry for the branch, we indicate *Probable Calling Branch* in the Special T field for the entry.
2. Whenever we have a target address misprediction, and redirect prefetching to a new address, then when we do the BTB lookup in the new quadword, we notice whether there is a *Probable Calling Branch* immediately prior to our entry point into the quadword. If there is, we assume that the branch that was just mispredicted is a subroutine return, and we indicate *Subroutine Return* in its Special T field.
3. Whenever we get a BTB hit with *Subroutine Return* indicated in the Special T field, we use the address on the top of the stack instead of the address in the Target Address field of the BTB.

3.6.5 Eliminating the Stack

Finally, the algorithm described above can be implemented directly in the BTB without the use of a stack [90]. This requires a new Special T state that indicates that an entry is a subroutine entry point. This works as follows:

1. Whenever we encounter a branch that could be a subroutine call (e.g., a BAL), then when we create the BTB entry for the branch, we indicate *Probable Calling Branch* in the Special T field for the entry.
2. Whenever we encounter a target address misprediction, and we redirect prefetching to a new address, then when we do the BTB lookup in the new quadword, we notice whether there is a *Probable Calling Branch* immediately prior to the entry point into the quadword. If there is, we assume that the branch that was just mispredicted is a subroutine return, and we indicate *Subroutine Return* in its Special T field.
 - (a) In addition, the Target Address field of the Probable Calling Branch in the target quadword specifies the address of the subroutine entry point (i.e., the beginning of the subroutine), and we know the

address of the returning branch (it is the address of the branch that was just mispredicted - for which we are currently doing the BTB search).

- (b) Create a new entry in the BTB at the subroutine entry point. We set its Special T field to indicate *Subroutine Entry Point*, and we set its Target Address field equal to the address of the returning branch; i.e., a Subroutine Entry Point entry does not indicate the presence of a branch instruction. It is merely a pointer to the returning branch - the branch for which we are now doing the search.
3. Whenever we get a BTB hit with *Subroutine Entry Point* indicated in the Special T field, we do not redirect prefetching, since this is not a branch. However, we do a BTB update. Specifically, we take the Target Address field (which points to the returning branch), and we update the entry corresponding to the returning branch. Specifically, we change its Target Address field to the address of the instruction that sequentially succeeds the branch that generated the current BTB hit, i.e., we set it to point to the sequential successor of the calling branch. (Remember that the branch that generated this hit is the calling branch.)
 4. Whenever we get a BTB hit with *Subroutine Return* indicated in the Special T field, its Target Address should be correct, since it was updated to the correct return point in Step 3.

This algorithm will work for subroutines having one or more entry points, and a single return point. If there are multiple return points, this algorithm can have problems unless each entry point correlates to a specific return point.

3.7 Putting It All Together

A number of points are made in this section about the operating environment (the context) of a predictor relative to the size of the instruction working-set, and of the statistical accuracy of predicting branches. Considering these things motivates (or not) the use of even more exotic mechanisms, and it relates branch prediction to instruction prefetching.

3.7.1 Working Sets and Contexts

We have said that (roughly) one in four instructions is a branch. While this was a statement about the dynamic frequency of branches, it is also roughly true of their static frequency. Since the job of the BTB is to remember the

branches in a program, the BTB is attempting to remember the same context as the instruction cache [113]. Therefore, it is interesting to consider which of these (the BTB or the instruction cache) remembers more context, and whether one of them can prefetch for the other.

If the average instruction size is one word (four bytes), and there is one branch per four instructions, then on the average, each quadword contains one branch. (This is primarily why we have discussed instruction prefetching in units of quadwords.) Therefore, a BTB that contains N branch entries holds (roughly) the same context as an instruction cache having N quadwords. To compare these two (in bytes) requires an estimate of the size of a BTB entry.

3.7.2 The Size of a BTB Entry

For the sake of putting down some approximate numbers, let's assume that we are working with 32-bit addresses, and that we are using full address tags. Next, we approximate the size of an entry in a 4K-entry, four-way set-associative BTB. Recall that an entry can have eight fields. With these assumptions, the size of each field is:

1. Valid - one bit.
2. Action - one or two bits.
3. Tag - If there are 4K entries and four sets, then ten bits are used for the congruence class selection (i.e., not needed in the tag). Since the tag discriminates at the quadword granularity, the low-order four bits of the address are not part of the tag. Thus, for a branch address, the tag is $32-10-4 =$ eighteen bits.
4. Offset - The low-order branch-address bits (the byte address within the quadword). For a quadword, this is four bits.
5. Target Address - The address of the target instruction - thirty-two bits.
6. Length - one bit.
7. Special T - one or two bits.
8. Check Bits - between zero and eight bits.

The total is in the range of 58-68 bits. So that we can work with round numbers, we'll call this eight bytes, which is half of a quadword.

Therefore, for an N -entry BTB to hold the same context as an N -quadword cache, it requires half the capacity of the cache in bytes. In practice, it would be unusual to make a BTB this large. In the first place, it is desirable to be able to cycle the BTB quickly - so that on a hit, the very next lookup can be done on the target address provided by the hit. (When the BTB cannot cycle this fast, one can then explore whether it is worthwhile to do *hedge-BTB*)

lookups in the event that a hit turns out to have been wrong.) Therefore, BTBs typically do not remember as much context as the instruction cache.

3.7.3 The BTB and the Instruction Cache: Economies of Size

If a BTB remembered *more* than the instruction cache, then running the BTB well ahead of the instruction stream would provide a vehicle for prefetching for the instruction cache [114, 115, 116, 117]. Since implementations usually have these relative sizes reversed, it is possible to consider multilevel BTBs (i.e., a hierarchy of pageable BTBs), and to use the cache miss process as a trigger to prefetch BTB information down the (posited) BTB hierarchy [118, 119, 120].

We should mention that it is possible to economize on the size of BTB entries by assuming that target addresses are physically proximate to branch addresses, which is usually true. If we assume that a target instruction will fall within one Kilobyte of the branch instruction, then we can shorten the Target Address field to eleven bits. We can also eliminate many of the high-order bits from the Tag without losing much [121, 122, 123]. There have been a number of proposals like this, which can cut the size of an entry appreciably without losing much accuracy.

3.7.4 More Exotic Prediction for the More Difficult Branches

Since we have demonstrated that a BTB entry already requires lots of storage (eight bytes), it is then reasonable (in a relative sense) to add even more information for some of the branches if that will improve the predictive accuracy further. Most of the branches in a program are very predictable (e.g., 99+% accurate), and the BTB will do just fine by remembering the last action and target address for these. In real commercial code, BTBs will be found to have accuracies in the 75-90% range.

Suppose that we achieve an accuracy of 90%. It is not the case that each branch in the program is 90% predictable. More likely, we will find that 80% of the branches in the program are nearly 100% predictable, and 20% (or so) are predicted with very poor accuracy (50% or so). This tells us that the way in which we are trying to predict this 20% does not work for these particular branches. Then increasing our aggregate accuracy from 90% to a much higher number requires doing something even more exotic for this (relatively small) subset of the branches.

On a side note, since it is a relatively small percentage of the branches that are apparently intractable, a very small cache can be used to store the alternate paths for only these branches. Since the cache is not used to store the alternate paths of the predictable branches, it can be very small, hence fast. Such a cache enables quicker recoveries when these intractable branches are mispredicted [124, 125, 126].

3.7.5 Branches and the Operand Space

On the other hand, a branch outcome is not a random event. It is the result of a calculation or a test on the state of an operand [127, 128, 129, 130, 131]. Many of the least-predictable branches depend of the test of a single byte, and frequently the state of that byte is set (by a store) well ahead of the test (e.g., hundreds of cycles). In addition, it is sometimes possible to predict the values of the operands themselves [132, 133].

There have been proposals for additional tables that maintain entries (indexed by the addresses of the relevant test-operands) that contain a pointer to the branch instruction affected by the test-operand, the manner in which that test-operand is tested to determine the outcome of the branch, and the previous outcome of that branch [127, 128, 134].

When the address of a store operand hits in this table, the operand being stored is tested (by the prediction hardware) in the manner described in the table, and if the new (determined) branch outcome differs from the historical outcome, an update is sent to the affected branch entry in the BTB. If the store occurs enough ahead of the instruction prefetch (which is typically the case), then the BTB will make a correct prediction.

3.7.6 Branches and the Operand-Address Space

There is another subset of unpredictable branches that cannot be predicted in this manner because the operand that is tested is not always fetched from the same address. The test instruction that determines the branch outcome may test different operands at different times. For many of these instructions, there is a strong correlation between the operand that is tested and the branch outcome. (Generally, the operands being tested are statically defined constants.)

When this is the case, the branch is very predictable once it is known which of the operands is to be tested. Since the branch outcome (taken or not taken) depends on the branch itself (discriminated by the branch address) and on the operand that is tested (discriminated by the operand address), this method is called an *Instruction Cross Data* (IXD) method.

Unfortunately, the correct operand address is not generally known until the test instruction (which immediately precedes the branch instruction) performs address generation (i.e., this instruction generates the operand address that we need to know). This is too late to influence the instruction prefetching, so this technique is more applicable at decode-time [135].

Since this table is used at decode time to predict branch action, its entry is small (one bit). The table is indexed by a hashing of the instruction address with the operand address. It looks very much like predictor shown in Figure 3.6.b, except that an operand address is used instead of the *Global Actions* that are shown in the figure.

3.7.7 Tandem Branch Prediction

In real machines, since prefetch-time mechanisms are costly, and (therefore) cannot remember much context, and since decode-time mechanisms are simple, and can remember lots of context, and since the two operate at different points in the pipeline, we generally use both mechanisms in tandem [136] [137] [138] [139] [70].

For example, with eight bytes per entry, an eight Kilobyte BTB remembers one Kilobranches, which corresponds to a sixteen Kilobyte working set. With one bit per entry, an eight Kilobyte BHT remembers sixty-four Kilobranches, which corresponds to a one Megabyte working set.

It will frequently be the case that the instruction decoder encounters branches in the instruction stream that the BTB did not know were there (this is not a distinguishable event if the Valid bit is used as a Taken indicator - hence the use of these distinct fields). Then, although the instruction prefetching had not been redirected by the BTB, it is at least still possible to predict the branch as *taken* (if it is) at decode time if we are also using a BHT (a.k.a., DHT). This allows us to mitigate some of the penalty associated with the BTB miss (assuming that the branch is taken).

Thus, in a real machine, there is nothing as simple as *the* branch predictor. There will typically be several mechanisms at work (both independently and in tandem) at different points in the pipeline. There will also be different points at which recovery is done (based on branch presence or target or action becoming known, or based on a more reliable predictor supplanting the guess made by a less reliable predictor). There will also be different opportunities to hedge (do BTB searches and instruction prefetches) down alternate paths, depending on the bandwidth available, and the complexity of the buffering that we will want to deal with.

Predicting a branch is not a single event. It is an ongoing process as branch information makes its way through a pipeline. Although quoting some theoretical accuracy (derived from an execution trace), and some nominal penalty for a wrong guess is helpful in getting a rough idea about branch performance, simulation in a cycle-accurate timer (which includes modeling all table lookups and updates explicitly) is needed for an accurate picture of processor performance [140].

3.7.8 Accuracy and the Updating of Tables

As we have already mentioned, *accuracy* as derived from an execution trace can be wrong when prediction mechanisms depend on *last* branch action, because in a real machine, the *last* action has not yet happened at the time that the prediction is made.

Another crucial aspect of prediction that we have not yet mentioned is whether the required updates (to our various tables) have been able to occur by the time that they are needed. When doing analysis from trace tapes,

it is generally assumed that the updates magically get done, and that if the information in the table is correct, that the branch will be predicted correctly in a real machine. This is not always the case [141].

For example, we have seen comparative simulations in which higher performance is achieved by making the caches smaller. Why this occurs is as follows. If the BTB search has priority over the BTB update, then in branch-laden code, the BTB is constantly being searched, and the updates to it do not get done. Because the BTB does not get updated, the branches are repeatedly guessed wrong (although in an execution trace analysis, they would be counted as *right*, since the updates should have occurred). When the cache size is reduced, cache misses cause the processor to stall, and when the instruction prefetching stops, the BTB updates get made. Then, although the processor takes an occasional cache miss, the branches get predicted correctly, and the overall effect is that the processor runs faster.

Therefore, it is important to consider how updates to the BTB are prioritized relative to searches. It is also important to make sure that there is adequate update bandwidth. (This is why we had mentioned that it can be advantageous to organize a BTB on double-quadwords when the instruction prefetch is a quadword.)

3.7.9 Predictor Bandwidth and Anomalous Behaviors

Another phenomenon worth mentioning is that because of the dynamics of searching and updating a BTB simultaneously, in some configurations (meaning the read-write priorities on a finite number of ports) it is possible for a running program to create multiple entries for the same branch instruction with predictions that disagree. After this happens, the hit-logic may encounter coincident hits with opposite predictions for a given branch.

Some prediction algorithms require much more update bandwidth than others, and this needs to be taken into account when comparing them. For example, an algorithm that requires knowing the last N actions for a specific branch requires updating the table each time the branch executes. A simpler algorithm may only require an update when the action changes (from taken to not taken).

An execution-trace analysis may show that the first algorithm achieves higher accuracy. In a running machine, the update bandwidth may interfere with the ability to search the table in a timely way, and the simpler algorithm - although less accurate - may achieve higher performance. Therefore, better accuracy does not necessarily imply better performance.

3.7.10 The Importance of Fast Prediction Mechanisms

In the section on predicting branch action, we had mentioned that predictors that take more than a cycle (such as those involving two-stage lookups, or for large BTBs) are problematic. At that time, we had not yet gotten into

the details of instruction prefetching, and had just said that it was not useful to get a correct prediction too late. In fact, this is particularly damaging for prefetch-time predictors.

When a predictor takes more than a single cycle to predict, the prediction must be done out of phase (and ahead of) instruction prefetching, otherwise it cannot drive the prefetching correctly. When the first wrong prediction occurs, prefetching is redirected to the correct target address (as determined by the processor), and a quadword of instructions is prefetched. If no prediction is available by the end of the next cycle (and none will be, because we are assuming that the predictor takes more than a single cycle), then instruction prefetching has no choice but to fetch the next sequential quadword blindly.

It is very likely that each quadword of instructions contains a branch. Since the prefetch mechanism cannot respond within a cycle, instruction prefetching is always done blindly following a wrong prediction (and if at a quadword per cycle rate, usually incorrectly) until a branch prediction (pertaining to a branch in the first quadword that was prefetched) becomes available. Since the prediction comes later than when its associated prefetch should have been initiated, the prediction is (*de facto*) *wrong* even when it is factually correct, because it came too late to redirect the prefetching to the correct target address.

This will trigger another recovery action, and for the same reason, the next branch prediction is also (likely) missed. Hence, because of timing, a single wrong prediction can snowball into a sequence of *wrong* predictions, even when the predictor has the correct information about all branches subsequent to one that is first guessed wrong.

It is very important to keep the timing of the prediction mechanism to one cycle. In high-frequency designs where the pipeline stages are very lean, this can be problematic, since tables may have to be kept very small, hence inherently inaccurate.

3.8 Very High ILP Environments

Throughout the 1990s, processor complexity was increased in an effort to achieve very high levels of *Instruction Level Parallelism* (ILP). Doing this requires decoding and executing multiple instructions per cycle.

3.8.1 Superscalar Processing and the Monolithic Prediction of Branch Sequences

In the late 1970s, the first *Reduced Instruction Set Computer* (RISC) machine, the 801 [12], decoded and executed a very specific four instructions

per cycle. By doing a fixed-point operation (to manipulate loop indices), a load (or a store), a floating-point operation, and a branch (e.g., a loop), the 801 could do vector processing in superscalar mode. When running this kind of program at full speed, the 801 encounters a branch every cycle.

By the mid 1990s, people were talking about eight-at-a-time superscalar machines, and by the end of the 1990s the ante was up to sixteen-at-a-time, and even more [16]. With a branch occurring every four instructions, decoding more than four instructions per cycle requires predicting more than one branch per cycle [142] [143] [144] [145] [146] [147] [148] [17], as well as doing multiple disjoint instruction prefetches per cycle [142] [143] [149]. The complexity of this is considerable.

Basically, this requires that BTB entries contain monolithic information about sequences of branches (including multiple <branch address; target address> pairs), and that it generate multiple instruction prefetches (at each of the predicted target addresses) each cycle. If the amount of information in a BTB grows too large (say it becomes a quadword - the same size as the instruction sequence it is predicting), then one needs to question the efficacy of using a BTB (and very high ILP techniques) against simply making the caches larger. In this regime, ILP techniques like trace-caches start to make a lot more sense [150].

3.8.2 Predicting Branches in a Multithreaded Environment

Currently, the quest for higher ILP has abated as the efficient use of power has become paramount [15], and modern designs are becoming much more energy-conscious. At the same time, multithreading is a growing trend, and this requires multiple disjoint instruction streams in flight simultaneously.

Multithreading has some of the same aspects as very wide-issue superscalar. It requires doing BTB (if it is used) lookups on behalf of multiple instruction streams. While these lookups might be simultaneous (requiring multiple banks or ports) or not, there is another aspect to this that is problematic. If the instruction streams are not disjoint, they nonetheless can operate on different data, and the same branch (in two streams, say) may have different outcomes. Thus, each stream will damage the predictive accuracy of the other streams.

If the instruction streams are disjoint, since BTB lookups are done with virtual addresses, the BTB (as it is) cannot distinguish the streams, so it can provide one stream with (erroneous) branch predictions that are generated by different streams. (E.g., the BTB will find branches that don't exist.) This can be fixed by adding *stream tags* to the BTB. However, since the required number of branch entries per virtual quadword is roughly equal to the number of streams, the set-associativity needs to be increased accordingly, or thrashing will greatly reduce the accuracy of predictions - even with stream tags.

Although a highly set-associate (hence more complicated) BTB would be able to discriminate between streams because of the tagging, the overhead of doing this is less reasonable in a BHT (a.k.a., DHT), where an entry is 1 bit.

Hence the likely use of a BHT is to allow all of the streams to update and use the BHT in mutual collision. This will tend to randomize the predictions, and reduce their accuracy. The alternative is to use stream tags (essentially, to have a separate BHT for each stream). While this works, remember that for a fixed number of bits, dividing them among the streams means that each stream will remember much less context - hence have a lower predictive accuracy.

3.8.3 Limitations

It costs a lot to run fast. And many of the mechanisms required to do this start breaking down as certain thresholds are exceeded. We had made the case in Section 3.2 that ILP is very fundamentally limited by the branches. Many of those limitations in high ILP environments and in highly pipelined machines are due to complicated second-order effects which go well beyond the scope of Minsky's conjecture.

Running fast requires resolving sequences of branches as quickly as possible so as to facilitate an ample supply of instructions to the execution end of the pipeline. There have been (and continue to be) proposals for *runahead threads* which attempt to run the branch portion of the instruction stream speculatively ahead of the canonical instruction processing so as to resolve the branches early [7] [8] [151] [152] [10] [153].

3.9 Summary

The branch instruction is the workhorse of modern computing. Processor performance is fundamentally limited by the behaviors of the branch instructions in programs. Achieving high performance requires preserving the autonomy of instruction prefetching [19], which requires that the branch instructions be accurately anticipated, and predicted correctly.

3.9.1 Simplicity

Conceptually, predicting branches is simple. The vast majority of branches usually do exactly the same thing each time they are executed. Predictors based on this principle are fairly straightforward to implement, and do a reasonably good job. To do an even better job of predicting branches requires much more complexity. Also, the details of implementation in a real machine can make the realities of branch prediction more complex still. We touched on some of those details in this chapter.

3.9.2 Complexity

Because of the complexity of operations within a real pipeline, we emphasized that getting a true picture of the efficacy of branch prediction requires simulating it in a cycle-accurate timer model in which the details of the prediction mechanism (including updating it) are modeled explicitly [140]. We also tried to make clear that in a real machine, there is no such thing as *the* prediction mechanism. Branch prediction involves interaction between the states of several mechanisms (including the processor itself) at different stages in the pipeline.

Since there are several ways in which a prediction can be wrong, and these become manifest in different stages of the pipeline, there are many different penalties involved. For this reason, it is not particularly meaningful to talk about *the accuracy* of an algorithm. *The accuracy* usually refers to some abstract analysis performed on an execution trace. This number does not necessarily transfer to a real machine in a direct way.

3.9.3 Two Saving Graces

While the complexity and the details of branch prediction can appear daunting, there are two saving graces. First, simple algorithms work pretty well. Many complex algorithms can outsmart themselves when actually implemented.

Second, a prediction is just that: a prediction. It does not *have* to be right. In this sense, predicting branches is simpler than operating the rest of the machine. In our algorithms, we can make approximations, take shortcuts, and shave-off lots of bits, and still do pretty well. The things that we are predicting (branches) are capricious - there is no point in obsessing excessively over predicting something that has inherent unpredictability.

3.9.4 Implementing Real Branch Prediction Mechanisms

It can be very interesting to ponder evermore clever prediction strategies, and to extrapolate their theoretical limits. It is equally interesting and challenging to ponder the complexities of a real environment (threads, ILP, tag aliasing, virtual/real aliasing, overlays, pipelining, changing operand state, register pointers, etc.), and the ways in which the vagaries and vicissitudes of that environment can foil a prediction mechanism. When all is said and done, our predictors will still generate predictions that can be wrong for multiple reasons, or that can be right by fortuitous accident.

While achieving high predictive accuracy is necessary for running very fast, adding too much complexity (to achieve that accuracy) can (ironically) hurt the accuracy because of second-order effects. If adequate performance suffices, the complexity should be kept small. If truly high performance is required,

the added complexity needs to be modeled in detail to be sure that it works in a real machine.

While doing good branch prediction is essential in modern computers, many of the more exotic schemes conceived, while boons to the deipnosophists, may yet be a step or two away from implementation. When designing a predictor for a real machine, the most important thing to practice is practicability.

References

- [1] A.W. Burks, H.H. Goldstine, and J. von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. In *in Aspray and Burks [AB87]*, volume 1, pages 97–142. The Institute of Advanced Study, Princeton, Report to the U.S. Army Ordnance Department.
- [2] J. von Neumann. *Collected Works*, volume 5. MacMillan, New York, 1963.
- [3] A. W. Burks and A. R. Burks. The eniac: First general purpose electronic computer. *Annals of the History of Computing*, 3(4):310–399, 1981.
- [4] K. Godel. Uber formal unentscheidbare satze der principia mathematica und verwandter systeme i. *Mh. Math. Phys.*, (38):173–198, 1931. (English translation in M. Davis, *The Undecidable*, pp. 4-38, Raven Press, Hewlett, N.Y., 1965.).
- [5] E. Morrison and P. Morrison. *Charles Babbage and His Calculating Engines*. New York, 1961.
- [6] J. Fotheringham Dynamic storage allocation in the atlas computer including an automatic use of backing store. *Communications of the ACM*, 4(10):435–436, 1961.
- [7] P.G. Emma, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Pipeline processing apparatus for executing instructions in three streams, including branch stream, pre-execution processor for pre-executing conditional branch instructions. U.S. Patent #4991080, assigned to IBM Corporation, Filed Mar. 13, 1986, Issued Feb. 5, 1991.
- [8] P.G. Emma, J.W. Knight, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Multiple sequence processor system. U.S. Patent #5297281, assigned to IBM Corporation, Filed Feb. 13, 1992, Issued Mar. 22, 1994.

- [9] K. Skadron, P.S. Ahuja, M. Martonosi, and D.W. Clark. Branch prediction, instruction-window size, and cache size: Performance trade-offs and simulation techniques. *IEEE Transactions on Computers*, 48(11):1260–1281, Nov. 1999.
- [10] A. Roth and G.S. Sohi. Speculative data-driven multithreading. In *Seventh International Symposium on High-Performance Computer Architecture*, pages 37–48, Jan. 19-24 2001.
- [11] K. Hwang and F.A. Briggs. *Computer Architecture and Parallel Processing*. McGraw Hill Book Company, 1984. p. 14.
- [12] G. Radin. The 801 minicomputer. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 39–47, Palo Alto, California, March 1982.
- [13] P. G. Emma. Understanding some simple processor-performance limits. *IBM Journal of Research & Development*, pages 215–232, Feb. 1997.
- [14] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 25–34, May 25-29 2002.
- [15] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P. Strenski, and P. Emma. Optimizing pipelines for power and performance. In *35th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 333–344, November 2002.
- [16] Y.N. Patt, S.J. Patel, M. Evers, D.H. Friendly, and J. Stark. One billion transistors, one uniprocessor, one chip. *Computer*, 30(9):51–57, Sept. 1997.
- [17] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 295–306, May 25-29 2002.
- [18] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–19, Sept./Oct. 1997.
- [19] G. Reinman, B. Calder, and T. Austin. Fetch directed instruction prefetching. In *Proceedings of the 32nd Annual International IEEE/ACM Symposium on Microarchitecture*, pages 16–27, Nov. 16-18 1999.
- [20] T.Y. Yeh and Y.N. Patt. Branch history table indexing to prevent pipeline bubbles in wide-issue superscalar processors. In *Proceedings of the 26th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 164–175, Dec. 1-3 1993.

- [21] A. Seznec and A. Fraboulet. Effective ahead pipelining of instruction block address generation. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 241–252, June 9-11 2003.
- [22] J. Cocke, B. Randell, H. Schorr, and E.H. Sussenguth. Apparatus and method in a digital computer for allowing improved program branching with branch anticipation, reduction of the number of branches, and reduction of branch delays. U.S. Patent #3577189, assigned to IBM Corporation, Filed Jan. 15, 1965, Issued May 4, 1971.
- [23] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo. The ibm system/360 model 91: Machine philosophy and instruction-handling. *IBM Journal of Research and Development*, 11(1):8–24, Jan. 1967.
- [24] W.F. Bruckert, T. Fossum, J.A. DeRosa, R.E. Glackenmeyer, A.E. Helenius, and J.C. Manton. Instruction prefetch system for conditional branch instruction for central processor unit. U.S. Patent #4742451, assigned to Digital Equipment Corporation, Filed May 21, 1984, Issued May 3, 1988.
- [25] P.G. Emma, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Using a small cache to hedge for a bht. *IBM Technical Disclosure Bulletin*, page 1737, Sept. 1985.
- [26] P.G. Emma, J.W. Knight, J.H. Pomerene, T.R. Puzak, and R.N. Rechtschaffen. Hedge fetch history table. *IBM Technical Disclosure Bulletin*, pages 101–102, Feb. 1989.
- [27] J. Pierce and T. Mudge. Wrong-path instruction prefetching. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 165–175, Dec. 2-4 1996.
- [28] T.C. Mowry and E.A. Killian. Method and apparatus for reducing delays following the execution of a branch instruction in an instruction pipeline. U.S. Patent #5696958, assigned to Silicon Graphics, Inc., Filed Mar. 15, 1995, Issued Dec. 9, 1997.
- [29] A. Klauser and D. Grunwald. Instruction fetch mechanisms for multi-path execution processors. In *Proceedings of the 32nd Annual International IEEE/ACM Symposium on Microarchitecture*, pages 38–47, Nov. 16-18 1999.
- [30] B. Simon, B. Calder, and J. Ferrante. Incorporating predicate information into branch predictors. In *Ninth International Symposium on High-Performance Computer Architecture*, pages 53–64, Feb. 8-12 2003.
- [31] A.K. Uht, V. Sindagi, and K. Hall. Disjoint eager execution: An optimal form of speculative execution. In *Proceedings of the 28th Annual*

- IEEE/ACM International Symposium on Microarchitecture*, pages 313–325, Nov. 29- Dec. 1 1995.
- [32] T.F. Chen. Supporting highly speculative execution via adaptive branch trees. In *Fourth International Symposium on High-Performance Computer Architecture*, pages 185–194, Feb. 1-4 1998.
 - [33] D.J. Lilja. Reducing the branch penalty in pipelined processors. *Computer*, 21(7):47–55, July 1988.
 - [34] P.G. Emma, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Group approach to branch prediction. *IBM Technical Disclosure Bulletin*, page 4339, Dec. 1984.
 - [35] P.G. Emma, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Comprehensive branch prediction mechanism for bc. *IBM Technical Disclosure Bulletin*, pages 2255–2262, Oct. 1985.
 - [36] E. Jacobsen, E. Rotenberg, and J.E. Smith. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 142–152, Dec. 2-4 1996.
 - [37] C.C. Lee, I.C.K. Chen, and T.N. Mudge. The bi-mode branch predictor. In *Proceedings of the 30th Annual International IEEE/ACM Symposium on Microarchitecture*, pages 4–13, Dec. 1-3 1997.
 - [38] M. Haungs, P. Sallee, and M. Farrens. Branch transition rate: A new metric for improved branch classification analysis. In *Sixth International Symposium on High-Performance Computer Architecture*, pages 241–250, Jan. 8-12 2000.
 - [39] P.K. Dubey and M.J. Flynn. Branch strategies: Modeling and optimization. *IEEE Transactions on Computers*, 40(10):1159–1167, Oct. 1991.
 - [40] J.P. Linde. Microprogrammed control system capable of pipelining even when executing a conditional branch instruction. U.S. Patent #4373180, assigned to Sperry Corporation, Filed July 9, 1980, Issued Feb. 8, 1983.
 - [41] D.R. Ditzel and H.R. McLellan. Arrangement and method for speeding the operation of branch instructions. U.S. Patent #4853889, assigned to AT&T Bell Laboratories, Filed May 11, 1987, Issued Aug. 1, 1989.
 - [42] S. Krishnan and S.H. Ziesler. Branch prediction and target instruction control for processor. U.S. Patent #6324643, assigned to Hitachi, Ltd., Filed Oct. 4, 2000, Issued Nov. 27, 2001.
 - [43] K. Wada. Branch advanced control apparatus for advanced control of a branch instruction in a data processing system. U.S. Patent #4827402, assigned to Hitachi, Ltd., Filed Apr. 22, 1986, Issued May 2, 1989.

- [44] H. Potash. Branch predicting computer. U.S. Patent #4435756, assigned to Burroughs Corporation, Filed Dec. 3, 1981, Issued Mar. 6, 1984.
- [45] D.R. Beard, A.E. Phelps, M.A. Woodmansee, R.G. Blewett, J.A. Lohman, A.A. Silbey, G.A. Spix, F.J. Simmons, and D.A. Van Dyke. Method of processing conditional branch instructions in scalar/vector processor. U.S. Patent #5706490, assigned to Cray Research, Inc., Filed June 7, 1995, Issued Jan. 6, 1998.
- [46] S. Mahlke and B. Natarajan. Compiler synthesized dynamic branch prediction. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 153–164, Dec. 2-4 1996.
- [47] D.I. August, D.A. Connors, J.C. Gyllenhaal, and W.M.W. Hwu. Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results. In *Third International Symposium on High-Performance Computer Architecture*, pages 84–93, Feb. 1-5, 1997.
- [48] T.Y. Yeh, M.A. Popligner, and M. Rahman. Optimized branch predictions for strongly predicted compiler branches. U.S. Patent #6427206, assigned to Intel Corporation, Filed May 3, 1999, Issued July 30, 2002.
- [49] M. Tremblay. Software branch prediction filtering for a microprocessor. U.S. Patent #6374351, assigned to Sun Microsystems, Inc., Filed Apr. 10, 2001, Issued Apr. 16, 2002.
- [50] J. Smith. Branch predictor using random access memory. U.S. Patent #4370711, assigned to Control Data Corporation, Filed Oct. 21, 1980, Issued Jan. 25, 1983.
- [51] R. Nair. Optimal 2-bit branch predictors. *IEEE Transactions on Computers*, 44(5):698–702, May 1995.
- [52] P.M. Paritosh, R. Reeve, and N.R. Saxena. Method and apparatus for a single history register based branch predictor in a superscalar microprocessor. U.S. Patent #5742905, assigned to Fujitsu, Ltd., Filed Feb. 15, 1996, Issued Apr. 21, 1998.
- [53] Y. Totsuka and Y. Miki. Branch prediction apparatus. U.S. Patent #6640298, assigned to Hitachi, Ltd., Filed Apr. 7, 2000, Issued Oct. 28, 2003.
- [54] A.J. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, 1981.
- [55] J.F. Brown III, S. Persels, and J. Meyer. Branch prediction unit for high-performance processor. U.S. Patent #5394529, assigned to Digital Equipment Corporation, Filed July 1, 1993, Issued Feb. 28, 1995.

- [56] R. Nair. Dynamic path-based branch correlation. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 15–23, Nov. 29 - Dec. 1 1995.
- [57] C. Young, N. Gloy, and M.D. Smith. A comparative analysis of schemes for correlated branch prediction. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 276–286, June 22-24 1995.
- [58] B. Fagin and A. Mital. The performance of counter- and correlation-based schemes for branch target buffers. *IEEE Transactions on Computers*, 44(12):1383–1393, Dec. 1995.
- [59] S. Reches and S. Weiss. Implementation and analysis of path history in dynamic branch prediction schemes. *IEEE Transactions on Computers*, 47(8):907–912, Aug. 1998.
- [60] S.C. Steely and D.J.Sagar. Past-history filtered branch prediction. U.S. Patent #5828874, assigned to Digital Equipment Corporation, Filed June 5, 1996, Issued Oct. 27, 1998.
- [61] T. Sherwood and B. Calder. Automated design of finite state machine predictors for customized processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 86–97, June 30 - July 4 2001.
- [62] G.D. Zuraski, J.S. Roberts, and R.S. Tupuri. Dynamic classification of conditional branches in global history branch prediction. U.S. Patent #6502188, assigned to Advanced Micro Devices, Inc., Filed Nov. 16, 1999, Issued Dec. 31, 2002.
- [63] R. Thomas, M. Franklin, C. Wilkerson, and J. Stark. Improving branch prediction by dynamic data flow-based identification of correlated branches from a large global history. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 314–323, June 9-11 2003.
- [64] D.A. Jimenez and C. Lin. Dynamic branch prediction with perceptrons. In *Seventh International Symposium on High-Performance Computer Architecture*, pages 197–206, Jan. 19-24 2001.
- [65] M. Kampe, P. Stenstrom, and M. Dubois. The fab predictor: Using Fourier analysis to predict the outcome of conditional branches. In *Eighth International Symposium on High-Performance Computer Architecture*, pages 223–232, Feb. 2-6, 2002.
- [66] D.A. Jimenez. Fast path-based neural branch prediction. In *Proceedings of the 36th Annual International IEEE/ACM Symposium on Microarchitecture*, pages 243–252, Dec. 3-5 2003.

- [67] D.B. Sand. Branch prediction apparatus and method for a data processing system. U.S. Patent #4430706, assigned to Burroughs Corporation, Filed Oct. 27, 1980, Issued Feb. 7, 1984.
- [68] J. J. Losq, G. S. Rao, and H. E. Sachar. Decode history table for conditional branch instructions. U.S. Patent #4477872, assigned to IBM Corporation, Filed Jan. 15, 1982, Issued Oct. 16, 1984.
- [69] S.G. Tucker. The IBM 3090 system: An overview. *IBM Systems Journal*, 25(1):4–19, 1986.
- [70] D.R. Kaeli and P.G. Emma. Improving the accuracy of history-based branch prediction. *IEEE Transactions on Computers*, 46(4):469–472, Apr. 1997.
- [71] K. Driesen and U. Holzle. The cascaded predictor: Economical and adaptive branch target prediction. In *Proceedings of the 31st Annual International IEEE/ACM Symposium on Microarchitecture*, pages 249–258, Nov. 30 - Dec. 2 1998.
- [72] P.G. Emma, J.W. Knight, J.H. Pomerene, T.R. Puzak, and R.N. Rechtschaffen. Improved decode history table hashing. *IBM Technical Disclosure Bulletin*, page 202, Oct. 1989.
- [73] P.G. Emma, D.R. Kaeli, J.W. Knight, J.H. Pomerene, and T.R. Puzak. Aliasing reduction in the decoding history tables. *IBM Technical Disclosure Bulletin*, pages 237–238, June 1993.
- [74] A.N. Eden and T.N. Mudge. The yags branch prediction scheme. In *Proceedings of the 31st Annual International IEEE/ACM Symposium on Microarchitecture*, pages 69–77, Nov. 30 - Dec. 2 1998.
- [75] C.M. Chen and C.T. King. Walk-time address adjustment for improving the accuracy of dynamic branch prediction. *IEEE Transactions on Computers*, 48(5):457–469, May 1999.
- [76] H. Patil and J. Emer. Combining static and dynamic branch prediction to reduce destructive aliasing. In *Sixth International Symposium on High-Performance Computer Architecture*, pages 251–262, Jan. 8-12, 2000.
- [77] T.Y. Yeh and Y.N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124–134, May 19-21 1992.
- [78] T.Y. Yeh and Y.N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 257–266, May 1993.

- [79] S. Sechrest, C.C. Lee, and T. Mudge. The role of adaptivity in two-level adaptive branch prediction. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 264–269, Nov. 29- Dec. 1 1995.
- [80] M. Evers, S.J. Patel, R.S. Chappell, and Y.N. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 52–61, June 27 - July 1 1998.
- [81] T. Juan, S. Sanjeevan, and J.J. Navarro. Dynamic history-length fitting: A third level of adaptivity for branch prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 155–166, June 27 - July 1 1998.
- [82] A.R. Talcott. Methods and apparatus for branch prediction using hybrid history with index sharing. U.S. Patent #6510511, assigned to Sun Microsystems, Inc., Filed June 26, 2001, Issued Jan 21, 2003.
- [83] G.P. Giacalone and J.H. Edmondson. Method and apparatus for predicting multiple conditional branches. U.S. Patent #6272624, assigned to Compaq Computer Corporation, Filed Apr. 4, 2002, Issued Aug. 7, 2001.
- [84] D.A. Jimenez, S.W. Keckler, and C. Lin. The impact of delay on the design of branch predictors. In *Proceedings of the 33rd Annual International IEEE/ACM Symposium on Microarchitecture*, pages 67–76, Dec. 10-13, 2000.
- [85] P.G. Emma, J.W. Knight, J.H. Pomerene, T.R. Puzak, R.N. Rechtschaffen, and J.R. Robinson. Multi-prediction branch prediction mechanism. U.S. Patent #5353421, assigned to IBM Corporation, Filed Aug. 13, 1993, Issued Oct. 4, 1994.
- [86] P.H. Chang. Branch predictor using multiple prediction heuristics and a heuristic identifier in the branch instruction. U.S. Patent #5687360, assigned to Intel Corporation, Filed Apr. 28, 1995, Issued Nov. 11, 1997.
- [87] P.Y. Chang, E. Hao, and Y.N. Patt. Alternative implementations of hybrid branch predictors. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 252–257, Nov. 29 - Dec. 1 1995.
- [88] S. Mallick and A.J. Loper. Automatic selection of branch prediction methodology for subsequent branch instruction based on outcome of previous branch prediction. U.S. Patent #5752014, assigned to IBM Corporation, Filed Apr. 29, 1996, Issued May 12, 1998.
- [89] A. Falcon, J. Stark, A. Ramirez, K. Lai, and M. Valero. Prophet / critic hybrid branch prediction. In *Proceedings of the 31st Annual In-*

- ternational Symposium on Computer Architecture*, pages 250–261, June 19–23, 2004.
- [90] P.G. Emma, J.W. Knight, J.H. Pomerene, R.N. Rechtschaffen, F.J. Sparacio, and C.F. Webb. Subroutine return through the branch history table. U.S. Patent #5276882, assigned to IBM Corporation, Filed July 27, 1990, Issued Jan. 4, 1994.
 - [91] N. Suzuki. Microprocessor having branch prediction function. U.S. Patent #5327536, assigned to NEC Corporation, Filed May 22, 1991, Issued July 5, 1994.
 - [92] C.N. Tran and W.K. Lewchuk. Branch prediction unit which approximates a larger number of branch predictions using a smaller number of branch predictions and an alternate target indication. U.S. Patent #5974542, assigned to Advanced Micro Devices, Inc., Filed Oct. 30, 1997, Issued Oct. 26, 1999.
 - [93] T.M. Tran. Branch prediction mechanism employing branch selectors to select a branch prediction. U.S. Patent #5995749, assigned to Advanced Micro Devices, Inc., Filed Nov. 19, 1996, Issued Nov. 30, 1999.
 - [94] E.H. Sussenguth. Instruction sequence control. U.S. Patent #3559183, assigned to IBM Corporation, Filed Feb. 29, 1968, Issued Jan. 26, 1971.
 - [95] S. Hanatani, M. Akagi, K. Nigo, R. Sugaya, and T. Shibuya. Instruction prefetching device with prediction of a branch destination address. U.S. Patent #4984154, assigned to NEC Corporation, Filed Dec. 19, 1988, Issued Jan. 8, 1991.
 - [96] J.S. Liptay. Design of the ibm enterprise system/9000 high-end processor. *IBM Journal of Research and Development*, 36(4):713–731, July 1992.
 - [97] D.B. Fite, J.E. Murray, D.P. Manley, M.M. McKeon, E.H. Fite, R.M. Salett, and T. Fossum. Branch prediction. U.S. Patent #5142634, assigned to Digital Equipment Corporation, Filed Feb. 3, 1989, Issued Aug. 25, 1992.
 - [98] T. Morisada. System for controlling branch history table. U.S. Patent #5345571, assigned to NEC Corporation, Filed Aug. 5, 1993, Issued Sept. 6, 1994.
 - [99] M. Kitta. Arrangement for predicting a branch target address in the second iteration of a short loop. U.S. Patent #5394530, assigned to NEC Corporation, Filed Feb. 22, 1994, Issued Feb. 28, 1995.
 - [100] K. Shimada, M. Hanawa, K. Yamamoto, and K. Kaneko. System with reservation instruction execution to store branch target address for use upon reaching the branch point. U.S. Patent #5790845, assigned to Hitachi, Ltd., Filed Feb. 21, 1996, Issued Aug. 4, 1998.

- [101] C. Joshi, P. Rodman, P. Hsu, and M.R. Nofal. Invalidating instructions in fetched instruction blocks upon predicted two-step branch operations with second operation relative target address. U.S. Patent #5954815, assigned to Silicon Graphics, Inc., Filed Jan. 10, 1997, Issued Sept. 21, 1999.
- [102] P.G. Emma, J.W. Knight, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Branch target table. *IBM Technical Disclosure Bulletin*, page 5043, Apr. 1986.
- [103] C.H. Perleberg and A.J. Smith. Branch target buffer design and optimization. *IEEE Transactions on Computers*, 42(4):396–412, Apr. 1993.
- [104] B. Calder and D. Grunwald. Fast and accurate instruction fetch and branch prediction. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 2–11, April 18-21 1994.
- [105] P.G. Emma, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Highly accurate subroutine stack prediction mechanism. *IBM Technical Disclosure Bulletin*, pages 4635–4637, Mar. 1986.
- [106] C.F. Webb. Subroutine call and return stack. *IBM Technical Disclosure Bulletin*, page 221, April 1988.
- [107] P.G. Emma, J.W. Knight, J.H. Pomerene, T.R. Puzak, and R.N. Rechtschaffen. Indirect targets in the branch history table. *IBM Technical Disclosure Bulletin*, page 265, Dec. 1989.
- [108] D.R. Kaeli and P.G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 34–42, May 27-30 1991.
- [109] S. Narita, F. Arakawa, K. Uchiyama, and H. Aoki. Branching system for return from subroutine using target address in return buffer accessed based on branch type information in bht. U.S. Patent #5454087, assigned to Hitachi, Ltd., Filed Oct. 23, 1992, Issued Sept. 26, 1995.
- [110] B.D. Hoyt, G.J. Hinton, D.B. Papworth, A.K. Gupta, M.A. Fetterman, S. Natarajan, S. Shenoy, and R.V. D'Sa. Method and apparatus for resolving return from subroutine instructions in a computer processor. U.S. Patent #5604877, assigned to Intel Corporation, Filed Jan. 4, 1994, Issued Feb.18, 1997.
- [111] K. Skadron, P.S. Ahuja, M. Martonosi, and D.W. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings of the 31st Annual International IEEE/ACM Symposium on Microarchitecture*, pages 259–271, Nov. 30 - Dec. 2 1998.
- [112] P.M. Ukai, K. Tashima, and A. Aiichiro. Predicted return address selection upon matching target in branch history table with entries in

return address stack. U.S. Patent #6530016, assigned to Fujitsu, Ltd., Filed Dec. 8, 1999, Issued Mar. 4, 2003.

- [113] S.P. Kim and G.S. Tyson. Analyzing the working set characteristics of branch execution. In *Proceedings of the 31st Annual International IEEE/ACM Symposium on Microarchitecture*, pages 49–58, Nov. 30 - Dec. 2 1998.
- [114] P.G. Emma, J.W. Knight, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Multiple branch analyzer for prefetching cache lines. U.S. Patent #4943908, assigned to IBM Corporation, Filed Dec. 2, 1987, Issued July 24, 1990.
- [115] P.G. Emma, J.W. Knight, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Prefetch deconfirmation based on bht error. *IBM Technical Disclosure Bulletin*, page 4487, Mar. 1987.
- [116] P.G. Emma, J.W. Knight, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Next-sequential prefetching using a branch history table. *IBM Technical Disclosure Bulletin*, pages 4502–4503, Mar. 1987.
- [117] J.B. Keller, P. Sharma, K.R. Schakel, and F.M. Matus. Training line predictor for branch targets. U.S. Patent #6647490, assigned to Advanced Micro Devices, Inc., Filed Oct. 14, 1999, Issued Nov. 11, 2003.
- [118] J.H. Pomerene, T.R. Puzak, R.N. Rechtschaffen, P.L. Rosenfeld, and F.J. Sparacio. Pageable branch history table. U.S. Patent #4679141, assigned to IBM Corporation, Filed Apr. 29, 1985, Issued July 7, 1987.
- [119] T.Y. Yeh and H.P. Sharangpani. Method and apparatus for branch prediction using first and second level branch prediction tables. U.S. Patent #6553488, assigned to Intel Corporation, Filed Sept. 8, 1998, Issued Apr. 22, 2003.
- [120] D.R. Stiles, J.G. Favor, and K.S. Van Dyke. Branch prediction device with two levels of branch prediction cache. U.S. Patent #6425075, assigned to Advanced Micro Devices, Inc., Filed July 27, 1999, Issued July 23, 2002.
- [121] P.G. Emma, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Dynamic distinction of distinguished segments. *IBM Technical Disclosure Bulletin*, pages 4065–4069, Feb. 1986.
- [122] B. Fagin and K. Russell. Partial resolution in branch target buffers. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 193–198, Nov. 29- Dec. 1 1995.
- [123] B. Fagin. Partial resolution in branch target buffers. *IEEE Transactions on Computers*, 46(10), Oct. 1997.

- [124] P.G. Emma, J.W. Knight, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Branch wrong guess cache. *IBM Technical Disclosure Bulletin*, pages 310–311, May 1988.
- [125] J.O. Bondi, A.K. Nanda, and S. Dutta. Integrating a misprediction recovery cache (mrc) into a superscalar pipeline. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 178–190, Dec. 2-4 1996.
- [126] S. Wallace, D.M. Tullsen, and B. Calder. Instruction recycling on a multiple-path processor. In *Fifth International Symposium on High-Performance Computer Architecture*, pages 44–53, Jan. 9-13 1999.
- [127] P.G. Emma, J.H. Pomerene, G.S. Rao, R.N. Rechtschaffen, H.E. Sachar, and F.J. Sparacio. Store-time updates to the branch history table. U.S. Patent #4763245, assigned to IBM Corporation, Filed Oct. 30, 1985, Issued Aug. 9, 1988.
- [128] P.G. Emma, J.H. Pomerene, G.S. Rao, R.N. Rechtschaffen, H.E. Sachar, and F.J. Sparacio. Branch prediction mechanism in which a branch history table is updated using an operand sensitive branch table. U.S. Patent #4763245, assigned to IBM Corporation, Filed Oct. 30, 1985, Issued Aug. 9, 1988.
- [129] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes. In *Proceedings of the 31st Annual International IEEE/ACM Symposium on Microarchitecture*, pages 59–68, Nov. 30 - Dec. 2 1998.
- [130] T.H. Heil, Z. Smith, and J.E. Smith. Improving branch predictors by correlating on data values. In *Proceedings of the 32nd Annual International IEEE/ACM Symposium on Microarchitecture*, pages 28–37, Nov. 16-18 1999.
- [131] L. Chen, S. Dropsho, and D.H. Albonesi. Dynamic data dependence tracking and its application to branch prediction. In *Ninth International Symposium on High-Performance Computer Architecture*, pages 65–76, Feb. 8-12 2003.
- [132] P.G. Emma, J.H. Pomerene, T.R. Puzak, R.N. Rechtschaffen, and F.J. Sparacio. Operand history table. *IBM Technical Disclosure Bulletin*, pages 3815–3816, Dec. 1984.
- [133] A. Sodani and G.S. Sohi. Understanding the differences between value prediction and instruction reuse. In *Proceedings of the 31st Annual International IEEE/ACM Symposium on Microarchitecture*, pages 205–215, Nov. 30 - Dec. 2 1998.
- [134] P.G. Emma, J.W. Knight, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Methods and apparatus for insulating a branch prediction

mechanism from data dependent branch table updates that result from variable test operand locations. U.S. Patent #5210831, assigned to IBM Corporation, Filed Oct. 30, 1989, Issued May 11, 1993.

- [135] P.G. Emma, J.H. Pomerene, G. Rao, R.N. Rechtschaffen, and F.J. Sparacio. Ixd branch history table. *IBM Technical Disclosure Bulletin*, pages 1723–1724, Sept. 1985.
- [136] P.G. Emma, J.W. Knight, J.H. Pomerene, R.N. Rechtschaffen, and F.J. Sparacio. Unencountered branch indication. *IBM Technical Disclosure Bulletin*, pages 5036–5037, Apr. 1986.
- [137] P.G. Emma, J.W. Knight, J.W. Pomerene, T.R. Puzak, R.N. Rechtschaffen, J. Robinson, and J. Vannorstrand. Redundant branch confirmation for hedge fetch suppression. *IBM Technical Disclosure Bulletin*, page 228, Feb. 1990.
- [138] S. McFarling. Combining branch predictors. Technical report, DEC WRL TN-36, June 1993.
- [139] R. Yung. Design decisions influencing the ultrasparc's instruction fetch architecture. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 178–190, Dec. 2-4 1996.
- [140] B. Calder, D. Grunwald, and J. Emer. A system level perspective on branch architecture performance. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 199–206, Nov 29- Dec. 1 1995.
- [141] T.M. Tran and D.B. Witt. Superscalar microprocessor which delays update of branch prediction information in response to branch misprediction until a subsequent idle clock. U.S. Patent #5875324, assigned to Advanced Micro Devices, Inc., Filed Oct. 8, 1997, Issued Feb. 23, 1999.
- [142] P.G. Emma, J.W. Knight, J.H. Pomerene, and T.R. Puzak. Simultaneous prediction of multiple branches for superscalar processing. U.S. Patent #5434985, assigned to IBM Corporation, Filed Aug. 11, 1992, Issued July 18, 1995.
- [143] D.N. Pnevmatikatos, M. Franklin, and G.S. Sohi. Control flow prediction for dynamic ilp processors. In *Proceedings of the 26th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 153–163, Dec. 1-3 1993.
- [144] S. Dutta and M. Franklin. Control flow prediction with tree-like subgraphs for superscalar processors. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 258–263, Nov. 29- Dec. 1 1995.

- [145] M. Rahman, T.Y. Yeh, M. Poplingher, C.C. Scafidi, and A. Choubal. Method and apparatus for generating branch predictions for multiple branch instructions indexed by a single instruction pointer. U.S. Patent #5805878, assigned to Intel Corporation, Filed Jan 31, 1997, Issued Sept. 8, 1998.
- [146] S . Wallace and N. Bagherzadeh. Multiple branch and block prediction. In *Third International Symposium on High-Performance Computer Architecture*, pages 94–103, Feb. 1-5, 1997.
- [147] A.R. Talcott, R.K. Panwar, R. Cherabuddi, and S. Patel. Method and apparatus for performing multiple branch predictions per cycle. U.S. Patent #6289441, assigned to Sun Microsystems Inc., Filed Jan. 9, 1998, Issued Sept. 11, 2001.
- [148] R. Rakvic, B. Black, and J.P. Shen. Completion time multiple branch prediction for enhancing trace cache performance. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 47–58, June 10-14 2000.
- [149] T.M. Conte, K.N. Menezes, P.M. Mills, and B.A. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 333–334, June 22-24 1995.
- [150] Q. Jacobson, E. Rotenberg, and J.E. Smith. Path-based next trace prediction. In *Proceedings of the 30th Annual International IEEE/ACM Symposium on Microarchitecture*, pages 14–23, Dec. 1-3 1997.
- [151] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A study of slipstream processors. In *Proceedings of the 33rd Annual International IEEE/ACM Symposium on Microarchitecture*, pages 269–280, Dec. 10-13 2000.
- [152] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 2–13, June 30 - July 4 2001.
- [153] R.S. Chappell, F. Tseng, A. Yoaz, and Y.N. Patt. Microarchitectural support for precomputation microthreads. In *Proceedings of the 35th Annual International IEEE/ACM Symposium on Microarchitecture*, pages 74–84, Nov. 18-22 2002.

Chapter 4

Trace Caches

Eric Rotenberg

North Carolina State University

4.1	Introduction	87
4.2	Instruction Fetch Unit with Trace Cache	89
4.3	Trace Cache Design Space	97
4.4	Summary	104
	References	104

4.1 Introduction

To put processor performance growth in perspective: The number of in-flight instructions in various stages of execution at the same time, inside a high-performance processor, has grown from tens to hundreds of instructions in the past decade. The growth in instruction-level parallelism (ILP) is due partly to significantly deeper pipelining, and partly to increasing the superscalar issue width, as shown in Table 4.1.

TABLE 4.1: Growth in instruction-level parallelism (ILP), measured by in-flight instruction capacity

Processor Generation	Pipeline Depth (fetch to execute)	Issue Width	In-flight Instructions
Pentium	5	1 instr.	~5
Pentium-III	10	3 μ-ops	~40
Pentium-IV	20	3 μ-ops	126
IBM Power4	12	5 instr.	200

Both factors – increasing width and depth – place increasing pressure on the instruction fetch unit. Increasing the superscalar issue width requires a corresponding increase in the instruction fetch bandwidth, simply because instructions cannot issue faster than they are supplied.

The relationship between increasing pipeline depth and fetch bandwidth is more subtle. Due to data dependences among instructions, finding enough in-

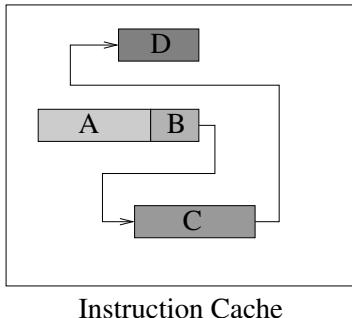


FIGURE 4.1: A conventional instruction cache stores instructions in their static program order.

dependent instructions that can execute in parallel requires examining a much larger “window” of speculative instructions. Branch mispredictions cause the window to fill with useless instructions that are eventually flushed, leaving the window empty. A deep pipeline will take a long time to replenish the window after a flush. As a result, it takes a long time to ramp up execution to its peak parallel efficiency, due to an inadequate pool of instructions from which to expose independent instructions. In this case, a fetch unit that can deliver high instruction fetch bandwidth can rapidly replenish the window in a sort of “burst” mode, ramping up overall execution faster.

Unfortunately, increasing instruction fetch bandwidth is impeded by several factors. Instruction cache misses and branch mispredictions constrain instruction fetch bandwidth by disrupting the supply of useful instructions for extended periods of time. However, even absent these discrete performance-degrading events, sustained instruction fetch bandwidth is limited on a continuous basis by frequent *taken branches* in the dynamic instruction stream. Conventional fetch units cannot fetch the instructions following a taken branch in the same cycle as the branch, since the blocks containing the branch and the target instructions are noncontiguous in the instruction cache. For example, it takes a minimum of three cycles to fetch the dynamic instruction sequence composed of basic blocks A, B, C, and D, shown in Figure 4.1, due to taken branches at the end of basic blocks B and C.

The taken-branch bottleneck did not really surface until microarchitects began pondering very aggressive superscalar processors, with >200-instruction windows and >10 peak issue bandwidth (16-issue processors became a popular substrate for a number of years [13, 18, 26]). Since taken branches typically occur once every 5 to 8 instructions [22] in integer benchmarks, a new approach to fetch unit design was needed.

The problem lies with the organization of the instruction cache, which stores instructions in their *static program order* even though instructions must be presented to the decoder in *dynamic program order*. Reconstructing a long sequence of dynamic instructions (say 16 or more) typically takes multi-

ple cycles, since the constituent static blocks are in noncontiguous locations throughout the instruction cache.

The trace cache was proposed to reconcile the way instructions are stored with the way they are presented to the decoder [15, 14, 10, 19, 23, 22, 17, 4]. A trace cache stores instructions in their dynamic program order. A trace is a long sequence of dynamic instructions, possibly containing multiple branches, some or all of which may be taken branches. Upon predicting and fetching a trace for the first time, the conventional instruction cache constructs the trace sequentially over several cycles, and supplies it to the decoder and execution engine. This first-time trace is also presented to and stored in the trace cache, for later use. Then, when the trace is needed again, as predicted by the branch prediction mechanism, it is found in the trace cache and the entire trace is supplied in a single cycle to the decoder, thereby exceeding the taken-branch bandwidth limit. Figure 4.2 depicts the process of filling the trace cache with a new trace (trace cache miss) and then later reusing the same trace from the trace cache (trace cache hit).

Alternative high-bandwidth fetch mechanisms have been proposed that assemble multiple noncontiguous fetch blocks from a conventional instruction cache [27, 1, 25]. The underlying idea is to generate multiple fetch addresses that point to all of the noncontiguous fetch blocks that must be assembled. The instruction cache is highly multiported, to enable fetching multiple disjoint cache blocks in parallel. This is achieved using true multiple ports (area-intensive), or, more cheaply, using multiple banks, although bank conflicts limit fetch bandwidth and steering addresses to their corresponding banks introduces complexity. Finally, the noncontiguous fetch blocks are assembled into the desired trace via a complex interchange/mask-shift network, which reorders the cache blocks and collapses away their unwanted instructions.

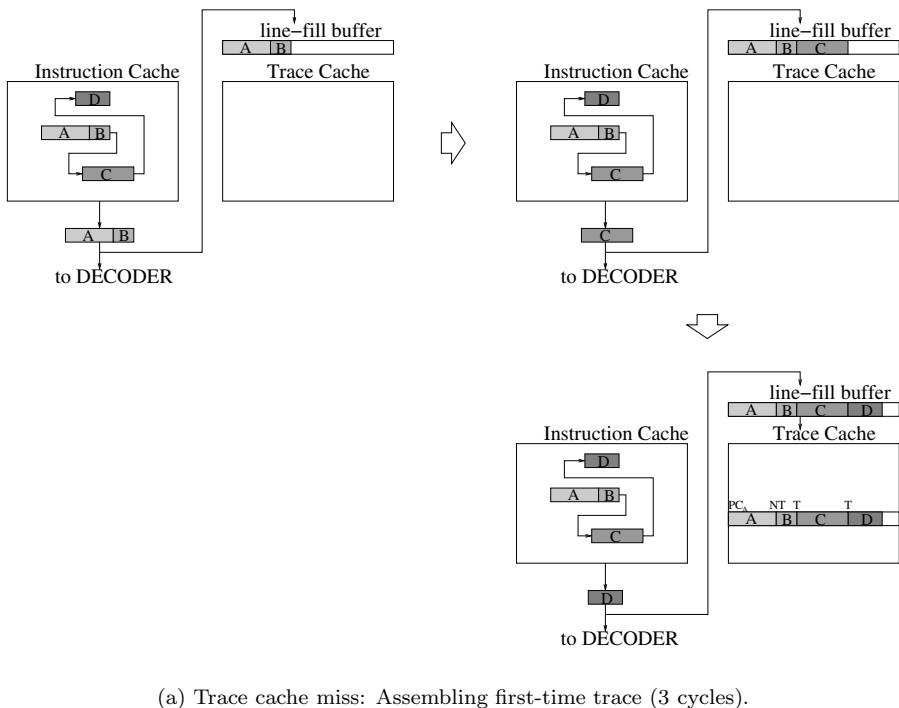
In essence, these alternative high-bandwidth fetch mechanisms repeatedly assemble traces on-the-fly from the conventional instruction cache. The trace cache moves this complexity off the critical fetch path, to the fill side of the trace cache, thus assembling traces only once and then reusing them many times afterwards.

In the following sections, we describe an instruction fetch unit with a trace cache, and then discuss in-depth issues regarding the trace cache design space.

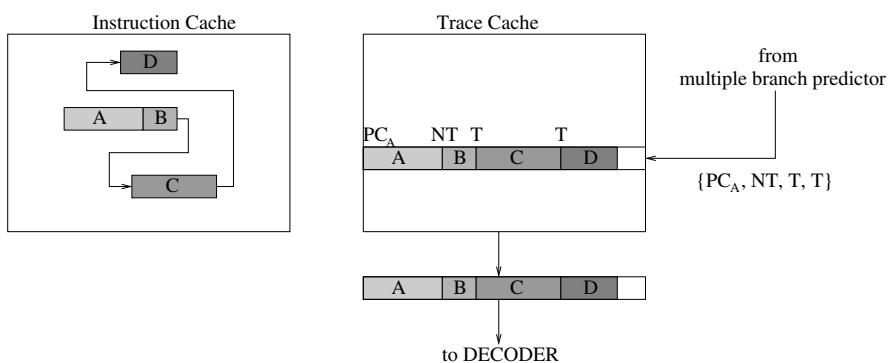
4.2 Instruction Fetch Unit with Trace Cache

4.2.1 Traces

A *trace* is a sequence of dynamic instructions containing multiple, possibly noncontiguous basic blocks. A trace is uniquely identified by the PC of the



(a) Trace cache miss: Assembling first-time trace (3 cycles).



(b) Trace cache hit: Reuse trace.

FIGURE 4.2: High-level view of trace cache operation.

first instruction in the trace, the start PC, and the sequence of taken/not-taken conditional branch outcomes embedded within the trace. The start PC plus sequence of conditional branch outcomes is sometimes referred to as the *trace identifier*, or trace id. Trace ids are fundamental in that they provide the means for looking up traces in the trace cache, regardless of specific policies for managing the trace cache (e.g., indexing policies), of which there are multiple alternatives.

The hardware limits the maximum number of instructions and branches in a trace. The maximum number of instructions in a trace is governed by the line size of the trace cache. The maximum number of branches in a trace is governed by the prediction mechanism. In order to predict the next trace to be fetched, a *multiple-branch predictor* [27, 23, 17] is needed to generate multiple branch predictions in a single cycle, corresponding to a particular path through the program rooted at the current fetch PC. Thus, the maximum number of branches in a trace is determined by the throughput of the multiple-branch predictor, i.e., the number of branch predictions it can produce per cycle.

There are typically other constraints that may terminate traces early, before the maximum number of instructions/branches is fulfilled. Traces are typically terminated at subroutine return instructions, indirect branches, and system calls. The targets of returns and indirects are variable, thus their outcomes cannot be represented with the usual taken/not-taken binary indicator. Embedding returns/indirects would require expanding the trace id representation to include one or more full target addresses (depending on the maximum number of allowable returns/indirects). Although it is feasible to design a trace cache that permits embedded returns/indirects, terminating traces at these highly-variable control transfers reduces the number of unique traces that are created, reducing trace cache pressure (conflicts) and thereby improving instruction fetch performance despite reducing the average length of traces.

The criteria for forming (delineating) traces are collectively referred to as the *trace selection* policy. Overall instruction fetch bandwidth is affected by a number of interrelated trace cache performance factors, e.g., average trace length, trace cache hit rate, trace prediction accuracy, etc. The trace selection policy typically affects all of these performance factors, providing significant leverage for balancing tradeoffs among these factors.

4.2.2 Core Fetch Unit Based on Instruction Cache

An instruction fetch unit incorporating a trace cache is shown in Figure 4.3. At its core is a high-performance conventional fetch unit, comprised of: (1) a conventional instruction cache, (2) a branch target buffer (BTB) [12], (3) a conditional branch predictor, and (4) a return address stack (RAS) [11].

The instruction cache is two-way interleaved, in order to provide a full cache line's worth of sequential instructions, even if the access is not aligned

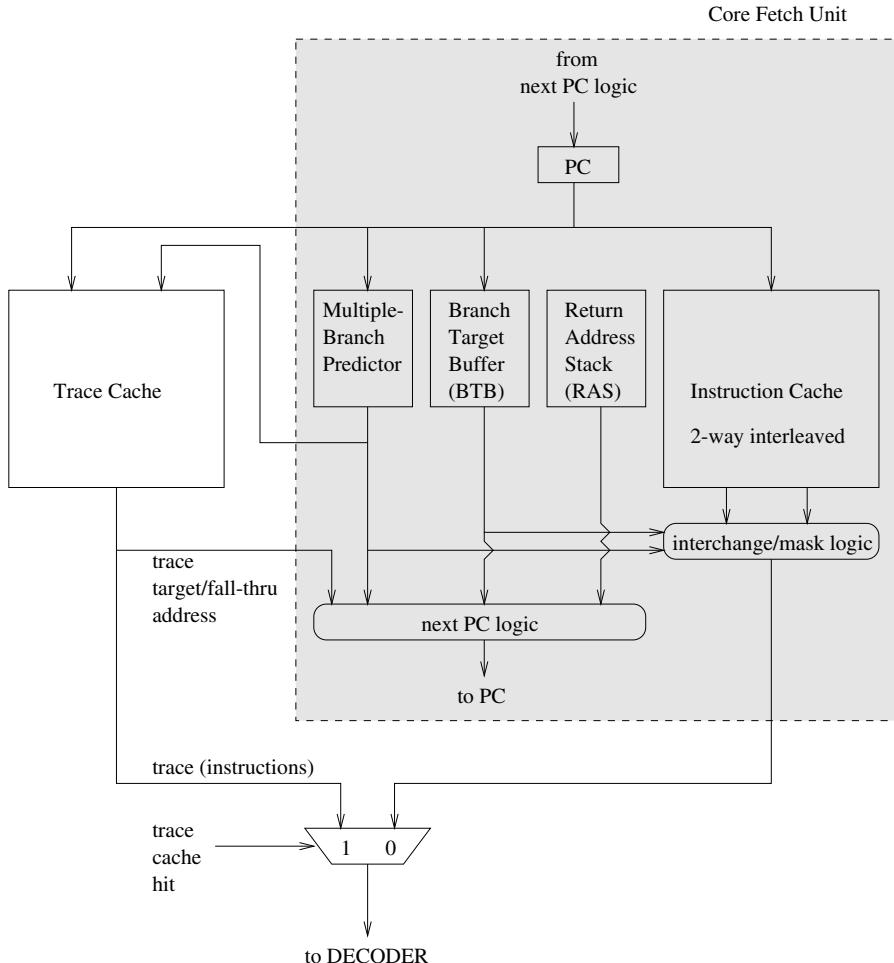


FIGURE 4.3: Instruction fetch unit with trace cache.

on a cache line boundary [6]. Without interleaving, an unaligned access supplies fewer instructions than the number of instructions in the cache line, because the fetch PC begins in the middle of the cache line. With two-way interleaving, two sequential cache lines are fetched in parallel from two banks containing only even-address and odd-address lines, respectively. Thus, although an unaligned access begins in the middle of one of the lines (either even or odd, depending on the fetch PC), additionally fetching instructions from the next sequential line enables gathering a full line of instructions, as shown in Figure 4.4.

The BTB detects and predecodes branches among the instructions fetched from the instruction cache, in the same cycle they are fetched. This enables the next fetch PC to be calculated in time for the next cycle, before the

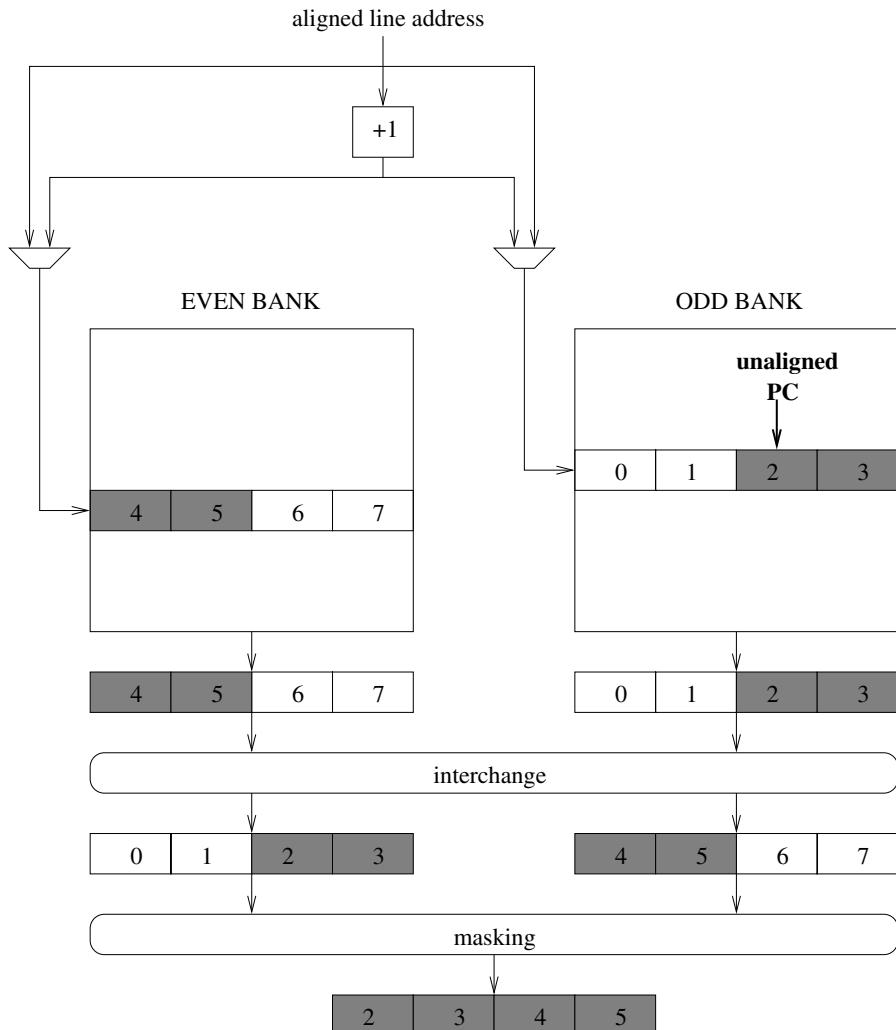


FIGURE 4.4: Two-way interleaved instruction cache.

decode stage. If a control transfer is a conditional branch and the conditional branch predictor predicts the branch is taken, then the BTB also provides the precomputed taken target. If the control transfer is a return instruction, then the RAS is popped to predict the target of the return.

As stated earlier, the conditional branch predictor must be capable of generating multiple branch predictions per cycle, if it is to guide fetching traces from the trace cache. We will discuss the operation of the trace cache in the next subsection. But, first, note that the core fetch unit based on the instruction cache may also exploit multiple branch predictions. A conventional instruction cache can supply multiple *sequential* basic blocks (i.e., their branches are all predicted not-taken), up to the first predicted-taken branch. This aggressive core fetch unit has been referred to as SEQ.n [22, 24], highlighting the ability to fetch multiple sequential basic blocks in a single cycle, as opposed to SEQ.1 which can only fetch one basic block per cycle, even if its terminal branch is predicted as not-taken.

The instruction cache is indexed using the low bits of the current fetch PC. The upper bits of the PC are used in the usual way to detect a cache hit or miss (compared against one or more tags). One or more hits in the BTB locate branches within the fetched instructions from the instruction cache. Branch predictions from the multiple-branch predictor are matched up with branches detected by the BTB. The *alignment* logic after the instruction cache swaps the two lines fetched from the banks, if instructions from the odd bank logically precede instructions from the even bank. *Masking* logic combines the BTB information (location of branches) and the multiple-branch predictions to select only instructions before the first predicted-taken branch.

4.2.3 Trace Cache Operation

In early trace cache implementations, such as the one shown in Figure 4.3, the instruction cache and trace cache are accessed in parallel. The current fetch PC plus multiple taken/not-taken predictions are used to access both caches. If the predicted trace is found in the trace cache, then instructions supplied by the instruction cache are discarded since they are a subset of the trace supplied by the trace cache. Otherwise, instructions supplied by the instruction cache are steered to the decoder. Fetching multiple sequential basic blocks from the instruction cache was explained in the previous subsection.

In one of the most basic trace cache designs, the trace cache is indexed with the low bits of the current fetch PC. Each trace in the trace cache is tagged with its trace id, i.e., start PC plus embedded branch outcomes. The low bits of the start PC are omitted since they are implied in the trace cache index (same concept applies to conventional cache tags). The embedded branch outcomes are called the *branch flags*. Each branch flag is one bit and encodes whether or not the corresponding branch is taken. In addition, a *branch mask* effectively encodes the number of embedded branches in the trace. The branch mask is used by the hit logic to compare the correct number of branch

predictions with branch flags, according to the actual number of branches within the trace. Also, several bits are included to indicate whether or not the last instruction in the trace is a branch, and its type. Finally, two address fields are provided, a *fall-through address* and a *target address*. These are explained later. The content of a generic trace cache line is shown in Figure 4.5, including the various fields outlined above and the trace’s instructions. For illustration, the maximum number of branches in a trace is 4 (the maximum number of instructions is not explicitly shown – 16 is fairly typical).

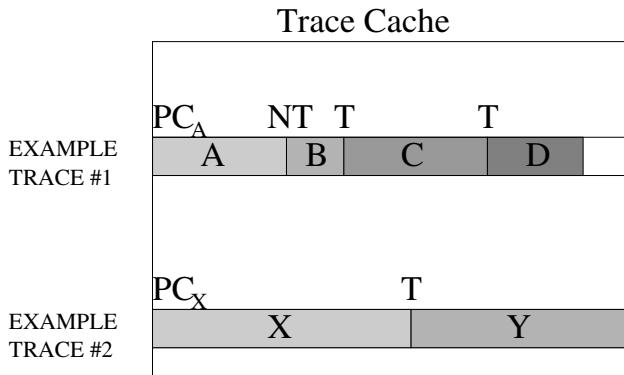
The trace cache hit logic compares the fetch PC (excluding index bits) plus multiple branch predictions with the trace id read from the trace cache. A hit is signaled if the fetch PC matches the trace’s start PC and the branch predictions match the trace’s branch flags. The branch mask provides a quick means for comparing only as many predictions flags as there are branches internal to a trace. Only internal branches, and not a terminal branch (a branch that is the last instruction in the trace), distinguish the trace and are considered by the hit logic. The branch mask is a sequence of (zero or more) leading 1’s followed by (zero or more) trailing 0’s. The number of leading 1’s is equal to the number of internal branches in the trace (again, this excludes a terminal branch). Thus, there is a trace cache hit if (1) the fetch PC matches the indexed trace’s start PC and (2) the following test is true: $(\text{branch_predictions} \& \text{branch_mask}) == (\text{branch_flags} \& \text{branch_mask})$.

Note that, if the maximum branch limit is b branches overall, then the maximum number of embedded branches is no more than $b - 1$. Thus, there are $b - 1$ branch flags and the branch mask is $b - 1$ bits long. (This assumes a trace selection policy that terminates a trace once the b^{th} branch is brought into the trace. A subtle alternative is to try extending the trace after the b^{th} branch until the maximum instruction limit is fulfilled, but stopping just before the next branch so that the maximum branch limit is still observed. We assume the former policy.)

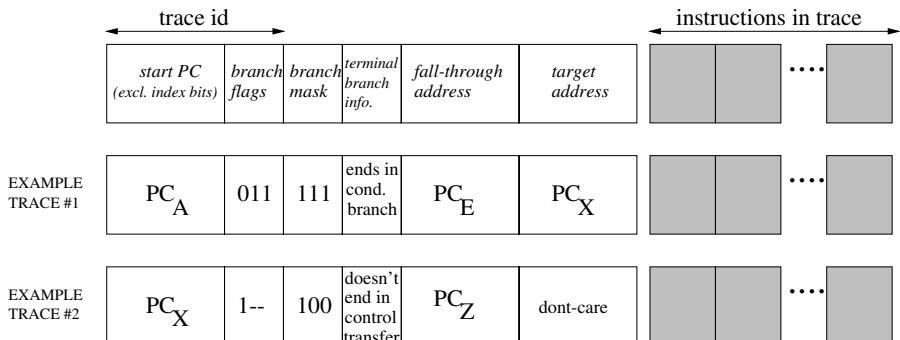
In the case of a trace cache hit, the branch mask determines how many of the predictions supplied by the multiple-branch predictor are actually “consumed” by the matching procedure. If the trace ends in a conditional branch, then one additional prediction is consumed, for predicting this terminal branch (more on this when we explain the two address fields). (Since the maximum branch limit is observed regardless of specific nuances of trace selection policies, the multiple-branch predictor provides enough branch predictions.)

Like conventional caches, the trace cache can be set-associative. In this case, multiple trace ids are read from the trace cache, equal to the number of ways. The trace cache hit logic is replicated to match the fetch PC plus predictions against all of the trace ids read from the trace cache.

The trace’s fall-through and target addresses are required because the BTB cannot generate the next PC following the trace, in the case of a trace cache hit. The trace typically extends beyond the scope of the current BTB access, by virtue of fetching past multiple taken branches. The current BTB access extends only through the first predicted-taken branch, supplying that branch’s



(a) Example traces in trace cache.



(b) Contents of trace line, and representations for two example traces.

FIGURE 4.5: Contents of trace cache line (with two examples).

taken target, corresponding to an internal target within the trace and not the overall trace’s target. (Moreover, for a less aggressive core fetch unit implementation than the one described in subsection 4.2.2, the BTB access only detects and provides a target for the first branch, whether taken or not-taken.)

The trace’s fall-through and target addresses are managed according to the last instruction in the trace (terminal instruction). If the terminal instruction is a conditional branch, then the two address fields are equal to the branch’s fall-through and target addresses, respectively. If the trace hits and the terminal branch is predicted taken, then the next PC is set equal to the trace’s target address. If the terminal branch is predicted not-taken, then the next PC is set equal to the trace’s fall-through address.

If the last instruction is not a control transfer instruction, then the trace’s fall-through address is equal to the last instruction’s PC plus one. If the trace hits, then the next PC is set equal to the trace’s fall-through address, unconditionally (the trace’s target address is ignored).

If the last instruction is a return instruction, then both address fields are irrelevant. If the trace hits, then the next PC is obtained from the RAS, like usual. (If there are one or more calls embedded within the trace and the trace ends in a return, then a RAS “hazard” exists: the last call’s target must be forwarded to the return instruction, effectively implementing a RAS bypass.)

4.3 Trace Cache Design Space

4.3.1 Path Associativity

The basic trace cache design presented in the previous section uses only the fetch PC to index into the trace cache. Thus, if the trace cache is direct-mapped, then two different traces rooted at the same start PC cannot be cached simultaneously.

Path associativity refers to the ability to cache multiple traces rooted at the same start PC. Note that a set-associative trace cache provides some degree of path associativity, since it can cache multiple traces with the same start PC within a set. Traces with the same start PC always map to the same set and thus this implicit approach for achieving path associativity is not totally flexible. Nonetheless, research has shown that conventional 2-way and 4-way set-associativity is effective for providing path associativity [17, 24].

A different indexing strategy is needed to endow a direct-mapped trace cache with path associativity, as discussed in subsection 4.3.2.

4.3.2 Indexing Strategy

In the basic trace cache design described in the previous section, the trace cache index is formed using only PC bits. However, additional information from the predicted trace id can also be used, i.e., one or more branch predictions (taken/not-taken bits). One or more predictions can either be concatenated or exclusive-or'ed with the low bits of the PC. This has the advantage of mapping multiple traces with the same start PC to different sets. In this way, even a direct-mapped cache can provide path associativity.

A potential flaw with this approach is that the number of embedded branches within the trace to be fetched, indicated by the branch mask, is not known until it is fetched, too late for forming the index. A one-size-fits-all approach must be used, i.e., a fixed number of predictions is used to form the index. For traces that have fewer embedded branches than the number of branch predictions hashed into the trace cache index, redundant copies of the trace may be created within the trace cache.

Thus, this approach is best used with a predictor that produces trace ids explicitly (in contrast, a conventional multiple-branch predictor always generates the maximum number of branch predictions without regard for trace ids explicitly). A predictor that predicts trace ids explicitly is called a *trace predictor* [8, 24].

4.3.3 Partial Matching

Partial matching is another facet of the trace cache design space. Partial matching refers to the ability to supply part of a trace if there is no cached trace that wholly matches the predicted trace, but there is a trace with a matching prefix.

Partial matching can soften the impact of not providing path associativity. If two traces with the same start PC cannot be cached simultaneously, then one can be pinned in the cache such that one of the traces always hits entirely, and the other trace partially hits due to a common prefix. (To implement this policy, a partial match should be treated as a hit instead of a miss, so that the conflicting traces do not repeatedly replace each other and one trace is preferred. It is not clear how to determine the preferred trace automatically.)

Implementing partial matching may be more trouble than it is worth. Partial matching is more expensive due to the need to include extra fall-through and target addresses for embedded branches, since the BTB can only supply the target for the first embedded branch. Without partial matching, only the overall trace fall-through and target addresses are needed, as described in Section 4.2.3. Moreover, accessing the instruction cache in parallel with the trace cache achieves some of the benefits of partial matching, in that the instruction cache may be able to supply instructions up to the first predicted-taken branch if the trace cache misses. In any case, if partial matching is implemented, it should be designed efficiently so that it supplements rather

than competes with the partial matching already provided by the instruction cache.

Of course, there are arguments one way or the other for shifting complexity away from the core fetch unit and focusing efforts on the trace cache, for instance: only access the instruction cache if the trace cache misses, do not provide an L1 instruction cache at all, etc. (these alternatives are explored in Sections 4.3.9, 4.3.10, and 4.3.11). In this case, partial matching in the trace cache may be more than just a luxury.

4.3.4 Coupling Branch Prediction with the Trace Cache

In some sense, traces in the trace cache reflect recent histories of branches. This can be exploited to tie branch prediction to the trace cache [19], rather than use a separate branch prediction mechanism.

If there is no external branch prediction mechanism, the trace cache is indexed using only the PC and it returns a predicted trace. This implies the trace cache supplies only the most recent trace rooted at a given PC, implicitly predicting a path through the control-flow region. From the standpoint of branch prediction, this is equivalent to using a one-bit counter to predict branches. (Nonetheless, redundant copies of static branches in the trace cache enables specialization of static branches, to some extent, exploiting limited context.) Alternatively, two-bit counters could be associated with each embedded branch within a trace. A trace is displaced by an alternate trace rooted at the same PC, only if the two-bit counter of the first differing branch switches polarity.

A potential drawback of tying branch prediction to the trace cache is degraded accuracy compared to using a separate state-of-the-art branch predictor, since an integrated approach is typically limited to simple bimodal prediction, as discussed above.

A hybrid approach, called branch promotion [16], exploits the trace cache for predicting highly biased branches but still uses an external predictor to predict unbiased (yet predictable) branches. A branch is promoted when its outcome is perceived to be invariant, in which case the branch predictor stops predicting it. Implicitly, the trace cache predicts promoted branches by virtue of only caching traces reflecting their favored directions. While promotion reduces the cost of the branch predictor and/or improves its accuracy (by off-loading biased branches to the trace cache), it does not altogether eliminate the external branch predictor and thus does not achieve the simplicity of fully coupling branch prediction with the trace cache. Nonetheless, promotion provides a subtle example of exploiting the trace cache to predict branches.

4.3.5 Trace Selection Policy

The trace selection policy determines how the dynamic instruction stream is delineated into traces. A base policy is dictated by hardware constraints:

maximum trace length, maximum branch prediction throughput, hardware support (or lack thereof) for embedding indirects/returns, etc.

More sophisticated policies build on top of the base policy and provide leverage for affecting various performance factors of the trace cache. The key factors that affect overall instruction fetch bandwidth are average trace length, trace cache hit rate, and trace prediction accuracy. Trace selection significantly affects these performance factors, sometimes quite subtly [24]. Yet, trace selection has yet to be systematically explored. One reason is that a policy choice typically affects all factors simultaneously, making it difficult to discover more than just basic rules of thumb that have already been observed through ad hoc means and experience.

Typically, there is a tradeoff between maximizing trace length and minimizing trace cache misses. Reducing constraints in order to grow the length of traces also tends to increase the number of frequently-used unique traces, since there are more embedded branches and thus more possible traces (although, sometimes, fewer traces may be created if there is good coverage by the longer traces). One approach for reducing trace pressure, even with very long traces, is to terminate traces at branches [16, 17], i.e., avoid splitting a basic block if possible. This prevents scenarios where nearly identical traces are created except for their start points being slightly shifted.

On the other hand, it has been observed that a large working set of traces can be exploited for higher trace prediction accuracy [24]. A trace predictor that uses a history of trace ids seems to benefit from the shifting effect described above (although this has not been directly confirmed – at present, it is only a hypothesis). The shifting effect indirectly conveys a tremendous amount of history and provides a very specific context for making predictions, by revealing specific information regarding the path that was taken up to the current point in the program.

As another common example of trading trace length for lower miss rates, terminating traces at call instructions significantly limits the average trace length. Nonetheless, stopping at call instructions significantly reduces the number of unique traces by “re-synchronizing” trace selection at the entry points of functions, regardless of the call site.

4.3.6 Multi-Phase Trace Construction

When fetching and executing a particular sequence of dynamic instructions, there is a corresponding sequence of traces, according to the trace selection policy. However, the same sequence of dynamic instructions can be covered by a different trace sequence, that is slightly shifted or “out of phase” with respect to the current trace sequence. (In fact, there are many different trace sequences, having slightly different phases, that cover the same dynamic instructions.)

Typically, the trace cache is only updated with traces from the current sequence of traces. Yet, it is possible to update the trace cache with other

potential trace sequences, which have different phases but otherwise cover the same dynamic instructions. For example, we can begin constructing a new trace after every branch [19], or even after every instruction. This approach requires multiple trace constructors, for simultaneously assembling multiple traces that are slightly shifted with respect to each other. This technique is tantamount to prefetching (preconstructing [9]) traces, based on the heuristic that the present control-flow region will be revisited in the future, only with a slightly shifted phase. This may reduce the perceived trace cache miss rate. On the other hand, a potential negative side-effect is polluting the trace cache with trace sequences that will never be observed.

4.3.7 Managing Overlap between Instruction Cache and Trace Cache

In one school of thought, the trace cache is viewed as a fetch “booster” that can be kept small if all that can be done is done to improve the instruction cache’s performance.

One way to reduce pressure in the trace cache is to avoid obvious cases of overlap with the instruction cache’s capabilities. The instruction cache can be highly optimized to fetch multiple contiguous basic blocks. Thus, the trace cache should not cache traces with zero taken branches, since the instruction cache can supply these traces very efficiently (and more efficiently than the trace cache, since the trace cache does not exploit spatial locality) [21].

Moreover, a profiling compiler can explicitly off-load more traces from the trace cache to the instruction cache, by converting frequently-taken branches into frequently-not-taken branches [1, 20]. By increasing the total number of instructions supplied by the instruction cache vs. the trace cache, a smaller trace cache can be used to target only traces that cannot be addressed by any other means.

4.3.8 Speculative vs. Non-speculative Trace Cache Updates

Another choice is whether to construct new traces speculatively as instructions are fetched from the instruction cache, or non-speculatively as instructions are retired. Constructing traces and filling them into the trace cache speculatively has the advantage of enabling trace cache hits for new traces that are reaccessed soon after the first access. This scenario may occur in small loops. Moreover, if the trip counts are low for these loops, filling the trace cache with new traces at retirement time (non-speculatively) may be altogether too late. On the other hand, filling the trace cache with speculative traces runs the risk of polluting the trace cache with useless traces.

4.3.9 Powerful vs. Weak Core Fetch Unit

If the trace cache performs sufficiently well (hits often and supplies large traces), one can make a case for simplifying the core fetch unit since most instructions are supplied by the trace cache in any case. For instance, the instruction cache could be non-interleaved since the trace cache handles unaligned accesses just fine. Also, the instruction cache could limit bandwidth to only one basic block per cycle. Both of these suggestions reduce complexity of the logic after the instruction cache needed to swap and mask instructions, as well as the BTB and next PC logic. These simplifications reduce area, static and dynamic power, and latency of the core fetch unit, although these savings must be weighed against a more complex (e.g., larger) trace cache, which we used to justify the downsizing of the core fetch unit, in the first place.

The above argument rests on the tenuous case that the trace cache can always compensate for a weak core fetch unit, and more. The counterargument is that the trace cache is not always reliable, due to lack of spatial locality and occasional explosions in the number of frequently-used unique traces. Trace caches fail to exploit spatial locality because instructions are not individually accessible, only whole traces are. Poor trace selection coupled with a large working set of static instructions may cause temporary or perpetual trace cache overload, even if there is significant instruction cache locality. From this standpoint, the core fetch unit should be designed to maximize sequential-fetch performance, for an overall robust design that is tolerant of marginal trace cache performance.

4.3.10 Parallel vs. Serial Instruction Cache Access

Up until now, we have assumed that the instruction cache and trace cache are accessed in parallel. The advantage of this approach is that the penalty for missing in the trace cache is not severe. Since the instruction cache is accessed in parallel, part of the trace is supplied to the decoder with no delay, despite the trace cache miss.

A downside of parallel lookups is power consumption. If the trace cache hits, then the instruction cache is needlessly powered up and accessed. A solution to this problem is to access the instruction cache only when it is for certain that the trace cache does not have the trace. While this saves power, unfortunately, there is a performance hit due to delaying the instruction cache access when the trace cache misses.

To sum up, parallel access optimizes performance for the less common case of a trace cache miss, whereas serial access optimizes power for the more common case of a trace cache hit. Which is better depends on how common a trace cache hit is in actuality, almost certainly benchmark dependent (among other factors). It also depends on which of the two factors, power or performance, is deemed more important for the given situation. In a hybrid approach, one

could try to predict the likelihood of a trace cache hit, and access the instruction cache either in parallel (trace cache hit unlikely) or serially (trace cache hit likely), accordingly [7].

4.3.11 L1 vs. L2 Instruction Cache

With a sufficiently high trace cache hit rate, one could perhaps make a case for eliminating the L1 instruction cache altogether. In this case, new (or displaced) traces are constructed (or reconstructed) by fetching instructions from the L2 cache. Some research provides evidence that trace construction latency is not critical to overall performance (e.g., even ten cycles to construct a trace and fill it in the trace cache has shown little effect on performance [17]).

The problem with eliminating the L1 instruction cache is that the trace cache may, at times, perform poorly. In this case, an L1 instruction cache can offset trace cache misses better than an L2 instruction cache can.

If the trace cache is only backed by the L2 cache (no L1 instruction cache), the relatively high penalty of trace cache misses can be mitigated by preconstructing/prefetching traces into the trace cache [9].

4.3.12 Loop Caches

Trace caches implicitly unroll loops with arbitrary internal control-flow, compensating for cases where the compiler may not be able to practically and accurately loop-unroll statically [3]. However, the generality of trace caches means that loops are not specifically exploited, yet, focusing on loop constructs may improve instruction storage efficiency, multiple-branch prediction efficiency and accuracy, and — ultimately — overall fetch bandwidth.

The simplest *loop caches*, used in some embedded processors, focus on simple tight loops (i.e., no internal control-flow and small loop bodies) [5]. It was recognized that significant energy is expended fetching the same sequence of instructions repeatedly from the large instruction cache. Hence, significant energy can be saved by having a much smaller cache before the instruction cache that holds very recently accessed loops.

More advanced loop caches permit control-flow within the loop body [3]. Because internal control-flow is permitted, entire loop visits are analyzed, in order to detect frequently occurring identical visits suitable for caching. The loop cache is coupled with a loop predictor [2], which essentially predicts the aggregate control-flow for a whole loop visit with only a single prediction. (Thus, a loop predictor predicts the trip count, i.e., number of iterations, and the control-flow pattern within each iteration individually.)

Even a generic trace cache can benefit from trace selection policies that are loop-aware. For example, embedding only a whole number of loop iterations tends to have a “re-synchronizing” effect, avoiding the creation of many traces that are offset by a few instructions but otherwise identical.

4.4 Summary

High instruction fetch bandwidth is important as pipelines become wider and deeper. A wide execution engine is ultimately limited by the rate at which instructions are supplied to it. Therefore, raw instruction fetch bandwidth must increase to accommodate wider superscalar processors. Even relatively narrow processors can benefit from large instruction fetch bursts that quickly replenish an empty window, reducing the ramp-up delay after pipeline flushes.

The trace cache provides a means for overcoming the taken-branch bottleneck, which otherwise fundamentally constrains the peak instruction fetch bandwidth. A trace cache exceeds the bandwidth limit of conventional instruction caches, by enabling fetching past one or more taken branches in a single cycle. This is achieved by storing instructions in their dynamic program order, i.e., storing otherwise noncontiguous instructions in a single contiguous trace cache line.

In addition to describing the basic operation of a fetch unit with a trace cache, this chapter also highlighted many design alternatives and tradeoffs that trace caches present. Some of the design alternatives have not yet been fully explored, such as trace selection, suggesting areas for future research. Future adaptive policies may hold the key for reconciling the many tradeoffs among various design alternatives.

Very wide superscalar processors (e.g., 16-issue) – the initial impetus for exploring trace caches in the mid to late 90s – have not materialized commercially. And, for the most part, researchers have returned to modest superscalar cores as substrates for exploring solutions to other bottlenecks such as the memory wall. Nonetheless, ILP seems to come and go in cycles. The author cautiously predicts that wide processors will be revisited when high processor frequency becomes intractable from a complexity and power standpoint. Before that happens, researchers must crack the branch misprediction bottleneck with compelling solutions, in order to expose abundant ILP in programs.

References

- [1] T. Conte, K. Menezes, P. Mills, and B. Patel. Optimization of instruction fetch mechanisms for high issue rates. *22nd International Symposium on Computer Architecture*, pages 333–344, June 1995.

- [2] M. de Alba and D. Kaeli. Path-based hardware loop prediction. *4th International Conference on Control, Virtual Instrumentation and Digital Systems*, Aug 2002.
- [3] M. de Alba and D. Kaeli. Characterization and evaluation of hardware loop unrolling. *Boston Area Architecture Workshop (BARC-2003)*, Jan 2003.
- [4] D. Friendly, S. Patel, and Y. Patt. Alternative fetch and issue policies for the trace cache fetch mechanism. *30th International Symposium on Microarchitecture*, pages 24–33, Dec 1997.
- [5] A. Gordon-Ross, S. Cotterell, and F. Vahid. Exploiting fixed programs in embedded systems: A loop cache example. *Computer Architecture Letters*, 1, Jan 2002.
- [6] G. F. Grohoski. Machine organization of the ibm rs/6000 processor. *IBM Journal of Research and Development*, 34(1):37–58, Jan 1990.
- [7] J. S. Hu, M. J. Irwin, N. Vijaykrishnan, and M. Kandemir. Selective trace cache: A low power and high performance fetch mechanism. Technical Report CSE-02-016, Department of Computer Science and Engineering, Pennsylvania State University, Oct 2002.
- [8] Q. Jacobson, E. Rotenberg, and J. E. Smith. Path-based next trace prediction. *30th International Symposium on Microarchitecture*, pages 14–23, Dec 1997.
- [9] Q. Jacobson and J. E. Smith. Trace preconstruction. *27th International Symposium on Computer Architecture*, pages 37–46, June 2000.
- [10] J. Johnson. Expansion caches for superscalar processors. Technical Report CSL-TR-94-630, Computer Science Laboratory, Stanford University, June 1994.
- [11] D. Kaeli and P. Emma. Branch history table prediction of moving target branches due to subroutine returns. *18th International Symposium on Computer Architecture*, pages 34–42, May 1991.
- [12] J. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 21(7):6–22, Jan 1984.
- [13] M. Lipasti and J. Shen. Superspeculative microarchitecture for beyond AD 2000. *IEEE Computer, Billion-Transistor Architectures*, 30(9):59–66, Sep 1997.
- [14] S. Melvin and Y. Patt. Performance benefits of large execution atomic units in dynamically scheduled machines. *3rd International Conference on Supercomputing*, pages 427–432, June 1989.

- [15] S. Melvin, M. Shebanow, and Y. Patt. Hardware support for large atomic units in dynamically scheduled machines. *21st International Symposium on Microarchitecture*, pages 60–66, Dec 1988.
- [16] S. Patel, M. Evers, and Y. N. Patt. Improving trace cache effectiveness with branch promotion and trace packing. *25th International Symposium on Computer Architecture*, pages 262–271, June 1998.
- [17] S. Patel, D. Friendly, and Y. Patt. Critical issues regarding the trace cache fetch mechanism. Technical Report CSE-TR-335-97, Department of Electrical Engineering and Computer Science, University of Michigan, 1997.
- [18] Y. Patt, S. Patel, M. Evers, D. Friendly, and J. Stark. One billion transistors, one uniprocessor, one chip. *IEEE Computer, Billion-Transistor Architectures*, 30(9):51–57, Sep 1997.
- [19] A. Peleg and U. Weiser. Dynamic flow instruction cache memory organized around trace segments independent of virtual address line. US Patent No. 5,381,533, Jan 1995.
- [20] A. Ramirez, J. L. Larriba-Pey, C. Navarro, J. Torrellas, and M. Valero. Software trace cache. *13th International Conference on Supercomputing*, pages 119–126, June 1999.
- [21] A. Ramirez, J. L. Larriba-Pey, and M. Valero. Trace cache redundancy: Red & blue traces. *6th International Symposium on High-Performance Computer Architecture*, pages 325–336, Jan 2000.
- [22] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. *29th International Symposium on Microarchitecture*, pages 24–34, Dec 1996.
- [23] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. Technical Report 1310, Computer Sciences Department, University of Wisconsin - Madison, Apr 1996.
- [24] E. Rotenberg, S. Bennett, and J. E. Smith. A trace cache microarchitecture and evaluation. *IEEE Transactions on Computers, Special Issue on Cache Memory*, 48(2):111–120, Feb 1999.
- [25] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud. Multiple-block ahead branch predictors. *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 116–127, Oct 1996.
- [26] J. E. Smith and S. Vajapeyam. Trace processors: Moving to fourth-generation microarchitectures. *IEEE Computer, Billion-Transistor Architectures*, 30(9):68–74, Sep 1997.

- [27] T-Y. Yeh, D. T. Marr, and Y. N. Patt. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. *7th International Conference on Supercomputing*, pages 67–76, July 1993.

Chapter 5

Branch Predication

David August

Princeton University

5.1	Introduction	109
5.2	A Generalized Predication Model	112
5.3	Compilation for Predicated Execution	115
5.4	Architectures with Predication	119
5.5	In Depth: Predication in Intel IA-64	123
5.6	Predication in Hardware Design	127
5.7	The Future of Predication	129
	References	129

5.1 Introduction

The performance of instruction-level parallel (ILP) processors depends on the ability of the compiler and hardware to find a large number of independent instructions. Studies have shown that current wide-issue ILP processors have difficulty sustaining more than two instructions per cycle for non-numeric programs [1, 2, 3]. These low speedups are a function of a number of difficult challenges faced in extracting and efficiently executing independent instructions.

The extraction of ILP can be done by the compiler, the hardware, or both. The compiler and hardware have individual strengths and the task of extracting ILP should be divided accordingly. For example, the compiler has the ability to perform detailed analysis on large portions of the program, while the hardware has access to specific dynamic outcomes which can only be known at run time. While extracting ILP is necessary to achieve large speedups in ILP processors, the architecture must be able to execute the extracted ILP in an efficient manner. Performance gained through the extraction of ILP is often lost or diminished by stalls in the hardware caused by the long latency of operations in certain exceptional conditions. For example, a load from memory may stall the processor while the data is fetched from the relatively slow main memory. Hardware techniques which minimize these stalls, such as cache in case of memory, are essential for realizing the full potential of ILP.

One of the major challenges to effectively extracting and efficiently executing ILP code is overcoming the limitations imposed by branch control flow.

At run time, the branch control flow introduces uncertainty of outcome and non-sequentiality in instruction layout, which limit the effectiveness of instruction fetch mechanisms. Sophisticated branch prediction techniques, speculative instruction execution, advanced fetch hardware, and other techniques described in this book are necessary today to reduce instruction starvation in high-performance processor cores [4, 5, 6, 7].

For non-numeric benchmarks, researchers report that approximately 20% to 30% of the dynamic instructions are branches, an average of one branch for each three to five instructions. As the number of instructions executed concurrently grows, the sustained rate of branches executed per cycle must grow proportionally to avoid becoming the bottleneck. Handling multiple branches per cycle requires additional pipeline complexity and cost, which includes multi-ported branch prediction structures. In high issue rate processors, it is much easier to duplicate arithmetic functional units than to predict and execute multiple branches per cycle. Therefore, for reasons of cost, ILP processors will likely have limited branch handling capabilities which may limit performance in non-numeric applications. In those processors which do support multiple branches per cycle, the prediction mechanisms are subject to diminishing prediction accuracy created by the additional outcomes of execution. Even branch prediction mechanisms with good accuracy on a single branch per cycle often have undesirable scaling characteristics.

5.1.1 Overcoming Branch Problems with Predication

Many forms of speculation have been proposed to alleviate the problems of branching control discussed above. A radically different approach taken in Explicitly Parallel Instruction Computing (EPIC) architectures allows the compiler to simply eliminate a significant amount of branch control flow presented to the processor. This approach, called *predication* [8, 9], is a form of compiler-controlled speculation in which branches are removed in favor of fetching multiple paths of control. The speculative trade-off the compiler makes involves weighing the performance cost of potential branch mispredictions with the performance cost associated with fetching additional instructions.

Specifically, predication is the conditional execution of instructions based on the value of a Boolean source operand, referred to as the predicate. If the value in the predicate source register is true (a logical **1**), a predicated instruction is allowed to execute normally; otherwise (a logical **0**), the predicated instruction is nullified, preventing it from modifying the processor state. The values in these predicate registers are manipulated by predicate defining instructions.

Figure 5.1 contains a simple example to illustrate the concept of predication. Figure 5.1(a) shows two if-then-else construct, often called hammocks or diamonds, arranged sequentially within the code. The outcome of each

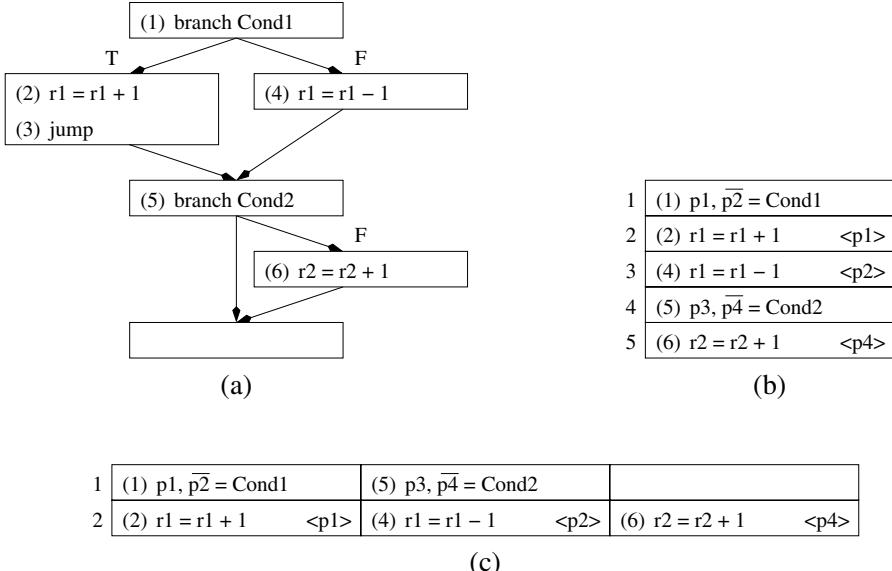


FIGURE 5.1: A simple code segment with and without predication.

branch is determined by the evaluation of the independent branch conditions $Cond1$ and $Cond2$, which may be $r3 < 10$ or any other register value comparison. Depending on the outcome of the first branch register $r1$ is either incremented or decremented. Register $r2$ is incremented only when $Cond2$ is false. In order to respect the branch control dependences, the target of the first branch must be selected before the second branch can execute.

The basic compiler transformation to exploit predicated execution is known as *if-conversion* [10, 11]. If-conversion systematically replaces conditional branches in the code with predicate define instructions that define a set of predicates. Instructions control dependent on the replaced branches are then converted to predicated instructions, utilizing the appropriate predicate register. In this manner, control dependences from a branch to a set of guarded instructions become data dependences from a predicate define to those same instructions now guarded with the defined predicate register.

Figure 5.1b shows the code segment after if-conversion, and Figure 5.1c shows the code after scheduling. Here the two branch conditions are now part of two predicate defines. Four predicates are defined to be true or false depending on the evaluation of the conditions. For example, if $Cond1$ is satisfied, P_1 is set to true and P_2 is set to false. In the next cycle, instructions 2, 4, and 6 are executed based on the values stored in the predicate registers. Notice that these instructions do not need to respect positional constraints with any branches or with each other. They are now free to be scheduled simultaneously as they are here. As a result, meaningful ILP (ILP discounting

the nullified instructions) can be increased over this original code segment even assuming perfect branch prediction in the original branch code. Another benefit is the potential reduction in instruction count. Instruction 3 is no longer necessary since within this block of code execution conditions are no longer determined solely by fetch pattern. Further, fetch hardware now has a single very simple fetch pattern to perform instead of four patterns of which only one is correct for every execution of this code segment. The removal of branches with predication improves performance by eliminating any branch misprediction penalties. In particular, the removal of frequently mispredicted branches can yield large performance gains [12, 13, 14].

The remainder of this chapter covers predication from several perspectives. The next section presents a generalized model of predicated architectures. Using this model as a point of reference, Section 5.3 details issues in the compilation for predication. Following this, Section 5.4 surveys several research and commercial predicated architectures, and Section 5.5 provides a detailed analysis of predication in action on the Itanium 2 processor. Section 5.6 outlines the challenges predication presents in future hardware design. Finally, the chapter concludes with a look toward predication's role in the future.

5.2 A Generalized Predication Model

This section presents the *IMPACT EPIC* architecture model which serves as a generalized model of architectures with predicated execution support and serves as a basis for the discussion of predication in the remainder of the chapter. The *IMPACT EPIC* architecture model [15], a statically scheduled, in-order issue, EPIC architecture originated as a generalization of the Cydra 5 and the HPL PD architectures [9, 16] described later.

Any architecture supporting predicated execution must be able to conditionally nullify the side effects of selected instructions based on the value of its predicate source register. Additionally, the architecture must support efficient computation of predicate values. To do this, the *IMPACT EPIC* model of predicated execution contains N independently addressable single-bit predicate registers. Predicate register 0 is defined always to hold the value 1. All instructions in the model take a guard predicate source register. The architectural also includes a set of predicate defining instructions. These predicate defining instructions are classified broadly as predicate comparison instructions, predicate clear/set instructions, and predicate save/restore instructions.

Predicate register file. As previously mentioned, an N by 1-bit register file to hold predicates is added to the baseline architecture. The choice of introducing a new register file to hold predicate values rather than using the

TABLE 5.1: IMPACT EPIC predicate deposit types.

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

existing general purpose register file was made for several reasons. First, it is inefficient to use a single general register to hold each one bit predicate. Second, register file porting is a significant problem for wide-issue processors. By keeping predicates in a separate file, additional port demands are not added to the general purpose register file. Within the architecture, the predicate register file behaves no differently than a conventional register file. For example, the contents of the predicate register file must be saved during a context switch. Furthermore, the predicate file is partitioned into caller and callee save sections based on the chosen calling convention. **Predicate comparison instructions.** The most common way to set predicate register values is with a new set of predicate comparison instructions. Predicate comparison instructions compute predicate values using semantics similar to those for conventional comparison instructions. There is one predicate comparison instruction for each integer, unsigned, float, and double comparison instruction in an unpredicated ISA. Unlike traditional comparison instructions, the predicate comparison instructions have up to two destination operands and these destination operands refer to registers in the predicate register file. The predicate comparison instruction format is as shown here.

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

This computes the condition $C = src_0 [cmp] src_1$ and optionally assigns a value to two destination predicates, P_{dest_1} and P_{dest_2} , according to the predicate types ($type_1$ and $type_2$, respectively), C , and the guarding predicate P_{guard} , as shown in Table 5.1. (In the table, a “-” indicates that no value is deposited.) The IMPACT EPIC Architecture defines six deposit types which specify the manner in which the result of a condition computation and the guard predicate are deposited into a destination predicate register. The (U)nconditional, (O)r, (A)nd, and (C)onditional types are as defined in the HP Labs PD Specification [16]. The and- (A), or- (O), conjunctive- (\wedge) and disjunctive- (\vee) type predicate deposits are termed parallel compares since multiple definitions of a single one of these types can commit simultaneously to the same predicate register. While simple if-conversion generates only unconditional and or-type defines [11], the other types are useful in optimization of predicate define networks; so an effective predicate analysis system should fully support their general use [17].

Unconditional destination predicate registers are always defined, regardless of the value of P_{guard} and the result of the comparison. If the value of P_{guard}

is **1**, the result of the comparison is placed in the predicate register (or its complement for \overline{U}). Otherwise, a **0** is written to the predicate register. Unconditional predicates are utilized for blocks that are executed based on a single condition, i.e., they have a single control dependence.

The OR-type predicates are useful when execution of a block can be enabled by multiple conditions, such as logical AND (`&&`) and OR (`||`) constructs in C. OR-type destination predicate registers are set if P_{guard} is **1** and the result of the comparison is **1** (**0** for \overline{OR}); otherwise, the destination predicate register is unchanged. Note that OR-type predicates must be explicitly initialized to **0** before they are defined and used. However, after they are initialized multiple OR-type predicate defines may be issued simultaneously and in any order on the same predicate register. This is true since the OR-type predicate either writes a **1** or leaves the register unchanged which allows implementation as a wired logical OR condition. This property can be utilized to compute an execution condition with zero dependence height using multiple predicate define instructions.

The AND-type predicates are analogous to the OR-type predicates. AND-type destination predicate registers are cleared if P_{guard} is **1** and the result of the comparison is **0** (**1** for \overline{AND}); otherwise, the destination predicate register is unchanged. The AND-type predicate is particularly useful for transformations such as control height reduction [18, 19].

The conditional type predicates have semantics similar to regular predicate instructions, such as adds. If the value of P_{guard} is **1**, the result of the comparison is placed in the destination predicate register (or its complement for \overline{C}). Otherwise, no actions are taken. Under certain circumstances, a conditional predicate may be used in place of an OR-type predicate to eliminate the need for an initialization instruction. However, the parallel issue semantics of the OR-type predicates are lost with conditional predicates.

Two predicate types are used to generate efficient code using Boolean minimization techniques by allowing predicate defines to behave functionally as logical-and and logical-or gates [17]. These are referred to as *disjunctive-type* ($\vee T$ or $\vee F$) and *conjunctive-type* ($\wedge T$ or $\wedge F$). Table 5.1 (right-hand portion) shows the deposit rules for the new predicate types. The $\wedge T$ -type define clears the destination predicate to **0** if either the source predicate is FALSE or the comparison result is FALSE. Otherwise, the destination is left unchanged. Note that this behavior differs from that of the and-type predicate define, in that the and-type define leaves the destination unaltered when the source predicate evaluates to FALSE. The conjunctive-type thus enables a code generator to easily and efficiently form the logical conjunction of an arbitrary set of conditions and predicates.

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

Predicate clearing and setting. The stylized use of OR-type and AND-type predicates described previously requires that the predicates be precleared and preset, respectively. While preclearing and presetting can be performed by unconditional predicate comparison instructions, a set of instructions are defined to make this process more efficient.

First, instructions to clear and set groups of registers using a mask are provided. These instructions are aptly called *pclr* and *pset*. These instructions set a contiguous group of predicate registers to zero or one using a mask. Thus, any combination of the predicates can be cleared or set using these instructions. For architectures with many predicate registers and limited instruction encoding, the mask may refer only to a subset of the predicate register file.

Second, an integer to predicate collection register move is provided. A predicate collection register is a special-purpose integer register which holds the value of a collection of consecutive predicate registers. For example, a machine with 64 predicate registers may have single 64-bit predicate collection register whose value is tied to the state of the entire predicate register file. The predicate collection register move instruction can move the value of the predicate collection register to and from an integer general purpose register. While this value is contained in a general purpose register, it can be manipulated in the usual manner.

Predicate saving and restoring. During procedure calls, predicate registers need to be saved and restored according to calling convention. Such saving and restoring can be performed using moves to and from the predicate collection register. A similar operation can be performed by the operating system during context switches.

Should the number of predicate registers needed by the compiler exceed the number of architectural predicate registers, special predicate spill and fill operations should be used. These operations, called *pspill* and *pfill*, allow an individual predicate register to be loaded from and stored to stack memory. In this manner, the compiler has the freedom to handle predicate registers in the same way as the conventional register types.

5.3 Compilation for Predicated Execution

Since predication is completely compiler inserted and managed, compilers for predicated architectures are necessarily more complex. A compiler with predication support must simultaneously handle codes with branches and predicated instructions since programs will contain both. To be effective, optimizations and analyses must be able to handle predicate and branch control of instructions with equal strength. As we will see in this section, the

complexity added by predication is a challenge to compiler writers, yet it is not without benefits.

5.3.1 If-Conversion

At the most basic level, compilers support predicated execution by performing if-conversion for some set of branches at some point in the compilation process. The decision of what branches to if-convert and when in the compilation process to if-convert them can have large effects on overall code performance.

As mentioned earlier, predication is a trade-off between the benefits of removing a branch and the cost of processing multiple paths. Unfortunately, these factors are hard to estimate during compilation. For example, the compiler must estimate the misprediction rate and bias for each branch under consideration. Further, the compiler must also estimate the effects if-conversion will have on subsequent compiler optimizations. Profile information may help with characterizing the misprediction rate, but estimating the effect removing the branch will have on subsequent optimizations is a real problem which has implications on when in the compilation process if-conversion is best applied [20].

Previous work has demonstrated the value of introducing predication in an early compilation phase to enhance later optimization; in particular the *hyperblock* compilation framework has been shown to generate efficient predicated code [21, 22]. The hyperblock compilation framework is able to deliver better performance than late if-conversion because all subsequent optimizations have more freedom to enhance the predicated code. Unfortunately, early if-conversion makes the decision of what to if-convert very difficult. The over-application of if-conversion with respect to a particular machine will result in the over-saturation of processor fetch and potentially execution resources. A delicate balance between control flow and predication must be struck in order to maximize predication's potential. Early if-conversion decisions are made based on the code's characteristics prior to further optimization. Unfortunately, code characteristics may change dramatically due to optimizations applied after if-conversion, potentially rendering originally compatible traces incompatible. In cases where estimates of final code characteristics are wrong, the final code may perform worse for having had predication introduced. This problem may be alleviated by *partial reverse if-conversion* techniques which operate at schedule time to balance the amount of control flow and predication present in the generated code. Partial reverse if-conversion replaces some predicated code with branch control flow by the re-introduction of branches [20].

Predicating late in the compilation process, or applying post-optimization if-conversion, has the advantage of not having to predict the effect of subsequent optimizations. This leads to more consistent, but less impressive

performance gains [23, 24]. This is the consequence of not capitalizing on optimization opportunities exposed during if-conversion.

5.3.2 Predicate Optimization and Analysis

In a compiler supporting predication, optimizations fall into one of three categories. These are:

1. optimizations simply made predicate-aware
2. optimizations enhanced by predication
3. predicate-specific optimizations

Most traditional optimizations fall into the first category. Dead code elimination, loop invariant code removal, common subexpression elimination, register allocation, and others simply need to understand that control is no longer just positional. This is typically accomplished providing by these optimizations with dataflow and control flow analyses which factor predication into their results. This analysis problem is, however, not a simple one.

Several predicate dataflow analysis systems have been proposed to address this issue [25, 26, 27, 28]. The simplest method to deal with predication is to augment dataflow analyses with the notion of a conditional writer. For example, in liveness analysis predicated instructions simply do not kill their definitions. This is however extremely conservative and leads to suboptimal results. In the case of live variable analysis, the degradation in accuracy dramatically reduces the effectiveness of several optimizations including register allocation [29].

More sophisticated predicate dataflow engines consider the relationship among predicates. Some make approximations for efficiency [25, 27, 28] while others represent the relationship completely for accuracy [26]. This information about the predicates' relationships is then used to perform the dataflow analysis itself. To do this, some analysis engines simply create a shadow control flow graph encoding the predicate information upon which traditional dataflow analysis can be performed [28, 27]. Other dataflow analysis systems customize the dataflow analysis itself for predication [25].

In addition to those optimizations which are simply made predicate-aware, there are those which are enhanced by predication. Scheduling and ILP optimizations are encumbered by branches. Generally, these transformations must handle control dependences separately from data dependences. Typically, compilers avoid this complication by limiting these transformations to within a single basic block. Optimization of a code window containing multiple basic blocks requires either transformation of the code, usually with duplication, or application of complex and expensive global optimization techniques. Global basic block techniques often need to make trade-offs between the performance of different paths. The uncertainty of branch outcome forces the

compiler to make path selection decisions based on estimates which can often be inaccurate. Predication eliminates these control dependences, in effect turning optimizations designed to handle a single basic block into optimizations which can handle larger predicated regions formed from a portion of the control flow graph.

Beyond expanding the scope of optimization and scheduling, predicated execution also provides an efficient mechanism by which a compiler can explicitly present the overlapping execution of multiple control paths to the hardware. In this manner, processor performance is increased by the compiler's ability to find ILP across distant multiple program paths. Another, more subtle, benefit of predicated execution is that it facilitates the movement of redundant computation to less critical portions of the code [18].

Predication also allows more freedom to code motion in loops. By controlling the execution of an instruction in a loop with predication, that instruction can be set to execute in some subset of the iterations. This is useful, for example, in loop invariant code insertion. While loop invariant code insertion may seem like a pessimization, it does in fact produce more compact schedules for nested loops in which the inner loop iterates a few times for each invocation. This is a common occurrence in many codes including some SPEC-CPU integer benchmarks. This freedom is also useful in removing partial dead code elimination from loops. It also allows for kernel-only software pipelined loops and increases the applicability of software pipelining to loops with control flow [30].

Finally, several optimizations only exist to operate on predicated codes. One such technique, called Predicate Decision Logic Optimization (PDLO), operates directly on the predicate defines [31]. PDLO extracts program semantics from the predicate computation logic. Represented as Boolean expressions in a binary decision diagram (BDD), which can also be used for analysis, the predicate network is optimized using standard factorization and minimization techniques. PDLO then reformulates the program decision logic back as more efficient predicate define networks. Combined with aggressive early if-conversion and sophisticated late partial reverse if-conversion, PDLO can be used to optimize predicate defines as well as branches.

Another predicate-specific optimization is called *predicate promotion*. Predicate promotion is a form of compiler-controlled speculation that breaks predicate data dependences. Using predicate promotion to break predicate data dependences is analogous to using compiler-controlled control speculation to break control dependences. In predicate promotion, the predicate used to guard an instruction is *promoted* to a predicate with a weaker condition. That is, the instruction is potentially nullified less often, potentially never, during execution. Since any unnecessary extra executions are not identified until later, all executions of the instruction must be treated as speculative. Promotion must be performed carefully with potentially accepting instructions as the extra executions may throw a spurious exception, an exception not found in the original program. To address this problem, several architectures provide a

form of delayed exception handling which is only invoked for the non-spurious exceptions [28, 17, 32]. Predicate promotion is essential to realizing the full potential of predication to extract ILP.

5.3.3 The Predicated Intermediate Representation

Even when predicated execution support in the targeted hardware is minimal or non-existent, a predication may be useful in the compiler’s intermediate representation. The predicated intermediate representation is obtained by performing if-conversion early in the compilation process, before optimization. This predicated representation provides a useful and efficient model for compiler optimization and scheduling. The use of predicates to guard instruction execution can reduce or even completely eliminate the need to manage branch control dependences in the compiler, as branches are simply removed through the if-conversion process. Control dependences between branches and other instructions are thus converted into data dependences between predicate computation instructions and the newly predicated instructions. In the predicated intermediate representation, control flow transformations can be performed as traditional data flow optimizations. In the same way, the predicated representation allows scheduling among branches to be performed as a simple reordering of sequential instructions. Removal of control dependences increases scheduling scope and affords new freedom to the scheduler [30]. Before code emission for a machine without full predication support, the predicated instructions introduced must be converted to traditional branch control flow using a process called *reverse if-conversion* [33].

5.4 Architectures with Predication

Many architectures have included predication in their designs. This section describes several different implementations of predication in the context of these architectures. In each case, the type of predicate support implemented reflects the design goals typical of their respective application domains.

5.4.1 Hewlett-Packard Laboratories PD

The basis for predication in the IMPACT EPIC architecture is the Hewlett-Packard PD (formerly HP PlayDohTM) architecture developed at Hewlett-Packard Laboratories [16]. As a research platform, HP-PD was designed to be as general as possible. Evaluations made with HP-PD influenced the design of Intel’s IA-64 architecture described in Section 5.5.

PD is a parameterized Explicitly Parallel Instruction Computing (EPIC) architecture intended to support public research on ILP architectures and compilation [16]. HP-PD predicate define instructions generate two Boolean values using a comparison of two source operands and a source predicate. An HP-PD predicate define instruction has the form:

$$pD_0\text{-}type_0, pD_1\text{-}type_1 = (src_0 \text{ cond } src_1) \langle pSRC \rangle.$$

The instruction is interpreted as follows: pD_0 and pD_1 are the destination predicate registers; $type_0$ and $type_1$ are the predicate types of each destination; $src_0 \text{ cond } src_1$ is the comparison, where cond can be *equal* ($==$), *not equal* ($!=$), *greater than* ($>$), etc.; $pSRC$ is the source predicate register. The value assigned to each destination is dependent on the predicate type. HP-PD defines three predicate types, *unconditional* (UT or UF), *wired-or* (OT or OF), and *wired-and* (AT or AF). Each type can be in either normal mode or complement mode, as distinguished by the T or F appended to the type specifier (U, O, or A). Complement mode differs from normal mode only in that the condition evaluation is treated in the opposite logical sense.

For each destination predicate register, a predicate define instruction can either deposit a 1, deposit a 0, or leave the contents unchanged. The predicate type specifies a function of the source predicate and the result of the comparison that is applied to derive the resultant predicate. The eight left-hand columns of Table 5.1 show the deposit rules for each of the HP-PD predicate types in both normal and complement modes, indicated as “T” and “F” in the table, respectively.

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

With traditional approaches to generating predicated code, these limitations are not serious, as there is little need to support logical operations amongst predicates. The Boolean minimization strategy described in Section 5.3, however, makes extensive use of logical operations on arbitrary sets of both predicates and conditions. In this approach, intermediate predicates are calculated that contain logical subexpressions of the final predicate expressions to facilitate reuse of terms or partial terms. The intermediate predicates are then logically combined with other intermediate predicates or other compare conditions to generate the final predicate values. Without efficient support for these logical combinations, gains of the Boolean minimization approach are diluted or lost.

```

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

```

FIGURE 5.2: Example of if-then-else predication in the Cydra 5.

5.4.2 Cydrome Cydra 5

The Cydrome Cydra 5 system is a VLIW, multiprocessor system utilizing a directed-dataflow architecture [9, 34]. Developed in the 1980’s, Cydra 5 was a pioneer in the use of predication and was aimed at a broad range of supercomputing applications. Predication was primarily included in the architecture to enhance the modulo scheduling of loops.

Each Cydra 5 instruction word contains seven operations, each of which may be individually predicated. An additional source operand added to each operation specifies a predicate located within the predicate register file. The predicate register file is an array of 128 Boolean (one bit) registers. The content of a predicate register may only be modified by one of three operations: *stuff*, *stuff_bar*, or *brtop*. The *stuff* operation takes as operands a destination predicate register and a Boolean value as well as an input predicate register. The Boolean value is typically produced using a comparison operation. If the input predicate register is one, the destination predicate register is assigned the Boolean value. Otherwise, destination predicate is assigned to 0. The *stuff_bar* operation functions in the same manner, except the destination predicate register is set to the inverse of the Boolean value when the input predicate value is one. The semantics of the unconditional predicates are analogous to those of the *stuff* and *stuff_bar* operations in the Cydra 5. The *brtop* operation is used for loop control in software pipelined loops and sets the predicate controlling the next iteration by comparing the contents of a loop iteration counter to the loop bound.

Figure 5.2 shows the previous example after if-conversion for the Cydra 5. To set the mutually exclusive predicates for the different execution paths shown in this example requires three instructions. First, a comparison must

be performed, followed by a *stuff* to set the predicate register for the true path (predicated on P_1) and a *stuff-bar* to set the predicate register for the false path (predicated on P_2). This results in a minimum dependence distance of 2 from the comparison to the first possible reference of the predicate being set.

In the Cydra 5, predicated execution is integrated into the optimized execution of modulo scheduled inner loops to control the prologue, epilogue, and iteration initiation [35, 36]. Predicated execution in conjunction with rotating register files eliminates almost all code expansion otherwise required for modulo scheduling. Predicated execution also allows loops with conditional branches to be efficiently modulo scheduled.

5.4.3 ARM

The Advanced RISC Machines (ARM) processors consist of a family of processors, which specialize in low cost and very low power consumption [37]. They are targeted for embedded and multi-media applications. Consequently, ARM uses a relatively low-overhead version of predication. The ARM instruction set architecture supports the conditional execution of all instructions. Each instruction has a four bit condition field that specifies the context for which it is executed. By examining the condition field of an instruction and the condition codes in a processor status register, the execution condition of each instruction is calculated. The condition codes are typically set by performing a compare instruction. The condition field specifies under what comparison result the instruction should execute, such as equals, less than, or less than or equals.

5.4.4 Texas Instruments C6X

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

5.4.5 Systems with Limited Predicated Execution Support

Many contemporary processors added some form of limited support for predicated execution with the goal of gaining some of the benefits of conditional execution while maintaining backward compatibility. A conditional move instruction was added to the DEC Alpha, SPARC V9, and Intel Pentium Pro processor instruction sets [38, 39, 40]. A conditional move is functionally equivalent to that of a predicated move. The move instruction is augment-

ed with an additional source operand which specifies a condition. As with a predicated move, the contents of the source register are copied to the destination register if the condition is true. Otherwise, the instruction does nothing. The DEC GEM compiler can efficiently remove branches utilizing conditional moves for simple control constructs [41].

Another approach was taken by the HP PA-RISC instruction set designers. PA-RISC provides all branch, arithmetic, and logic instructions the capability to conditionally nullify the subsequent instruction in the instruction stream [42].

The Multiflow Trace 300 series machines supported limited predicated execution by providing *select* instructions [43]. Select instructions provide more flexibility than conditional moves by adding a third source operand. The semantics of a select instruction in C notation are as follows:

```
select dest,src1,src2,cond
dest = ( (cond) ? src1 : src2 )
```

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

Vector machines have had support for conditional execution using mask vectors for many years [44]. A mask of a statement S is a logical expression whose value at execution time specifies whether or not S is to be executed. The use of mask vectors allows vectorizing compilers to vectorize inner loops with if-then-else statements.

Although there are a great variety of architectural styles for predicated execution, most can be mapped to the generalized IMPACT EPIC model presented earlier.

5.5 In Depth: Predication in Intel IA-64

Perhaps the mostly widely known architecture employing predication is Intel’s IA-64 [32]. Predication in IA-64 is the product of research performed on the HP-PD and IMPACT EPIC architectures. The IA-64 architecture is implemented by Intel’s Itanium line of processors. As an aggressive design with full predication support, the Itanium processor line makes an excellent tool for the evaluation of predication and for predicate compilation research.

TABLE 5.2: The five comparison type combinations available with IA-64 compare instructions. The comparison types used in each combination are defined in Table 5.1

<i>ctype</i>	<i>p</i> ₁	<i>p</i> ₂
none	CT	CF
unc	UT	UF
or	OT	OT
and	AT	AT
or.andcm	OT	AF

The Intel IA-64 architecture supports full predication with a bank of 64 predicate registers and a limited set of two-destination predicate defining instructions. Large general purpose register files combined with the cost of encoding this level of predication led to a larger operation width. Up to three non-NOP operations exist in every 128-bit instruction bundle. This cost was deemed worthy for the potentially branch intensive server and supercomputing workloads targeted by Intel designers.

Predicate defines are simply called *compares* in IA-64, a consequence of the fact that the only comparison instructions that exist all have predicate, not integer, destination registers. The compare instructions have the format:

$$(qp) \text{ cmp}.crel.ctype \ p_1, p_2 = r_2, r_3$$

where the *qp* is the qualifying predicate, the *crel* is the comparison relation (eq, ne, lt, le, gt, ge, ltu, leu, gtu), and the *ctype* is the comparison type. IA-64 supports the four basic comparison types proposed in the HP-PD model in its compare instructions, although only in limited combinations within a single instruction. These are shown in Table 5.2.

Since not all combinations of deposit types can be combined into a single predicate defining instruction, the compiler is forced to generate sub-optimal comparison code in some situations when compared to the HP-PD or IMPACT EPIC models. Further, some compilation techniques presented for predicate optimization are only applicable with significant modification and, even then, are likely to lead to performance robbing inefficiencies [31]. Nevertheless, it is interesting to explore the role predication plays in a commercial product such as the Itanium 2.

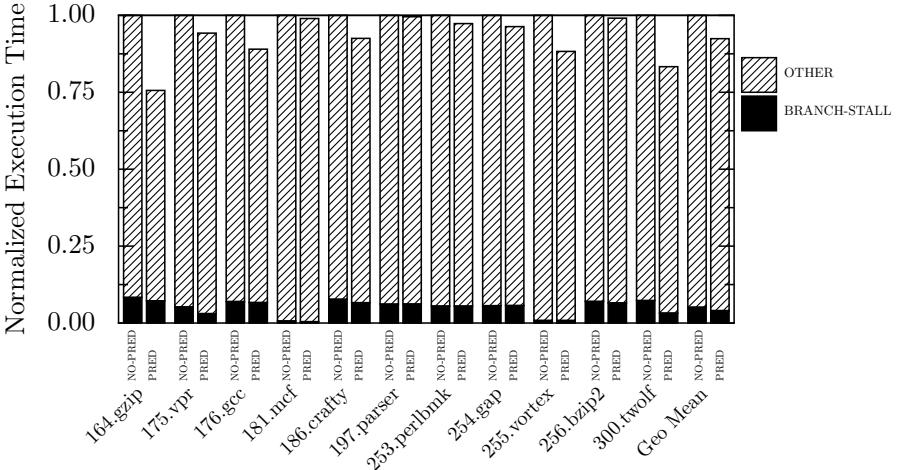


FIGURE 5.3: Execution time of SPEC benchmarks with (PRED) and without (NO-PRED) predication (normalized to NO-PRED). Total time is subdivided into branch misprediction stalls and other.

5.5.1 Predication in the Itanium 2 Processor

As of this writing, the Itanium 2 processor represents the most advanced implementation of Intel's IA-64 ISA. The Itanium 2 is a six-issue processor with six ALUs, two FPUs, and three branch units. The two of the six ALUs also serve as load units, and another two serve as store units. Itanium 2 issues instructions in order and does not perform register renaming. The Itanium 2 employs an aggressive branch predictor, and instruction fetch is decoupled from the back end by a 48-operation buffer [45].

All ILP achieved is expressed in advance by the compiler in the form of instruction groups. These groups are delimited by the compiler using stop bits in the executable. As an implementation of IA-64, almost all instructions in the Itanium 2 are guarded by a qualifying predicate. In addition, compiler controlled data and control speculation are supported.

Figures 5.3 and 5.4 show the impact predication has on performance of the Itanium 2. These figures show characteristics of SPEC 2000 codes generated by version 7.1 of Intel's Electron compiler using aggressive optimization settings running on 900Mhz Itanium 2 processors. Electron performs if-conversion late in the compilation process and, as a consequence, cannot take advantage of more aggressive uses of predication [24]. (Sias et al. perform a thorough analysis of predication and other ILP enhancing techniques with early if-conversion using the IMPACT compiler on Itanium 2 [22].) A breakdown of the performance is obtained using the hardware performance monitoring features of the Itanium 2, collected using Perfmon kernel support and the Pfmon tool developed by Hewlett-Packard Laboratories [46].

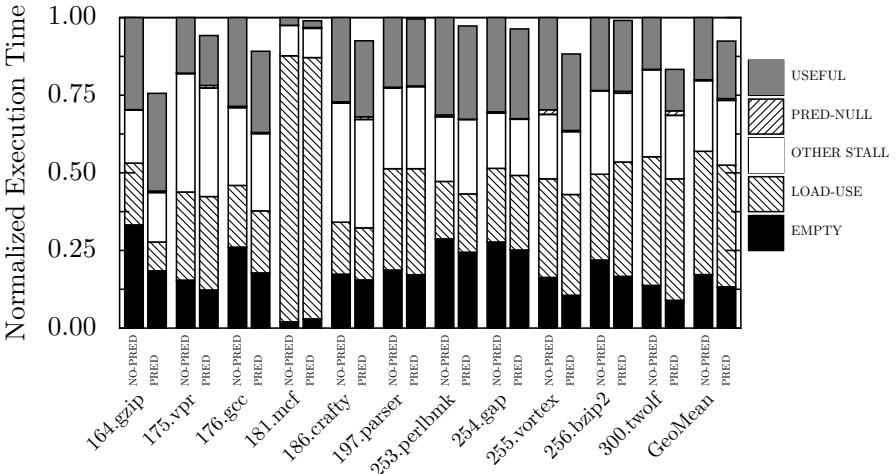


FIGURE 5.4: Execution time of SPEC benchmarks with (PRED) and without (NO-PRED) predication (normalized to NO-PRED). Total time is subdivided into time wasted performing no work (empty execution slots), stalls due to load-use (cache misses), other types of stalls, nullifying instructions, and executing useful work.

Figure 5.3 shows the performance of the benchmarks without and with predication. Overall, predication helps to achieve a 8.2% reduction in cycle count. 164.gzip, the benchmark responding best to predication, shows a 25% reduction in cycles is achieved. Interestingly, the change in the amount of time spent in branch stalls is only slightly reduced. This suggests that, contrary to the expectations of many, the performance enhancement derived from predication is not due to a reduction in total branch mispredictions. This is consistent with measurements by others [47].

Figure 5.4 shows that predication improves performance primarily by reducing load-use stalls and by increasing utilization of empty issue slots. Both of these effects are a consequence of the increased scheduling freedom and optimization opportunities provided by prediction. In the case of load-use stalls, this scheduling freedom enables the compiler to find useful work to hide cache misses. Since the Itanium 2 stalls on use, work can be performed on code following a load missing in the cache so long as it doesn't use the value. Scheduling freedom provided by predication frees up more instructions for this purpose. In the case of empty issue slots, the added scheduling freedom allows instructions to be issued in cases where execution resources would otherwise go unused. This increase in ILP exposed by the compiler translates directly to performance during execution.

1	(1) $p_1, \bar{p}_2 = \text{Cond1}$		
2	(2) $r1 = 0$	< $p_1>$	(3) $r1 = 1$
3	(4) $r2 = r1 + 5$		< $p_2>$

FIGURE 5.5: A simple code segment illustrating the multiple definition problem.

5.6 Predication in Hardware Design

As described earlier in this chapter, by removing branches, predication reduces the need to speculate early in the pipeline. Instead, in using predication, the compiler speculates that the cost of fetching additional instructions with predication is offset by the savings obtained by removing branch mispredictions, by reducing branch resource consumption, and by enhancing optimizations in the compiler. There are however new hardware design issues raised by predication which must also be considered.

While predication eliminates the need to make decisions that necessitate speculation early in the pipeline, predication still requires decisions to be made later in the pipeline. These decisions may ultimately require speculation in more aggressive architectures. These decisions stem from the *multiple-definition problem*.

Figure 5.5 shows a very simple example of the multiple-definition problem. In this example, the predicate define instruction 1 in cycle 1 sets the value of predicates P_1 and its complement, P_2 . In cycle 2, only one of the two instructions issued concurrently (2 or 3) will execute while the other is nulled. Finally, in cycle 3, instruction 4 will either get the value of $r1$ from instruction 2 or from instruction 3 depending on the value of the predicates P_1 and P_2 . At this point, $r2$ will either have the value 5 or 6. An extremely simple in-order machine has no problem with this sequence of events. This, however, is not true for more aggressive architectures with bypassing.

Consider this code example in a modern in-order pipeline. In such a case, the value produced in cycle 2 may not have time to be recorded in the single architectural register file. Instead, register bypass logic may be necessary to deliver the value of $r1$ to instruction 4. Since the bypass either forwards the data from the execution unit handling instruction 2 or the execution unit handling instruction 3 depending on the value of the predicates, the value of the predicate must be known in time. This may be problematic, however, as the value of these predicates are only known after the execution of the predicate define. For instance, in a machine with incomplete bypass, the issue logic may need to know the value of the predicate before issuing instructions 2 and 3. This information is unlikely to be available as there is ideally no time between the completion of instruction 1 and the start of execution of instructions 2 and 3.

1	(1) $p1, \bar{p2} = \text{Cond1}$	
2	(2) $r1 = 0$	$\langle p1 \rangle$
3	(3) $\text{if } (p2) \ r1 = 1 \ \text{else } r1 = r1$	
4	(4) $r2 = r1 + 5$	

FIGURE 5.6: Serialization to solve the multidef problem.

In a machine with a renaming stage, such as is found in architectures supporting out-of-order execution, the problem is much worse. In such machines, a renaming stage must give physical register names to the instances of $r1$ in instructions 2, 3, and 4. This renaming must respect the actual dependence at run-time, a dependence determined by the value of the predicate. Since the rename stage is before the execute stage, the predicate information is typically unavailable at the time registers are renamed, as is the case in this example. The naïve solution, to stall the processor at rename until predicates are computed, is in general unsatisfactory.

Several solutions have been proposed in the literature to address this problem. Perhaps the simplest is to serialize predicated instructions 2 and 3 in the compiler or hardware [43, 48]. This solution is illustrated in Figure 5.6. With serialization, instruction 3 consumes the value of instruction 2. In cases where instruction 3 would be nullified, the instruction now is still executed. Instead of doing nothing, the instruction now performs a physical register move of the result of instruction 2 into the register destined to be sourced by instruction 4. While this eliminates the need to stall in the rename stage, the cost of this solution is still high as it causes the serialization of predicated instructions. The solution also requires an additional source operand, as can be seen in instruction 3, that adds complexity to the design of the datapath. A similar technique is used to deal with the CMOV, conditional move instruction, in the Alpha 21264 [49].

To reduce this serialization, Wang et al. proposed an approach based upon the notion of static-single-assignment (SSA) [50]. The idea defers renaming decisions by inserting additional instructions, called select- μ ops which resemble phi-nodes, before instructions which need the computed value. These additional instructions are responsible for selecting between the set of definitions based on predicate values. The presence of the select- μ ops allows destination registers of predicated instructions, instructions 2 and 3 in the figure, to be named without regard to the predicate relationship between the multiple definitions and the original use.

Another solution employs a form of speculation, called predicate prediction, has been proposed [51]. The proposed predicate prediction mechanism speculatively assumes a value for the predicate to be used in the renaming process, and provides a clever means to recover from mispredictions which avoids resource consumption by known nullified instructions. Consequently, the penalty due to mispredicting a predicate was shown to be less severe

than mispredicting a branch. In the compiler, predicate mispredictions influence factors used by the compiler heuristics, but leave the basic principles unchanged.

5.7 The Future of Predication

While predication has realized significant performance gains in actual processors, predication's true potential has not yet been achieved. As described earlier in Section 5.3, predication enhances and augments ILP optimizations even in the presence of perfect branch prediction. While researchers working on predicated compilations techniques see many untapped opportunities today, there likely exist many more unidentified opportunities.

As seen in Section 5.5, predication's effect on ILP is more significant than its effect removing stalls due to branch misprediction in Itanium 2. This, however, is likely to change. The longer pipelines necessary to enhance performance exacerbate the branch misprediction problem. Novel architectural and compiler techniques which speculatively execute larger and disjoint regions of code may also increase reliance on good branch predictions. Predication is likely to play a role in addressing problems introduced by these new techniques by avoiding the need to make such speculative predictions.

References

- [1] N. P. Jouppi and D. W. Wall, Available instruction-level parallelism for superscalar and superpipelined machines, in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 272–282, April 1989.
- [2] M. D. Smith, M. Johnson, and M. A. Horowitz, Limits on multiple instruction issue, in *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 290–302, April 1989.
- [3] M. A. Schuette and J. P. Shen, An instruction-level performance analysis of the Multiflow TRACE 14/300, in *Proceedings of the 24th International Workshop on Microprogramming and Microarchitecture*, pp. 2–11, November 1991.

- [4] T. M. Conte, K. Menezes, P. Mills, and B. Patel, Optimization of instruction fetch mechanisms for high issue rates, in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 333–344, June 1995.
- [5] E. Rotenberg, S. Bennett, and J. E. Smith, Trace cache: A low latency approach to high bandwidth instruction fetching, in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 24–34, December 1996.
- [6] J. E. Smith, A study of branch prediction strategies, in *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 135–148, May 1981.
- [7] T. Y. Yeh and Y. N. Patt, Two-level adaptive training branch prediction, in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pp. 51–61, November 1991.
- [8] P. Y. Hsu and E. S. Davidson, Highly concurrent scalar processing, in *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 386–395, June 1986.
- [9] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, The Cydra 5 departmental supercomputer, *IEEE Computer*, vol. 22, pp. 12–35, January 1989.
- [10] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, Conversion of control dependence to data dependence, in *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pp. 177–189, January 1983.
- [11] J. C. Park and M. S. Schlansker, On predicated execution, Tech. Rep. HPL-91-58, Hewlett Packard Laboratories, Palo Alto, CA, May 1991.
- [12] D. N. Pnevmatikatos and G. S. Sohi, Guarded execution and branch prediction in dynamic ILP processors, in *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 120–129, April 1994.
- [13] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu, Characterizing the impact of predicated execution on branch prediction, in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 217–227, December 1994.
- [14] G. S. Tyson, The effects of predicated execution on branch prediction, in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 196–206, December 1994.
- [15] D. I. August, K. M. Crozier, J. W. Sias, D. A. Connors, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu, The IMPACT EPIC 1.0 Architec-

- ture and Instruction Set reference manual, Tech. Rep. IMPACT-98-04, IMPACT, University of Illinois, Urbana, IL, February 1998.
- [16] V. Kathail, M. S. Schlansker, and B. R. Rau, HPL PlayDoh architecture specification: Version 1.0, Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, Palo Alto, CA, February 1994.
 - [17] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu, Integrated predication and speculative execution in the IMPACT EPIC architecture, in *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 227–237, June 1998.
 - [18] M. Schlansker, V. Kathail, and S. Anik, Height reduction of control recurrences for ILP processors, in *Proceedings of the 27th International Symposium on Microarchitecture*, pp. 40–51, December 1994.
 - [19] M. Schlansker and V. Kathail, Critical path reduction for scalar programs, in *Proceedings of the 28th International Symposium on Microarchitecture*, pp. 57–69, December 1995.
 - [20] D. I. August, W. W. Hwu, and S. A. Mahlke, A framework for balancing control flow and predication, in *International Symposium on Microarchitecture*, pp. 92–103, 1997.
 - [21] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, R. A. Bringmann, and W. W. Hwu, Effective compiler support for predicated execution using the hyperblock, in *Proceedings of the 25th International Symposium on Microarchitecture*, pp. 45–54, December 1992.
 - [22] J. W. Sias, S. zee Ueng, G. A. Kent, I. M. Steiner, E. M. Nystrom, and W. mei W. Hwu, Field-testing IMPACT EPIC research results in Itanium 2, in *Proceedings of the 31st Annual International Symposium on Computer Architecture*.
 - [23] A. Klauser, T. Austin, D. Grunwald, and B. Calder, Dynamic hammock predication for non-predicated instruction set architectures, in *Proceedings of the 18th Annual International Conference on Parallel Architectures and Compilation Techniques*, pp. 278–285, October 1998.
 - [24] J. Bharadwaj, K. Menezes, and C. McKinsey, Wavefront scheduling: Path based data representation and scheduling of subgraphs, in *Proceedings of the 32nd International Symposium on Microarchitecture (MICRO)*, pp. 262–271, November 1999.
 - [25] R. Johnson and M. Schlansker, Analysis techniques for predicated code, in *Proceedings of the 29th International Symposium on Microarchitecture*, pp. 100–113, December 1996.
 - [26] J. W. Sias, W. W. Hwu, and D. I. August, Accurate and efficient predicate analysis with binary decision diagrams, in *Proceedings of 33rd An-*

- nual International Symposium on Microarchitecture, pp. 112–123, December 2000.
- [27] D. I. August, *Systematic Compilation for Predicated Execution*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 2000.
 - [28] S. A. Mahlke, *Exploiting Instruction Level Parallelism in the Presence of Conditional Branches*. PhD thesis, University of Illinois, Urbana, IL, 1995.
 - [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 International Symposium on Microarchitecture*, pp. 114–125, December 1996.
 - [30] N. J. Warter, *Modulo Scheduling with Isomorphic Control Transformations*. PhD thesis, University of Illinois, Urbana, IL, 1993.
 - [31] D. I. August, J. W. Sias, J. Puiatti, S. A. Mahlke, D. A. Connors, K. M. Crozier, and W. W. Hwu, The program decision logic approach to predicated execution, in *Proceedings of the 26th International Symposium on Computer Architecture*, pp. 208–219, May 1999.
 - [32] Intel Corporation, *IA-64 Application Developer's Architecture Guide*, May 1999.
 - [33] N. J. Warter, S. A. Mahlke, W. W. Hwu, and B. R. Rau, Reverse if-conversion, in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pp. 290–299, June 1993.
 - [34] G. R. Beck, D. W. Yen, and T. L. Anderson, The Cydra 5 minisupercomputer: Architecture and implementation, *The Journal of Supercomputing*, vol. 7, pp. 143–180, January 1993.
 - [35] J. C. Dehnert and R. A. Towle, Compiling for the Cydra 5, *The Journal of Supercomputing*, vol. 7, pp. 181–227, January 1993.
 - [36] J. C. Dehnert, P. Y. Hsu, and J. P. Bratt, Overlapped loop support in the Cydra 5, in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 26–38, April 1989.
 - [37] A. V. Someren and C. Atack, *The ARM RISC Chip, A Programmer's Guide*. Reading, MA: Addison-Wesley Publishing Company, 1994.
 - [38] D. L. Weaver and T. Germond, *The SPARC Architecture Manual*. S-PARC International, Inc., Menlo Park, CA, 1994.
 - [39] Digital Equipment Corporation, *Alpha Architecture Handbook*. Maynard, MA: Digital Equipment Corporation, 1992.

- [40] Intel Corporation, Santa Clara, CA, *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, 1997.
- [41] D. S. Blickstein, P. W. Craig, C. S. Davidson, R. N. Faiman, K. D. Glosop, R. B. Grove, S. O. Hobbs, and W. B. Noyce, The GEM optimizing compiler system, *Digital Technical Journal*, vol. 4, no. 4, pp. 121–136, 1992.
- [42] Hewlett-Packard Company, Cupertino, CA, *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 1990.
- [43] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg, The Multiflow Trace scheduling compiler, *The Journal of Supercomputing*, vol. 7, pp. 51–142, January 1993.
- [44] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers*. Reading, MA: Addison-Wesley Publishing Company, 1991.
- [45] C. McNairy and D. Soltis, Itanium 2 processor microarchitecture, *IEEE MICRO*, vol. 23, pp. 44–55, March-April 2003.
- [46] S. Eranian, Perfmon: Linux performance monitoring for IA-64. Downloadable software with documentation, <http://www.hpl.hp.com/research/linux/perfmon/>, 2003.
- [47] Y. Choi, A. Knies, L. Gerke, and T.-F. Ngai, The impact of if-conversion and branch prediction on program execution on the intel Itanium processor, in *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 182–191, IEEE Computer Society, 2001.
- [48] P. Y. Chang, E. Hao, Y. Patt, and P. P. Chang, Using predicated execution to improve the performance of a dynamically scheduled machine with speculative execution, in *Proceedings of the 1995 International Conference on Parallel Architectures and Compilation Techniques*, June 1995.
- [49] R. Kessler, The Alpha 21264 microprocessor, *IEEE Micro*, vol. 19, p. p. 24–36, March/April 1991.
- [50] P. H. Wang, H. Wang, R.-M. Kling, K. Ramakrishnan, and J. P. Shen, Register renaming and scheduling for dynamic execution of predicated code, in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 15–25, January 2001.
- [51] W. Chuang and B. Calder, Predicate prediction for efficient out-of-order execution, in *Proceedings of the 17th Annual International Conference on Supercomputing (ICS)*, pp. 183–192, ACM Press, 2003.

Chapter 6

Multipath Execution

Augustus K. Uht

University of Rhode Island

6.1	Introduction	135
6.2	Motivation and Essentials	136
6.3	Taxonomy and Characterization	137
6.4	Microarchitecture Examples	148
6.5	Issues	154
6.6	Status, Summary, and Predictions	157
	References	157

6.1 Introduction

To branch or not to branch? Why not do both?

These are the essential questions Riseman and Foster asked in their classic paper on performance and cost limits on the execution of branchy code [1]. They had been stymied by the intractability of getting any kind of reasonable performance from branch-intensive code, and sought to find out just what the performance and hardware cost limits were.

Riseman and Foster performed many simulations on typical branch- or control-flow intensive code (SPEC was not then in existence) to study *eager execution*. In this scenario, whenever the program counter (PC) comes to a branch, execution continues speculatively down both sides or *paths* of the branch. The incorrect path's results are discarded when the branch condition is evaluated (the branch is *resolved*). This operation is repeated indefinitely (and recursively), limited only by the resources of the target machine.

Riseman and Foster determined that for unlimited resources, e.g., Processing Elements (PE), *pure* eager execution would completely eliminate branch delays and result in very high performance; in their measurements they found that the harmonic mean *Instruction-Level Parallelism (ILP)* of such eagerly executed general-purpose code was about a factor of 25. That is, an eager execution machine could execute 25 single-cycle instructions in one cycle.

However, the catch-22 of the situation was that the only way to get a sizeable fraction of this high performance was to use excessive resources. This

is due to the construction of the execution *branch tree*. Starting from one unresolved branch, the number of active paths grows exponentially with the *depth* of branches encountered. The result is the well-known binary tree.

Eager execution's cost was so abysmal that it discouraged further investigations into reducing branch penalties for about a decade. Eager execution is the first known general form of multipath code execution. However, a limited form was implemented for instruction-fetching only, somewhat earlier, in 1967 in the IBM 360/91 [2].

In general, *multipath execution* is the execution of code down both paths of one or more branches, discarding incorrect results when a branch is resolved. There are many dimensions to the differences between the many types of multipath execution. This chapter explores these dimensions.

The remainder of this chapter is organized as follows. In Section 6.2 we further motivate our study of multipath execution, and present its basic concepts and terminology. Section 6.3 presents a taxonomy of the dimensions and types of multipath execution. This section also contains a description of the little multipath theory that exists, in part of subsection 6.3.1.3.1. Some of the known microarchitectural implementations of multipath execution are examined in Section 6.4. Key issues that multipath execution raises are discussed in Section 6.5. Section 6.6 suggests some areas for future multipath research, and presents our own personal multipath prognoses.

6.2 Motivation and Essentials

Most forms of multipath execution use some form of branch prediction, presented earlier in this book. In typical speculative execution a branch's final state, taken or not-taken, is predicted when the branch is encountered. Then execution proceeds *only* down the predicted path. This can be repeated for any number of dynamic branches, constrained only by machine resources. This is called single path or *unipath* execution; see Figure 6.1 tree (a).

Multipath execution differs from unipath execution in that *both* paths of a branch may be speculatively executed simultaneously. Thus, multipath execution is a branch predictor's way of hedging its bets: if the prediction is wrong, some or all of the other path has already been executed, so the machine can use the not-predicted path's results immediately. Thus, any branch *misprediction penalty* normally occurring in unipath execution is either reduced or eliminated, improving performance. Other branch-penalty reduction methods such as predication, discussed in a prior chapter, can also be combined with multipath execution, further improving performance.

Speculative code can consist of both multipath and unipath sections; that is, execution may proceed down one or both paths of a branch, and this situation

may change over time. This may sound odd, but there are sound theoretical and practical reasons for doing this, as we will see.

Confidence estimation [3] of a branch prediction has played a key role in some multipath implementations. For example, a branch might only be eagerly executed (multipath execution) if its predictions are frequently wrong, that is, when there is little confidence in their accuracy; otherwise, the branch only executes down the predicted path (unipath execution).

We now briefly return to our discussion of multipath's performance potential; the latter can generally be summarized as the amount of *Instruction-Level Parallelism (ILP)* present in the typical code to be executed by a target machine. How well a microarchitecture exploits this potential parallelism is the number of *Instructions Per Cycle (IPC)* the microarchitecture actually executes and yields from the potential ILP. There have been numerous ILP limit studies over the years, e.g. [1, 4], demonstrating very high levels of ILP if the ill effects of control flow can be alleviated [5]. ILPs in the tens or even hundreds have been found.

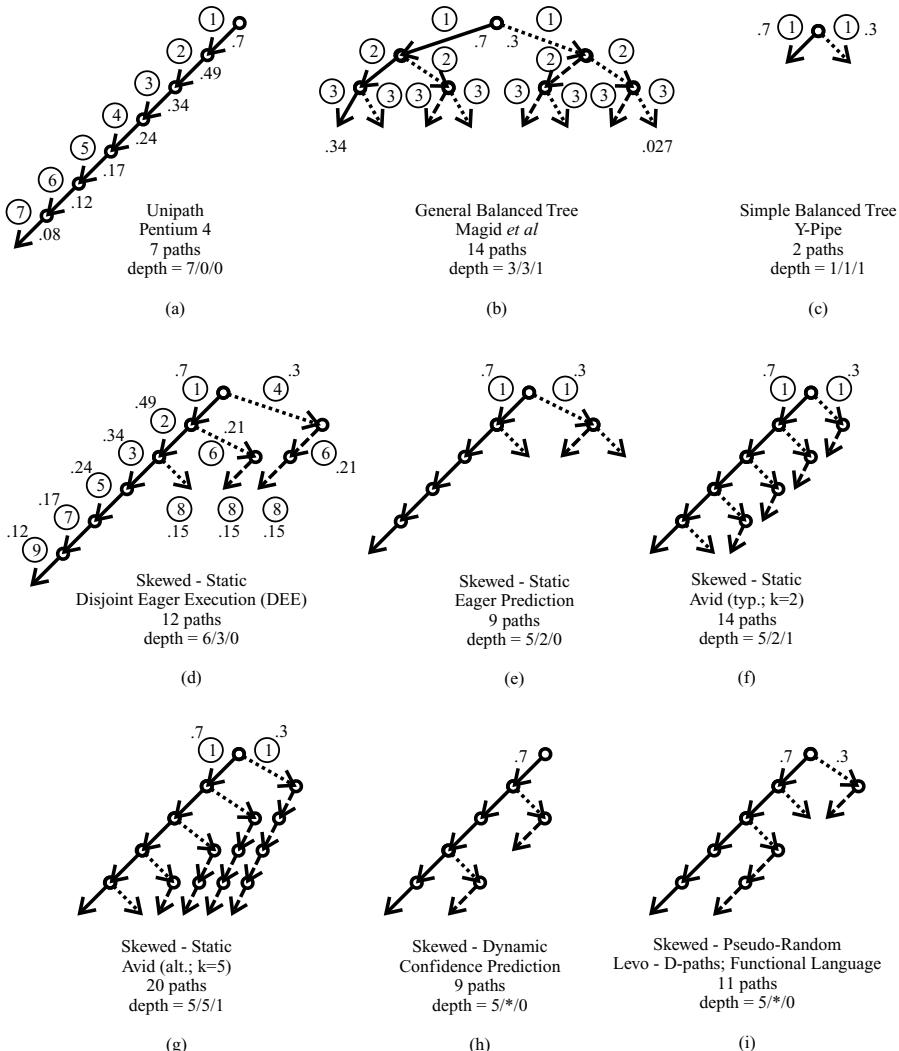
A word or two on terminology. A *branch path* consists of the dynamic code between two conditional branches with no intervening conditional branches. This is similar to but different from the more common *basic block*. They are usually about the same size.

When a branch is *forked* execution proceeds down both paths of the branch at the same time. When the two paths begin execution at different times, then the later path is *spawned* from the branch. Lastly, a branch is *split* if either the branch is forked or spawned; it includes both possibilities.

6.3 Taxonomy and Characterization

We now present the various taxonomical dimensions of multipath execution and analyze the characteristics of each dimension's attributes. Sample multipath branch trees are shown in Figure 6.1. The taxonomy is summarized in Table 6.1; it includes example machines and methods and their classifications, as well as cross-references to both corresponding text sections and to representative trees in Figure 6.1. Referring to the Figure, a tree's *Mainline* path is the complete group of predicted paths starting at the root of the tree; in other words, it is the most likely path of execution. A group of paths split off of the Mainline path via one not-predicted branch is called a *Sideline* path.

In general, multipath execution can be applied to all types of branches: conditional, unconditional, calls, returns, indexed, indirect, etc. In practice, only two-way conditional branches are usually split. The other kinds of branches can either be treated as special cases of conditional branches or as unsplittable. N-way branches can sometimes be split by using the compiler to convert them

**FIGURE 6.1:** Some possible branch trees.

Notes: All of the trees are multipath trees except (a).

The text beneath a tree indicates the following:

- Line 1 - Tree name/classification
- Line 2 - Tree implementation example: machine or method
- Line 3 - Number of branch paths in the sample tree, which will vary with a machine's design or operation
- Line 4 - Depth three-tuple - See the right side of the **Legend**.
These, too, may vary. (*: dynamically variable).

TABLE 6.1: Multipath taxonomy and machine/model characteristics

Section:											
Machine/Study:											
Dimension/ Section Characteristic											
6.3.1	Tree geometry	Riseman et al. [1], 1972 IBM 360/91 [2], 1967 I-Fetch-DEE [6], 2003	6.1 6.4.1 -	Y-Pipe [7], 1992 Eag. Prdct. [8], 1994 Sel. Dual [9], 1996 Lim. Dual [10], 1997 TMEF [11], 1998 PolyPath [12], 1998 PrincePath [13], 1998	6.4.1.1 - 6.3.1.3.3 6.3.1.3.3 6.3.4.2 6.4.1.2 6.4.1.3	f b d h i g h ?					?
6.3.1.2	<i>Balanced tree</i>	✓ ✓	✓		✓						✓
6.3.1.3	<i>Skewed</i>										
6.3.1.3.1	Theory-Based										
6.3.1.3.2	Heuristic:Static										
6.3.1.3.3	Heuristic:Dynamic confidence-based										
6.3.1.3.4	Pseudo-Random										
6.3.2	Path ID	?						?	?		?
6.3.2.1	<i>Binary Tag</i>										
6.3.2.1.1	Taken/Not-T.		✓								
6.3.2.1.2	Predicted/Not-P.			✓	✓	✓	✓	✓	✓		
6.3.2.2	<i>Implied</i>	✓	✓	✓	✓	✓	✓	✓	✓		
6.3.2.3	<i>Other ID</i>										
6.3.2.3.1	Thread					✓					
6.3.2.3.2	Process					✓					
6.3.3	Phases of Operation										
6.3.3.1	<i>Pipe Stage</i>								?		
	<i>Split/Live/Prune/All</i>										
	Pre-Fetch							P			
	I-Fetch	S A A	S S S S S S S	S S S	A S						
	Decode	L	P L ? L L L	L L A L	L A P A A S						
	Execute	P	P P P P P L	P A P A P P	P A A S P						
	D-cache			P	P						
6.3.3.2	<i>Disjoint</i>	✓	✓		✓ ✓ ✓	?				✓	
6.3.4	Granularity										
6.3.4.1	<i>Fine-Instruction</i>	✓ ✓ ✓	✓ ✓ ✓ ✓	✓ ✓ ✓ ✓	✓ ✓ ✓ ✓	✓ ✓ ✓	✓ ✓ ✓				
6.3.4.2	<i>Thread</i>			✓				✓			
6.3.4.3	<i>Coarse-Procedure</i>										
6.3.4.4	<i>Process</i>						✓	✓			
6.3.4.5	<i>Function</i>										
6.3.5	Predication					✓	✓	✓	✓	✓	
6.3.6	Data Speculation					✓	✓	✓	✓	✓	
6.3.7	Compiler-Assisted				✓			✓	✓	✓	
	Scheduling							?	✓	✓	
	Confidence Hints							?	✓	✓	

to combinations of two-way conditional branches. An example is the binary code implementation of `switch` statements.

6.3.1 Branch Tree Geometry

Perhaps the greatest differentiator of multipath execution schemes is the geometry of the ‘live’ branch tree. This geometry is formed by the pattern of unresolved conditional branches in the machine. Resolved conditional branches can be present in the tree, but they do not delineate branch paths in the tree; only unresolved conditional branches do that. Many branches can be live at the same time; exactly how many depends on the particular machine implementing the tree, and the code dynamics.

Branch trees fall into one of three broad categories: *unipath*, *balanced multipath*, and *skewed multipath*. In the following discussion refer to Figure 6.1 for pictures of various styles of branch trees.

In the Figure an uncircled number next to a branch path is the path’s *cumulative probability (cp)* of execution, formed by multiplying the corresponding branch’s *Branch Prediction Accuracy (BPA)* by the cp’s of other branch paths between it and the root of the tree. For the sake of illustration, a BPA of 70% was used for all of the branches; a typical BPA is much higher, say 90%.

A circled number next to a branch path in the figure is the branch path’s priority for resources. A lower number is a higher priority.

6.3.1.1 Unipath Tree

The unipath (*Single Path* [15]) tree is both a reference point and is frequently used as part of multipath trees. Unipath execution is just simple branch prediction: when a branch is encountered, its direction is predicted and execution proceeds only down the predicted path. This process is usually repeated.

The net result of the latter effect is to decrease the usefulness of instructions executed in lower paths of the tree (later in time). This is indicated by the rapidly decreasing cumulative probabilities of the branch paths as the depth of prediction or speculation increases.

6.3.1.2 Balanced Multipath Tree

Riseman and Foster’s [1] eager execution tree, discussed in Section 6.1, is the canonical example of a balanced branch tree. Every branch encountered is *forked* to its two paths, one corresponding to the taken direction of the branch, and the other path to the not-taken direction of the branch.

The major disadvantages of a balanced tree is its exponential growth in required execution resources as the branch depth increases, and its relatively low performance. The latter is due to the small cumulative probabilities of most of the tree’s branch paths. Their results are not likely to be needed.

An advantage of a balanced tree is that since branches are always forked, it does not use a branch predictor. This is true only for balanced trees.

An example machine using a single-level balanced tree is the Y-Pipe machine [7]; see tree (c) in Figure 6.1. Only one branch is forked at a time. An example of a multi-level balanced tree implementation is Magid et al.’s machine [14].

6.3.1.3 Skewed Multipath Trees

The latter two sections discussed the two extremes of branch tree construction. Both unipath execution and balanced trees have serious problems in high-performance microarchitectures: they both exhibit diminishing returns with increasing available resources.

Skewed trees are the middle ground of multipath trees; they make up the bulk of the trees realistically considered for multipath execution. We classify a skewed tree into one of the following categories: *Theory-Based*, *Heuristic:Static*, *Heuristic:Dynamic*, or *Pseudo-Random*.

6.3.1.3.1 Theory-Based A theory-based machine’s tree construction is based on some mathematically-based principle, e.g., a tree-construction theorem minimizing resource usage.

An example is the *dynamic DEE tree*. The idea is to assign branch path *resources*, such as Processing Elements (PE), to the most likely-to-be-executed branch paths first (i.e., those paths with the greatest cumulative probabilities). (An example of this construction, but with constant probabilities, is given in the next section.) This assignment was proven to be theoretically optimal, giving the best performance for constrained resources [15], where the quantity of resources is directly proportional to the number of branch paths in the tree.

DEE is also shown to be theoretically optimal or close to optimal in related situations in [26]. Further, Gaysinsky et al. [27] considered block fetching in file systems by looking at the theoretical implications of combined pipelining and caching. Depending on the assumptions made, DEE is shown to be either a theoretically-optimal or at least a near-optimal scheduling algorithm.

Theory-based machines tend to have dynamically-changing branch trees; see Figure 6.1 tree (h). The advantage of these machines is that they may operate beautifully [16], but the empirically-observed disadvantage is that they tend to be costly and complex.

6.3.1.3.2 Heuristic:Static Some possible heuristic:static configurations are shown in Figure 6.1, branch trees (d)-(g). These trees’ shapes are based on some heuristic and fixed at machine design-time (static).

The *static DEE tree* (d) is a heuristic of the theoretically-optimal dynamic DEE tree discussed in the last section. The static DEE tree is a dynamic DEE tree having all of the branches’ BPAs equal. In operation, at the root branch the predicted path ($0.7 > 0.3$) is assigned the first resources after the branch

has been predicted; the cp of path one is the largest of the outstanding paths. Likewise, the next branch is predicted yielding a predicted-path having a cp of $0.7 * 0.7 = 0.49$ and a not-predicted path with a cp of $0.7 * 0.3 = 0.21$. Hence, the 0.49 path is the path with the second largest cp, so it gets the next set of resources, as indicated by its circled ‘2’. This process continues until we have exhausted the available resources. It becomes interesting after the fourth branch has been predicted, yielding the 0.24 cp predicted path. Now, in this case $0.24 < 0.3$ of the not-predicted path of the first branch, so the latter is assigned resources next, as indicated by the circled ‘4’. This is the time-‘disjoint’ resource assignment of disjoint eager execution.

In a real dynamic DEE tree the BPA of one branch is likely to be different from the BPA of another, so instead of the tree having the nice pseudo-symmetric shape shown in (d), a dynamic version of the tree might look something closer to tree (h). Tree (d) is suboptimal, BUT, it has the nice heuristic property that with the statically-fixed cp’s the shape of the tree can be fixed at machine design time, and no dynamic cp calculations need be done. These calculations would otherwise entail many costly and slow multiplications whenever the tree is dynamically modified.

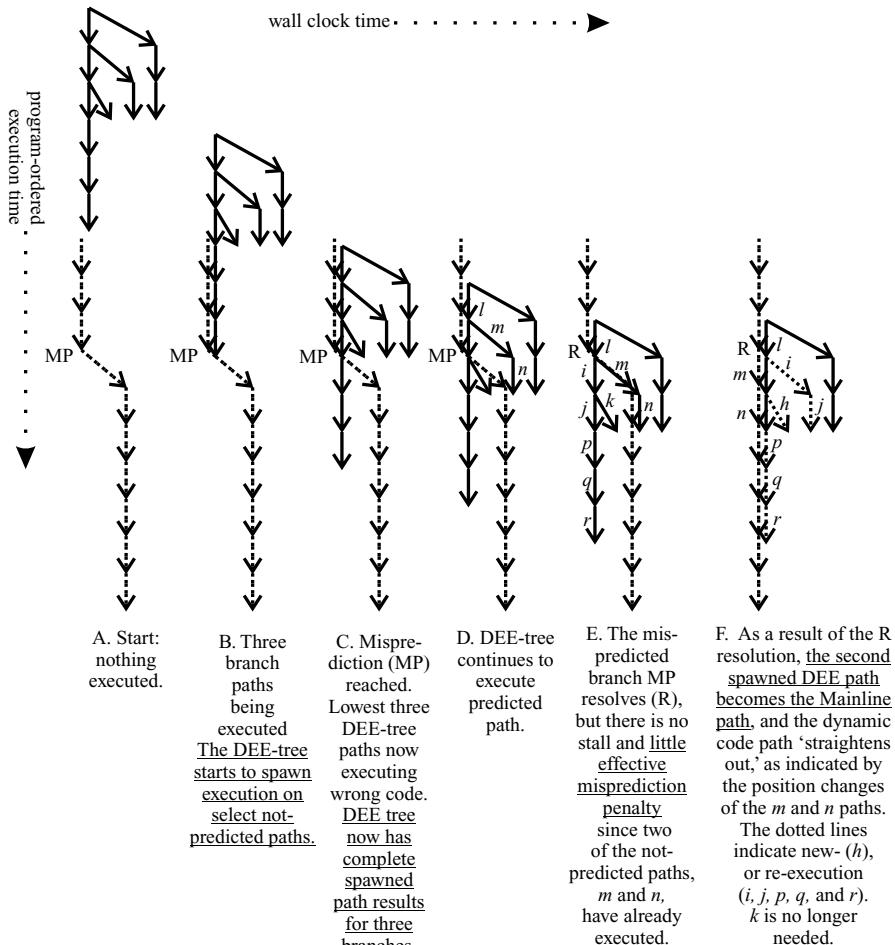
The ‘statically-fixed shape’ attribute is what makes heuristic:static trees so attractive, regardless of their actual shape. Both cost and time are saved, and operation is simplified. The static DEE tree of (d) is nice in particular since it is an approximation of a theoretically-optimal tree, and hence gives good performance [15]. An example of static DEE tree code execution is shown in Figure 6.2. Especially note the lack of a significant branch misprediction penalty.

6.3.1.3.3 Heuristic:Dynamic In this category a branch tree is formed dynamically by some heuristic; see Figure 6.1 tree (h). This is perhaps the most common form of multipath execution.

The most common operational example [9, 10, 12] is confidence-based forking. When a machine fetches a branch, not only is it predicted but the quality of the prediction is checked with a confidence estimator; see Section 6.2. If the branch confidence is low, the branch is forked and execution proceeds down both paths. Otherwise, unipath operation is assumed and the branch is executed in the predicted direction.

Either one [9, 10] or multiple [12] forked-branches can be live at the same time. In the former case, tree (h) would only show one forked branch. Single forked-branch machines are simpler and less costly than the multiple forked ones, but give lower performance.

6.3.1.3.4 Pseudo-Random Trees Pseudo-random trees are a by-product of physical constraints. For example, Levo’s [17] branch tree is a physically-based heuristic on top of the static DEE tree heuristic; see Figure 6.1 tree (i) and Section 6.4.2.3. The branch tree then has a less obvious structure.



NOTES:

1. Each arrow is a branch path.
2. Branch targets are at the arrowheads.
3. Solid and dotted arrows comprise the static DEE tree.
4. Dashed arrows comprise the dynamic code execution.
5. Vertical arrows (pointed down) are *predicted* paths.
6. Angled arrows (to the right) are *not-predicted* paths.

MORE NOTES:

- a. Execution occurs only on the solid and dotted paths, that is, the DEE tree paths.
- b. Correct execution occurs only where a solid (DEE tree) path is 'over' a dashed (actual) code execution path.
- c. The *i, j, p, q*, and *r* results are corrected as needed, also reducing the effective misprediction penalty.

7. A branch prediction accuracy of about 70% assumed for illustration purposes.
8. The dynamic code has one misprediction.
9. At F, DEE is two branch paths ahead in execution after the misprediction, as compared to a unipath execution. (Pipeline effects are not included.)

FIGURE 6.2: Static DEE tree [15] operating on a dynamic code stream.

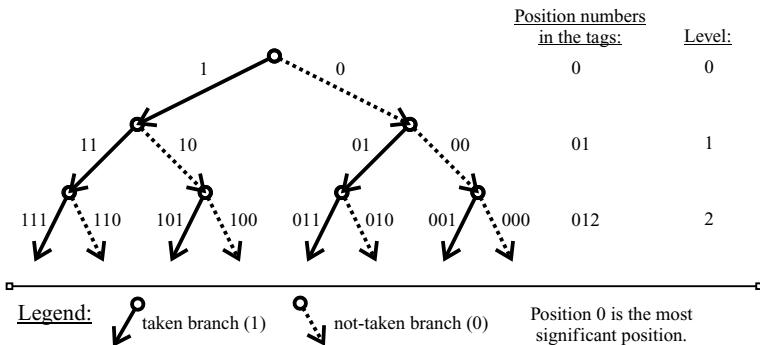


FIGURE 6.3: Typical branch path ID numbering scheme.

Since pseudo-random trees are tuned to a specific microarchitecture they may provide high performance with ease of operation. However, it may be hard to analyze such a tree to estimate a paper machine's performance.

6.3.2 Branch Path/Instruction ID

6.3.2.1 Binary Path Tag

In multipath execution a tag is used to uniquely identify an instruction's tree branch-path of origin (where it came from). This tag differentiates versions of the same static instruction and is used for pruning. Figure 6.3 shows the most common tag numbering schemes' foundation. Normally, branch paths are indicated by a *path ID tag*.

The path ID tags use the path labels shown in the figure, one ID position per level. These tags use the fewest number of positions needed to identify a path, i.e., one position for level 0, etc. For ease of use there may be as many positions as levels. A two-bit per position encoding can be used to distinguish among three possible cases: branch taken, branch not-taken, and branch invalid ('X'), e.g., [12]. The 'invalid' case occurs if a branch has not yet been predicted or has been resolved. From the figure, an example of this 3-valued tag using all positions would be: '0XX' for the level 0 not-taken path.

For a pruning example, assume that the tree of Figure 6.3 is indicative of the current live branch paths throughout the processor. Now say the second branch at the top of level 1 is mispredicted not-taken. Then the path ID tag corresponding to the mispredicted path tag, '00X', is broadcast throughout the processor. Instructions compare their path ID tags to the broadcast tag, ignoring the 'X' positions. All instructions having matching values for level positions 0 and 1, '00*', know that they are on the mispredicted path or its descendants' paths and are pruned (their results are discarded and their resources re-allocated). Other paths' instructions continue execution.

Whenever the root branch resolves, all of the tags in the machine are shifted left by one position. This frees up ID space on the right of the tags for new branches on the new, lowest level. The elegance and ease of operation of the overall scheme explains why it is so widely used.

6.3.2.1.1 Taken/Not-Taken Path ID's typically directly indicate branch execution direction. This can help with multipath operation.

6.3.2.1.2 Predicted/Not-Predicted Considering branches as predicted or not-predicted abstracts away the actual direction of a branch, making such a scheme especially suitable for theoretical analysis and general multipath execution modeling, e.g., Figure 6.1.

6.3.2.2 Implied

Some machines use ID tags to indicate other speculative entities in the machine; thus, actual branch path ID's are *implied*. For example, Levo uses ID tags for speculative groups of instructions whether they share the same branch ancestors or not; see Section 6.4.2.3.

The decision to use explicit or implicit tags is mainly dependent on the details of a specific microarchitecture.

6.3.2.3 Other ID

The basic branch path ID tag can be augmented with tags for other purposes, e.g., to exploit other levels of parallelism. Different threads and processes may also be active in a machine at the same time, share resources, and thus append thread and process ID's to instruction path ID tags to completely differentiate instruction instances.

6.3.2.3.1 Thread The actual construction of a thread ID for a multipath machine is implementation dependent, but it is not hard [11, 18].

6.3.2.3.2 Process Process ID construction is also not hard [18]. It can be as simple as appending the process ID or equivalent from the Translation Look-Aside Buffer (TLB) to the path ID tag. In existing microarchitectures the process ID may already be part of an instruction tag.

6.3.3 Phases of Operation

In this section we mainly consider the three phases of multipath operation, namely when/where in a machine: a branch is *split*, multiple paths are *live* and operate, and path *pruning* is performed after a branch resolves.

6.3.3.1 Pipeline Stage(s)

Table 6.1 shows common pipeline stages and where multipaths Split, are Live, and are Pruned for the example machines. Note that a particular machine may have a multipath phase be active in more than one stage, be active throughout a stage, or be active just at the beginning or end of a stage.

Phase location affects the benefits a machine gets from multipath operation. Intuitively, the longer the multipaths stay speculative, the better the performance but the greater the cost and complexity of operation. However, the precise relationship of performance to phase operation or location is unknown.

In the table the execution stage is assumed to include memory loads, speculative or not. Stores in the execution stage are always speculative.

Speculative memory caches [28] allow multipath execution to go about as far into the memory system as possible, at least with near-in technology. However, this may be undesirable, since every version of a store may be sent to the speculative cache, increasing data bandwidth requirements.

Pruning can be done in a single location or be distributed among the sections of a machine, depending on the details of the machine. Classically-based (superscalar) machines such as PolyPath [12] often perform pruning in one location; this keeps the operation simple although it may create a bottleneck and lengthen cycle time. Non-classical machines such as Kin [18] may distribute pruning throughout the execution stage, avoiding a bottleneck.

6.3.3.2 Disjoint Splitting

The vast majority of multipath machines do not split paths at different times, i.e., they are not disjoint (they don't spawn paths), but always fork paths. Forking is a straightforward way to implement splitting.

However, DEE theory tells us that forking is usually suboptimal, and that spawning the second path at a different time may be preferable; it works for Levo (see Section 6.4.2.3). In the final analysis, whether or not spawning is worth doing likely depends on an individual machine's characteristics.

6.3.4 Granularity

6.3.4.1 Fine-Instruction

The vast majority of past and present multipath schemes operate on individual instructions. This is quite natural, since the basic enabler of multipath execution is the branch instruction, a fine-grain element itself.

6.3.4.2 Thread

Early on, multithreaded-machine researchers realized that unused hardware thread-resources could, with a little extra effort, also be used for speculatively executing a Sideline path in other threads [11], that is, realize multipath

execution. A mixture of threads from both the same and different programs can be active simultaneously, so that both throughput and single-program performance improve.

6.3.4.3 Coarse-Procedure

I know of only one study or machine using this level of granularity, that of Hordijk and Corporaal [24]. Briefly, they used a compiler to partition a program into tasks, where each task was a procedure. A procedure invocation corresponds to how we have been viewing branch splitting. They simulated the resulting code on a multiprocessor model, using a DEE construct, and obtained an ILP of about 65. To achieve this the DEE tree only needed to go three levels deep. (Presumably, this means three nested procedure calls.) It was primarily a limit study. We will return to this work in Section 6.4.3.2. It appears that there is a lot of potential here, and that it has implications for multipath execution on Chip Multiprocessors (CMP).

6.3.4.4 Process

Not much work has been done on multipath execution at the process level, to my knowledge. The closest work is the Kin processor [18], targeted to execute instructions from different processes simultaneously, but this is not necessarily executing processes down multiple paths.

6.3.4.5 Function

Functional-language and logic-language machines can benefit from a type of multipath execution called *eager evaluation*. (Unfortunately, over the years ‘eager execution’ and ‘eager evaluation’ have often been used interchangeably or with their opposite meanings. We use the current seemingly-accepted definitions; see Section 6.4.4.) A function’s evaluation is the basic unit of granularity for such machines. Tubella and González [25] show that judicious use of eager evaluation can lead to substantial performance gains.

6.3.5 With Predication

Predication exhibits synergistic gains with DEE [15]. Intuitively, predication should add to the cost and complexity of a machine. However, it could be argued that Levo’s implementation of hardware-based predication is actually simpler and cheaper than a more classical approach to typical branch execution. Predication is uncommon in hardware-based multipath machines, while it is common in software-based multipath methods; see [29].

6.3.6 With Data Speculation

Levo uses a data speculation/multipath combination. While little performance gain has been realized to date, the combination has great potential; IPC speedups of a factor of 2-3 may be possible [30].

6.3.7 Compiler-Assisted

Little work has been done here. The Trace Scheduling-2 multipath compiler was proposed for VLIW machines, but the compiler complexity proved to be daunting [29]. The proposed DanSoft machine [21] uses a simple mechanism of statically-generated *confidence hints* from the compiler to help the processor determine whether or not to dynamically fork a branch for dual-path I-Fetch.

6.4 Microarchitecture Examples

The examples are categorized into four sections. *Hardware: Classically Based* concentrates on multipath machines based on typical superscalar cores. Although they are easy to build, they are not likely to provide substantial performance gains. (“If you want big changes, you have to change a lot.” - after Prof. Donald Hartill.) The *Hardware: Non-Classically Based* section looks at many multipath machines with radically new processor cores. In the *Multiprocessors* section we examine several approaches to multipath multiprocessors. We end with something completely different: *Functional or Logic Language Machines*.

6.4.1 Hardware: Classically Based

The IBM 360/91 [2] is an early example of multipath execution; it kept dual paths for instruction fetch. While multipath execution restricted to instruction fetching or pre-fetching continues to be used in some current microarchitectures, this is done usually to keep such classical superscalar pipelines fed with instructions; thus, no big performance wins are likely.

However, an exception to this is the Y-Pipe machine [7]. We also discuss two high-end multipath machines based on superscalar cores.

6.4.1.1 Y-Pipe

The Y-Pipe machine [7] is one of the earliest and simplest of modern multipath machines. Y-Pipe assumes a simple five-stage pipeline (IF-ID-EX-M-WB), and the ability to fetch two instructions in one cycle. Y-Pipe achieves 0 cycle (no) branch penalty without a branch predictor as follows.

The code is scheduled so that every conditional branch will be dynamically immediately preceded by its associated compare instruction, and not another conditional branch. At run time, every conditional branch is forked at instruction-fetch. Its dual paths stay live until the end of the decode stage, when the branch's compare instruction has just executed, the branch is resolved, and the wrong path is pruned. Thus, there is no branch penalty! And without a branch predictor!

Y-Pipe may be ideal for embedded applications with its low cost, likely low-power, relatively high performance and deterministic branch penalty (0 cycles). The latter is a great attraction for embedded systems' real-time programming. The dual instruction-fetch per cycle requirement is less of an issue in an embedded environment.

6.4.1.2 Polypath

The advanced Polypath machine is an implementation of *Selective Eager Execution (SEE)* [12]. Polypath can simultaneously follow many more than two paths at one time. Branches can resolve out-of-order. Polypath's front-end uses a JRS confidence estimator [3] to make the fork/no-fork decision. While predictable branches stay unipath, unpredictable branches (low confidence) are forked and executed simultaneously.

Polypath uses a *context tag* to show instruction's branch history; this tag is a version of the path ID labeling scheme of Section 6.3.2.1. Every instruction proceeds through the execution stage with its tag. Thus, typical register operation structures can be used, suitably augmented to store context tags.

Functional units send forked branch resolution information to the instruction window on *branch resolution buses*, one bus for each resolution desired per cycle. If a mispredicted branch's path tag is received by the window, the mispredicted path and its descendants are pruned and their results squashed. Mispredicted low-confidence multipath branches exhibit no branch penalty since they are forked. Mispredicted high-confidence unipath branches have the same (high) penalty as a pure unipath processor.

The added hardware is somewhat costly and complex, though not a large fraction of the overall cost of the core. The added performance is modest. In simulations fetch bandwidth was kept the same for both uni- and multipath machines, perfect caches (no misses) and perfect memory disambiguation were assumed, and the processor was 8-way issue, with many functional units. The overall performance improvement was about 14%, from 3.85 to 4.4 IPC, from a unipath to a multipath model, respectively. The low accuracy common to confidence estimators likely contributed to wasted resources or missed forking opportunities and hence low performance gain. I-Fetch memory bandwidth requirements increased substantially.

6.4.1.3 PrincePath

PrincePath [13] [our name, for “Princeton multiPath”] is similar to the Polypath machine. PrincePath also uses confidence estimation [3] to select which paths to fork at I-Fetch time, and allows multiple forked paths to be live at any given time. The equivalent to context tags is also used. As is typical equal priority is given to the forked paths, that is, disjoint splitting is not used; see Section 6.3.3.2.

Unlike the Polypath machine, PrincePath uses a global ARB (Address Resolution Buffer) [28] in front of the memory system to hold speculative stores suitably tagged. Therefore, PrincePath is able to keep multiple paths alive for probably the longest time realistically possible. When pruning occurs if the resolving branch is the earliest unresolved branch in the machine, the wrong path store versions in the ARB are discarded and the non-speculative store is committed to the remainder of the memory system.

The simulations’ benchmarks were similar to those used in the PolyPath simulations. The PrincePath results indicated that branches with a misprediction rate greater than 35% should be forked, with lower rate branches unsplit. PrincePath achieved similar speedups to PolyPath: about 15.5% over the baseline case. PrincePath is similarly likely constrained by classical super-scalar assumptions. Speculative caching does not seem to me to help; this may be due to short-lived speculative paths. PrincePath did not experience a substantial increase in fetch bandwidth requirements.

6.4.2 Hardware: Non-Classically Based

6.4.2.1 Magid et al.

Little was published on this machine [14]. We know that it is a proposed implementation of the original Riseman and Foster eager execution model [1], using a multi-level balanced tree. Since the pure eager execution model’s required resources are so large for so little gain, the Magid machine would not be likely to realize a high IPC. It may have been the first machine to use the elegant binary path ID scheme described earlier in Section 6.3.2.1.

6.4.2.2 Condé-2-Based - Early DEE

The Condé-2 -based static DEE tree implementation [15] was an early version of the Levo machine. Since this early version was costly, not scalable, cumbersome to operate, and has been superseded by the current Levo, we will not dwell on it. The machine was never simulated, although the static DEE tree concept itself was [15]. DEE with MCD gave an ILP of about 30.

6.4.2.3 Levo - Current DEE

It is my experience that it is very hard to implement a realistic multipath processor that follows the DEE theory closely. Thus, the Levo [17] imple-

mentation of multipath execution is a multipath heuristic on top of another multipath heuristic; to wit, the Levo microarchitecture uses a modified version of the standard static DEE tree, in particular a version that meshes well with the rest of the Levo microarchitecture. Even with this double DEE approximation, Levo benefits greatly from multipath execution. Levo is described in much greater detail in a later chapter of this book, so we will only describe its broad characteristics here.

Figure 6.1 tree (i) is an approximation of Levo’s branch tree. It is pseudo-random because the Sideline path lengths are not ordered by their size, as is the case with the static DEE tree, tree (d). However, unlike tree (h), the unresolved split branches’ predicted branch paths are all normally adjacent.

Unlike Levo, in most multipath machines a misprediction of a split branch causes all of the instructions following the Sideline path’s instructions to be flushed, and their results squashed. In Levo the Mainline instructions after the mispredicted branch’s spawned path are not flushed nor are their results discarded. Instead the spawned path broadcasts its result data forward to cause the re-execution of only directly and indirectly dependent instructions.

Levo uses implied path IDs; sections of instructions have an ID corresponding to either a part of the Mainline path or an entire Sideline path. Typically, only 8 partial-Mainline and 8 Sideline-path are present, so the implied ID tags are just a few bits long. Levo gave about 5 IPC with detailed simulations using realistic assumptions, with the potential to realize IPCs in the 10’s.

6.4.2.4 Adaptive Branch Trees

The ABT machine proposal [16] is an implementation of the optimal dynamic DEE tree. The machine dynamically computes and updates the cumulative probabilities of all of the branch paths in its window, assigning execution resources to the most likely-to-be-used paths. The ABT machine does not use reduced control dependencies such as either the Minimal Control Dependencies of [15], or typical software or hardware-base predication.

ABT realization is difficult because of both the massive amount of computation necessary to keep the cumulative probabilities updated, and the difficulty of the calculations needed to dynamically sort the branch paths by their cumulative probabilities (to determine which path gets resources).

However, the ABT simulations partially verified the initial DEE work [15]. ABT with an optimal tree performed very well and better than the corresponding suboptimal static DEE tree. Also, ABT didn’t perform substantially better, indicating the usefulness of the static DEE tree heuristic.

6.4.2.5 Kin and Avid Execution

Kol devised and studied a new form of multipath execution called *Avid Execution*, to be implemented in a novel asynchronous machine called ‘Kin’ [18]. A typical Avid branch tree model is shown in Figure 6.1 tree (f). The two key features of this tree are: every conditional branch is forked, and the length

of each forked path is the same for every branch (with the caveat that forked paths cannot exist past the end of the Mainline path).

Kol defines a parameter k equal to the length in branch paths of a forked path. In tree (f) $k = 2$. Tree (g) is a special Avid case in which the forked path depth is equal to the Mainline path length; in this case $k = 5$. We will concentrate on the most useful configuration, tree (f). Kin's multipath operation is similar to others previously mentioned, except that the multipath operation phases are organized a bit differently and are spread throughout most of the machine; see the ‘Avid/Kin’ column in Table 6.1.

Limit-type simulations on some of the SPECint95 benchmarks including `gcc` indicated speedups of about a factor of two for branch trees with $k = 1$ over $k = 0$. A $k = 2$ tree performed about the same as a $k = 1$ tree. These simulations used a branch predictor with about a 90% BPA. Also interesting, I-Fetch requirements actually DECREASED for trees with $k = 0$ to $k = 1$.

The results are quite remarkable: little multipath execution needs to be done for big gains. The DEE simulations gave similar results, and also showed that branches tended to resolve at or close to the root of the DEE tree. Combining the Avid and DEE results seems to indicate that less multipath execution than either Avid or DEE used is needed for good performance. Kol also suggests that a confidence estimator could be used to dynamically vary the forked path length to further improve performance and resource utilization; this was not simulated. Kol also states that the length could even go to zero; I must disagree with this. At some point, every conditional branch should be split. We’ll come back to these results and analyses in Section 6.5.4.

6.4.2.6 Dynamic Conditional Execution (DCE)

In general-purpose code simulations dos Santos et al. [19] found that most mispredictions are in *short* branches (little code between the branch and its target) and most branches are short branches. They argued that most short branches can fit in a reasonably sized cache line. So when a proposed machine fetches a line containing a short branch, the machine is also effectively fetching down both paths of the branch automatically. Multipath execution of these paths ensues. This is multipath execution with just about no extra I-Fetch bandwidth requirements (maybe 0), and on the cheap. Some SPECint simulations, including `gcc`, indicate that about a factor of two performance gain over a standard superscalar machine may be possible. The proposed machine is evolving, and the gains may increase. This idea is clever and most promising.

6.4.3 Multiprocessors

6.4.3.1 Chip MultiProcessors (CMP)

Chidester et al. [22] proposed a fine-grain multipath CMP containing eight specialized processors. Compiler support is needed to insert special instruc-

tions for forking and other information into the machine’s programs. A mapping approach that worked well used one processor for the Mainline path and the other processors for Sideline paths. Most of the performance gain was realized with a small degree of forking. Some of the SPECint benchmarks, including `gcc`, were simulated on the CMP; the average performance gain was 12.7%. The researchers found that the main inhibitor of performance was the on-chip interconnect’s effect on the transmission latency of intermediate results from one processor to another.

In another approach, Sundaramoorthy et al. [23] proposed a CMP version of confidence-estimation based dual-path execution. Two processors execute the same program, just a bit differently: each processor executes a different path of a low-confidence branch. When the branch resolves, only control-flow and data-flow dependent state are transferred from the correct path processor to the incorrect path processor. The transfer enables the processor on the incorrect path to correct its results and to catch up to the correct path’s processor, ideally before the latter reaches the next low-confidence branch. While the design is elegant, the negative effect of the interconnect latency resulted in a small performance gain, about 5% on `gcc`.

6.4.3.2 Coarse-Grain

We now return to Hordijk and Corporaal’s study [24], first examined in Section 6.3.4.3. To briefly review: they simulated a coarse-grain multiprocessor consisting of individual ILP processors. Procedure calls were the basic unit of granularity. The compiler did the partitioning. DEE execution was used.

Minimal Control Dependencies (MCD) (essentially predication) were used, but it did not provide much benefit. This is in contrast to our fine-granularity study [15] in which both MCD and DEE were needed to get the best performance. This raises interesting questions for future research.

In the study, the basic assumption was that resources were infinite. (However, they found that the multipath level only had to be three procedures deep to get most of the performance gain; so resources were in some sense limited.) Data dependencies were honored, only RAW for register operands, and RAW, WAR, and WAW for memory operands. Perfect disambiguation was assumed. A one-cycle communications latency was assumed; while this is a bit optimistic for a multi-chip multiprocessor, it is certainly in the realm of the possible for a CMP. Many SPECint benchmarks were simulated.

An oracle gave an average ILP of 66.7. The average ILP of the DEE approach without MCD was 63.0! `cpp` also did well with an ILP of 26.2.

In summary, this is a limit study with important significance for other multipath techniques, especially CMP. We have seen that most CMP approaches are aimed at the fine-grain or instruction level. This places great constraints on multipath CMPs’ design and operation. The Hordijk results just presented argue strongly for coarse-grain multipath CMPs. This is intuitive: one wants as little communications as possible between the processors of a CMP

at opposite ends of a chip. Much more work needs to be done to further characterize multipath CMP operation, especially at the coarse-grain level.

6.4.4 Functional or Logic Language Machines

Such machines typically execute programs using *lazy evaluation*, in which code is only executed non-speculatively, and only when its results are needed. This is a very safe approach, but it severely limits performance.

The alternative is eager evaluation, in which as many paths as possible execute code simultaneously, similarly to unlimited multipath execution on an imperative machine. The major problem with eager evaluation is that in operation the machine can completely run out of memory, then be unable to execute down the lazy (Mainline) path, and deadlock occurs.

Tubella and González [25] proposed the parallel execution of Prolog via a compromise between lazy and eager evaluation. Breadth-first execution is done until resources are exhausted, then the mode of execution goes back to depth-first; when resources again become available, eager evaluation resumes. The tree shape can be dynamically changed for different program phases' execution. No compiler or interpreter modifications are needed. An SPMD machine model was used in the simulations. Speedups of x4-x63 over sequential execution were demonstrated for four Prolog benchmarks. The best results were for more non-deterministic programs (analogous to imperative programs with unpredictable branches).

6.5 Issues

6.5.1 Branch Prediction

Branch predictors are typically initially modeled and rated by themselves, with the predictor-state updated immediately after every prediction. In such a scenario predictors can give very high accuracies, say 95%. However, in a real machine predictions occur early in the pipeline, while branch results are not known until late in the pipeline. The resulting predictor-update delay can be 10-20 cycles or more for a Pentium 4 class processor.

For example, the PrincePath machine (Section 6.4.1.3) used a nominally accurate compound predictor having two two-level predictors, one local and one global. With the inherent update delays, it only gave an effective average BPA of 86.7% on the SPECint go, gcc, compress, li, and perl benchmarks.

Since pipelines are getting deeper, effective BPAs will continue to decline, increasing the benefits of multipath execution.

6.5.2 Confidence Estimation

Delayed updating likely affects confidence estimators similarly. However, the situation is exacerbated since the confidence estimators start off providing rather low accuracy estimations. Therefore it will be even less advisable in the future to completely depend on confidence information for splitting decisions.

6.5.3 Pipeline Depth

Deeper pipelines not only have linearly increasing branch penalties with depth, but the negative performance effects of such increased penalties grow superlinearly with greater depth, due to the equivalent of Amdahl's Law. Multipath execution can greatly reduce the performance loss where unipath execution does not; see the next section.

6.5.4 Implications of Amdahl's Law - ILP Version

Amdahl's Law is:

$$S = \frac{1}{f_S + \frac{f_P}{P}}$$

Where: S is the net overall speedup, P is the speedup of the parallel code region, and f_S and f_P are the fractions of the Serial and Parallel code regions, respectively. As is well known, this expression tells us that a relatively small sequential region or f_S can result in a big drop in performance. Something similar happens in pipelined machines, where the effective branch penalty corresponds to Amdahl's sequential code. (These are, of course, just rough analytical models.)

Reformulating Amdahl's Law for ILP pipelined machines, we get:

$$R = \frac{1}{c_{MIS} + \frac{c_P}{P}}$$

Where: R is the net execution rate in IPC, c_{MIS} is the extra cycle fraction due to the misprediction penalty, c_P is the cycles per instruction due to normal non-mispredicted execution, and P is the Oracle parallelism, i.e., IPC with no mispredictions. We assume that P includes any parallelism bottlenecks in the machine, e.g., limited issue-width. Expanding:

$$R = \frac{1}{\frac{t_{MIS}*(1-BPA)}{BPL} + \frac{1}{P}}$$

t_{MIS} is the *effective* misprediction penalty in cycles, BPA is the Branch Prediction Accuracy, and BPL is the Branch Path Length in instructions.

Only one or two mispredictions can kill the overall performance of a machine. A misprediction gives worse than sequential execution for a period

of time; it gives NO execution for that time. (Note that we are ignoring predication and MCD in this analysis.)

Seeing what the R expression means, take: $BPL = 6$ instructions; use PrincePath's $BPA = 0.867$; take: $P = 5$ IPC; and, assuming unipath execution, $t_{MIS} = 20$ cycles, similar to a common Pentium 4. Then: $R = 1.55$ IPC. This is just 31% of the peak performance.

Now, assume a PolyPath multipath implementation; not every branch is forked but the misprediction rate is still reduced by a factor of two, i.e., $(1 - BPA)$ is halved. Then: $R = 2.37$ IPC. This is much better, getting 47% of the peak performance, an improvement of 53% over the unipath case, but still not where we'd like to be.

Now consider Avid multipath execution with its fork depth $k = 3$ (see Section 6.4.2.5). Then the effective branch penalty is: $t_{MIS} = 20 - (k * BPL) = 2$ cycles. Second-order effects such as double mispredictions do not change this formulation significantly. Then: $R = 4.09$ IPC. This is 82% of the peak performance, and an improvement of 164% over the unipath case. Multipath execution can give great gains!

It is likely that all conditional branches should be forced to split. This is done in Avid, eager prediction, and DEE, corresponding to the static trees (d)-(g) of Figure 6.1. Confidence estimation is not likely to get the big gains.

6.5.5 Memory Bandwidth Requirements

6.5.5.1 Instruction Fetch

Different studies give conflicting indications of the multipath effects on I-Fetch bandwidth. I believe that extra bandwidth is NOT necessarily needed.

Examining the static DEE tree model, we can readily see the possibilities. The same instructions fetched for the Mainline path can be used for the Side-line paths; no bandwidth increase here. Further, the Mainline path is much shorter than the unipath of a conventional machine, so fewer instructions need to be fetched there. Hence, multipath execution can actually lead to a decreased I-Fetch bandwidth requirement!

6.5.5.2 Data Accesses

The overall effect of multipath execution on data bandwidth is not clear. It has never been measured, to our knowledge. Intuitively multipath execution will increase the necessary load bandwidth. However, explicit or implicit prefetching, or load re-use, may even reduce load bandwidth requirements.

For in-order commit multipath machines store bandwidth should stay the same or even decrease, since stores may combine before an actual commit. For out-of-order commit, such as with speculative caches, store bandwidth may well increase, since many speculative stores will be sent to the speculative cache.

Therefore, multipath machines need not require greater memory bandwidth than classical unipath machines, and may even need less.

6.6 Status, Summary, and Predictions

High-performance microprocessors are running out of options for improved performance. There is little headroom for further substantive increases in clock frequency. Increasing cache sizes is giving diminishing returns.

There is only one option: exploit more ILP. However, unipath methods cannot do this, e.g., branch prediction is becoming a dead-end for greater effective improved accuracies. We need multipath execution.

Fortunately, there is renewed interest in multipath execution [20]. Also, existing research efforts are continuing at both our lab [17] and at others.

There are many promising areas of future multipath research, such as: combining data speculation with multipath execution, deriving other optimal forms of multipath execution, investigating new and existing multipath heuristics, and studying coarse(r)-grain CMP machines. Multipath techniques may also be applicable both to other areas of computer science and engineering, such as database engines, and even possibly to other disciplines.

Multipath execution has come a long way since the IBM 360/91, and it's going to go farther. It will be standard in high-performance architectures.

References

- [1] Riseman, E. M. and Foster, C. C. The Inhibition of Potential Parallelism by Conditional Jumps. *IEEE Transactions on Computers*, C-21(12):1405–1411, December 1972.
- [2] Anderson, D. W., Sparacio, F. J., and Tomasulo, R. M. The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling. *IBM Journal of Research and Development*, 11(1):8–24, January 1967.
- [3] Jacobsen, E., Rotenberg, E., and Smith, J. E. Assigning Confidence to Conditional Branch Prediction. In *Proceedings of the 29th International Symposium on Microarchitecture (MICRO-29)*, pages 142–152. IEEE and ACM, December 1996.

- [4] Lam, M. S. and Wilson, R. P. Limits of Control Flow on Parallelism. In *Proc. of the 19th Annual International Symposium on Computer Architecture, Gold Coast, Australia*, pages 46–57. IEEE/ACM, May 1992.
- [5] Uht, A. K., Sindagi, V., and Somanathan, S. Branch Effect Reduction Techniques. *IEEE COMPUTER*, 30(5):71–81, May 1997.
- [6] Valia, S. K., Koblenksi, S. A., and Janes, D. R. Multipath Execution Using Dynamic Relative Confidence. Technical report, Department of Computer Sciences, University of Wisconsin, Madison, Madison, WI, USA, Spring 2003. Instructor: Prof. David Wood.
- [7] Knieser, M. J. and Papachristou, C. A. Y-Pipe: A Conditional Branching Scheme Without Pipeline Delays. In *Proc. of the 25th International Conference on Microarchitecture*, page 125128. ACM/IEEE, 1992.
- [8] Messaris, S. A. Combining Speculative with Eager Execution to Reduce the Branch Penalty on Instruction-Level Parallel Architectures. Master’s thesis, Department of Computer Science, Michigan Technological University, Houghton, MI, USA, 1994.
- [9] Heil, T. H. and Smith, J. E. Selective Dual Path Execution. Technical report, Department of Electrical and Computer Engineering, University of Wisconsin-Madison, Madison, WI, USA, November 8, 1996.
- [10] Tyson, G., Lick, K., and Farrens, M. Limited Dual Path Execution. Technical Report CSE-TR-346-97, Dept. of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, USA, 1997.
- [11] Wallace, S., Calder, B., and Tullsen, D. M. Threaded Multiple Path Execution. In *25th Annual International Symposium on Computer Architecture*, pages 238–249. ACM, June 1998.
- [12] Klauser, A., Paithankar, A., and Grunwald, D. Selective Eager Execution on the PolyPath Architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture, Barcelona, Spain*, pages 250–259, June 27 - July 01, 1998.
- [13] Ahuja, P. S., Skadron, K., Martonosi, M., and Clark, D. W. Multipath Execution: Opportunities and Limits. In *Proceedings of the 12th International Conference on Supercomputing*. ACM, July 1998.
- [14] Magid, N., Tjaden, G., and Messinger, H. Exploitation of Concurrency by Virtual Elimination of Branch Instructions. In *Proc. of the 1981 Intl. Conference on Parallel Processing*, pages 164–165, Aug. 1981.
- [15] Uht, A. K. and Sindagi, V. Disjoint Eager Execution: An Optimal Form of Speculative Execution. In *Proc. of the 28th International Symposium on Microarchitecture, Ann Arbor, MI*, pages 313–325, Nov./Dec. 1995.

- [16] Chen, T. F. Supporting Highly Speculative Execution via Adaptive Branch Trees. In *Proc. of the 4th International Symposium on High Performance Computer Architecture*, pages 185–194. IEEE, Jan. 1998.
- [17] Uht, A. K., Morano, D., Khalafi, A., and Kaeli, D. R. Levo - A Scalable Processor With High IPC. *The Journal of Instruction-Level Parallelism*, 5, August 2003.
- [18] Kol, R. *Self-Timed Asynchronous Architecture of an Advanced General Purpose Microprocessor*. PhD thesis, Dept. of Electrical Engineering, The Technion - Israel Institute of Technology, Haifa, Israel, Nov. 1997.
- [19] dos Santos, R., Navaux, P., and Nemirovsky, M. DCE: The Dynamic Conditional Execution in a Multipath Control Independent Architecture. Technical Report UCSC-CRL-01-08, University of California, Santa Cruz, Santa Cruz, CA, USA, June 2001.
- [20] Acosta, C., Vajapeyam, S., Ramrez, A., and Valero, M. CDE: A Compiler-driven, Dependence-centric, Eager-executing Architecture for the Billion Transistor Era. In *Proceedings of the Workshop on Complexity Effective Design (WCED'03), at the International Symposium on Computer Architecture (ISCA '03)*, July 2003.
- [21] Gwennap, L. DanSoft Develops VLIW Design. *Microprocessor Report*, 11(2), February 17, 1997.
- [22] Chidester, M. C., George, A. D., and Radlinski, M. A. Multiple-Path Execution for Chip Multiprocessors. *Elsevier Journal of Systems Architecture: The EUROMICRO Journal*, 49(1-2):33–52, July 2003.
- [23] Sundaramoorthy, K., Purser, Z., and Rotenberg, E. Multipath Execution on Chip Multiprocessors Enabled by Redundant Threads. Technical Report CESR-TR-01-2, Center for Embedded System Research (CESR), Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC, USA, October 23, 2001.
- [24] Hordijk, J. and Corporaal, H. The Impact of Data Communication and Control Synchronization on Coarse-Grain Task Parallelism. In *Proceedings of the 2nd Annual Conference of ASCI, Lommel, Advanced School for Computing and Imaging, Delft, The Netherlands*, January 1996.
- [25] Tubella, J. and González, A. Exploiting Path Parallelism in Logic Programming. In *Proceedings of the Euromicro Workshop on Parallel and Distributed Processing*, pages 164–173, January 25-27, 1995.
- [26] Raghuvaran, P., Shachnai, H., and Yaniv, M. Dynamic Schemes for Speculative Execution of Code. In *Proc. of the Sixth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), Montreal, Canada*, July 1998.

- [27] Gaysinsky, A., Itai, A., and Shachnai, H. Strongly Competitive Algorithms for Caching with Pipelined Prefetching. In *Proc. of the 9th Annual European Symposium on Algorithms, Aarhus*, pages 49–61. Springer-Verlag Lecture Notes In Computer Science, August 2001.
- [28] Franklin, M. and Sohi, G. S. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.
- [29] Fisher, J. A. Global Code Generation for Instruction-Level Parallelism: Trace Scheduling-2. Technical Report HPL-93-43, Computer Research Center, Hewlett-Packard Laboratories, Palo Alto, CA, USA, June 1993.
- [30] González, J. and González, A. Limits on Instruction-Level Parallelism with Data Speculation. Technical Report UPC-DAC-1997-34, Department Architettura de Computadores, Universitat Polytechnica Catalan, Barcelona, Spain, 1997.

Chapter 7

Data Cache Prefetching

Yan Solihin¹ and Donald Yeung²

North Carolina State University,¹ University of Maryland at College Park²

7.1	Introduction	161
7.2	Software Prefetching	162
7.3	Hardware Prefetching	173
	References	181

7.1 Introduction

Due to the growing gap in processor and memory speeds, a cache miss is becoming more expensive. Data prefetching is a technique to hide cache miss latency by bringing data closer to the processor ahead of the processor's request for it. One way to initiate prefetching is by inserting prefetch instructions into the program. This insertion is typically performed statically by the compiler. We will discuss it in the *Software Prefetching* section (Section 7.2). Prefetching can also be initiated dynamically by observing the program's behavior. Since this dynamic prefetch generation is typically performed by hardware, we will discuss it in the *Hardware Prefetching* section (Section 7.3). Note that this boundary is not firm; for example prefetch instructions may be inserted dynamically by run-time optimizer, whereas some hardware prefetching techniques may be implemented in software.

There are three metrics that characterize the effectiveness of a prefetching technique: coverage, accuracy, and timeliness. *Coverage* is defined as the fraction of original cache misses that are prefetched, and therefore become cache hits or partial cache misses. *Accuracy* is defined as the fraction of prefetches that are useful, i.e., they result in cache hits. And finally, *timeliness* measures how early the prefetches arrive, which determines whether the full cache miss latency is hidden, or only a part of the miss latency is hidden. An ideal prefetching technique should exhibit high coverage so that it eliminates most of the cache misses, high accuracy so that it does not increase memory bandwidth consumption, and timeliness so that most of the prefetches hide the full cache miss latency. Achieving this ideal is a challenge. By aggressively issuing many prefetches, a prefetching technique may achieve a high coverage

but low accuracy. By conservatively issuing only prefetches that are highly predictable, a prefetching technique may achieve a high accuracy but a low coverage. Finally, timeliness is also important. If a prefetch is initiated too early, it may pollute the cache, and may be replaced from the cache or prefetch buffer before it is used by the processor. If it is initiated too late, it may not hide the full cache miss latency.

In addition to the performance metrics, there are also other practical considerations that are important, such as the cost and complexity of a hardware implementation and whether it needs code recompilation.

Most of the prefetching techniques discussed in this chapter were designed for uniprocessor systems, although they can also be applied to multiprocessor systems. Multiprocessor-specific prefetching techniques will not be discussed in this chapter. Finally, due to a very high number of publications in data cache prefetching, it is impossible to list and discuss all of them in this chapter. The goal of this chapter is to give a brief overview of existing prefetching techniques. Thus, it will only focus on a subset of representative techniques that hopefully motivate readers into reading other prefetching studies as well.

7.2 Software Prefetching

Software prefetching relies on the programmer or compiler to insert explicit prefetch instructions into the application code for memory references that are likely to miss in the cache. At run time, the inserted prefetch instructions bring the data into the processor’s cache in advance of its use, thus overlapping the cost of the memory access with useful work in the processor. Historically, software prefetching has been quite effective in reducing memory stalls for scientific programs that reference array-based data structures [3, 7, 24, 31, 32]. More recently, techniques have also been developed to apply software prefetching for non-numeric programs that reference pointer-based data structures as well [22, 28, 39].

This section discusses both types of software prefetching techniques. We begin by reviewing the architectural support necessary for software prefetching. Then, we describe the algorithms for instrumenting software prefetching, first for array-based data structures and then for pointer-based data structures. Finally, we describe the interactions between software prefetching and other software-based memory performance optimizations.

7.2.1 Architectural Support

Although software prefetching is a software-centric technique, some hardware support is necessary. First, the architecture’s instruction set must pro-

vide a *prefetch instruction*. Prefetch instructions are non-blocking memory loads (i.e., they do not wait for the reply from the memory system). They cause a cache fill of the requested memory location on a cache miss, but have no other side effects. Since they are effectively NOPs from the processor's standpoint, prefetch instructions can be ignored by the memory system (for example, at times when memory resources are scarce) without affecting program correctness. Hence, prefetch instructions enable software to provide "hints" to the memory system regarding the memory locations that should be loaded into cache.

In addition to prefetch instructions, software prefetching also requires lockup-free caches [26, 40]. Since software prefetching tries to hide memory latency underneath useful computation, the cache must continue servicing normal memory requests from the CPU following cache misses triggered by prefetch instructions. For systems that prefetch aggressively, it is also necessary for the cache and memory subsystem to support multiple outstanding memory requests. This allows independent prefetch requests to overlap with each other.

Unless the compiler or programmer can guarantee the safety of all prefetches, another requirement from the architecture is support for ignoring memory faults caused by prefetch instructions. This support is particularly useful when instrumenting prefetch instructions speculatively, for example to prefetch data accessed within conditional statements [3].

Finally, a few researchers have investigated software prefetching assuming support for *prefetch buffers* [7, 24]. Instead of prefetching directly into the L1 cache, these approaches place prefetched data in a special data buffer. On a memory request, the processor checks both the L1 cache and the prefetch buffer, and moves prefetched data into the L1 cache only on a prefetch buffer hit. Hence, prefetched blocks that are never referenced by the processor do not evict potentially useful blocks in the L1 cache. In addition, usefully prefetched blocks are filled into the L1 cache as late as possible—at the time of the processor's reference rather than the prefetch block's arrival—thus delaying the eviction of potentially useful blocks from the L1 cache. On the other hand, prefetch buffers consume memory that could have otherwise been used to build a larger L1 cache.

7.2.2 Array Prefetching

Having discussed the architectural support for software prefetching, we now examine the algorithms for instrumenting prefetches into application code. We begin by focusing on techniques for array references performed within loops that give rise to *regular memory access patterns*. These memory references employ array subscripts that are affine—i.e., linear combinations of loop index variables with constant coefficients and additive constants. (We also discuss prefetching for indirect array references that give rise to *irregular memory access patterns*, though to a lesser extent.)

Affine array references are quite common in a variety of applications, including dense-matrix linear algebra and finite-difference PDE solvers as well as image processing and scans/joins in relational databases. These programs can usually exploit long cache lines to reduce memory access costs, but may suffer poor performance due to cache conflict and capacity misses arising from the large amounts of data accessed. An important feature of these codes is that memory access patterns can be identified exactly at compile time, assuming array dimension sizes are known. Consequently, programs performing affine array references are good candidates for software prefetching.

7.2.2.1 Mowry’s Algorithm

The best-known approach for instrumenting software prefetching of affine array references is the compiler algorithm proposed by Mowry et al. [33, 31]. To illustrate the algorithm, we use the 2-D Jacobi kernel in Figure 7.1a as an example, instrumenting it with software prefetching using Mowry’s algorithm in Figure 7.1b. The algorithm involves three major steps: locality analysis, cache miss isolation, and prefetch scheduling.

Locality analysis determines the array references that will miss in the cache, and thus require prefetching. The goal of this step is to avoid *unnecessary prefetches*, or prefetches that incur runtime overhead without improving performance because they hit in the cache. The analysis proceeds in two parts. First, reuse between dynamic instances of individual static array references is identified. In particular, locality analysis looks for three types of reuse: spatial, temporal, and group. Spatial reuse occurs whenever a static array reference accesses locations close together in memory. For example, every array reference in Figure 7.1a exhibits spatial reuse. Since the i loop performs a unit-stride traversal of each inner array dimension, contemporaneous iterations access the same cache block. In contrast, temporal reuse occurs whenever a static array reference accesses the same memory location. (None of the array references in Figure 7.1a exhibit temporal reuse since all dynamic instances access distinct array elements.) Lastly, group reuse occurs whenever two or more different static array references access the same memory location. In Figure 7.1a, all four B array references exhibit group reuse since many of their dynamic instances refer to the same array elements.

After identifying reuse, locality analysis determines which reuses result in cache hits. The algorithm computes the number of loop iterations in between reuses, and the amount of data referenced within these intervening iterations. If the size of this referenced data is smaller than the cache size, then the algorithm assumes the reuse is captured in the cache and the dynamic instances do not need to be prefetched. All other dynamic instances, however, are assumed to miss in the cache, and require prefetching.

The next step in Mowry’s algorithm, after locality analysis, is *cache miss isolation*. For static array references that experience a mix of cache hits and misses (as determined by locality analysis), code transformations are per-

```

a) for (j=2; j <= N-1; j++) {
    for (i=2; i <= N-1; i++)
        A[j][i]=0.25*(B[j][i-1]+B[j][i+1]+B[j-1][i]+B[j+1][i]);

b) for (j=2; j <= N-1; j++) {
    for (i=2; i <= PD; i+=4) {           // Prologue
        prefetch(&B[j][i]);
        prefetch(&B[j-1][i]);
        prefetch(&B[j+1][i]);
        prefetch(&A[j][i]);
    }
    for (i=2; i < N-PD-1; i+=4) {       // Steady State
        prefetch(&B[j][i+PD]);
        prefetch(&B[j-1][i+PD]);
        prefetch(&B[j+1][i+PD]);
        prefetch(&A[j][i+PD]);
    }
    A[j][i]=0.25*(B[j][i-1]+B[j][i+1]+B[j-1][i]+B[j+1][i]);
    A[j][i+1]=0.25*(B[j][i]+B[j][i+2]+B[j-1][i+1]+B[j+1][i+1]);
    A[j][i+2]=0.25*(B[j][i+1]+B[j][i+3]+B[j-1][i+2]+B[j+1][i+2]);
    A[j][i+3]=0.25*(B[j][i+2]+B[j][i+4]+B[j-1][i+3]+B[j+1][i+3]);
}
for (i=N-PD; i <= N-1; i++)           // Epilogue
    A[j][i]=0.25*(B[j][i-1]+B[j][i+1]+B[j-1][i]+B[j+1][i]);
}

```

FIGURE 7.1: Example illustrating Mowry's algorithm. a) 2-D Jacobi kernel code b) instrumented with software prefetching using Mowry's algorithm [32].

formed to isolate the misses from the hits in separate static array references. This enables prefetching of the cache-missing instances while avoiding unnecessary prefetches fully statically, i.e., without predicating prefetch instructions with IF statements that incur runtime overhead.

The appropriate cache miss isolation transformation depends on the type of reuse. For spatial reuse, *loop unrolling* is performed. Figure 7.1b illustrates loop unrolling for the 2-D Jacobi kernel (see the second nested loop). As described earlier, the i loop of the Jacobi kernel performs a unit-stride traversal of the A and B inner array dimensions. Assuming 8-byte array elements and a 32-byte cache block, cache misses for each static array reference occur every $\frac{32}{8} = 4$ iterations. By unrolling the loop 4 times, the leading array references incur cache misses every iteration while the remaining unrolled references never miss, thus isolating the cache misses. For temporal reuse, cache misses are isolated via *loop peeling*. In this case, some subset of iterations (usually the first) incur all the cache misses, and the remaining iterations hit in the cache. The cache-missing iteration(s) are peeled off and placed in a separate loop. Lastly, for group reuse, no explicit transformation is necessary since cache misses already occur in separate static array references. Analysis is performed simply to identify which static array reference is the leading reference in each group, and hence will incur all the cache misses on behalf of the other static references.

Once the cache misses have been isolated, prefetches are inserted for the static array references that incur them. Figure 7.1b illustrates the prefetch instrumentation for the 2-D Jacobi kernel. Assuming the group reuse between $B[j][i+1]$ and $B[j][i-1]$ is captured by the cache but the group reuse across outer loop iterations (i.e., the j subscripts) is not, then there are four static array references that incur all the cache misses— $A[j][i]$, $B[j][i+1]$, $B[j-1][i]$, and $B[j+1][i]$. Prefetches are instrumented for these four array references, as illustrated in Figure 7.1b (again, see the second nested loop). Notice, prefetches are not instrumented for the remaining static array references, thus avoiding unnecessary prefetches and reducing prefetch overhead.

The last step in Mowry's algorithm is *prefetch scheduling*. Given the high memory latency of most modern memory systems, a single loop iteration normally contains insufficient work under which to hide the cost of memory accesses. To ensure that data arrive in time, the instrumented prefetches must initiate multiple iterations in advance. The minimum number of loop iterations needed to fully overlap a memory access is known as the *prefetch distance*. Assuming the memory latency is l cycles and the work per loop iteration is w cycles, the prefetch distance, PD , is simply $\lceil \frac{l}{w} \rceil$. Figure 7.1b illustrates the indices of prefetched array elements contain a PD term, providing the early prefetch initiation required.

By initiating prefetches PD iterations in advance, the first PD iterations are not prefetched, while the last PD iterations prefetch past the end of each array. These inefficiencies can be addressed by performing *software pipelining* to handle the first and last PD iterations separately. Software pipelining cre-

ates a *prologue loop* to execute PD prefetches for the first PD array elements, and an *epilogue loop* to execute the last PD iterations without prefetching. Along with the original loop, called the *steady state loop*, the transformed code initiates, overlaps, and completes all prefetches relative to the computation in a pipelined fashion. Figure 7.1b illustrates the prologue, steady state, and epilogue loops created by software pipelining for the 2-D Jacobi kernel.

7.2.2.2 Support for Indexed Arrays

In addition to affine arrays, another common type of array is *indexed arrays*. Indexed arrays take the form $A[B[i]]$, in which a data array and index array are composed to form an *indirect array reference*. While the inner array reference is affine and regular, the outer array reference is *irregular* due to memory indirection. Indexed arrays arise in scientific applications that attempt complex simulations. In computational fluid dynamics, meshes for modeling large problems are sparse to reduce memory and computation requirements. In N-body solvers for astrophysics and molecular dynamics, data structures are irregular because they model the positions of particles and their interactions. Indexed arrays are frequently used to express these complex data relationships since they are more efficient than pointers. Unfortunately, the cache performance of indexed arrays can be poor since both spatial and temporal locality is low due to the irregularity of the access pattern.

Software prefetching of indexed arrays was first studied by Callahan et al. [3]. Later, Mowry extended his algorithm for affine arrays discussed in Section 7.2.2.1 to handle indexed arrays [34, 31]. Mowry’s extended algorithm follows the same steps discussed earlier, with some modifications. The modifications stem from two differences between indexed and affine array references. First, static analysis cannot determine locality information for consecutively accessed indexed array elements since their addresses depend on the index array values known only at runtime. Hence, the algorithm conservatively assumes no reuse occurs between indexed array references, requiring prefetching for all dynamic instances. Moreover, since the algorithm assumes all dynamic instances miss in the cache, there is also no need for cache miss isolation.

Second, before a data array reference can perform, the corresponding index array reference must complete since the index value is used to index into the data array. Hence, pairs of data and index array references are serialized. This affects prefetch scheduling. As in affine array prefetching, the computation of the prefetch distance uses the formula, $PD = \lceil \frac{l}{w} \rceil$. However, the adjustment of array indices for indexed arrays must take into consideration the serialization of data and index array references. Since data array elements cannot be prefetched until the index array values they depend on are available, prefetches for index array elements should be initiated *twice* as early as data array elements. This ensures that an index array value is in cache when its corresponding data array prefetch is issued.

7.2.3 Pointer Prefetching

In Section 7.2.2, we discussed software prefetching techniques for array-based codes. This section focuses on techniques for *pointer-chasing codes*. Pointer-chasing codes use linked data structures (LDS), such as linked lists, n-ary trees, and other graph structures, that are dynamically allocated at run time. They arise from solving complex problems where the amount and organization of data cannot be determined at compile time, requiring the use of pointers to dynamically manage both storage and linkage. They may also arise from high-level programming language constructs, such as those found in object-oriented programs.

Due to the dynamic nature of LDS creation and modification, pointer-chasing codes commonly exhibit poor spatial and temporal locality, and experience poor cache behavior. Another significant performance bottleneck in these workloads is the *pointer-chasing problem*. Because individual nodes in an LDS are connected through pointers, access to additional parts of the data structure cannot take place until the pointer to the current portion of the data structure is resolved. Hence, LDS traversal requires the sequential reference and dereference of the pointers stored inside visited nodes, thus serializing all memory operations performed along the traversal.

The pointer-chasing problem not only limits application performance, but it also potentially limits the effectiveness of prefetching. As discussed earlier, an important goal of software prefetching is to initiate prefetches sufficiently early to tolerate their latency. Doing so requires knowing the prefetch memory address in advance as well. This is straightforward for array data structures since the address of future array elements can be computed given the desired array indices. However, determining the address of a future link node in an LDS requires traversing the intermediate link nodes due to the pointer-chasing problem, thus preventing the early initiation of prefetches.

LDS prefetching techniques try to address the pointer chasing problem and its impact on early prefetch initiation. We present the techniques in two groups: those that use the natural pointers in the LDS only, and those that create special pointers called *jump pointers* for the sole purpose of prefetching.

7.2.3.1 Natural Pointer Techniques

The simplest software prefetching technique for LDS traversal is *greedy prefetching*, proposed by Luk and Mowry [28]. Greedy prefetching inserts software prefetch instructions immediately prior to visiting a node for all possible successor nodes that might be encountered by the traversal. To demonstrate the technique, Figure 7.2 shows the prefetch instrumentation for two types of LDS traversals. Figure 7.2a illustrates greedy prefetching for a loop-based linked list traversal. In this case, a single prefetch is inserted at the top of the loop for the next link node in the list. Figure 7.2b illustrates greedy prefetching for a recursive tree traversal. In this case, prefetches are inserted at the top of the recursive function for each child node in the sub-tree.

```

a)   struct node {data, next}
      *ptr, *list_head;

      ptr = list_head;
      while (ptr) {
          prefetch(ptr->next);
          ...
          ptr = ptr->next;
      }
}

b)   struct node {data, left, right}
      *ptr;

      void recurse(ptr) {
          prefetch(ptr->left);
          prefetch(ptr->right);
          ...
          recurse(ptr->left);
          recurse(ptr->right);
      }
}

```

FIGURE 7.2: Example illustrating greedy pointer prefetching for a) linked list and b) tree traversals using greedy prefetching [28].

Greedy prefetching is attractive due to its simplicity. However, its ability to properly time prefetch initiation is limited. For the linked list traversal in Figure 7.2a, each prefetch overlaps with a single loop iteration only. Greater overlap is not possible since the technique cannot prefetch nodes beyond the immediate successor due to the pointer-chasing problem described earlier. If the amount of work in a single loop iteration is small compared to the prefetch latency, then greedy prefetching will not effectively tolerate the memory stalls. The situation is somewhat better for the tree traversal in Figure 7.2b. Although the prefetch of `ptr->left` suffers a similar problem as the linked list traversal (its latency overlaps with a single recursive call only), the prefetch of `ptr->right` overlaps with more work—the traversal of the entire left sub-tree. But the timing of prefetch initiation may not be ideal. The latency for prefetching `ptr->right` may still not be fully tolerated if the left sub-tree traversal contains insufficient work. Alternatively, the prefetched node may arrive too early and suffer eviction if the left sub-tree traversal contains too much work.

Another approach to prefetching LDS traversals is *data linearization prefetching*. Like greedy prefetching, this technique was proposed by Luk and Mowry [28] (a somewhat similar technique was also proposed by Stoutchini et al.[44]). Data linearization prefetching makes the observation that if contemporaneously traversed link nodes in an LDS are laid out linearly in memory, then prefetching a future node no longer requires sequentially traversing the intermediate nodes to determine its address. Instead, a future node's memory address can be computed simply by offsetting from the current node pointer, much like indexing into an array. Hence, data linearization prefetching avoids the pointer-chasing problem altogether, and can initiate prefetches as far in advance as necessary.

The key issue for data linearization prefetching is achieving the desired linear layout of LDS nodes. Linearization can occur either at LDS creation, or after an LDS has been created. From a cost standpoint, the former is more desirable since the latter requires reorganizing an existing LDS via data copying at runtime. Moreover, linearizing at LDS creation is feasible especially if the order of LDS traversal is known *a priori*. In this case, the allocation and linkage of individual LDS nodes should simply follow the order of node

traversal. As long as the memory allocator places contemporaneously allocated nodes regularly in memory, the desired linearization can be achieved. Notice, however, periodic “re-linearization” (via copying) may be necessary if link nodes are inserted and deleted frequently. Hence, data linearization prefetching is most effective for applications in which the LDS connectivity does not change significantly during program execution.

7.2.3.2 Jump Pointer Techniques

Both greedy and data linearization prefetching do not modify the logical structure of the LDS to perform prefetching. In contrast, another group of pointer prefetching techniques have been studied that insert special pointers into the LDS, called *jump pointers*, for the sole purpose of prefetching. Jump pointers connect non-consecutive link nodes, allowing prefetch instructions to name link nodes further down the pointer chain without traversing the intermediate link nodes and without performing linearization beforehand. Effectively, the jump pointers increase the dimensionality and reduce the diameter of an LDS, an idea borrowed from Skip Lists [37].

Several jump pointer techniques have been studied in the literature. The most basic approach is *jump pointer prefetching* as originally proposed by Luk and Mowry [28]. Roth and Sohi [39] also investigated jump pointer prefetching, introducing variations on the basic technique in [28] that use a combination of software and hardware support to pursue the jump pointers. Figure 7.3a illustrates the basic technique, applying jump pointer prefetching to the same linked list traversal shown in Figure 7.2a. Each linked list node is augmented with a jump pointer field, called *jump* in Figure 7.3a. During a separate initialization pass (discussed below), the jump pointers are set to point to a link node further down the linked list that will be referenced by a future loop iteration. The number of consecutive link nodes skipped by the jump pointers is the prefetch distance, PD , which is computed using the same approach described in Section 7.2.2 for array prefetching. Once the jump pointers have been installed, prefetch instructions can prefetch through the jump pointers, as illustrated in Figure 7.3a by the loop labeled “Steady State.”

Unfortunately, jump pointer prefetching cannot prefetch the first PD link nodes in a linked list because there are no jump pointers that point to these early nodes. In many pointer-chasing applications, this limitation significantly reduces the effectiveness of jump pointer prefetching because pointer chains are kept short by design. To enable prefetching of early nodes, jump pointer prefetching can be extended with *prefetch arrays* [22]. In this technique, an array of prefetch pointers is added to every linked list to point to the first PD link nodes. Hence, prefetches can be issued through the memory addresses in the prefetch arrays before traversing each linked list to cover the early nodes, much like the prologue loops for array prefetching prefetch the first PD array

```

a) struct node {data, next, jump}
    *ptr, *list_head, *prefetch_array[PD], *history[PD];
int i, head, tail;

for (i = 0; i < PD; i++) // Prologue Loop
    prefetch(prefetch_array[i]);

ptr = list_head;
while (ptr->next) { // Steady State Loop
    prefetch(ptr->jump);
    ...
    ptr = ptr->next;
}

b) for (i = 0; i < PD; i++) history[i] = NULL;
tail = 0;
head = PD-1;

ptr = list_head;
while (ptr) { // Prefetch Pointer Generation Loop
    history[head] = ptr;
    if (!history[tail])
        prefetch_array[tail] = ptr;
    else
        history[tail]->jump = ptr;
    head = (head+1)%PD;
    tail = (tail+1)%PD;
    ptr = ptr->next;
}

```

FIGURE 7.3: Example illustrating jump-pointer and prefetch-array pointer prefetching for a linked list traversal [22].

elements. Figure 7.3b illustrates the addition of a prologue loop that performs prefetching through a prefetch array.

As described earlier, the prefetch pointers must be installed before prefetching can commence. Figure 7.3b shows an example of prefetch pointer installation code which uses a *history pointer array* [28] to set the prefetch pointers. The history pointer array, called **history** in Figure 7.3b, is a circular queue that records the last PD link nodes traversed by the initialization code. Whenever a new link node is traversed, it is added to the head of the circular queue and the head is incremented. At the same time, the tail of the circular queue is tested. If the tail is NULL, then the current node is one of the first PD link nodes in the list since PD link nodes must be encountered before the circular queue fills. In this case, we set one of the **prefetch_array** pointers to point to the node. Otherwise, the tail's jump pointer is set to point to the current link node. Since the circular queue has depth PD , all jump pointers are initialized to point PD link nodes *ahead*, thus providing the proper prefetch distance. Normally, the compiler or programmer ensures the prefetch pointer installation code gets executed prior to prefetching, for example on the first traversal of an LDS. Furthermore, if the application modifies the LDS after

the prefetch pointers have been installed, it may be necessary to update the prefetch pointers either by re-executing the installation code or by using other fix-up code.

7.2.4 Relationship with Data Locality Optimizations

There have been several software-centric solutions to the memory bottleneck problem; software prefetching is just one of these software-based solutions. Another important group of software techniques is *data locality optimizations*. Data locality optimizations enhance the locality of data memory references, either by changing the layout of data in memory or by changing the order of accesses to the data, thus improving the reuse of data in the cache.

The type of optimization depends on the memory reference type. For affine array references, *tiling* is used [45]. Tiling combines strip-mining with loop permutation to form small tiles of loop iterations which are executed together. Alternatively, for indexed array references, *runtime data access reordering* is used [12, 29, 30]. This technique reorders loop iterations at runtime using an inspector-executor approach [11] to bring accesses to the same data closer together in time. Finally for pointer references, *cache-conscious heap allocation* [2, 8] is used. Cache-conscious heap allocation places logically linked heap elements physically close together in memory at memory allocation time to improve spatial locality. (This technique is related to data linearization described in Section 7.2.3.1.)

Although both software prefetching and data locality optimizations improve memory performance, they do so in very different ways. Prefetching initiates memory operations early to hide their latency underneath useful computation, whereas data locality optimizations improve reuse in the data cache. The former is a *latency tolerance* technique while the latter is a *latency reduction* technique.

Because prefetching only tolerates memory latency, it requires adequate memory bandwidth to be effective. As software prefetches bring required data to the processor increasingly early, the computation executes faster, causing a higher rate of data requests. If memory bandwidth saturates, prefetching will not provide further performance gains, and the application will become limited by the speed of the memory system. In contrast, data locality optimizations reduce the average memory latency by eliminating a portion of the application's cache misses. Hence, it not only reduces memory stalls, it also reduces traffic to lower levels of the memory hierarchy as well as the memory contention this traffic causes.

Compared to data locality optimizations, software prefetching can potentially remove more memory stalls since it can target *all* cache misses. Provided ample memory bandwidth exists, software prefetching can potentially remove the performance degradation of the memory system entirely. Data locality optimizations cannot remove all stalls, for example those arising from cold misses. Another advantage of software prefetching is software transforma-

tions are purely speculative—they do not change the semantics of the program code. In contrast, data locality optimizations can affect program correctness; hence, compilers must prove the correctness of transformations before they can be applied. In some cases, this can reduce the applicability of data locality optimizations compared to software prefetching.

7.3 Hardware Prefetching

The previous section has given an overview of software prefetching techniques. This section will give an overview of hardware prefetching techniques.

Based on which party initiates the prefetching, hardware prefetching techniques can be categorized as *processor-side* or *memory-side*. In the former approach, the processor or an engine in its cache hierarchy issues the prefetch requests [5, 6, 20, 21, 23, 25, 36, 38, 41, 47, 10]. In the latter approach, the engine that prefetches data for the processor is in the main memory system [42, 43, 1, 4, 17, 35, 46].

The advantages of processor-side prefetching is the abundant information that can be used by the processor, which includes the program counter, virtual addresses of memory references, etc., that are not always available to the memory system. One advantage of memory-side prefetching is that it eliminates the overheads and state bookkeeping that prefetch requests introduce in the paths between the main processor and its caches. In addition, the prefetcher can exploit its proximity to the memory to its advantage, for example by storing its state in memory and by exploiting low latency and high bandwidth access to the main memory. In microprocessors, processor-side prefetching is typically implemented at the level of L1 or L2 cache, using simple engines that detect and prefetch spatial locality of a program, such as in the Intel Pentium 4 [14], and IBM Power4 [19]. At the system level, memory-side prefetching is typically implemented in the memory controller that buffers the prefetched data from the memory and targets prefetching strided accesses, such as in NVIDIA’s DASP engine [35].

Based on the type of data access patterns that can be handled, *sequential prefetching* detects and prefetches for accesses to contiguous locations. *Stride prefetching* detects and prefetches accesses that are s -cache block apart between consecutive accesses, where s is the amount of the stride. Therefore, an s -strided access would produce address trace of $A, A + s, A + 2s, \dots$. Note that sequential prefetching is a stride prefetching where $s \leq 1$. Finally, some accesses that are not strided may appear random to the prefetcher, and is therefore not easy to detect and prefetch. These accesses may result from a linked data structure traversal, if the allocation sequence of the instances is such that they are not located in contiguous (or strided) memory locations.

They may also result from indirect array references, such as in $a[b[i]]$. There are ways to deal with this. One type of technique relies on the fact that although the accesses appear random, their sequence is often repeated during program execution. Therefore, by recording the sequence, address correlation can be learned and used for prefetching. This type of technique is called *correlation prefetching*. Finally, for accesses that result from dereferencing pointers in linked data structures, prefetching can be initiated if a pointer is identified and the data pointed is prefetched, before the actual dereferencing. This technique is called *content-directed prefetching*.

Finally, there are various places where prefetching can be initiated and where the destination where the prefetched data can be placed. Prefetching can be initiated in the L1 cache level, L2 cache level, the memory controller, or even in the memory chips. The prefetched data are usually stored at the level where the prefetch is initiated, either in the prefetch-initiating cache, or in a separate prefetch buffer at the same level as the prefetch-initiating cache to avoid polluting the cache with prefetched data. In some cases, the initiating level is not the same as the destination. For example, a memory-side prefetching may be initiated near the memory but the prefetched data are pushed into the L2 cache.

7.3.1 Stride and Sequential Prefetching

Early stride and sequential prefetching studies include the Reference Prediction Table by Chen and Baer [6], and stream buffers by Jouppi [21]. This section will describe stream buffers scheme, which is attractive due to its simplicity. A stream buffer allows prefetching sequential accesses. A stream buffer is a FIFO buffer, where each entry contains a line that has the same size as a cache line, an address (or tag) of the line, and an “available” bit. To prefetch for multiple streams in parallel, more than one stream buffers can be used, where each buffer prefetches from one stream.

On a cache access, the cache and the head entries of the stream buffers are checked for a match. If the requested line is found in the cache, no action on the stream buffers is performed. If the line is not found in the cache, but found at the head of a stream buffer, the line is moved into the cache. The head pointer of the stream buffer moves to the next entry, and the buffer prefetches the last entry’s successor into the freed entry, i.e., if the last entry’s line address is L , $L + 1$ is prefetched. If the line is not found in both the cache and all the stream buffers, a new stream buffer is allocated. The line’s successors are prefetched to fill the stream buffers. While a prefetch is in flight, the available bit is set to ‘0’, and is set to ‘1’ only when the prefetch has completed.

Figure 7.4 illustrates the stream buffer operation with three stream buffers. Suppose that the addresses from access streams are $A, B, A+2, B+1, A+4, B+2, \dots$ and that the cache and stream buffers do not contain any of the blocks initially. The access to A results in a cache miss, which results in allocating a

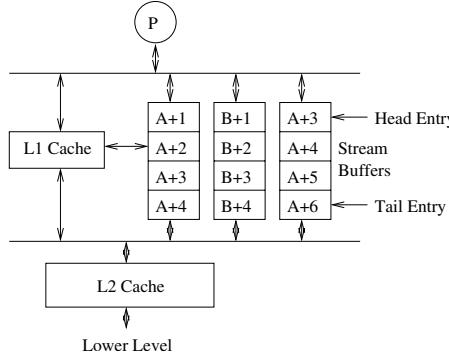


FIGURE 7.4: Illustration for stream buffer operation.

stream buffer for A 's successors, which are $A + 1, A + 2, A + 3, A + 4$. Similarly, the access to B results in a cache miss, and a stream buffer is allocated for B 's successors $B + 1, B + 2, B + 3, B + 4$. Next, for an access to $A + 2$, the cache and the head entries of the stream buffers are checked for a match. Since there is no match, a new stream buffer is allocated for $A + 2$'s successors $A + 3, A + 4, A + 5, A + 6$. However, an access to $B + 1$ will find a match in the stream buffer, and $B + 1$ will be moved into the L1 cache, and the freed entry is used to prefetch the successor of the tail entry.

Note that now we have two stream buffers allocated for A 's successors and $A + 2$'s successors, and some entries overlap in two stream buffers ($A + 3$ and $A + 4$). This is caused by the inability to identify a stride of 2 for A 's stream. In fact, it cannot handle all non-sequential accesses efficiently, including non-unit strides ($s > 1$) or even a negative sequential access ($s = -1$). For non-unit stride accesses, each access could result in a stream buffer miss because the address is only compared to the head entries of the buffers. Not only can this produce overlapping entries, a single stream can also occupy several stream buffers. To overcome that, Palacharla and Kessler [36] suggested two techniques to enhance the stream buffers: allocation filters and a non-unit stride detection mechanism. The filter waits until there are two consecutive misses for the same stream before allocating a stream buffer, making sure that a stream buffer is only allocated for strided accesses. A dynamic non-unit stride detection was also proposed by determining the minimum signed difference between the miss address and the past N miss addresses. Farkas et al. [13] attempt to accurately detect the stride amount by using not only load addresses, but also the PCs of the load instructions. A stride address prediction table, indexed by the PC of a load, is used to learn and record dynamic strides. Note that, with larger cache blocks, more strided access patterns will appear sequential. Sherwood et al. [41] augments the stream buffers with a correlation-based predictor that detects non-strided accesses.

Overall, prefetching for sequential and strided accesses is achievable with relatively simple hardware. Several recent machines, such as Intel Pentium 4 and IBM Power4 architectures, implement a hardware prefetcher that is similar to stream buffers at the L2 cache level [14, 19]. For example, the L2 cache in the Power4 can detect up to eight sequential streams. The L2 cache line is 128 bytes, and therefore a sequential stream detector can catch most of strided accesses. On a cache miss to a line in the L2 cache, the fifth successor line is prefetched. This is similar to a 4-entry stream buffer. The L3 cache block is four times larger than the L2 (512 bytes), and therefore it only prefetches the next successive line on an L3 miss. Since the prefetching is based on physical addresses, prefetching of a stream is terminated when a page boundary is encountered. However, for applications with long sequential accesses, continuity in the prefetching can be supported if large pages (16-MB) are used.

7.3.2 Correlation Prefetching

While prefetching strided or sequential accesses can be accomplished with relatively simple hardware, such as stream buffers, there is less clear solution to prefetching for non-strided (*irregular*) accesses. This irregularity often results from accesses in linked data structures (pointer dereferences) or sparse matrices (matrix or array dereferences). Fortunately, although these accesses appear random, they are often repeated. For example, in a linked list, if the structure is traversed, the addresses produced will be repeated as long as the list is not modified too often. For a more irregular linked data structure such as a tree or a graph, it is still possible to have access repetition if the traversal of the structures is fairly unchanged. This behavior is exploited by correlation prefetching.

Correlation Prefetching uses past sequences of reference or miss addresses to predict and prefetch future misses [42, 43, 1, 5, 20, 27, 41, 9, 15, 16]. It identifies a correlation between pairs or groups of addresses, for example between a miss and a sequence of successor misses. A typical implementation of pair-based schemes uses a *Correlation Table* to record the addresses that are correlated. Later, when a miss is observed, all the addresses that are correlated with its address are prefetched. Correlation prefetching has general applicability in that it works for any access or miss patterns as long as the miss address sequences repeat. This includes strided accesses, even though it will be more cost-effective to use a specialized stride or sequential prefetcher instead. The drawbacks of correlation prefetching are that it needs to be trained to record the first appearance of the sequences, and to be effective, its table size needs to scale with the working set size of the program.

Several studies have proposed hardware-based implementations [1, 5, 20, 27, 41], typically by placing a custom prefetch engine and a hardware correlation table between the processor and L1 caches or between the L1 and L2 caches, or in the main memory for prefetching memory pages. Figure 7.5a

shows how a typical (*Base*) correlation table, as used in [5, 20, 41], is organized. Each row stores the tag of an address that missed, and the addresses of a set of *immediate* successor misses. These are misses that have been seen to *immediately* follow the first one at different points in the application. The parameters of the table are the maximum number of immediate successors per miss (*NumSucc*), the maximum number of misses that the table can store predictions for (*NumRows*), and the associativity of the table (*Assoc*). According to [20], for best performance, the entries in a row should replace each other with a LRU policy.

Figure 7.5a illustrates how the algorithm works. The figure shows two snapshots of the table at different points in the miss stream ((i) and (ii)). Within a row, successors are listed in Most Recently Used (MRU) order from left to right. At any time, the hardware keeps a pointer to the row of the last miss observed. When a miss occurs, the table learns by placing the miss address as one of the immediate successors of the last miss, and a new row is allocated for the new miss unless it already exists. When the table is used to prefetch ((iii)), it reacts to an observed miss by finding the corresponding row and prefetching all *NumSucc* successors, starting from the MRU one. In the figure, since the table has observed a, b and a, d in the past, when a is observed, both b and d are prefetched. Note that the figure shows two separate steps: the learning step, where the table records and learns the address patterns ((i) and (ii)), and prefetching step, where the table is used for prefetching. In practice, the prefetching performs better when both steps are combined [20, 42, 43].

There are enhancements to the *Base* correlation prefetching. Sherwood et al. combines the prefetcher with traditional stream buffers, combining the ability to prefetch regular and irregular accesses [41]. However, its location near the processor limits the size of the table, limiting its effectiveness for programs with large working sets. Lai et al. uses the PC information as well as addresses to index the correlation table [27]. Although the correlation information can be made more accurate, it is more complicated because PCs need to be obtained. It also requires longer training.

A drawback of the *Base* approach is that to be effective, it needs a large correlation table. Proposed schemes typically need a 1-2 Mbyte on-chip SRAM table [20, 27], while some applications with large footprints even need a 7.6 Mbyte off-chip SRAM table [27]. In addition, prefetching only the *immediate* successors for each miss [5, 20, 41] limits the prefetcher's performance. The prefetching needs to wait until a "trigger" miss occurs before it can prefetch its immediate successors. The trigger misses cannot be eliminated since they are needed by the prefetcher to operate. Therefore, it has a low *coverage*, defined as the fraction of the original of misses that are prefetched [20]. In addition, the immediate successor prefetch may not be timely enough to hide its full memory latency.

One possible extension to the *Base* scheme is to reindex the table and follow the successor chains starting from the current miss address. This approach, which can potentially increase coverage and prefetching timeliness,

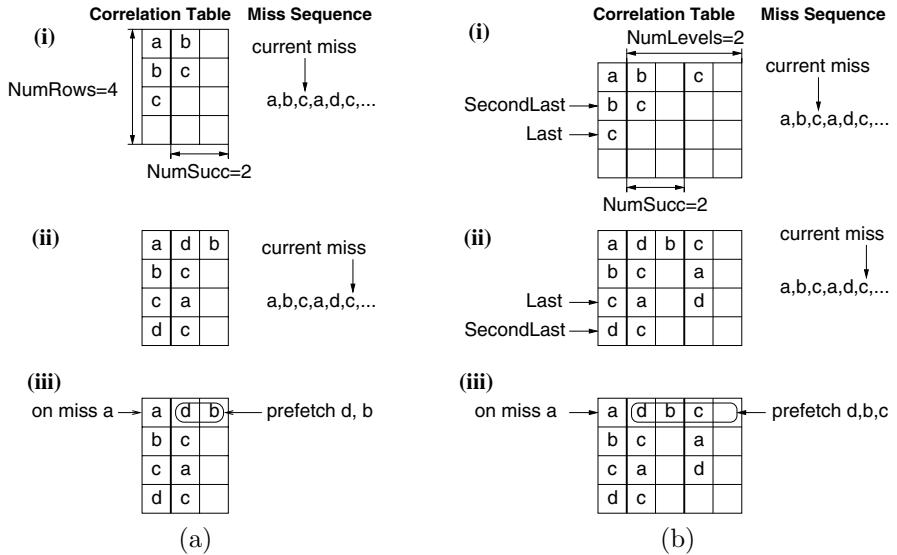


FIGURE 7.5: Correlation prefetching algorithms: *Base* (a) and *Replicated* (b). The figure is modified from [42].

has a drawback in that the accuracy of prefetching quickly degrades as the chain is traversed [42, 43]. To deal with the drawbacks of *Base* correlation prefetching, Solihin et al. proposed an alternative *Replicated* correlation table [42, 43] that records multiple levels of successors in each entry of the correlation table. This scheme is shown in Figure 7.5b. As shown in the figure, *Replicated* keeps *NumLevels* pointers to the table. These pointers point to the entries for the address of the last miss, second last, and so on, and are used for efficient table access. When a miss occurs, these pointers are used to access the entries of the last few misses, and insert the new address as the MRU successor of the correct level ((i) and (ii)). In the figure, the *NumSucc* entries at each level are MRU ordered. Finally, prefetching in *Replicated* is simple: when a miss is seen, all the entries from different successor levels in the corresponding row are prefetched ((iii)).

Replicated can prefetch multiple levels of successors more accurately compared to traversing a successor chain, because they contain the *true MRU* successors at each level. This is the result of grouping together all the successors from a given level, irrespective of the path taken. In [42, 43], three levels of successors can be predicted without losing much accuracy.

Correlation prefetching can be implemented in hardware [1, 5, 20, 27, 41, 15, 16] or in software [42, 43]. A software implementation is slower, and therefore is more suitable for an implementation near the memory. An advantage of a software implementation is that the correlation tables can be stored in the main memory, eliminating the need for an expensive specialized storage.

In addition, the prefetching algorithm can be customized based on the application characteristics [42, 43].

7.3.3 Content-Based Prefetching

Another class of prefetching examines the content of data being fetched to identify whether it contains an address or pointer that is likely to be accessed or dereferenced in the future, and prefetch it ahead of time. Note that this technique only works for prefetching pointers in linked data structures, and do not tackle regular or indirect matrix/array indexing.

Roth et al. [38] proposed a technique to identify a pointer load, based on whether a load (consumer) is an address produced by another load instruction (producer). If the producer and consumer are the same static load instructions, they are categorized as *recurrent loads*. Recurrent load results from pointer chasing, for example in the form of $p = p \rightarrow \text{next}$. Otherwise, they are categorized as *traversal loads*. Other loads that do not fetch pointers are categorized as *data loads*. A hardware that dynamically detects and prefetches the various pointer loads was proposed. Ibanez et al. characterized the frequency of different types of loads, including pointer loads in the Spec benchmarks and proposed a prefetching scheme for them [18].

Cooksey et al. [10] proposed content-directed prefetching, where pointers are identified and prefetched based on a set of heuristics applied to the content of a memory block. The technique borrows from conservative garbage collection, in that when data is demand-fetched from memory, each address-sized word of the data is examined for a “likely” address. Candidate addresses need to be translated from the virtual to the physical address space and then issued as prefetch requests. As prefetch requests return data from memory, their contents are also examined to retrieve subsequent candidates. A set of heuristics are needed to identify “likely” addresses.

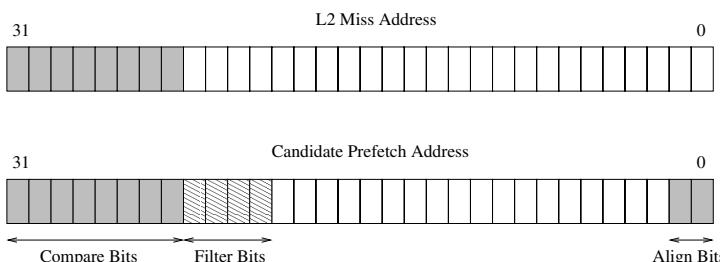


FIGURE 7.6: Heuristics for pointer identification. The figure is taken from [10], with minor modifications.

The heuristics are based on scanning each four-byte chunk on every loaded cache line. Each chunk is divided into several sections, as shown in Figure 7.6. The figure shows an L2 miss address that causes a block to be brought into the L2 cache, and a four-byte chunk from the block just brought in that is being considered as a candidate prefetch address. The first heuristic compares a few upper bits (*compare bits*) of the candidate prefetch address with ones from the miss address. If they match, the candidate prefetch address may be an address that shares the same base address as the miss address. This heuristic relies on the fact that linked data structure traversal may dereference many pointers that are located nearby, i.e., they share the same base address. Other pointers that jump across large space, for example a pointer from the stack to heap region, cannot be identified by the heuristic as addresses. The heuristics works well except for the two regions where the compare bits are all 0's or all 1's. Small non-address integers with 0's in their compare bits should not be confused as addresses, and therefore need a different heuristic to distinguish them from addresses. In addition, large negative numbers may have all 1's in the compare bits, and therefore need a different heuristic to avoid them to be identified as addresses. One alternative would be to not consider prefetch addresses that have all 0's and all 1's in their compare bits. Unfortunately, these cases are too frequent to ignore because many operating systems allocate stack or heap data in those locations. To get around it, a second heuristic is employed. If the compare bits are all 0's, the next several bits (*filter bits*) are examined. If a non-zero bit is found within the filter bit range, the data value is deemed to be a likely address. In addition, if the compare bits are all 1's, the next several bits (*filter bits*) are examined. If a zero bit is found within the filter bit range, the data value is deemed to be a likely address. Finally, a few lower bits (*align bits*) also help in identifying pointers. Due to memory alignment restriction, it is likely that addresses pointed by a pointer begin at the start of 4-byte chunks in memory. Therefore, the lower 2 bits of the pointer should be '00' in this case.

One inherent limitation of content-based prefetching is that a prefetch cannot be issued until the content of a previous cache miss is available for examination. Therefore, for dependent pointer loads, such as in linked data structure traversal, pointer chasing of n -linked nodes still result in n times the latency of a single pointer load. If each pointer load results in a cache miss, the pointer chasing critical path may still be significant. This limitation does not apply to sequential/stride and correlation prefetching which do not need to observe pointer load dependences.

References

- [1] T. Alexander and G. Kedem. Distributed Predictive Cache Design for High Performance Memory Systems. In *the Second International Symposium on High-Performance Computer Architecture*, pages 254–263, February 1996.
- [2] B. Calder, C. Krintz, S. John, and T. Austin. Cache-Conscious Data Placement. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 139–149, San Jose, CA, October 1998. ACM.
- [3] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.
- [4] J. B. Carter et al. Impulse: Building a Smarter Memory Controller. In *the 5th International Symposium on High-Performance Computer Architecture*, pages 70–79, January 1999.
- [5] M. J. Charney and A. P. Reeves. Generalized Correlation Based Hardware Prefetching. *Technical Report EE-CEG-95-1, Cornell University*, February 1995.
- [6] T. F. Chen and J. L. Baer. Reducing Memory Latency via Non-Blocking and Prefetching Cache. In *the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, October 1992.
- [7] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. mei W. Hwu. Data Access Microarchitectures for Superscalar Processors with Compiler-Assisted Data Prefetching. In *Proceedings of the 24th International Symposium on Microarchitecture*, 1991.
- [8] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-Conscious Structure Layout. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, Atlanta, GA, May 1999. ACM.
- [9] J. Collins, S. Sair, B. Calder, and D. Tullsen. Pointer Cache Assisted Prefetching. In *Proc. of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov 2002.
- [10] R. Cooksey, S. Jourdan, and D. Grunwald. A Stateless, Content-Directed Data Prefetching Mechanism. In *ASPLOS*, 2002.

- [11] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, Sept. 1994.
- [12] C. Ding and K. Kennedy. Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 229–241, Atlanta, GA, May 1999. ACM.
- [13] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. Memory-system Design Considerations for Dynamically-scheduled Processors. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, 1997.
- [14] G. Hinton and D. Sager and M. Upton and D. Boggs and D. Carmean and A. Kyker and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, (First Quarter), 2001.
- [15] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior. In *29th International Symposium on Computer Architecture*, May 2002.
- [16] Z. Hu, M. Martonosi, and S. Kaxiras. Tag Correlating Prefetchers. In *9th Intl. Symp. on High-Performance Computer Architecture*, Feb 2003.
- [17] C. J. Hughes. Prefetching Linked Data Structures in Systems with Merged DRAM-Logic. Master's thesis, University of Illinois at Urbana-Champaign, May 2000. Technical Report UIUCDCS-R-2001-2221.
- [18] P. Ibanez, V. Vinals, J. Briz, and M. Garzaran. Characterization and Improvement of Load/Store Cache-Based Prefetching. In *International Conference on Supercomputing*, pages 369–376, July 1998.
- [19] IBM. *IBM Power4 System Architecture White Paper*, 2002. <http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.html>.
- [20] D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. In *the 24th International Symposium on Computer Architecture*, pages 252–263, June 1997.
- [21] N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *the 17th International Symposium on Computer Architecture*, pages 364–373, May 1990.
- [22] M. Karlsson, F. Dahlgren, and P. Stenstrom. A Prefetching Technique for Irregular Accesses to Linked Data Structures. In *Proceedings of the 6th International Conference on High Performance Computer Architecture*, Toulouse, France, January 2000.

- [23] M. Karlsson, F. Dahlgren, and P. Stenstrom. A Prefetching Technique for Irregular Accesses to Linked Data Structures. In *the 6th International Symposium on High-Performance Computer Architecture*, pages 206–217, January 2000.
- [24] A. C. Klaiber and H. M. Levy. An Architecture for Software-Controlled Data Prefetching. In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 43–53, Toronto, Canada, May 1991. ACM.
- [25] D. Koufaty and J. Torrellas. Comparing Data Forwarding and Prefetching for Communication-Induced Misses in Shared-Memory MPUs. In *International Conference on Supercomputing*, pages 53–60, July 1998.
- [26] D. Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of 8th International Symposium on Computer Architecture*, pages 81–87. ACM, May 1981.
- [27] A. Lai, C. Fide, and B. Falsafi. Dead-Block Prediction and Dead-Block Correlating Prefetchers. In *the 28th International Symposium on Computer Architecture*, pages 144–154, June 2001.
- [28] C.-K. Luk and T. C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Cambridge, MA, October 1996. ACM.
- [29] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving Memory Hierarchy Performance for Irregular Applications. In *Proceedings of the 1999 ACM International Conference on Supercomputing*, Rhodes, Greece, June 1999.
- [30] N. Mitchell, L. Carter, and J. Ferrante. Localizing non-affine array references. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Newport Beach, CA, Oct. 1999.
- [31] T. Mowry. Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching. *Transactions on Computer Systems*, 16(1):55–92, February 1998.
- [32] T. Mowry and A. Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.
- [33] T. Mowry, M. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 62–73, Boston, MA, October 1992. ACM.

- [34] T. C. Mowry. Tolerating Latency Through Software-Controlled Data Prefetching, PhD Thesis. Technical report, Stanford University, March 1994.
- [35] NVIDIA. Technical Brief: NVIDIA nForce Integrated Graphics Processor (IGP) and Dynamic Adaptive Speculative Pre-Processor (DASP). <http://www.nvidia.com/>.
- [36] S. Palacharla and R. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *the 21st International Symposium on Computer Architecture*, pages 24–33, April 1994.
- [37] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6), June 1990.
- [38] A. Roth, A. Moshovos, and G. Sohi. Dependence Based Prefetching for Linked Data Structures. In *the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, October 1998.
- [39] A. Roth and G. S. Sohi. Effective Jump-Pointer Prefetching for Linked Data Structures. In *Proceedings of the 26th International Symposium on Computer Architecture*, Atlanta, GA, May 1999.
- [40] C. Scheurich and M. Dubois. Concurrent Miss Resolution in Multiprocessor Caches. In *Proceedings of the 1988 International Conference on Parallel Processing*, 1998.
- [41] T. Sherwood, S. Sair, and B. Calder. Predictor-Directed Stream Buffers. In *the 33rd International Symposium on Microarchitecture*, pages 42–53, December 2000.
- [42] Y. Solihin, J. Lee, and J. Torrellas. Using a User-level Memory Thread for Correlation Prefetching. In *29th International Symposium on Computer Architecture (ISCA)*, May 2002.
- [43] Y. Solihin, J. Lee, and J. Torrellas. Correlation Prefetching with a User-Level Memory Thread. *IEEE Transactions on Parallel and Distributed Systems*, June 2003.
- [44] A. Stoughton, J. N. Amaral, G. R. Gao, J. C. Dehnert, S. Jain, and A. Douillet. Speculative Pointer Prefetching of Induction Pointers. In *Compiler Construction 2001, European Joint Conferences on Theory and Practice of Software*, Genova, Italy, April 2001.
- [45] M. E. Wolf and M. S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM, June 1991.
- [46] C.-L. Yang and A. R. Lebeck. Push vs. Pull: Data Movement for Linked Data Structures. In *International Conference on Supercomputing*, pages 176–186, May 2000.

- [47] Z. Zhang and J. Torrellas. Speeding up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching. In *the 22nd International Symposium on Computer Architecture*, pages 188–199, June 1995.

Chapter 8

Address Prediction

Avi Mendelson

Intel Mobil Micro-Processor Architect

8.1	Introduction	187
8.2	Taxonomy of Address Calculation and Prediction	189
8.3	Address Prediction	194
8.4	Speculative Memory Disambiguation	200
8.5	Summary and Remarks	211
	References	212

8.1 Introduction

As the gap between processor speed and memory access time increases, the average service time of memory-based operations is becoming a dominant factor in the overall performance of modern processors. RISC processors, such as MIPS [10], can access the cache memory within a single cycle and guarantee a latency of three cycles between the time a Load instruction was fetched and the time the instructions that depends on the value of the Load can use it (i.e., *fetch-load-to-use* delay). However, the machine itself can execute at most a single instruction per cycle and uses a relatively slow clock frequency. Modern processors can execute several instructions per cycle in an *Out-of-Order* (OOO) fashion, run in much faster clock frequency, due to its deep pipeline design, and use several levels of memory hierarchies. However, it takes them from two to five cycles to access the fastest level of the cache hierarchy, depending on the size and the layout of the cache. Additionally, the *fetch-load-to-use* delay can vary from five to hundreds of cycles, even if the data reside in the fastest cache.

Most of the current processors handle memory operations and arithmetic operations in a different manner. RISC architectures make this distinction at the Instruction Set Architecture (ISA) level by separating Load and Store instructions from all other kinds of operations. CISC architectures, such as IA32, make the distinction at the micro-architectural level by breaking up the CISC instructions that allow a mix of arithmetic and memory operations into a sequence of internal operations, termed micro-ops, which preserve the same separation used for RISC architectures.

Memory operations can be divided into *Load* operations, which move data from a memory location to the processor, and *Store* operations, which move data from the processor to a memory location. Load and Store operations have different contributions to performance. *Load* operations need to be performed as soon as possible to allow operations on the critical path to be executed as soon as possible. *Store* operations have only limited direct impact on the performance. Thus, for a long time, most of the research in advanced architectures focused on reducing the *fetch-load-to-use* delay. This delay is affected by different factors, including (1) fetch and decode latencies, (2) optimal scheduling of *Loads* to access the memory, (3) the time it takes to retrieve the data from the memory hierarchy, and (4) the time it takes to deliver the data to the dependent instructions. Recently, more attention has been given to *Store* instructions. Although *Store* instructions have a minimal direct impact on performance, the dependencies between *Loads* and *Stores* can have a major impact on the *fetch-load-to-use* latency. Previous chapters of this book deal with different components of the *fetch-load-to-use* delay, such as the impact of the physical organization of the memory hierarchy on the access time of the data, the use of prefetching techniques to reduce the latencies, and more. This chapter focuses on techniques that take advantage of address prediction and dependency prediction to reduce the *fetch-load-to-use* delay.

For a given frequency, the performance a program can achieve when run on a particular processor depends on the internal dependencies within the program and the internal micro-architecture of processor. The program dependencies can be represented by the *Data Flow Graph* (DFG) of the program. A node in the DFG represents an operation the processor needs to perform and an edge in the graph represents one of the following two dependencies: register based dependency or memory based dependency. *Register based dependency* occurs when one instruction writes a register and a subsequent instruction reads the same register. *Memory based dependency* occurs when a *Store* instruction writes to a memory location and a subsequent *Load* instruction reads that same location.

Although register based dependencies and memory based dependencies appear very similar, they differ from each other in their nature. There are two main sources for these differences:

- 1. The time when the addresses are known and the ensuing resolution of when the dependencies can start.** For *register based operations*, the addresses of the registers are given as part of the decode information; therefore, the resolution of the dependencies can start at *after decode time*. For *memory based operations*, the addresses need to be calculated before the hardware can start resolving the dependencies. This calculation can take a short time if all the data needed are available, or a very long time if the data needed for the address calculation depends on another instruction that has not been completed.

2. **The time when the latencies of different instructions are known to the scheduler:** For *register based operations*, the latency of each instruction is fixed and known at fetch time; therefore, the scheduler can use this information to optimize the scheduling. For *memory based operations*, this time is known very late in the pipe since it depends on different parameters, such as the location of the information it is waiting for. As a result, scheduling memory based instructions is a very challenging task for the system.

The differences between these two classes of dependencies causes most systems to use different mechanisms for handling *register based operations* and *memory based operations* [1][10]. Out-of-Order based computers, such as Alpha[©] or Pentium^{©4}, keep all the register based operations that are *in-flight* in a structure called ROB (re-order buffer) and all the memory based *in-flight operations* in a structure called MOB (memory order buffer). In both structures, the instructions are kept in the “program order” so on one hand, the scheduler can decide, depending on the availability of the inputs and the availability of the resources, which operation to send for execution, and on the other hand, the system can preserve the in-order nature of the program at the retirement stage of the execution.

This chapter focuses on speculative methods for improving the *fetch-load-to-use* delay using address prediction and dependency prediction techniques in out-of-order computer architectures. Section 8.2 provides a short taxonomy of the address calculation and prediction techniques, Section 8.3 explores the different methods being proposed for predicting addresses, and Section 8.4 describes different techniques that use address prediction to improve the *fetch-load-to-use*. Section 8.5 concludes this chapter with a summary and remarks.

8.2 Taxonomy of Address Calculation and Prediction

The taxonomy of address calculation and address prediction techniques is included here to complete the discussion. We use this taxonomy to define the terminology used throughout this chapter and to draw a clear line between the topics covered in this chapter, related topics covered elsewhere in other chapters of this book, and a few other topics that are beyond the scope of this book and hence not covered. Readers who are interested in broadening their knowledge in this area can find more information in the following two surveys, [3] and [18].

8.2.1 Terminology and Definitions

We begin with a few definitions:

Definition 1 Address calculation: *An address is calculated for every Load, Store, or Prefetch operation, and is computed before the operation is scheduled for execution.*

Definition 2 Conflict of addresses: *If a Load operation is ready to be scheduled and there is an in-flight Store with an unresolved address that precedes it in the program's order, the addresses of the Store and the Load are conflicting.*

Definition 3 Collision of addresses: *If the address of a conflicting Store operation happens to access the same address as the Load that was ready to be scheduled, or that was speculatively scheduled to access the memory before the address of the Store was resolved, the addresses of the Store and the Load are colliding.*

Definition 4 Load promotion: *A Load is said to be promoted, if it is sent for execution out of its order with respect to other Loads or Stores in the program. Load promotion can be done either statically by the compiler or at run-time by the hardware.*

Definition 5 Address prediction: *An address is predicted if a Load, a Store, or a Prefetch operation is issued before the calculation of its memory addresses was completed, or if a Load is issued based on the prediction that a conflicting Store will not collide.*

Definition 6 Data bypassing: *If a Load gets its value directly from another Store operation, rather than from the upper level of the memory hierarchy (usually the main memory), the data was bypassed to the Load.*

Definition 7 Dependency prediction: *This aims at predicting whether different memory operations depend on each other. The predictor does not predict the addresses the Load or Store operations will access, it predicts whether or not they will access the same memory address. Current dependency predictors try to predict load-store dependencies as well as load-load dependencies.*

Note that some of these definitions can be applied to speculative techniques, as well as non-speculative techniques. For example, data can be bypassed based on a known address in a non-speculative fashion, or it can be applied speculatively. The remainder of this section provides a short description of different techniques used to accelerate address calculation and to perform address prediction. We classify the different methods into speculative vs. non-speculative techniques and denote which of them will be expanded upon later on in this chapter.

8.2.2 Non-speculative Address Calculation Techniques

Modern out-of-order computers present two different views of the system. Externally (i.e., architecture view), the system needs to execute operations in the order of the program (“in order”). However, internally (i.e., micro-architecture view), the system can issue the operations out of their original order, as long as the semantics of the program are maintained. The traditional way to allow memory based operations to be executed out of their original order is to force all Stores to preserve the order among them, but to allow Loads to be promoted under the following condition:

Definition 8 *If no speculation is allowed, a **Loads** can be promoted iff*

Its address is known

and

All addresses of the Stores that precede its execution are known

In order to accelerate the execution of *Loads* under *non-speculation* environment, several methods have been proposed:

Compiler based optimizations: If the addresses of the Load and all Stores that precede it can be calculated at compile time, or can be guaranteed not to access the same addresses, the compiler can change the original order of the Loads so they can be fetched and execute as soon as possible, without changing the semantics of the program [14].

Register tracking: In order to accelerate the address calculation, Bekerman et al. [2] suggest adding special hardware for address calculation to the front-end of the out-of-order machine. For example, their work shows that by tracking all operations that may affect the stack pointer, we can determine the addresses of future stack based operations as soon as they are decoded.

Decoupled architecture: Another way to advance address calculation is to devote part of the system to address calculation, as proposed for example by [21, 15, 16]. Here, the execution section of the processor attempts to calculate all the instructions that lead to the required address, so the result of the address calculation is available as soon as possible. For example, when clustered architecture is used, some systems suggest dedicating one cluster to address calculation. This kind of architecture can guarantee that instructions on the critical path of memory based operations will resolve their address calculation as soon as possible and will be scheduled for execution without unnecessary delays.

Helper threads: This is a name for a class of techniques, some of which are classified as non-speculative techniques and others are classified as speculative techniques. The technique is discussed in detail in a previous chapter of this book, and will therefore not be detailed here. However, it is important to note that helper threads can be considered a form of decoupled architecture, where special hardware (or an extension of the existing hardware) is used to accelerate the calculation of “hard to predict” Loads. In this case, the compiler can prepare the speculative code that will run in the presence of a long latency event, such as L2 cache miss [22, 4, 19].

This chapter focuses on speculative techniques for address calculation; therefore, most of the non-speculative techniques described here are beyond the scope of this chapter. Some of these techniques can also be applied speculatively (e.g., some hardware extensions allow compilers to issue speculative work) and are discussed later in this chapter.

8.2.3 Speculative Address Calculation Techniques

In order to increase the *memory level parallelism* (MLP), most modern architectures allow addresses to be speculated or predicted. *Next-line* prefetcher is an example of speculative address calculation, since it starts the calculation process before the data are actually demanded (requested) [17]. The use of a value prediction based mechanism to predict the outcome of an address calculation is an example of address prediction.

This subsection provides a short description of the classes of address prediction and calculation techniques.

Hit-Miss prediction: This simple form of address prediction calls predict whether an access to a specific level of the cache will result in a hit or miss. This information is very important to the scheduler in order to optimize its performance. Many modern processors (e.g., Pentium[©]4 [11]) assume that data always hit the first level of the memory hierarchy and only if proven wrong, the architecture recovers its state and re-fetches the right value.

The discussion on Hit-Miss prediction is beyond the scope of this chapter and will not be expanded upon.

Way prediction: For some systems, accessing different ways in the same set may have different access time. For such systems adding way prediction can be used to help to optimize the scheduler. The discussion on *Way prediction* is beyond the scope of this chapter and will not be expanded upon.

Prefetching: Prefetch mechanisms try to predict the addresses that memory based operations will access in the near future. Based on this assumption, the software and/or the hardware can speculatively start fetching

these addresses ahead of time. We can characterize different prefetching techniques with respect to the way they predict the addresses to be fetched:

- Prefetcher can start fetching addresses according to their basic characteristics. For example, based on locality of references, the prefetcher mechanism can anticipate that a neighbor's addresses will be accessed soon. Or, based on "stride of accesses" to the entire memory, or to specific regions of the memory (such as within a memory page), the hardware can decide to start prefetching data ahead of time.
- Address prediction: Any mechanism that can predict the outcome of calculations (value prediction) can be used to predict the outcome of address calculation, and, hence, can be used as a prefetcher [9]. Such a mechanism can be used to predict PC based access patterns or can be extended to predict more complicated patterns, such as anticipating the value of the next element in a link-list [6].

Prefetcher mechanisms are discussed in previous sections of this book. Thus, this chapter focuses on the comparison between the prediction based techniques and the address calculation techniques.

Speculative compiler based address calculation: Some instruction set architectures, such as Itanium[©] [20], add special instructions to allow the compiler to issue speculative Loads and recover from miss-speculations. The idea is to allow the compiler to define what the program needs to do if a Load collides with a Store at run-time.

Combining compiler based address prediction techniques with simple *in-order* architecture allows the system to achieve a high ILP level, without paying the price for complicated *out-of-order* machines.

Dependency prediction: Dependency prediction provides an early indication to distinguish between memory-based instructions that are dependent on each other and memory based operations that are independent of each other.

This information can be used by the system to achieve different goals:

- If a Load is predicted not to be dependent on any Store that precedes it, the schedule can promote the Load even before all the addresses of Stores that precede it in the program's order are known.
- A Load can be synchronized with a Store and thereby be prevented from being promoted, if the predictor indicates that the Load and Store may be dependent on each other.
- If Load and Stores are predicted to be dependent on each other, we can bypass the data between the Store to the Load through

internal buffers and before the addresses of the Load and the Store are resolved.

- If two Loads are predicted to access the same memory location, we can use the same “incoming” value for all Loads, even before their actual addresses are known.

8.2.4 Chapter Focus

This chapter focuses on speculative methods for reducing the *load-fetch-to-schedule*. In particular, we expand upon the following:

- **Address prediction:** We discuss the use of value prediction techniques as *address predictors* [26].
- **Comparison between address prediction and prefetching:** Although both techniques are used for the same purpose, we compare the characteristics of each technique.
- **Dependency prediction:** We focus on techniques that help identify dependencies, as well as techniques that help bypass information between different memory-based operations.
- **Compiler based speculative optimizations:** Some architecture adds special hardware hooks to allow speculative execution of compiler generated code. We focus on some of the techniques used by the Itanium[©] processor.

8.3 Address Prediction

Address prediction is a special case of value prediction, where the “values” being predicted are addresses of Load, Store, or Prefetch operation. Thus, all the techniques described in the chapter on Value Prediction can also be applied for address prediction.

There are different ways to predict an address of a memory operation; for example, all prefetch mechanisms described above need to predict addresses to be prefetched. One possible way of classifying the different address prediction techniques is to distinguish between non-history based techniques and history based techniques. Non-history based techniques predict the next address to be accessed, based on the characteristics of the program, such locality of references. History based address prediction, also termed “Table based prefetching mechanisms,” usually keeps the historical information, indexed by the PC of

the address of the memory address, and keeps information on the addresses that this instruction accessed in the past [9]. Stride prefetcher is a good example of such a mechanism.

This section focuses on the different aspects of address prediction, including: (1) characterization of the address prediction operations, (2) the relationship between value prediction and address prediction, and (3) the relationship between prefetch and address predictors.

8.3.1 Characterization of Address Predictability

High predictability of addresses is an enabler of many address prediction techniques. [26] classifies the predictability of Load instructions with respect to their memory location and their access pattern. The work suggests classifying the memory accesses with respect to their location attributes (i.e., whether they access the global, stack, or heap addresses) and with respect to the nature of their address generation. Address generation can be either scalar, generated by a single instruction, or non-scalar, generated by a sequence of instructions (e.g., access to an array element, a member of a list, etc.).

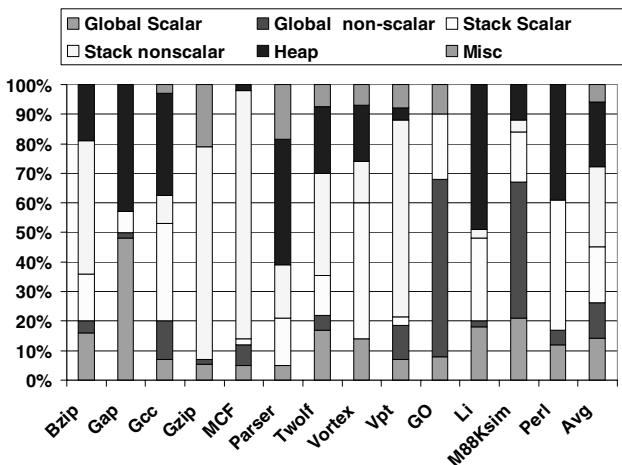


FIGURE 8.1: Load distribution among different applications.

As Figure 8.1 indicates, the distribution of Load instructions varies from one application to another. Although in MCF most address calculation was done for non-scalar addresses on the stack, GO application has very few such addresses, if at all.

Figure 8.2 and Figure 8.3 examine the correlation between the address calculation type and the predictability of the addresses belonging to that category. Figure 8.2 assumes three different types of address prediction: last value, stride, and context based prediction. This figure also provides a breakdown on the probability of predicting the address correctly (hit) or incorrectly (miss), assuming we know the type of address calculation.

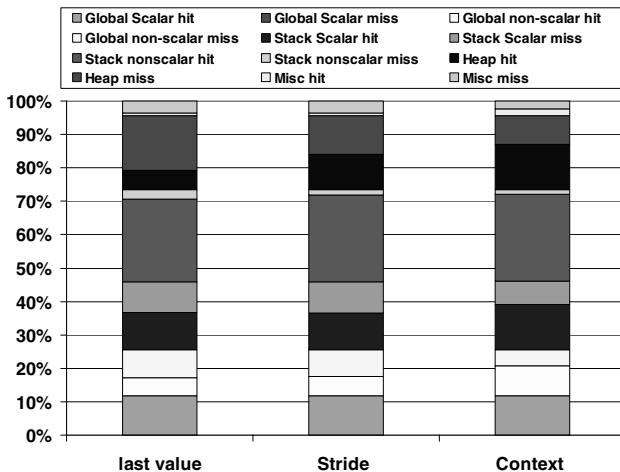


FIGURE 8.2: Breakdown of address predictability with respect to address classification.

Figure 8.3 uses the same address prediction mechanisms and gives the predictability (i.e., correct predictions out of all the predictions of that type) for each address calculation type.

As can be observed, the predictability of scalar operations when accessing global scalar data is quite high, while the predictability of heap operations is relatively small. Nonscalar accesses to the stack are somewhat surprisingly high, and the paper ([26]) explains this due to the nature of the accesses to the data on the stack.

In summary, the paper provides the overall predictability of addresses with respect to different address prediction techniques. The overall predictability of the addresses is given in Figure 8.4.

The results presented in Figure 8.4 were measured using infinite prediction tables and should be considered as an upper bound for the potential of address prediction. Nevertheless, the authors indicate that if different software techniques such as stack coloring are used, the prediction could be improved significantly over the numbers presented here.

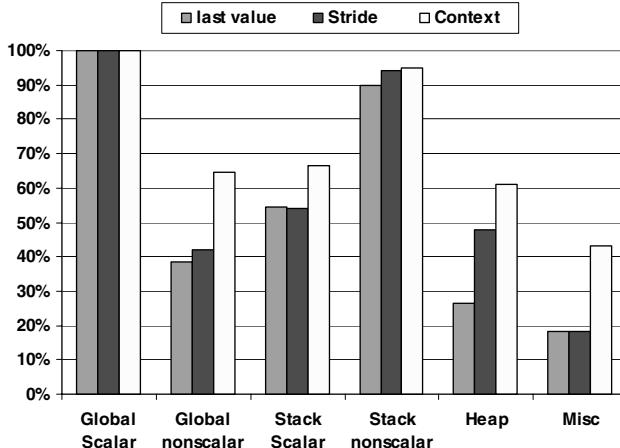


FIGURE 8.3: Predictability of addresses using different address predictors.

8.3.2 Address Predictability vs. Value Predictability

Address prediction and value prediction use the same hardware techniques. This makes it interesting to weigh the predictability of addresses as compared to the predictability of values. Gabbay et al. [8] compare the predictability of addresses and the predictability of values using two main criteria: (1) What percentage of the overall correctly predicted values in the system are used for address calculation? and (2) Can we predict both the address and the value of Load instructions, or only one of them?

Figure 8.5, based on data from [8], indicates the percentage used for address calculation out of all correctly predicted values. For completeness of the discussion, we add the distribution of Load instructions within the dynamic instruction stream of each program.

Figure 8.5 indicates that for most of the applications taken from SPECint, the predictability of the addresses is larger than their distribution within the dynamic instruction stream (i.e., the instructions that the processor executes at run-time). On average, 29.3% of all values being successfully predicted were used for address calculation, while their appearance in the dynamic instruction stream is limited to 23.4%. For some applications, such as m88ksim and compress, the predictability of the addresses is similar to their appearance in the instruction stream, and is approximately 20%. For other applications, such as go, li, jpeg, and vortex, the predictability of the addresses is significantly higher than the appearance of the Loads within the instruction stream.

A different aspect of the address predictability is provided by Figure 8.6 and Figure 8.7 (taken from [8]). For every Load operation, we compare whether its

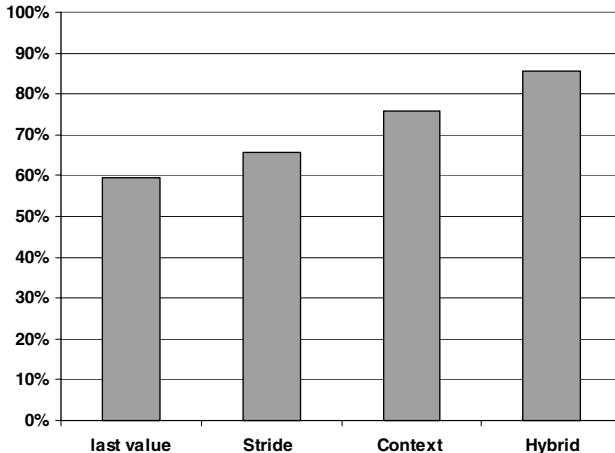


FIGURE 8.4: Predictability of addresses.

address and value were predictable. The results are divided into four possible combinations:

- Both the data values and addresses values can be predicted correctly
- Load references where only the address values can be predicted correctly
- Load references where only the data values can be predicted correctly
- Load references where neither the data values nor the address values can be predicted correctly.

Figure 8.6 compares these relationships between the value prediction and the address prediction for SPECint applications, while Figure 8.7 provides the same information for applications taken from SPECfp.

As we can observe, floating point based programs, taken from the SPECfp benchmark suite and integer based programs taken from the SPECint benchmark suite, behave quite differently with respect to their address and value predictability. For integer based applications (Figure 8.6), many of the Loads are not predictable at all, and we can expect the confidence level filter to prevent the offering of wrong predictions. For the rest of the Loads, more than 50% of the addresses are predictable in applications such as m88ksim and jpeg; many other applications also show good predictability of values and can be used to reduce the *fetch-load-to-use* delay. For all cases where both the address and value are predictable, we can improve both the fetch-load-to-use as well as the validation time of the operation.

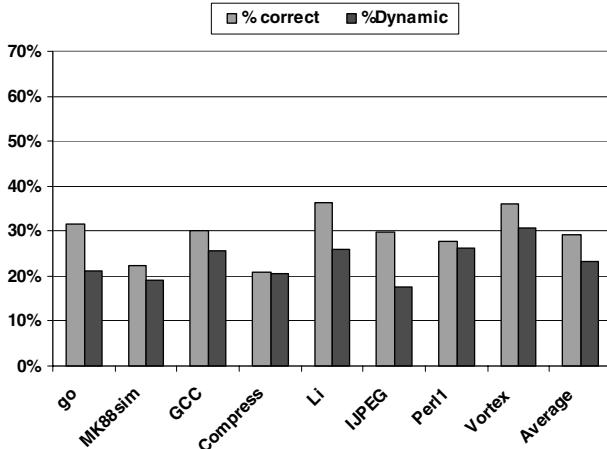


FIGURE 8.5: Address predictability out of all correctly predicted values - SPECint.

Floating-point based programs (Figure 8.7) present a different behavior. Here, for most of the Load operations, we can predict either only the address or both the address and the data. This behavior can be explained due to the fact that the majority of addresses in such programs are being used to walk matrices. In this case, a simple stride based address predictor should be sufficient to predict most of the addresses. Floating point values are rarely predicted. In the case of predictable values, we can assume that either sparse matrices were used and most of the values being loaded were zero (or other constant), or that the values measured were temporary values that have been saved and restored from the memory.

8.3.3 Combining Address Prediction with a Prefetching Mechanism

Several researchers suggest different techniques that combine value prediction and address prediction. For example, Gabbay and Mendelson [8, 7] suggest the use of two different stride based value predictions: a basic stride predictor that predicts only the value of the current instruction in the instruction queue and an advanced stride predictor that predicts all the values of all future appearances of the same instruction in the same instruction window. The advanced stride prefetcher value prediction serves as an efficient prefetcher when applied to instructions that aim to generate addresses and a machine that has a deep instruction window. The limited study of the performance

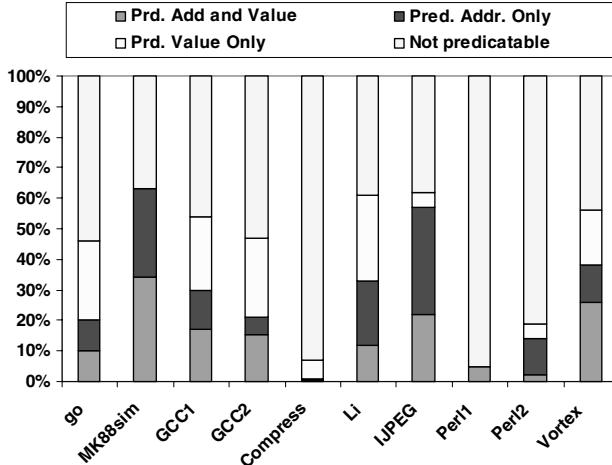


FIGURE 8.6: Value prediction vs. address prediction - SPECint.

presented there shows that such a prefetcher can be very useful for various applications such as vortex and m88ksim.

González [9] suggests another technique for combining value prediction together with a prefetch mechanism and integrating the new proposed technique as part of an out-of-order architecture. The technique, called “Address Prediction and Data Prefetching” (APDP), identifies memory based instructions whose effective address is highly predictable at decode time and starts executing them speculatively (as in the case of address prediction). For Load instructions, the proposed method also tries to predict the value of the next Load the instruction will use and start to prefetch this address as well.

The numbers presented in Figure 8.8 show that the use of prefetch techniques that are based on address prediction can contribute an average performance improvement of 13% for the architecture simulated in the paper. (The paper also uses address speculation techniques as part of this experiment, but Figure 8.8 ignores the effect of these techniques since it is explained in the next section.) For some applications, the improvement can be as high as 35%.

8.4 Speculative Memory Disambiguation

Out-of-order computers try to promote Loads and send them for execution as soon as possible, to reduce the *fetch-load-to-use* latency. A Load is said to

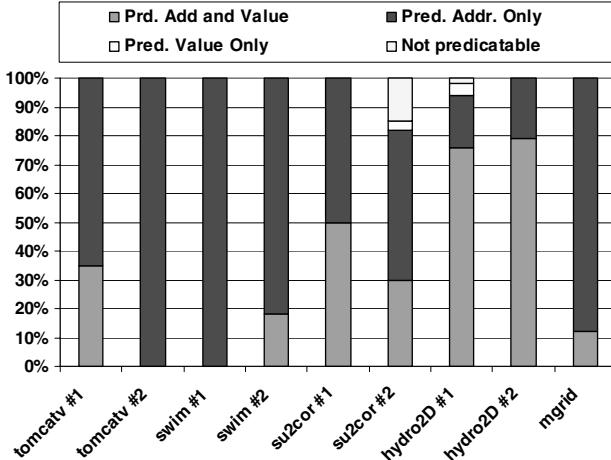


FIGURE 8.7: Value prediction vs. address prediction - SPECfp.

be safely promoted if it is scheduled only after its address is known and it can be guaranteed that no memory order violation will occur.

A Load is said to be speculatively promoted if it is scheduled based on the assumption that it will not access the same address (collide with) as any of the unknown Store addresses that precede it. The mechanism that is in charge of predicting memory order relations is termed *Speculative Memory Disambiguation* and can predict *read-after-write* as well as *read-after-read* dependencies, with respect to other memory operations with unknown addresses.

This section focuses on techniques that use address prediction or dependency prediction. We start this subsection with different characterizations of the use of address prediction for dynamic memory disambiguation and the performance potential of using these techniques. We continue the subsection with a description of the hardware implementations of different memory disambiguation techniques, which either aim to allow Load promotion or aim to bypass data from Store operations to Loads. We conclude this subsection with a description of software and hardware combinations that allow compilers to generate code that supports speculative Load promotion.

8.4.1 Basic Characterization

Modern out-of-order processors keep all *in-flight* memory operations sorted by their program order, in a structure called Memory Order Buffer (MOB). Memory operations can be scheduled out of this buffer when their address is known (or predicted); however, they need to be committed (retired) with respect to the MOB's order (program order)[10]. Most of the techniques used

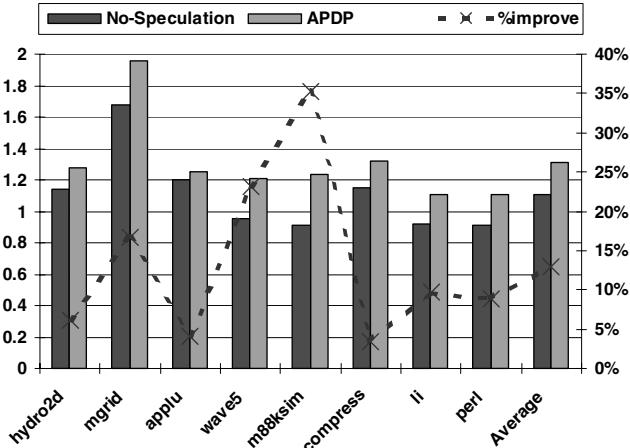


FIGURE 8.8: Performance gain for the APDP technique.

to handle out-of-order memory operations allow only Loads to be promoted, since recovery from miss-speculated out-of-order Store operations is much more complicated and requires special hardware. Because we are interested in speculative memory disambiguation, we begin this section with a summary of how many of these Loads are conflicting (see definitions 2), how many of them are colliding (see definitions 3), and what is the potential performance gain of dynamic memory disambiguation.

8.4.1.1 Conflicting vs. Colliding Memory Operations

Dynamic memory disambiguation is based on two assumptions: (1) Addresses are highly predictable (as seen in the previous section) and (2) Most memory operations do not depend on each other. Yoaz et al. [24] measured the distribution of colliding and conflicting Load instructions for a wide range of applications. Their experiment simulates an out-of-order machine; whenever a Load could be scheduled out of the instruction window, it was checked against all Stores in the MOB. If a Store with an unknown address that precedes it was found, they count it as a *conflict* Load. After the Store address was resolved, they again checked whether the Store actually *collided* with the Load's address.

Figure 8.4.1.1 shows the distribution of the Load instructions for an instruction window of 32 instructions.

Figure 8.4.1.1 indicates that although about 65% of the Loads were found to conflict with prior Stores, only about 10% of the Loads actually collided. As the size of the instruction window grows, the probability of a Load conflicting

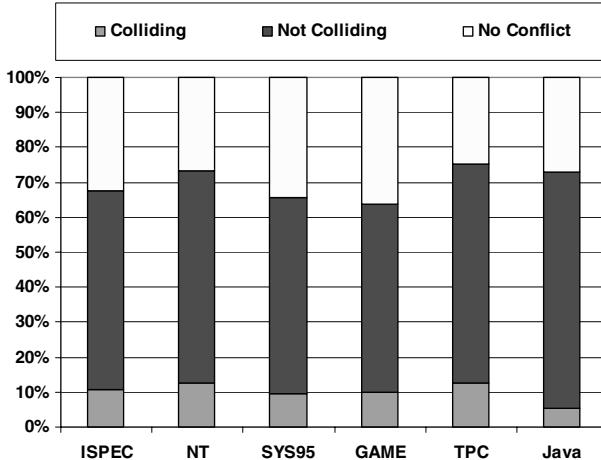


FIGURE 8.9: Percent of colliding loads vs. percent of conflicting loads.

with another Store and colliding with its address increases (as expected). However, the growing rate of the conflicting Loads is much higher than the growing rate of the colliding Loads.

These measurements indicate that as the instruction window increases, the use of the “traditional” techniques that wait for all the Store addresses to be resolved become inefficient. If a good address predictor can indicate with high confidence which of the Loads will collide with a Store and which of the Loads can be promoted, the potential for vast performance improvement increases with the size of the window.

8.4.1.2 Potential of Speculative Disambiguation

To evaluate the benefit of using aggressive memory disambiguation techniques, different papers, such as [18] and [5], measure the impact of using perfect address prediction on the overall performance of the system.

Figure 8.10 compares the execution of a set of applications on an out-of-order machine using traditional memory scheduling (similar to the Intel Pentium[®]Pro processor) where no speculative memory disambiguation is applied, with a similar processor that can correctly predict all the Store addresses. The results presented indicate that for some applications such as *jpeg* and *Perl*, the ability to promote Loads can increase by as much as 140% - 160%. For most of the applications, the gain was in the range of 75% - 105%, and for some applications such as *gcc* and *go*, the gain was as low as 20%. Note that the numbers presented by this experiment used system parameters that were representative a few years ago (i.e., 8 cycles access to L2 cache and 100 cycles

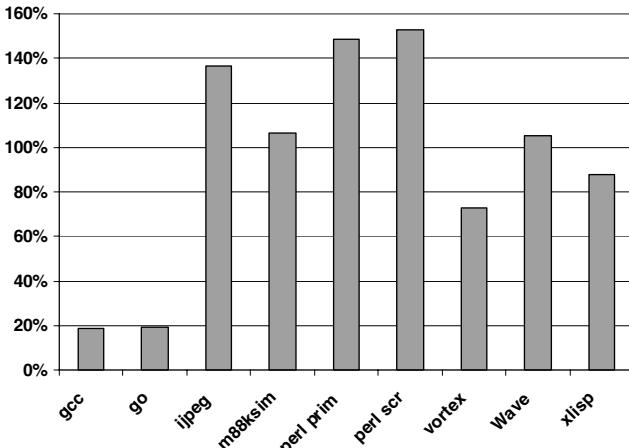


FIGURE 8.10: Speedup of perfect disambiguation over no speculation.

access time to the main memory). As the gap between the memory access time and the CPU frequency increases, the benefit of using such techniques increases as well.

8.4.2 Load Promotion

This section focuses on different techniques that use address prediction and conflict prediction to promote Loads speculatively. First, we describe some of the tradeoffs in designing such mechanisms and then extend the discussion to describe some of these techniques.

8.4.2.1 Design Tradeoffs

The techniques described in this section use address prediction to identify Load and Store pairs that communicate with each other. The predictor can predict a Load as one that will or will not collide. In return, the prediction can be either correct or incorrect. Thus, the memory accesses and the order prediction form five different combinations. It is important to explain the performance impact of each state in order to understand the design tradeoffs of the different dynamic memory disambiguation techniques.

1. **Not Conflicting:** A Load does not conflict with any of other memory locations at the time it is ready to be scheduled. This means that when the address of the Load is known, all the addresses of all Stores that precede this Load are known as well. If no Store accesses the same address as this Load, it can be scheduled safely. If a conflict exists, the

Load needs to wait until the dependency with that Store is completed and the Store value is bypassed to the Load.

2. **Actual Non-Collide, Predicted Non-Collide (ANCPNC):** If the schedule promotes a Load based on a prediction that it will not collide and the prediction was correct, the system can take full performance advantage to promote the Load.
3. **Actual Collide, Predicted Collide (ACPC):** If the scheduler did not promote a Load based on a correct prediction, the penalty paid can be justified.
4. **Actual Non-Collide Predicted Collide (ANCPC):** If the scheduler did not promote a Load, based on a wrong prediction, there is a loss of opportunity, but the penalty is relatively small.
5. **Actual Collide, Predicted Non-Collide (ACPNC):** If the scheduler promotes a Load based on a wrong prediction, the system needs to redo all the work it did based on the wrong assumption and reload the correct value.

The penalty the system pays for incorrect speculation depends on the complexity of the recovery mechanism. A simple recovery mechanism flushes the entire pipeline from the missed prediction point and re-executes all the instructions, including the miss-speculated Load. More complicated recovery mechanisms color all the instructions that may depend on a specific Load; if it was found to be miss-predicted, only the instructions that were based on the incorrect loaded value are re-executed. The design complexity of the selective recovery is beyond the scope of this chapter and we focus on the simple flush based technique.

8.4.2.2 Naive Dynamic Memory Disambiguation Technique

The simplest form of speculative memory disambiguation, termed naive speculation, allows all Loads to be promoted as long as we do not have prior knowledge that this Load collided in the past. Thus, the system keeps a list of addresses of Load instructions that caused a memory violation during their execution. Each Load that belongs to that list is not promoted speculatively.

Figure 8.11 shows the speedup of naive speculation as compared to the speedup of “perfect” memory, discussed in the previous section. It can be seen that in most the cases, the use of naive speculation gains much less performance than expected. For example jpeg gains only 6% out of 140% that the perfect memory gains, and the wave application loses performance when naive speculation is used. This is explained by the cost of the ACPNC type of disambiguation predictions. In this experiment, all the instructions after the misprediction point were squashed, re-fetched, and re-executed. This

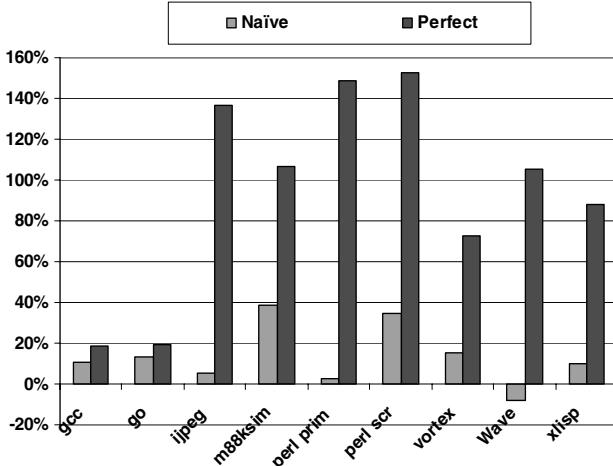


FIGURE 8.11: Speedup of naive speculation and perfect disambiguation over no speculation.

technique requires relatively simple support, but causes a significant performance overhead. Due to its simplicity, this mechanism is implemented by most existing high performance superscalar processors. A more sophisticated algorithm, termed selective recovery, is implemented by the Intel Pentium[®]4 processor [11]. This algorithm selectively squashes and selectively re-executes all the instructions that are dependent on the miss-predicted memory disambiguated Load. [17] provides a good comparison between these two classes of algorithms.

8.4.2.3 CHT-Based Implementation

An extension to the naive algorithm, proposed in [24], introduces the collision history table (CHT). The paper suggests using the CHT to keep additional information for each Load that causes a memory violation. In particular, the paper suggests adding information on the type of memory miss-prediction (e.g., ACPNC) and the maximum allowable promotion distance, to limit the distance a Load can be promoted. The latter is mainly needed for the case where the same Load collides with different Stores, each of them at a different distance. In order to avoid waiting too long, or scheduling the Load too soon, the paper suggests limiting the distance that the Load can move in any possible execution path.

The paper presents different possible implementations of the CHT table, including an area saving version that does not keep the address of the colliding Load, but keeps confidence level bits that indicate whether or not to

use the prediction. Using the confidence level estimation, together with the information on the types of miss-predictions, offers a very high accuracy rate with minimal execution overhead.

8.4.2.4 Store Set

The CHT method, described before, was based on the observation that a Load can collide with different Store instructions and thereby limit the distance that a Load can be promoted. The *Store Set* [5] technique extends this idea and call by allowing a Load to keep a set of all Stores it may interfere with and asking the scheduler to avoid promoting the Load as long as any of the Stores in the list are still in-flight with unknown addresses. The paper shows that this technique can accurately predict memory dependencies in the context of large instruction windows or out-of-order machines, and can achieve near-optimal performance compared to an instruction scheduler with perfect knowledge of memory dependencies.

The *Store Set* method starts with a mechanism similar to the naive speculation described above. For example, when an order violation is detected, an entry in the Store Set table is created. The next time the processor fetches that Load, it will not be promoted as long as the Store it depends on is “in-flight” with an unknown address. If the same Load creates a memory order violation with another Store, the address of that Store is added to the Store Set, and the scheduler will ensure that the Load waits for all Stores that belong to that set and are in-flight with unknown addresses.

The paper suggests the following simplifications to the hardware:

- Limits a Store to belong to at most a single *Store Set* at a time.
- Forces all the Stores belonging to the set to be executed in order.
- Makes a Load dependent on just one of the Stores in the set. Since all Stores are executed in-order, the Load can be dependent on the last one that was in-flight at the time the Load was decoded.

The paper suggests a detailed hardware description of the implementation for the *Store Set* technique. This description, as well as the detailed performance analysis of the methods, is beyond the scope of this section. The reader is advised to refer to [5] for more details.

8.4.3 Memory Bypassing

So far we focused on using address prediction and dependency prediction to speculatively schedule Loads that are predicted not to be dependent on any Store at the soonest possible time. This section describes techniques that take advantage of the same phenomenon, and use these techniques to accelerate the execution of Loads that are dependent on other Store operations. These techniques allow data to be transferred (bypassed) directly from the producer

of the memory operation (Store) to its consumer (Load), or between different Load operations that happen to occur simultaneously.

Most of the current out-of-order architectures postpone the execution of a Load that depends on a Store, until the former (Store) operation completes. As the latency to the main memory increases, the impact of such a delay increases and becomes important to the overall performance of the system. Passing values between Store and Load operations is more complicated than the “promotion” based algorithms, since in out-of-order machines, several instantiations of the same Load and Store instructions can be “in-flight” simultaneously in the system. Therefore:

1. It is not enough to predict the address of the Load and Store operations; we need to predict which Store will use the bypassing mechanism to pass information to which Load.
2. The Store operation can be executed non-speculatively; only Loads need to be speculated.
3. The Load value needs to be validated with respect to the real value in the main memory; if it fails, the system needs to recover from the miss-speculation.

Moshovos et al. [12] and Tyson and Austin [23] were the first to propose memory bypassing techniques to improve the load-to-use time of memory operations.

Tyson and Austin [23] present a technique called memory renaming that suggests adding two new data structures to the traditional architecture of the out-of-order computers: a *LoadStore* cache and a value cache. The Load/Store cache is used to identify Loads and Stores that may communicate with each other. The value file is used as an extension to the register file that can serve to transfer these values.

At decode time, each Load or Store looks up the LoadStore cache to get a number of a value file entry. If the entry does not exist, it creates a new entry in the cache and attaches to it a new entry from the value register file. If the entry exists, it checks to see (1) Whether it should use the value from the value file. It keeps a confidence level counter for this purpose. (2) Whether there is an in-flight Store that is about to update the value in the register file. In this case, it returns its entry number from the reorder buffer. Otherwise, it returns the last value that exists in the value file.

Because the proposed mechanism is based on speculative execution, the value being used needs to be validated against the correct value. Thus, all speculative and non-speculative Loads need to access the memory system. If the value is found to be correctly predicted, the processor can commit it; otherwise, the system needs to recover from the mispredicted speculation. The paper examines how memory renaming can be integrated into the pipeline of

the processor and reviews the benefit of the proposed technique under different recovery mechanisms.

Moshovos et al. [12] suggest a different technique, called memory cloaking, aimed at achieving similar goals. The authors observe that many of the memory dependent instructions are within the same inner loop; therefore, several versions of the same instruction may be in-flight at the same time in the MOB. The paper suggests adding three new data structures to the out-of-order architecture:

- **Dependence Detection Table (DDT):** Used to keep a list of all the Load and Store instructions in the system. The table keeps the PC address and memory address for each instruction.
- **Dependence Prediction and Naming Table (DPNT):** Used to define the pairs of Loads and Stores that may access the same address. For each such pair, two entries are allocated in DPNT table: one for the Load and one for the Store. The entries point to each other.
- **Synonym File (SF):** Used as a data transferring area between the Stores and the Loads.

The mode of operation in this technique is similar to regular instruction renaming. Each time a Store that has an entry in the DPNT structure is decoded, a new memory location in the SF area is defined. Each time a Load that has an entry in the DPNT structure is decoded, it will use the last known area of the corresponding Store.

Thus, if the same pair of instructions has several versions within the instruction queue, each of them uses a different communication buffer to transfer data between the Loads and the Stores.

Similar techniques for transferring information between memory operations can also be applied for *read-after-read* operations; Tyson and Austin [23] reported that when adding load data to the value file it improved the overall performance of the system, while Moshovos and Sohi report in [13] how the method they reported in [12] can be extended to support *read-after-read* memory relations as well.

8.4.4 Compiler Based Speculative Load Promotion

The Itanium[©] processor (IA64) presents a new approach for addressing data and address speculation. Itanium[©] is an in-order machine that embeds, as part of its ISA (instruction set architecture), special instructions to allow different types of speculation. Thus, the compiler can carry out address speculation and allow Load operations to be promoted over Store operations, even if it is not guaranteed that the Load and the Store do not access the same address. If at run-time, the system discovers that a memory order violation occurs, the compiler needs to define how the system should recover.

To implement this mechanism, Itanium[©] extends its in-order ISA with two instructions: ld.a (speculative Load) and chk.a (checkpoint for the speculation). This pair of instructions, together with the hardware that tracks memory collisions, enables compiler based speculation and safe execution—even if miss-speculation occurs.

The Itanium[©] uses a conflict resolution buffer structure called *Advanced Load Address Table* (ALAT) that detects memory order violations. This is accomplished by tracking the access to the memory location from the time the Load is issued to the time the data is checked. Every speculative Load from a memory location causes its address to be kept by the ALAT. Every Store to a memory location causes the Store’s address to be removed from the ALAT. When the CHK.S instruction is issued, it needs to check if the address still exists in the ALAT. If the address was found, the value brought by the Load is still valid (no Store occurred between the Load and the checkpoint). If the address was not found, the address may be valid or not. (Note, the address can be evicted from the ALAT structure for other reasons as well.) It is up to the software to decide what to do in the case of potential collision; however, it will try to re-load from the same address, using regular Load and not the speculative Load.

The recommended way to use this capability is to allow the compiler to promote Load speculatively over Store, if its analysis shows a high probability that the Load and Store will not collide. The compiler will put a speculative Load instruction in the new location of the Load; the compiler will put the checkpoint instruction at the location of the “non-speculative load” instruction.

For an example of the use of the new mechanism, we look at the following code taken from [25]. Suppose we try to optimize the following core:

```
foo (char *a, int *p) {
    *a = 1;
    b = *p + 5;
    ...
}
```

Let us assume the compiler cannot determine whether a and p point to the same location. Since a and p point to variables of different types, it is likely they do not point to the same location in memory (although the compiler cannot be sure of this). A smart compiler can optimize the code and produce the following new code:

```
ld4.a rt = [rp];
add rs = rt, 5;
...
st1 [ra] = 1;
chk.a rt, fixup_code2 reenter2:
...
fixup_code2: ld4 rt = [rp];;
```

```

add rs = rt, 5;
br reenter2;

```

The Load in the original code is replaced by a speculative Load and is promoted to be scheduled before the Store instruction.

When the ld.a executes, an entry is inserted into the ALAT. When the Store executes, the ALAT performs a comparison to check whether the address of the Store exists in the ALAT. When the chk.a is executed, if it can no longer find its address in the ALAT, it will force the code to jump to fixup_code2: at the point where the Load is reissued (in a non-speculative way), and the program can continue with its normal mode of operation.

Finding the right promotion point for the Loads is a difficult task for the compiler. If the speculative Load is not aggressive enough at the checkpoint, the instruction will have to wait for the Load. If the promotion point is too aggressive, the probability of the Load colliding with a Store, or not being on the right execution-path, increases and may slow down the overall execution.

8.5 Summary and Remarks

As the gap between the processor speed and the memory access time increases, the average service time of memory based operations become a dominant factor in the performance of modern processors. This chapter focused on the contribution of speculative address calculation techniques as well as speculative dependency prediction techniques for improving the *load-fetch-to-use* delay.

We provide a short taxonomy of the different methods for accelerating the address calculation and address prediction and we discuss the characteristics and the implementation details of different techniques that take advantage of these techniques.

We can assume that if the gap between the memory access time and the processor speed continues to grow, new techniques will have to be developed. Current techniques are limited in the amount of speculative work they can perform without committing the instructions, due to the requirement of retiring instructions in the order of the program. There are a number of directions toward the solution of that problem. One solution is to put the burden on the programmer and ask him or her to specify what the system should do in the case of long *fetch-load-to-use* delays. This can come in the form of multi-threaded code that gives the system an opportunity to continue its operation on a different stream of instructions, which are, hopefully, independent of the Load instruction that causes the long delay. A different approach to the problem tries to solve the problem within the hardware space, by allowing

instructions to speculatively commit and by creating a mechanism to recover if the speculation happens to be wrong.

We believe that a combination of all these techniques will be needed to ease the negative effect of the long latency to the memory. Such a method will require a major effort from both the hardware and the software communities, but such cooperation seems to be inevitable.

References

- [1] Todd M. Austin and Gurindar S. Sohi. Zero-cycle loads: microarchitecture support for reducing load latency. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 82–92. IEEE Computer Society Press, 1995.
- [2] Michael Bekerman, Adi Yoaz, Freddy Gabbay, Stephan Jourdan, Maxim Kalaev, and Ronny Ronen. Early load address resolution via register tracking. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 306–315. ACM Press, 2000.
- [3] Brad Calder and Glenn Reinman. A comparative survey of load speculation architectures. *Journal of Instruction-Level Parallelism*, 1(39):2–40, January 2000.
- [4] Robert Chappell, Jared Stark, Sangwook Kim, Steven Reinhardt, and Yale Patt. Simultaneous subordinate microthreading (ssmt). In *Proc. 26th International Symposium on Computer Architecture*, pages 186–195, May 1999.
- [5] George Z. Chrysos and Joel S. Emer. Memory dependence prediction using store sets. In *Proc. 25th International Symposium on Computer Architecture*, pages 142–153, Jun 1998.
- [6] Robert Cooksey, Stephan Jourdan, and Dirk Grunwald. A stateless, content-directed data prefetching mechanism. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 279–290, 2002.
- [7] Freddy Gabbay and Avi Mendelson. The effect of instruction fetch bandwidth on value prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 272–281. ACM Press, July 1998.

- [8] Freddy Gabbay and Avi Mendelson. Using value prediction to increase the power of speculative execution hardware. *ACM Trans. Comput. Syst.*, 16(3):234–270, 1998.
- [9] Jose González and Antonio González. Speculative execution via address prediction and data prefetching. In *Proc. 11th International Conference on Supercomputing*, pages 196–203, July 1997.
- [10] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2003.
- [11] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The microarchitecture of the Pentium processor. *Intel Technology Journal*, Feb 2001.
- [12] Andreas Moshovos, Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 181–193. ACM Press, 1997.
- [13] Andreas Moshovos and Gurindar S. Sohi. Read-after-read memory dependence prediction. In *Proc. 32nd International Symposium on Microarchitecture*, page 177–185, Nov 1999.
- [14] Alex Nicolau. *Parallelism, Memory-anti-aliasing and Correctness for Trace Scheduling Compilers*. Yale University, 1984.
- [15] Joan-Manuel Parcerisa and Antonio González. The latency hiding effectiveness of decoupled accessexecute processors. In *Proceedings of the Euromicro*, pages 293–300, Aug 1998.
- [16] Joan-Manuel Parcerisa and Antonio González. The synergy of multi-threading and access/execute decoupling. In *Proceedings of the Fifth International Symposium on High Performance Computer Architecture*, pages 59–63. IEEE Computer Society, 1999.
- [17] Glenn Reinman and Brad Calder. Predictive techniques for aggressive load speculation. In *Proc. 31st International Symposium on Microarchitecture*, page 127–137, Dec 1998.
- [18] Amir Roth, Ronny Ronen, and Avi Mendelson. Dynamic techniques for load and load-use scheduling. *Proceedings of the IEEE*, 89(11):1621–1637, 2001.
- [19] Amir Roth and Gurindar S. Sohi. Speculative data-driven multithreading. In *Proc. 7th International Conference on High Performance Computer Architecture*, pages 37–48, Jan 2001.
- [20] H. Sharangpani and K. Arora. Itanium processor microarchitecture. *IEEE, MICRO*, 20(5):24–43, September-Ocrober 2000.

- [21] James E. Smith. Decoupled access/execute computer architectures. *ACM Trans. Comput. Syst.*, 2(4):289–308, 1984.
- [22] Yan Solihin, Jaejin Lee, and Josep Torrellas. Using a user-level memory thread for correlation prefetching. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 171–182, 2002.
- [23] Gary S. Tyson and Todd M. Austin. Improving the accuracy and performance of memory communication through renaming. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 218–227. IEEE Computer Society, 1997.
- [24] Adi Yoaz, Mattan Erez, Ronny Ronen, and Stephan Jourdan. Speculation techniques for improving load related instruction scheduling. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 42–53. IEEE Computer Society, 1999.
- [25] Rumi Zahir, Jonathan Ross, Dale Morris, and Drew Hess. Os and compiler considerations in the design of the ia-64 architecture. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 212–221. ACM Press, 2000.
- [26] Jinsuo Zhang. The predictability of load address. *SIGARCH Comput. Archit. News*, 29(4):19–28, 2001.

Chapter 9

Data Speculation

Yiannakis Sazeides,¹ Pedro Marcuello,² James E. Smith,³ and Antonio González^{2,4}

¹ University of Cyprus; ² Intel-UPC Barcelona Research Center; ³ University of Wisconsin-Madison; ⁴ Universitat Politècnica de Catalunya

9.1	Introduction	215
9.2	Data Value Speculation	216
9.3	Data Dependence Speculation	224
9.4	Verification and Recovery	230
9.5	Related Work and Applications	234
9.6	Trends, Challenges, and the Future	235
9.7	Acknowledgments	237
	References	237

9.1 Introduction

The pursuit of higher levels of instruction level parallelism is fundamentally limited by architectural dependences involving both control and data. To alleviate serialization due to dependences, prediction and speculation mechanisms can be employed. These two mechanisms are central to the various techniques discussed in this book. In all cases, the key idea is to predict certain unknown information required by the processor (e.g., the outcome of branch or arithmetic instructions, or the existence of a memory dependence between a store and a load), and allow the processor to use the predicted information as if it was correct. In this way, instructions that directly or indirectly consume this information can execute earlier, and execution time may be reduced. Speculative execution requires a recovery scheme to ensure the correctness in case of misspeculation.

Branch prediction and speculation is an old technique that has been used in most processors for decades and it is present in practically all high-performance processors nowadays. On the other hand, techniques to predict and speculate on values and dependences are found in the literature after the middle of the nineties and their impact is still low in current processors. Some processors already implement data dependence speculation; however, as far as we know

data value speculation has not yet been implemented in current commercial designs.

One of the potential reasons why data value speculation had been practically ignored until recently is the fact that it seems a much harder problem than control speculation. Branches have Boolean outcomes, and therefore even naive schemes are likely to result in more than 50% of correct speculations. On the other hand, values may have up to a 64-bit range, which practically eliminates the possibility of correct predictions just by chance. Data dependences are again Boolean values (i.e., a dependence between two instruction either exists or not), and again may be more likely to be predictable.

This chapter is concerned with two different types of Data Speculation: Data Value and Data Dependence Speculation. It expands on the following essential mechanisms for Data Speculation: (a) techniques for predicting data values (Section 9.2) and for predicting data dependences (Section 9.3), (b) methods for verifying the correctness of predicted values/dependences and values produced by speculatively executed instructions (Section 9.4), and (c) schemes for recovering in case of a misprediction (Section 9.4). The chapter also considers implementation issues and related studies of Data Speculation (Sections 9.2-9.4) and provides discussion of research challenges and future applications (Section 9.6).

9.2 Data Value Speculation

Like other speculation techniques described in this book, data value speculation consists of interacting mechanisms for prediction, verification, and recovery. Prediction provides values that side-step value-dependences, verification determines the correctness of predicted values, and recovery is needed to remove the effects of value mispredictions, including the results of speculative instructions and speculated instructions themselves, if they have not yet completed execution.

The interaction of the various mechanisms is illustrated by the four instruction sequence in Figure 9.1.a. The example includes a serialized data dependence chain of three instructions that load a value from memory, mask a part of the value, and shifts the masked value by a fixed amount.

Without value speculation, the dependent instruction sequence executes in series, and may require several cycles to execute, with two (or more) cycles being spent by the load instruction alone (Figure 9.1.b). However, if the value to be loaded from memory is predicted, then the load-use dependence is broken and the execution of the dependent instructions can proceed speculatively, in parallel with the load (Figure 9.1.c). When the actual value loaded from

memory is available, the prediction must be verified. The actual value is compared with the predicted value (Figure 9.1.d). If the prediction is correct, execution proceeds normally. Otherwise, if the prediction is incorrect, the speculatively executed instructions need to be nullified (or *squashed*) and must be re-executed starting with the correct load value (Figure 9.1.e).

Data value speculation based on correct value predictions can increase instruction level parallelism, and therefore improve performance. However, mispredictions can degrade performance because misspecified instructions may be executed later than if no value speculation were done in the first place. This can occur because of overhead associated with misspeculation recovery and/or because of conservative recovery methods. For example, recovery from a value misprediction can be implemented by a complete squash of all instructions that follow the mispredicted instruction (Figure 9.1.e). In the example, the *add* instruction has no dependence on the predicted value, but it may be squashed along with the dependent instructions when the loaded value is mispredicted. To avoid this particular source of performance loss, a more selective recovery scheme may be used; selective methods are discussed later in Section 9.4.

Although the above ideas resemble mechanisms used by other forms of speculation, some differences emerge when the general mechanisms are applied to value speculation:

- Predicted values can come from a very large set (e.g., all 64-bit integers, all floating point numbers) and therefore multibit predictors are required (Section 9.2.1). This is in contrast to predictions of conditional branch directions or targets of conditional branches where the predicted values come from a much smaller set (e.g., *taken* or *not taken*).
- Because value prediction accuracy may be low in some situations, it is important to have the option of disabling speculation for certain predictions where the confidence of a correct prediction is low. This leads to hardware confidence estimators that supplement value predictions (Section 9.2.1 and 9.4).
- The instructions that may be affected by a value misprediction are only those that depend on the mispredicted value, either directly or transitively through other instructions. This may be a relatively small subset of all the instructions that follow a misprediction. This is in contrast to branch prediction, where many (sometimes all) of the instructions that follow a mispredicted branch depend on the branch outcome. Consequently, selective instruction squashing (Section 9.4) is often worthwhile with value prediction, but typically has much less payoff with branch prediction.

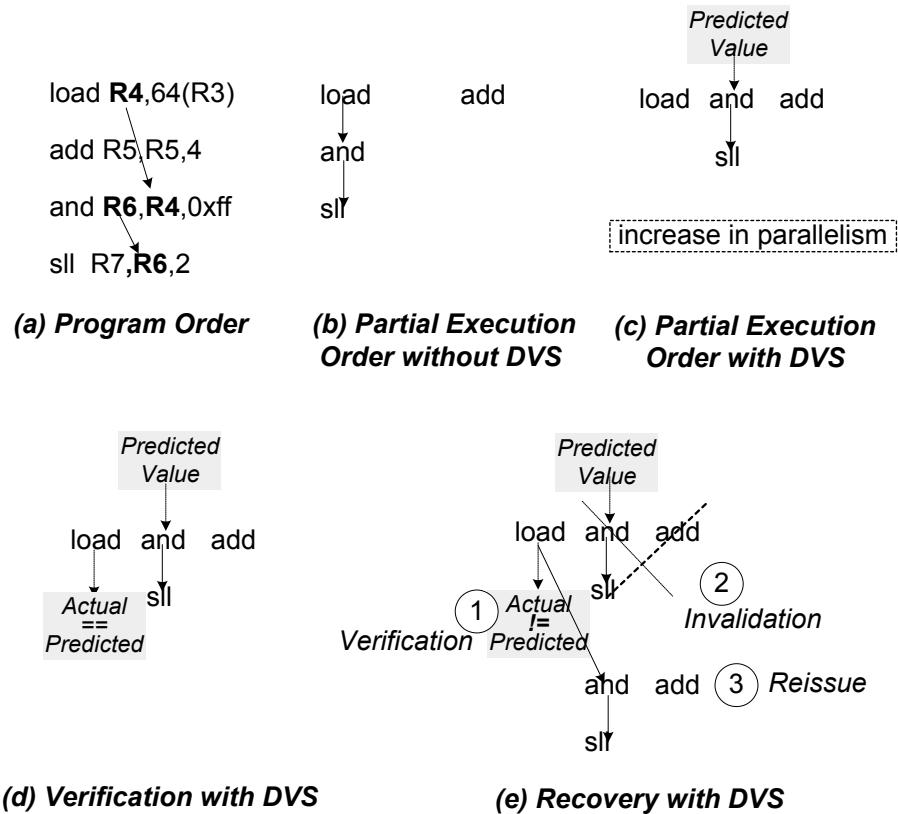


FIGURE 9.1: Implications of Data Value Speculation (DVS).

- Not all values provide the same performance benefit if they are correctly predicted. Values that reduce the program critical path are more likely to provide a significant performance benefit (Section 9.5).
- Additional microarchitectural paths and states may need to be introduced. For example, a value may be in several states: pending, speculative, and not-speculative (Section 9.4).

The rest of this section and Section 9.4 expand on the above distinct requirements of data value speculation.

9.2.1 Basic Value Predictors

Data value speculation is based on a mechanism for predicting data values. This subsection discusses the logical organization and some implementation issues of value predictors.

Most value predictors proposed to date are based on lookup table(s) containing values or value relationships. When a prediction is needed, the table(s) are accessed to obtain a prediction and later, when the correct value is available, the predictor may be updated to reflect the correct value. A value prediction can be associated with either the instruction that produces or consumes the value. Henceforth, we will associate a value prediction with the producing instruction. We refer to [38, 37] for a discussion on the issue of predicting input versus output values.

Figure 9.2 shows three different value predictors. The **last-value** predictor [39] consists of a single table containing data values. The table is indexed using a hashed version of the program counter (PC) of the instruction whose output value is to be predicted. Each table entry contains the last value produced by an instruction that hashes to that entry. The value read from the table is the prediction. Hence, the prediction is simply that the value will be the same as the last time the instruction was executed. Surprisingly often, an instruction does, indeed, produce the same value as on its most recent execution.

The **stride-value** predictor [18] also consists of a single prediction table that is indexed by the hashed PC of the instruction whose value is to be predicted. Each table entry contains two fields, the difference (stride) between the two most recent values produced by the instruction and the last value. The predicted value is found by adding the stride to the last data value. As its name suggests, a stride predictor is most effective at predicting sequences of values that differ by a constant.

The **context-based** predictor [55, 54] uses two tables. The history table is indexed with an instruction's hashed PC, and it contains a sequence of the n most recent values produced by the given instruction. This sequence is referred to as a "context". Then, a hashed version of the context is used to access the prediction table which contains the predicted value. This scheme essentially records the value that follows a particular sequence of n prior values, and predicts that same value when the context is repeated. The number of values in the context, n , is referred to as the predictor order. A context-predictor works well when a context is always followed by the same value.

What mainly distinguishes the various proposed value predictors are: (a) the information stored in each table entry, (b) the algorithm used to update the predictor, and (c) the index used to access the table(s). These three issues are discussed next.

The information stored in a prediction table entry determines how a prediction is obtained and the types of sequences an entry can predict. For the last-value and context-based predictors, the prediction is the content of the selected prediction table entry, whereas for a stride-value predictor, the table maintains a relationship between consecutive values and performs a simple computation to arrive at a predicted value. The last value and context predictors can only predict values they have seen previously, while a stride predictor can compute and predict values that have not yet been observed.

The most basic update algorithm is to always update each field with the most recent value information. However, this can sometimes result in reduced performance when a single anomalous value appears in an otherwise highly predictable stream. For example, a variable may sequence through a series of values with a constant stride, and then reset and start again at the beginning of the sequence. At the point of the reset there is a single anomalous stride value. To avoid multiple mispredictions due to single anomalous values, it is often better to incorporate some form of hysteresis into the update process [14, 39, 55, 54, 24, 4]. For example, an update algorithm may wait for two consecutive strides that are the same before changing the stride prediction.

The index value that accesses the table, e.g., a hash of the PC, tends to “localize” a stream of values so that they share some common property that makes them more predictable. Using the PC isolates those values that are produced by the same instruction. An implementation might also include (via the hash function) other instruction information such as certain register values or the sequence of recent branch outcomes leading up to the current instruction.

Using the basic predictors just introduced, a number of alternative data value predictors can be constructed. Some of these are surveyed in the following subsection.

9.2.2 Value Predictor Alternatives

In a given program, there is often a variety of value behavior. One single type of predictor may work for predicting some value sequences, but not others. Consequently, it has been proposed that hybrid value predictors, using a combination of the basic predictors, should be used [64, 50, 48]. In a hybrid predictor, a set of basic component predictors are supplemented with a selector. All the components predict a value, and the selector chooses the one deemed most likely to be correct. For example, a selector may keep a history of which component has been more accurate at predicting a given instruction’s output in the past. It will then choose the prediction with the highest confidence level based on the past history [50, 48]. Confidence estimation methods are described in the next subsection.

Wang and Franklin [64] propose a hybrid value predictor configuration that uses a stride predictor to capture constant and stride sequences and a two-level predictor for sequences that exhibit patterns that are not stride predictable. The two-level predictor maintains n unique values per instruction. The selection among the n values is based on a unique index associated with each value (if 4 values are maintained a 2 bit index is used). A history of p such indices, maintained per PC, provides a context that is used for accessing a selector table for choosing one of the n values. Another value prediction approach, similar to the two-level component in [64], selects among the n most recent values produced by a given instruction using the value prediction outcome history and a dynamic confidence estimator [7].

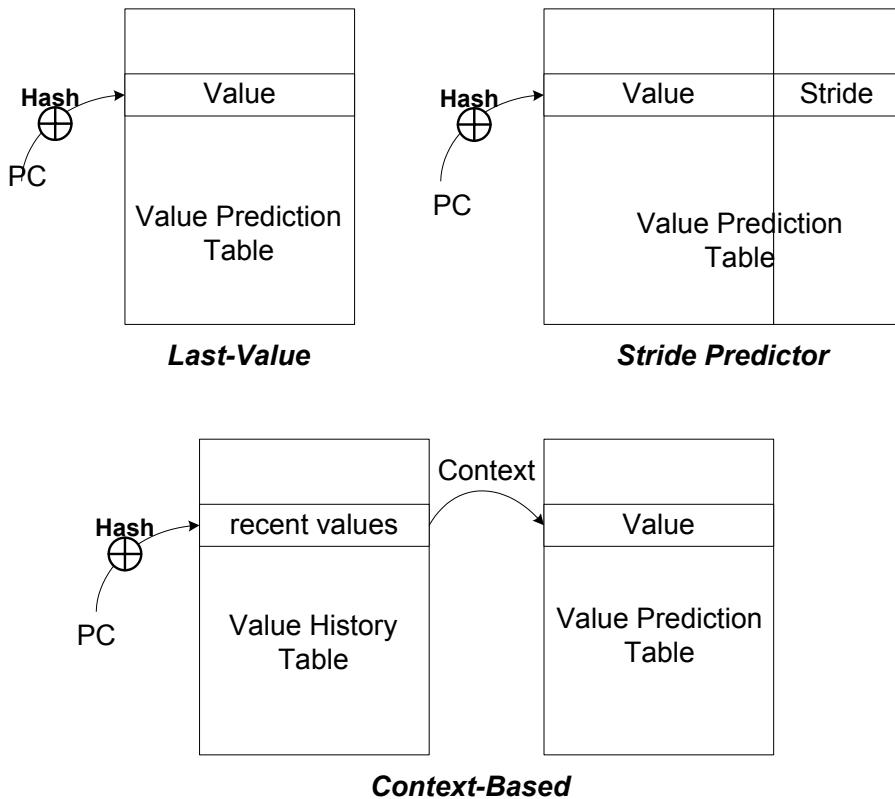


FIGURE 9.2: Various value predictors.

Another value prediction approach is the differential-FCM predictor [21, 51]. This predictor resembles the context-based predictor, but instead of learning the value that follows a context of values it learns the value-delta that follows a context of value-deltas. To get a prediction for an instruction’s output, the selected delta is added to the last value produced by the same instruction. This, in theory, can provide a more cost effective and accurate value predictor than basic context prediction because it can not only learn any repeating sequence, but it can also apply the same delta sequence to a different base address value and predict values that have not yet been observed.

A similar, recently proposed differential value predictor predicts a delta from the recent history of global deltas [67], rather than using the local (single PC) delta history. An alternative global value history prediction approach was proposed in [45]. A delta predictor was also proposed for a distributed microarchitecture that predicts the difference between input and output values of an instruction sequence using the control flow inside the sequence [42].

A novel value prediction scheme is the storageless value predictor [62]. This scheme does not predict a value per se; rather it predicts a register that contains the value that is to be produced by the predicted instruction. This scheme attempts to exploit the presence of the same value in different physical registers.

Recently, there have been proposals that attempt to predict values based on program structure [61, 52]. The program structure may represent more concisely and accurately the behavior of an instruction [56].

Finally, there have been proposals for value predictors that exploit compiler analysis to either reduce cost [19] or increase accuracy [62, 32].

The accuracy achieved by value predictors depends on many parameters, but the general trend is that the more sophisticated variations of context-based predictors are typically more expensive and have higher accuracy than simpler predictors such as last-value and stride. A version of a the *gDiff* predictor proposed in [67] achieves an accuracy of 73% when predicting all value producing instructions for SPECINT2000 benchmarks, whereas a stride predictor could provide accuracy close to 55%.

9.2.3 Confidence Estimation

To maximize gains when using prediction and speculative execution, it is imperative to have high prediction accuracy (and infrequent misspeculation). However, predictions of certain values or branch outcomes may be more accurate than others. Because a misspeculation can lead to performance loss, there are many situations where, for hard-to-predict values or branch outcomes, it is better not to speculate at all rather than risk the penalty of a misspeculation.

Misspeculation can be controlled with the use of confidence mechanisms [29] that prevent speculative execution when predictions have relatively low likelihood of being correct. A confidence table is used to assign a confidence

to a value prediction. Each entry in the table contains a summary of accuracy for recent predictions. One commonly used method for summarizing accuracy is to use saturating counters that increment up to some maximum value for each correct prediction and decrement down to some minimum value (usually zero) for each misprediction. The decrement value is often larger than the increment value, or the counter resets completely on a misprediction [29]. A prediction made at a high counter value is assigned a high confidence of being correct, and, conversely, a prediction made when the counter value is low is assigned a low confidence.

Confidence estimation is generally considered to be essential for value speculation because value prediction accuracy is not extremely high in the first place (when compared with branch prediction), and the penalty for a value-misspeculation may be high because a misspeculation (a) may cause the cancellation of correctly executed instructions, (b) may interfere with other forms of speculation and delay the execution of instructions, and (c) may delay the issue of a misspecified instruction even when the correct data is available. The effects of misspeculation are discussed in more detail in Section 9.4.

The use of saturating counters to control value speculation was proposed in [39]. Calder et al. [11] explored the use of confidence levels (low, medium, and high) for resetting counters and also proposed the propagation of confidence levels to dependent instructions. They showed that speculating on low confidence predictions lying on the critical program path can improve performance despite the low confidence. Bekerman et al.[4] suggest associating control flow history with confidence estimations. If the history leading up to a misprediction is repeated, then the subsequent prediction is assigned low confidence. The technique in [4] was proposed for addresses prediction but is also applicable to general value prediction.

An alternative to dynamically updated confidence tables is to assign static confidence levels based on program profiles [19] or based on patterns of correct/incorrect value predictions [9]. The former scheme requires instruction set modifications whereas the latter requires a hardware structure that maintains the pattern behavior per predicted instruction.

Experimental data show that very accurate confidence estimation for value prediction is possible. This comes at the cost of reducing the fraction of instructions that get predicted and therefore with a lower performance potential. Accuracy in excess of 90% is reported in [67] for the gDiff predictor when only predicting 64% of the value producing instructions. In [9] 99% accuracy is reported for a last-n value predictor but less than 40% of the instructions are getting predicted.

9.2.4 Implementation Issues

When implementing a value predictor, several issues need to be addressed, including: hardware cost, organization (table associativity, banking, etc.), hash functions, access latency, speculative-updates, checkpointing and recov-

ery of predictor state, bandwidth, energy, and other physical implementation issues. For studies of the effects of organization and hardware cost, we refer the reader to several studies listed in the references. These studies explore the trade-offs between accuracy, coverage, organization, and hardware cost for several predictors and confidence estimators. Below we address some of the other implementation issues.

An investigation of table lookup hash functions and their effects on the accuracy of context-based predictors and the performance of out-of-order processors is reported in [54, 51]. A function that maintains context values in hashed form, as opposed to keeping each value separately, is shown to provide good accuracy.

In many simulation-based value prediction studies, predictor tables are updated with the correct value immediately following a prediction. However, in a real implementation, it may take many cycles following the prediction before the correct value is known. This presents a design choice regarding the updating of value predictor tables: (a) wait until the correct value is known, or (b) update the table immediately with the predicted value, and correct it later if necessary. Waiting for the correct value is probably the simplest method, but unfortunately it is also the worst-performing method for most value predictors. With speculative updates, there must be a mechanism for checkpointing the old predictor state so that when a misprediction is detected the predictor state can be recovered. For a discussion on speculative updates and checkpoint the reader should refer to [54, 51, 33, 34, 67].

For superscalar processors with wider instruction issue widths, an implementation of value speculation may require multiple predictions during the same cycle, possibly for multiple instances of the same instruction. In [20] it is shown that the importance of value prediction increases with wider issue processors, and a scheme for high bandwidth value prediction is proposed. A scheme that provides high bandwidth prediction by moving the predictor to the pipeline back-end instead of the front-end (as is the common case) is proposed in [33]. Value predictor latency issues are also addressed in [33, 34].

Issues pertaining to bandwidth and energy efficiency for value prediction are explored in [5, 6, 40] where it is shown that value predictor energy consumption can be reduced drastically without significant degradations in performance.

9.3 Data Dependence Speculation

The presence or absence of data dependences are crucial for determining instruction level parallelism in a modern high performance processor. If two instructions are known to be independent, then they can be executed in parallel. On the other hand, if two instruction are dependent, then (in the absence of

value prediction) they must be executed serially. Therefore, an important aspect of extracting instruction level parallelism is detecting data dependences.

Data dependences involving register values can be determined easily in the front-end of a pipeline by simply comparing register specifiers. Determining dependences involving memory data is much more difficult, however, because it is necessary to first know the memory addresses where data values are held. Hence, potential parallelism between memory access instructions (loads and stores) can only be detected when it is known that their memory addresses do not overlap. Before memory instructions are executed (and their memory addresses are computed) the actual address values are often ambiguous. The process of computing and comparing memory addresses to detect independence is therefore known as “address disambiguation”.

To prevent the execution of a load instruction before any previous store to the same memory location, processors traditionally have employed **total-disambiguation**: a load instruction executes only when all previous store addresses have been computed, and a store instruction can execute only when all previous load and store addresses have been computed.

Figure 9.3 shows how total-disambiguation works. Consider the instruction sequence in Figure 9.3.a. The store instruction is at the end of the dependence chain of three instructions and the load instruction may depend on the store instruction (they may access the same memory address). With a total disambiguation approach the execution of the load instruction must be delayed until the previous store is known. This results in the serialized execution shown in Figure 9.3.b.

Obviously, this approach guarantees the correct execution of the program but it is often overly conservative. Independent memory instructions also have to wait for disambiguation with all previous memory store operations. That is, the execution of some memory instructions may be unnecessarily delayed due to false dependences.

Returning to the example of Figure 9.3, if the load instruction does not depend on the store, it can be executed as soon as all its input operands are available possibly in parallel or even before the execution of the store instruction. Approaches that allow the execution of memory instructions out-of-order are referred to as **partial-disambiguation**.

To implement partial disambiguation schemes, two main mechanisms are necessary: (a) the prediction of whether a memory instruction depends on a previous one, and (b) a scheme for detecting and recovering from misspeculation. These mechanisms are usually referred to as **data dependence speculation**. Notice that data dependence prediction differs from data value prediction in that it does not predict the value that flows from the producer instruction to the consumer one; it only predicts whether or not a given instruction depends on a previous one.

Figure 9.3.c shows how the example instruction sequence will be executed when there is a data dependence speculation mechanism. If the load is predicted not to be dependent on the previous store, then it can be issued in parallel

with the *mul* instruction. Obviously, as with all speculation mechanisms, the prediction has to be verified. Thus, when the address accessed by the store is computed, it is compared with the addresses accessed by speculative loads to detect misspeculation (Figure 9.3.d).

With data dependence speculation, there are two types of mispredictions: a) the predictor may speculate that an instruction depends on a previous instruction and in reality is independent, and b) it may speculate that an instruction is independent but it turns out to be dependent. In the former case, no recovery is needed; the results will be correct, but instruction execution is unnecessarily delayed. In the latter case, there is a dependence violation, and results may be incorrect unless a recovery action is undertaken. In this case, as shown in Figure 9.3.e, the offending dependent instruction must be squashed and re-executed. Some performance degradation may occur, depending on the added delay of the recovery scheme.

An implementation of partial-disambiguation requires hardware support for detecting misspeculation. This hardware buffers all the addresses accessed by speculatively executed load and store instructions. Then, when the address of a store instruction is computed, it is compared with the buffered speculative addresses. If there is a match, then recovery is invoked. Example implementations of these disambiguation mechanisms are the Address Reorder Buffer (ARB) [28] in the HP-PA8000 processors, the Advanced Load Address Table (ALAT) [36] in the Itanium and Itanium-2 processors, and the Address Resolution Buffer (ARB) [16] for the proposed Multiscalar processors [59].

In the next subsection, data dependence predictors are presented. The basic mechanisms for misspeculation recovery are similar to those for data value speculation and are discussed in Section 9.4.

9.3.1 Data Dependence Predictors

Data dependence prediction can be implemented either in hardware or in software. This section describes hardware data dependence predictors. Software approaches are briefly discussed in Section 9.5.2.

Data dependence speculation requires a predictor to decide whether to allow a ready-to-issue memory instruction to be speculatively executed prior to disambiguation with previous memory instructions. One of the simplest data dependence predictors always predicts one-way. A predictor that always predicts that a load instruction depends on any previous unresolved store implements a total disambiguation scheme. Alternatively, a predictor can always predict that a load instruction never depends on any previous unresolved stores. This naive scheme is currently implemented with the Address Reorder Buffer in the HP PA-8000 family [28], and was proposed for the Multiscalar processors [59].

However, more accurate predictions can be built based on information from previous executions of the same memory instructions. Two types of information can be used to implement more accurate data dependence predictors:

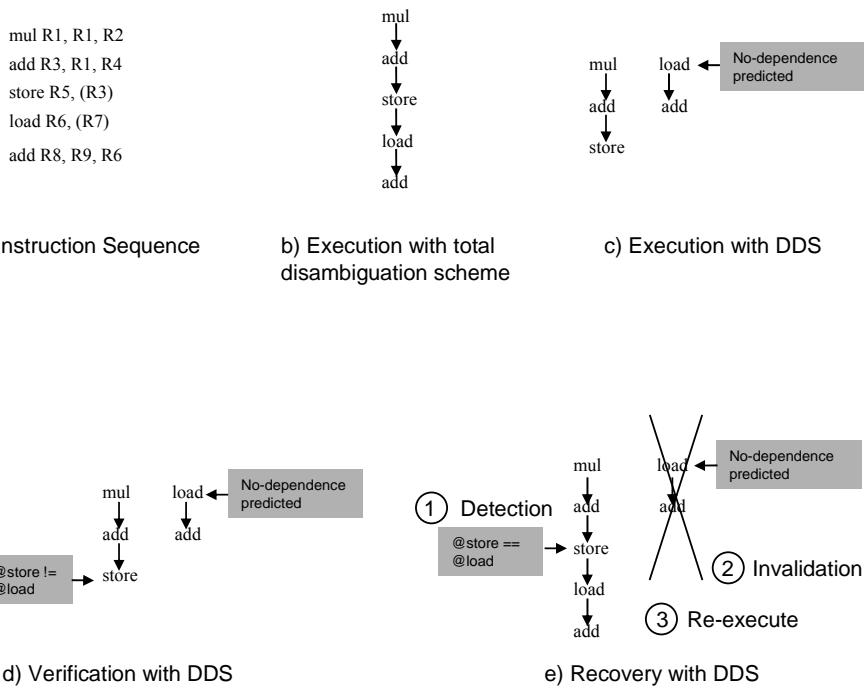


FIGURE 9.3: Data dependence speculation.

(a) predicted addresses for memory operations, and (b) the group of memory instructions a memory instruction depends on.

9.3.1.1 Data Dependence Prediction through Address Prediction

Address prediction has been widely studied for cache prefetching (cross reference Chapter 8). In prefetching studies, it has been shown that memory addresses are quite predictable. For instance, a simple stride sequence is followed by 90% of all the references in the SpecFp95 benchmark suite, which are primarily numerical applications. Stride sequences are often the result of traversing regular data structures such as arrays. For the integer-oriented SpecInt95 benchmark suite, more than 60% of memory references follow a stride pattern. These address prediction accuracies can be higher when more complex address predictors are considered [23].

Thus, if addresses are highly predictable, processors can predict memory data dependences by comparing predicted addresses. Examples of data dependence predictors that rely on stride address prediction are the Memory Address Prediction (MAP) [22] scheme and its follow-up work, the APDP (Address Prediction and Data Prefetching) [23]. Both schemes have similar functionality and are implemented on top of an ARB.

When a memory access instruction is decoded, the address to be accessed is predicted and can be used for speculative disambiguation with following memory instructions. Based on the speculative disambiguation, a memory access may also be performed speculatively. To check if a speculation is correct, the correct address, when computed, is compared with the predicted one. If a misspeculation occurs, then recovery is initiated. The APDP mechanism introduces additional hardware for prefetching values.

9.3.1.2 Data Dependence Prediction through Dependence History

It is also possible to predict data dependences based on dependences found in previously executed instructions. For example, if two memory instructions had a data dependence at some point in the recent past, then they are likely to have a similar dependence during future executions.

One such approach is described in [43] and in follow-up work [44]. The basic mechanism is based on the identification of pairs of static load-stores that are frequently dependent. The idea is then to establish a wait-signal protocol in such a way that the load can not be issued before the predicted dependent store. To capture these pairs, the loads belonging to the pair are detected by means of the ARB mechanism. Then, the static pairs are stored in a table, and when a later load is decoded, it accesses the table containing dependent pairs. If the load is in the table, it must wait for the completion of the corresponding store.

The follow-up work, referred to as Speculative Memory Cloaking [44], is similar to the above, but it addresses the problem of loads that depend on more than one store. Speculative Memory Cloaking also allows Speculative

Memory Bypassing, which allows a value to flow from the producer instruction of the data to be stored to the consumer instruction of a load.

Store Sets [13] is a similar dependence prediction approach with [43]; however the store set mechanism is able to deal with both the case where a load that depends on multiple stores and where multiple loads that depend on the same store. In this mechanism, each load instruction is associated with a store set, which consists of all those store instructions that the load has ever depended on. Initially, all load instructions have an empty store set, and when a dependence is detected, the PC of the store is added to the store set of the load. Thus, when a load is fetched, the store set is accessed to determine if there is any outstanding store (a preceding, not-executed store) that belongs to its store set. If so, the load instruction must wait for the execution of any prior stores belonging to its store set.

The store set approach can be implemented by means of two tables as shown in Figure 9.4. The Store Set Identifier Table (SSIT) is indexed by the Program Counter of memory access instructions and contains the identifier of the store set which is used to access a second table, the Last Fetched Store Table (LFST). This table contains the inum, a unique identifier for each currently executed instruction.

This mechanism works as follows. When a memory dependence is detected, then a store set for the load instruction is created. To do this, the corresponding entries of the SSIT for the load and the store are initialized with a store set identifier (SSID). When the same store is fetched again, it accesses the SSIT and if a valid SSID is found, then it accesses the LFST and deposits its unique identifier. Then, when the load is again fetched, it also accesses to the SSIT and looks up the LFST. If a valid inum is found, the load has to wait for the completion of the corresponding store to continue its execution.

If another dependence occurs for the same load, a new entry for the store is allocated in the SSIT with the same SSID. If both stores are outstanding when the load is fetched, only the second one will be found with a valid inum in the LFST; so in addition to placing its unique identifier in the table, the second store has to be synchronized with the previous store (i.e., they must be performed in order). The only constraint on this implementation of store sets is that a store cannot belong to more than one store set at any time.

The accuracy of the store-sets memory dependence predictor [13] was shown: (a) to be very close to 100% for most SPEC95 benchmarks, and (b) to essentially achieve the performance of an oracle dependence predictor for the processor and dependence predictor configuration reported in [13]. The dependence predictor proposed in [43] was also shown to be very accurate.

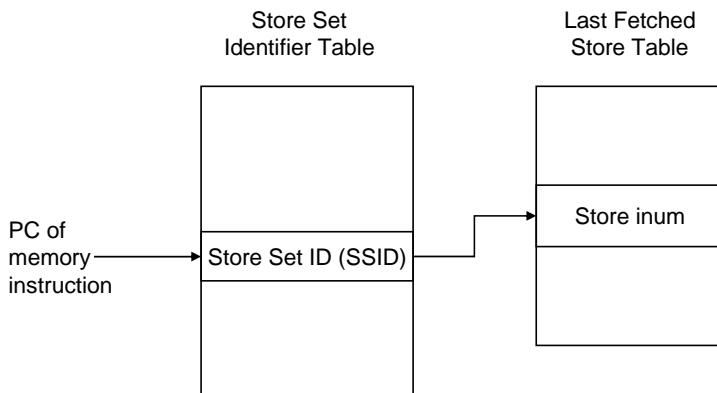


FIGURE 9.4: Store sets data dependence predictor.

9.4 Verification and Recovery

As noted earlier, the implementation of data value or data dependence speculation must include mechanisms for verification of speculatively executed instructions and recovery in case of misspeculation. However, verification and recovery may be implemented differently depending on whether the hardware or software performs the speculation functions. This section expands on the issues of verification and recovery in the context of a dynamically scheduled processor using a hardware based approach. The software based approach is discussed in Section 9.5.

The early contributions related to microarchitectural implementations of verification and recovery for data value speculation were made by Lipasti et al. [39, 38, 37]. Detailed discussion of the implications of data value speculation on the microarchitecture of a dynamically scheduled processor can be found in [37, 50, 51, 53]. The discussion in this section is based on [53]. First there is a discussion of verification and recovery, followed by a brief overview of the possible microarchitectural implications of data value speculation.

Note that verification and recovery mechanisms required by data dependence speculation [63] are similar to the ones described below for data value speculation and are not described separately.

9.4.1 Verification

When an instruction's output value is predicted correctly, verification is responsible for informing the instruction's direct and indirect successors that their input operand(s) are valid. Verification directly influences the release of *issue* resources (such as reservation stations) and *retirement* resources (such as reorder buffer entries). Considering that an instruction may need to hold a resource until all its input operands become *valid*, then fast verification can be critical for performance when resources are limited.

Fast verification latency is also important because it is critical that conditional branch predictions and memory accesses based on predicted values be resolved as quickly as possible. A problem with resolving branches using predicted/speculative values is that value prediction may be less accurate than branch prediction and hence may lead to additional branch mispredictions. Similarly, it is preferable that memory instructions access memory with valid addresses to avoid additional memory dependence violations because memory dependence predictors can be more accurate than value predictors [13, 43].

There are at least four approaches for performing verification:

Hierarchical Verification

With hierarchical verification a correctly predicted instruction can validate only its direct successors. The validated successors will then validate their direct successors. This process will be repeated until all successors get validated as being correct. Hierarchical verification can be implemented using the existing tag broadcasting mechanism implemented in many out-of-order superscalar processors for notifying dependent instructions when an input operand is ready. This verification approach can provide a performance improvement provided there are a number of separate data dependence chains in the instruction window. Otherwise, the increased execution parallelism from value speculation will be offset by the serialization in verification.

Retirement Based Verification

The retirement mechanism used in many dynamically scheduled superscalar processors can also be used to *verify* in multiple speculative instructions in parallel. That is, verification can be combined with retirement. However, this approach may have two pitfalls:

- (a) in each cycle only the w oldest instructions in the instruction window can be validated, where w is the retirement bandwidth of the processor. This may be undesirable if a younger instruction is otherwise valid but forced to hold a resource needlessly.
- (b) the additional functionality may stress the critical path for releasing retirement resources. Therefore, retirement resources may be freed only after additional delay.

Hybrid Retirement Based and Hierarchical Verification

This approach attempts to build on the strengths of the two previous approaches. *Retirement-based* verification is used for releasing resources with higher bandwidth whereas *hierarchical* verification is intended for faster de-

tection of mispredictions.

Flattened–Hierarchical Verification

In this scheme all direct and indirect successors of a correctly predicted instruction are validated in parallel. The *flattened–hierarchical* verification represents the verification method with the highest performance potential; however it is also likely to be the method with the highest implementation cost. This can be implemented using the data dependence tracking mechanism proposed in [12].

It is noteworthy that a microarchitecture may require a different verification approach depending on: (a) whether the issue and retirement resources are unified, and (b) whether branch and memory instructions are resolved with speculative/predicted values. Future research should investigate these important topics.

Parallel verification is assumed in a number of research papers. Hierarchical verification is explained in [50]. The first work to explore the effects of value speculation on branches was presented by Sodani and Sohi [58]. The authors compare the performance when branches are resolved with speculative/predicted values and when resolved only with valid values. That work also considers the effects of non-zero verification latencies.

In other related research [38, 49] branches are resolved out-of-order only when their operands are known to be non-speculative. More recent work explored issues related to the speculative resolution of branch instructions [3]. The study by [48] considered the combination of address, value, and dependence prediction for resolving speculative loads.

9.4.2 Recovery

When a value or dependence is predicted incorrectly, a recovery action must be taken to invalidate and re-issue misspecified instructions. Invalidation is responsible for informing the direct or indirect successors of a mispredicted instruction that they have received incorrect operands. This is essential to nullify the effects caused by a misprediction. Note that the invalidation mechanism has very similar functionality to the verification mechanism (Section 9.4.1), and in some cases one mechanism may be sufficient to implement both. Instruction re-issue is required so that misspecified instructions eventually execute with correct operands.

9.4.2.1 Invalidation

There are two basic models for invalidation: **complete invalidation** and **selective invalidation**. Complete invalidation treats a value misprediction similar to a branch misprediction and squashes all instructions following the one with the misprediction, regardless of whether they depend on it or not. This approach is simple, but may invalidate a number of instructions that were executed correctly.

Some researchers [39, 11, 62, 9] have compared the performance of selective and complete invalidation and observe small, but still positive performance gains when complete invalidation is used. Complete invalidation is relatively simple to implement and may be beneficial in terms of performance if value mispredictions are relatively rare. As noted in Section 9.2.3, value prediction can be enhanced by using confidence estimation [4, 11, 29, 7, 9].

Most papers adopt a hierarchical selective form of invalidation, where only instructions with incorrect results are squashed, and an invalid instruction can only invalidate its direct successors. The invalidated successors then invalidate their successors, etc. The implementation may be built on top of the existing tag broadcasting mechanism in superscalar processors [39, 49, 50, 63], or by using a dependence tracking mechanism as proposed in [12]. Thus far, the proposed designs for selective invalidation require that instructions with speculative/predicted operands continue to hold their instruction issue resources (such as reservation stations) after they issue, and until they have been verified.

The invalidation mechanism with the highest performance potential invalidates all direct and indirect successors of a mispredicted value in parallel. Selective parallel invalidation is effectively a flattened-hierarchical invalidation scheme. This is analogous to **flattened-hierarchical** verification. The invalidation latency can be crucial to performance because it may determine how quickly a misspeculated instruction reissues.

9.4.2.2 Instruction Re-Issue

An instruction that gets invalidated due to a misprediction must be reissued. Clearly, this means that the same instruction may issue multiple times. For this to occur, an instruction in an issue buffer or reservation station has an “enabled” flag which is set when all the instruction’s input values are available. When enabled, an instruction is said to be “awakened” and is ready to issue, subject to availability of hardware resources.

With selective invalidation, an instruction waits in its issue buffer or reservation at least until all its input operands have been validated as being correct. If it is invalidated, then the enable flag is cleared, and it can issue again, as soon as all of its input operands (including the previously invalid ones) again become available.

Sodani and Sohi [58] performed a comparison of two selective invalidation schemes: one can re-enable an instruction each time a new value becomes available for one of its inputs [49], and the other limits an instruction to at most two executions [39]. The two approaches have a subtle but important difference, the former effectively ignores speculative status of the operands and hence may reissue a misspecified instruction sooner. This also implies that some instructions may re-issue needlessly when they are not misspecified.

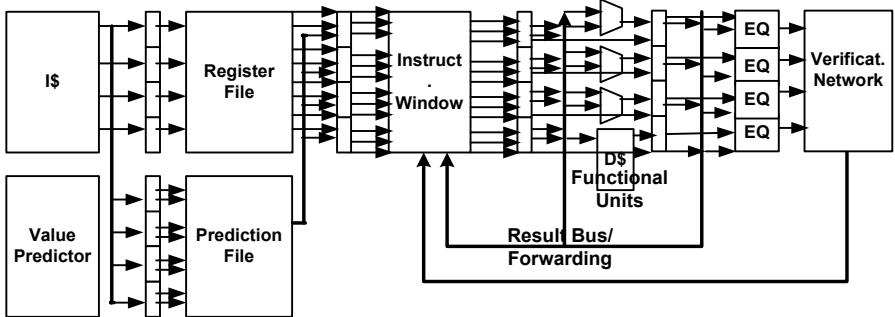


FIGURE 9.5: A microarchitecture with data value speculation.

9.4.3 Other Microarchitectural Implications of Data Value Speculation

The implementation of data value speculation can have important microarchitectural implications. Additional datapaths may be required so that predicted values can be routed and merged with the conventional execution datapath. The data dependence tracking mechanism may need to be modified to facilitate faster verification and recovery such as in [12]. The selection of instructions to be issued may assign different priorities depending on the presence of predicted and speculative operands. Finally, as noted above, the implementation of data value speculation may have indirect performance implications due to increased resource requirements and longer resource holding times.

Figure 9.5 shows some of the microarchitectural implications of data value speculation in the pipeline of an out-of-order processor. It is evident from this logical diagram that additional datapaths and functionalities will be required. For example, values need to flow from the predictor to reservations stations or the execution units, predicted values need to be checked, and verification needs to be performed by a data dependence tracking method. For a more detailed discussion of the microarchitectural consequences of data value speculation we refer the interested reader to [39, 53, 12].

9.5 Related Work and Applications

This section briefly presents some other data speculation related work.

9.5.1 Related Work: Data Value Speculation

Data value speculation can be applied to statically scheduled processors [17, 46] in addition to the dynamically scheduled processors we have focused on in this chapter. A static value speculation approach transfers responsibility for verification and recovery to software similar to other forms of software speculation. The prediction, however, can still be performed dynamically using a hardware predictor with explicit instructions reading and updating the value predictor [17, 46]. Recently there was a proposal for purely software based speculation, where prediction, verification, and recovery are all done in software [35].

The use of value prediction to reduce wire delays for distributed microarchitectures was studied in [47], and value speculation at coarser than single instruction granularity, i.e., instruction regions was studied in [65]. Value speculation was also considered for speculative multithreading [41]. Data value speculation may boost the performance of such architectures by predicting the values that flow from one thread to the other. If the processor is able to predict all the inter-thread data dependent values, concurrent threads can be executed as if they are independent.

The criticality of predicting various instruction types was investigated in [11]. It was shown that it is not vital that all instructions be predicted and that load instructions are more important to predict than other instruction types.

Value prediction for improving branch prediction was studied in [15, 27, 25, 1] and ways of improving confidence estimation were considered in [2]. Value prediction has also applied to compressing simulation traces [8]. The invariance of data values was studied in [10].

9.5.2 Related Work: Data Dependence Speculation

Data dependence speculation has been implemented in some commercially available microprocessors. Mechanisms to detect if a load has been issued before a previous store appears in the most current processors such as the Address Reorder Buffer of the HP-PA8000 [28] and the ALAT [57, 36] in Itanium processors, among others.

In speculative multithreaded processors, data dependence speculation has been intensively used in order to relax the parallelization constraints. Examples of both software and hardware proposals are [60, 31, 26, 30]. One recent study of the potential of compiler directed data speculation is reported in [66].

9.6 Trends, Challenges, and the Future

Run-time data dependence speculation is already present in some commercial processors. However, its potential benefit is very dependent on the underlying microarchitecture. In particular, for a centralized implementation of the load/store queue, a scheme that does not speculate on memory dependences and can issue a load request as soon as all previous stores have computed their effective addresses has on average just 6% lower performance than an ideal configuration with a perfect dependence predictor¹ for the SPEC2000 benchmarks, although in one of the programs, *quake*, the difference is as large as 58%.

However, in other scenarios run-time memory dependence speculation may be more attractive. One of them is for speculative multithreaded processors. As discussed in Chapter 13, data dependence speculation is a key component for that type of microarchitectures. Other scenarios where memory dependence speculation may be important is for distributed implementations of the load/store queue [68]. In these cases, each memory disambiguation logic has information for only a subset of all in-flight loads and stores, and speculation may help to reduce the amount of communication required among the different load/store queues, as well as reduce the penalties caused by the latency of such communications. Finally, memory dependence speculation may help reduce the latency of store-to-load forwarding. As load/store queues become bigger, the latency of forwarding is expected to increase, since it implies all-to-all comparisons. Simple schemes for memory dependence prediction may help to implement forwarding with a low latency.

Software-based data dependence speculation, which may require some hardware support, is present in some ISAs such as Intel Itanium Processors [57]. Unlike run-time schemes, in which the scope of this optimization is basically limited to dependences between loads and stores that are in-flight simultaneously, the scope of this optimization is much larger. In this case, these ISA extensions open new opportunities to compiler optimizations that may not be performed otherwise. Further research in compiler schemes is needed to better assess the potential of this approach.

On the other hand, data value speculation has not yet hit mainstream designs. The reasons may be several. First, the form of predictor adds significant complexity to a design. This added complexity not only comes from the predictor itself, but from the additional data-paths needed for moving predicted values from the predictor to the register file, as well as the added verification and recovery logic necessary. Second, the potential benefits re-

¹This corresponds to data obtained for an 8-way superscalar processor, with a 1024-entry ROB, 512-entry issue queue, and 512-entry load/store queue. If the ROB, issue queue, and load/store queue sizes are reduced to one fourth, the average benefit is just 3.6%.

ported so far for superscalar processors are moderate even if value predictors with relatively good accuracy (around 70% of all values [67] on average) have been proposed. This is partially due to the fact that sometimes the most predictable values are not the critical ones and because value predictions may interfere with other forms of speculation. It is also due to the fact that a superscalar processor has other performance bottlenecks that are not addressed by this technique (e.g., processor to memory gap). More research on how to predict the most critical values and how to leverage the benefits of value prediction with more cost-effective mechanisms may make this technology more attractive. For instance, data value speculation is again a key technique for speculative multithreaded processors (Chapter 13, since it is used to predict the values that are computed by one thread and used by others, which are normally just a few and by far the most critical ones).

To conclude, whereas schemes to speculate on dependences are used in some current microprocessors, the possibility of speculating on values is a fascinating technique from a conceptual standpoint but so far has had a minor impact in current designs. More research on novel microarchitectures and compiler techniques is needed to exploit data value speculation in a more cost-effective way.

9.7 Acknowledgments

We would like to thank Jaume Abella for providing us the data on the performance with a perfect and a realistic memory dependence predictor.

References

- [1] Aragon, J. L., González, J., García, J. M., and González, A. Confidence Estimation for Branch Prediction Reversal. In *Proceedings of the 8th International Conference on High Performance Computing* (December 2001).
- [2] Aragon, J. L., González, J., and González, A. Power-Aware Control Speculation through Selective Throttling. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture* (February 2003), pp. 103–112.

- [3] Balkan, D., Kalamatianos, J., and Kaeli, D. The Impact of Value Mis-speculation on Branch Resolution of Out-of-Order Processors. In *Proceedings of the 1st Annual Value Prediction Workshop (affiliated with ISCA-30)* (June 2003), pp. 3–9.
- [4] Bekerman, M., Jourdan, S., Ronen, R., Kirshenboim, G., Rappoport, L., Yoaz, A., and Weiser, U. Correlated load-address predictors. In *Proceedings of the 26th Annual International Symposium on Computer Architecture* (June 1999).
- [5] Bhargava, R., and John, L. Latency and energy aware value prediction for high-frequency processors. In *Proceedings of the 16th International Conference on Supercomputing* (June 2002), pp. 45–56.
- [6] Bhargava, R., and John, L. Performance and Energy Impact of Instruction-Level Value Predictor Filtering. In *Proceedings of the 1st Annual Value Prediction Workshop (affiliated with ISCA-30)* (June 2003), pp. 71–79.
- [7] Burtcher, M., and Zorn, B. G. Exploring Last n Value Prediction. In *International Conference on Parallel Architectures and Compilation Techniques* (1999).
- [8] Burtscher, M., and Jeeradit, M. Compressing extended program traces using value predictors. In *International Conference on Parallel Architectures and Compilation Techniques* (September 2003), pp. 159–169.
- [9] Burtscher, M., and Zorn, B. G. Prediction Outcome History-based Confidence Estimation for Load Value Prediction. *Journal of Instruction-Level Parallelism 1* (May 1999).
- [10] Calder, B., Feller, P., and Eustace, A. Value Profiling. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture* (December 1997), pp. 259–269.
- [11] Calder, B., Reinman, G., and Tullsen, D. Selective value prediction. In *Proceedings of the 26th Annual International Symposium on Computer Architecture* (June 1999).
- [12] Chen, L., Drposho, S., and Albonesi, D. H. Dynamic Data Dependence Tracking and its Application to Branch Prediction. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture* (February 2003), pp. 65–76.
- [13] Chrysos, G. Z., and Emer, J. S. Memory Dependence Prediction using Store Sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture* (June 1998), pp. 142–153.
- [14] Eickemeyer, R. J., and Vassiliadis, S. A Load Instruction Unit for Pipelined Processors. *IBM Journal of Research and Development* 37, 4 (July 1993), 547–564.

- [15] Farcy, A., Temam, O., Espasa, R., and Juan, T. Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture* (December 1998), pp. 59–68.
- [16] Franklin, M., and Sohi, G. S. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transaction on Computers* 6, 45 (May 1996), 552–571.
- [17] Fu, C., Jennings, M., Larin, S., and Conte, T. Value Speculation Scheduling for High Performance Processors. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1998).
- [18] Gabbay, F., and Mendelson, A. Speculative Execution Based on Value Prediction. Tech. Rep. EE-TR-96-1080, Technion, November 1996.
- [19] Gabbay, F., and Mendelson, A. Can Program Profiling Support Value Prediction? In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture* (December 1997), pp. 270–280.
- [20] Gabbay, F., and Mendelson, A. The Effect of Instruction Fetch Bandwidth on Value Prediction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture* (June 1998), pp. 272–281.
- [21] Goeman, B., Vandierndonck, H., and Bosschere, K. D. Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture* (January 2001).
- [22] González, J., and González, A. Memory Address Prediction for Data Speculation. Tech. Rep. TR-UPC-DAC-1996-51, Universitat Politècnica de Catalunya, 1996.
- [23] González, J., and González, A. Speculative Execution Via Address Prediction and Data Prefetching. In *Proceedings of the 11th Int. Conf. on Supercomputing* (1997).
- [24] González, J., and González, A. The Potential of Data Value Speculation. In *Proceedings of the 12th International Conference on Supercomputing* (June 1998), pp. 21–28.
- [25] González, J., and González, A. Control-Flow Speculation through Value Prediction for Superscalar Processors. In *International Conference on Parallel Architectures and Compilation Techniques* (September 1999).
- [26] Hammond, L., Willey, M., and Olukotun, K. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of the 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems* (1998).

- [27] Heil, T., Smith, Z., and Smith, J. E. Improving Branch Predictors by Correlating on Data Values. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture* (December 1999).
- [28] Hunt, D. Advanced Performance Features of the 64-bit PA-8000. In *Proceedings of the CompCon'95* (1995), pp. 123–128.
- [29] Jacobsen, E., Rotenberg, E., and Smith, J. E. Assigning Confidence to Conditional Branch Predictions. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture* (December 1996), pp. 142–152.
- [30] Kazi, I. H., and Lilja, D. J. Coarse-Grained Speculative Execution in Shared-Memory Multiprocessors. In *Proceedings of the 12th Int. Conf. on Supercomputing* (1998), pp. 93–100.
- [31] Krishnan, V., and Torrellas, J. Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor. In *Proceedings of the 12th Int. Conf. on Supercomputing* (1998), pp. 85–92.
- [32] Larson, E., and Austin, T. Compiler Controlled Value Prediction using Branch Predictor Based Confidence. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture* (2000).
- [33] Lee, S., Wang, Y., and Yew, P. Decoupled Value Prediction on Trace Processors. In *HPCA9* (2000).
- [34] Lee, S., and Yew, P. On Some Implementation Issues for Value Prediction on Wide-Issue ILP Processors. In *International Conference on Parallel Architectures and Compilation Techniques* (2000).
- [35] Li, X., Du, Z., Zhao, Q., and Ngai, T. Software Value Prediction for Speculative Parallel Threaded Computation. In *Proceedings of the 1st Annual Value Prediction Workshop (affiliated with ISCA-30)* (June 2003), pp. 18–25.
- [36] Lin, J., Cheng, T., Hsu, W.-C., and Yew, P.-C. Speculative Register Promotion Using Advanced Load Address Table (ALAT). In *Proceedings of the Int. Symp. on Code Generation and Optimization* (2003), pp. 125–134.
- [37] Lipasti, M. H. Value Locality and Speculative Execution. Tech. Rep. CMU-CSC-97-4, Carnegie Mellon University, May 1997.
- [38] Lipasti, M. H., and Shen, J. P. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture* (December 1996), pp. 226–237.

- [39] Lipasti, M. H., Wilkerson, C. B., and Shen, J. P. Value Locality and Data Speculation. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1996), pp. 138–147.
- [40] Loh, G. Width Prediction for Reducing Value Predictor Size and Power. In *Proceedings of the 1st Annual Value Prediction Workshop (affiliated with ISCA-30)* (June 2003), pp. 86–93.
- [41] Marcuello, P., and González, A. Clustered Speculative Multithreaded Processors. In *Proceedings of the 13th International Conference on Supercomputing* (1999).
- [42] Marcuello, P., and González, A. Value Prediction for Speculative Multi-threaded Architectures. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture* (1999).
- [43] Moshovos, A., Breach, S. E., Vijaykumar, T. J., and Sohi, G. Dynamic Speculation and Synchronization of Data Dependences. In *Proceedings of the 24th International Symposium on Computer Architecture* (June 1997), pp. 181–193.
- [44] Moshovos, A., and Sohi, G. S. Streamlining Inter-operation Memory Communication via Data Dependence Prediction. In *Proceedings of the 30th Int. Symp. on Microarchitecture* (1997), pp. 235–245.
- [45] Nakra, T., Gupta, R., and Soffa, M. L. Global Context-Based Value Prediction. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture* (February 1999), pp. 4–12.
- [46] Nakra, T., Gupta, R., and Soffa, M. L. Value prediction in vliw machines. In *Proceedings of the 26th Annual International Symposium on Computer Architecture* (June 1999).
- [47] Parcerisa, J., and González, A. Reducing Wire Delay Penalty through Value Prediction. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture* (2000).
- [48] Reinman, G., and Calder, B. Predictive Techniques for Aggressive Load Speculation. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture* (December 1998), pp. 127–137.
- [49] Rotenberg, E., Jacobson, Q., Sazeides, Y., and Smith, J. E. Trace Processors. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture* (December 1997), pp. 138–148.
- [50] Rychlik, B., Faistl, J., Krug, B., and Shen, J. P. Efficacy and Performance of Value Prediction. In *International Conference on Parallel Architectures and Compilation Techniques* (October 1998).

- [51] Sazeides, Y. An Analysis of Value Predictability and its Application to a Superscalar Processor. *PhD Thesis, University of Wisconsin-Madison* (1999).
- [52] Sazeides, Y. Dependence Based Value Prediction. Tech. Rep. CS-TR-02-00, University of Cyprus, February 2002.
- [53] Sazeides, Y. Modeling Value Speculation. In *Proceedings of the 8th International Symposium on High Performance Computer Architecture* (February 2002), pp. 211–222.
- [54] Sazeides, Y., and Smith, J. E. Implementations of Context-Based Value Predictors. Tech. Rep. ECE-TR-97-8, University of Wisconsin-Madison, Dec. 1997.
- [55] Sazeides, Y., and Smith, J. E. The Predictability of Data Values. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture* (December 1997), pp. 248–258.
- [56] Sazeides, Y., and Smith, J. E. Modeling Program Predictability. In *Proceedings of the 25th International Symposium on Computer Architecture* (June 1998), pp. 73–84.
- [57] Sharangpani, H., and Arora, K. Itanium Processor Microarchitecture. *IEEE Micro* 20, 5 (Sep./Oct. 2000), 24–43.
- [58] Sodani, A., and Sohi, G. S. Understanding the Differences between Value Prediction and Instruction Reuse. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture* (December 1998), pp. 205–215.
- [59] Sohi, G. S., Breach, S. E., and Vijaykumar, T. N. Multiscalar Processors. In *Proceedings of the 22nd Int. Symp. on Computer Architecture* (1995), pp. 414–425.
- [60] Steffan, J., and Mowry, T. The Potential of Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *Proceedings of the 4th Int. Symp. on High Performance Computer Architecture* (1998), pp. 2–13.
- [61] Thomas, R., and Franklin, M. Using Dataflow Based Context for Accurate Value Prediction. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques* (September 2001), pp. 107–117.
- [62] Tullsen, D. M., and Seng, J. S. Storageless value prediction using prior register values. In *Proceedings of the 26th Annual International Symposium on Computer Architecture* (June 1999).
- [63] Tyson, G., and Austin, T. Improving the Accuracy and Performance of Memory Communication through Renaming. In *Proceedings of the*

30th Annual ACM/IEEE International Symposium on Microarchitecture (December 1997).

- [64] Wang, K., and Franklin, M. Highly Accurate Data Value Prediction using Hybrid Predictors. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture* (December 1997), p. 281–290.
- [65] Wu, Y., Chen, D., and Fang, J. Better exploration of region-level value locality with integrated computation reuse and value prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture* (May 2001).
- [66] Wu, Y., Chen, L., Ju, R., and Fang, J. Performance potential of compiler-directed data speculation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software* (March 2003).
- [67] Zhou, H., Flanagan, J., and Conte, T. M. Detecting Global Stride Locality in Value Streams. In *Proceedings of the 30th International Symposium on Computer Architecture* (June 2003), pp. 324–335.
- [68] Zyuban, W., and Kogge, P. Inherently Low-Power High-Performance Superscalar Architectures. *IEEE Transactions on Computers* 50, 3 (March 2001), 268–3285.

Chapter 10

Instruction Precomputation

Joshua J. Yi,¹ Resit Sendag,² and David J. Lilja³

¹*Freescale Semiconductor Inc.*; ²*University of Rhode Island*; ³*University of Minnesota at Twin Cities*

10.1	Introduction	245
10.2	Value Reuse	246
10.3	Redundant Computations	247
10.4	Instruction Precomputation	248
10.5	Simulation Methodology	253
10.6	Performance Results for Instruction Precomputation	253
10.7	An Analytical Evaluation of Instruction Precomputation	260
10.8	Extending Instruction Precomputation by Incorporating Speculation	262
10.9	Related Work	263
10.10	Conclusion	265
10.11	Acknowledgment	265
	References	266

10.1 Introduction

As a program executes, some computations are performed over and over again. These redundant computations increase the program's execution time since they could require multiple cycles to execute and because they consume limited processor resources. To minimize the performance degradation that redundant computations have on the processor, instruction precomputation hardware can be used to dynamically remove these redundant computations. Instruction precomputation profiles the program to determine the highest frequency redundant computations. These computations then are loaded into the Precomputation Table before the program executes. During program execution, the processor accesses the Precomputation Table to determine whether or not an instruction is a redundant computation; instructions that are redundant receive their output value from the Precomputation Table and are removed from the pipeline. The key difference between instruction precomputation and value reuse – another microarchitectural technique that dynamically removes redundant computations – is that instruction precomputation

does not dynamically update the Precomputation Table with the most recent redundant computations since it already contains those that occur with the highest frequency. For a 2048-entry Precomputation Table, dynamically removing redundant computations yields an average speedup of 10.53%, while, by comparison, a 2048-entry Value Reuse Table produces an average speedup of 7.43%.

10.2 Value Reuse

During the course of a program’s execution, a processor executes many redundant computations. A redundant computation is a computation that the processor already performed earlier in the program. Since the actual input operand values may be unknown at compile time – possibly because they depend on the inputs to the program – an optimizing compiler may not be able to remove these redundant computations during the compilation process.

Redundant computations degrade the processor’s performance in two ways. First, executing instructions that are redundant computations consumes valuable processor resources, such as functional units, issue slots, and bus bandwidth, which could have been used to execute other instructions. Second, redundant computations that are on the program’s critical path can increase the programs overall execution time.

Value Reuse [1] is a microarchitectural technique that improves processor performance by dynamically removing redundant computations from the pipeline. During program execution, the Value Reuse hardware compares the opcode and input operand values of the current instruction against the opcodes and input operand values of all recently executed instructions, which are stored in the Value Reuse Table (VRT). If there is a match between the opcodes and input operand values, then the current instruction is a redundant computation and, instead of continuing its execution, the current instruction gets its output value from the result stored in the VRT. On the other hand, if the current instruction’s opcode and input operand values do not match those found in the VRT, then the instruction is not a recent redundant computation and it executes normally. After the instruction finishes execution, the Value Reuse hardware stores the opcode, input operand values, and output value for that instruction into the VRT.

While Value Reuse can improve processor performance, it does not necessarily target the redundant computations that have the most effect on program execution time. This shortcoming stems from the fact that the VRT is finite in size. Since the processor constantly updates the VRT, a redundant computation could be stored in the VRT, evicted, re-executed, and stored again. As

a result, the VRT could hold redundant computations that have a very low frequency of execution, thus decreasing the effectiveness of this mechanism.

To address this frequency of execution issue, instruction precomputation [2] uses profiling to determine the redundant computations with the highest frequencies of execution. The opcodes and input operands for these redundant computations then are loaded into the Precomputation Table (PT) before the program executes. During program execution, the PT functions like a VRT, but with two key differences: 1) The PT stores only the highest frequency redundant computations, and 2) The PT does not replace or update any entries. As a result, this approach selectively targets those redundant computations that have the largest impact on the program's performance.

10.3 Redundant Computations

Since instruction precomputation does not update the PT with the latest redundant computations, the redundant computations that are loaded into the PT must account for a sufficiently large percentage of the program's total dynamic instruction count or else instruction precomputation will not significantly improve the processor's performance. From another point of view, if the number of redundant computations needed by the instruction precomputation mechanism to improve the processor's performance significantly results in an excessively large PT, then instruction precomputation is not a feasible idea. Therefore, the key question is: Is there a small set of high frequency redundant computations that account for a large percentage of the total number of instructions executed?

To determine the frequency of redundant computations, the opcodes and input operands (hereafter referred to together as a “unique computation”) for all instructions are stored. Accordingly, any unique computation that has a frequency of execution greater than one is a redundant computation, since it is executed more than once in the program. After profiling each benchmark, the unique computations are sorted by their frequency of execution. Figure 10.1 shows the frequency distribution of the unique computations for selected benchmarks from the SPEC CPU 2000 benchmark suite (described in Section 10.5), using logarithmic frequency ranges. Logarithmic ranges were used since they produced the most compact results without affecting the content.

In Figure 10.1, the height of each bar corresponds to the percentage of unique computations that have a frequency of execution within that frequency range. With the exception of *gzip*, almost 80% of all unique computations have execution frequencies less than 10, while over 90% of all unique computations have execution frequencies less than 100. This result shows that most unique computations occur relatively infrequently in a program. Consequently, the

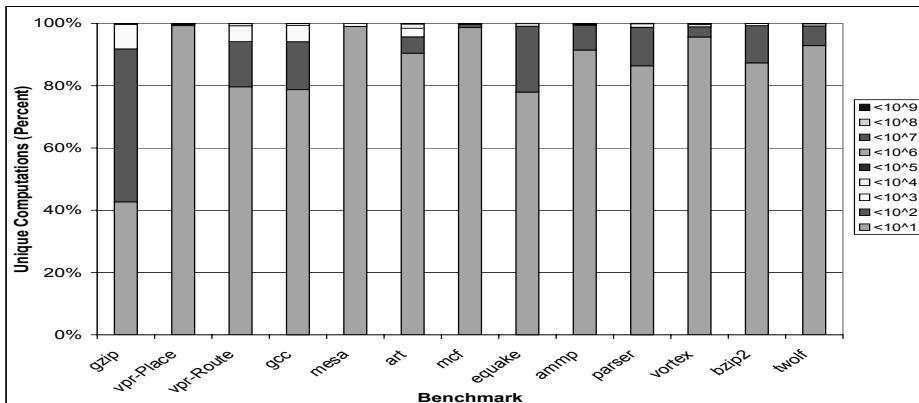


FIGURE 10.1: Frequency distribution of unique computations.

performance benefit of caching most of the unique computations is relatively low since they only execute a few times.

A unique computation's frequency of execution corresponds to the number of dynamic instructions that that unique computation represents. Figure 10.2 shows the percentage of the total number of dynamic instructions that are accounted for by the unique computations in each frequency range for the different benchmark programs. For each frequency range, comparing the heights of the bars in Figures 10.1 and 10.2 shows the relationship between the unique computations and dynamic instructions. For example, in *vpr-Place*, only 3.66% of all dynamic instructions produce more than 99% of all unique computations.

More than 90% of the unique computations account for only 2.29% (*mesa*) to 29.66% (*bzip2*) of the total number of instructions. In other words, a very large percentage of the unique computations cover a disproportionately small percentage of the total number of instructions. On the other hand, a small set of unique computations accounts for a disproportionately large number of instructions. Therefore, simply storing the highest frequency unique computations will cover a significant percentage of a program's instructions.

Table 10.1 shows the percentage of dynamic instructions that are represented by the highest frequency 2048 unique computations. The top 2048 unique computations, which account for a very small percentage of the total unique computations (0.002% - 0.162%), represent a significant percentage of the total dynamic instructions (14.68% - 44.49%). The conclusion from the results in this section is that a significant percentage of a program's dynamic instructions is due to a small number of high-frequency unique computations.

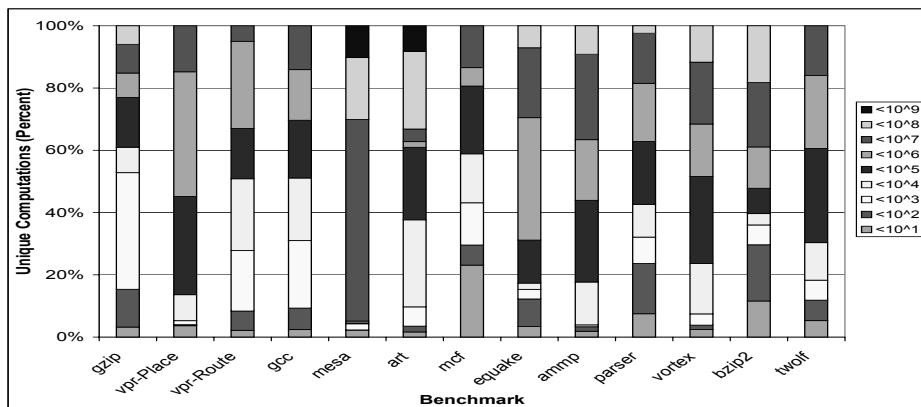


FIGURE 10.2: Percentage of dynamic instructions due to the unique computations in each frequency range.

TABLE 10.1: Characteristics of the 2048 most frequent unique computations

Benchmark	% of Total Unique Computations	% of Total Instructions
gzip	0.024	14.68
vpr-Place	0.029	40.57
vpr-Route	0.162	23.44
gcc	0.032	26.25
mesa	0.010	44.49
Art	0.010	20.24
mcf	0.005	19.04
quake	0.017	37.87
ammp	0.079	23.93
parser	0.010	22.86
vortex	0.033	25.24
bzip2	0.002	26.83
twolf	0.026	23.54

TABLE 10.2: Number of unique computations present in two sets of the 2048 most frequent unique computations, using two different input sets

Benchmark	In Common	Percentage
gzip	2028	99.02
vpr-Place	527	25.73
vpr-Route	1228	59.96
gcc	1951	95.26
mesa	589	28.76
art	1615	78.86
mcf	1675	81.79
quake	1816	88.67
ammp	1862	90.92
parser	1309	63.92
vortex	1298	63.38
bzip2	1198	58.50
twolf	397	19.38

10.4 Instruction Precomputation

As described above, instruction precomputation consists of two main steps: static profiling and dynamic removal of redundant computations. In the profiling step, the compiler executes the program with a representative input set to determine the unique computations that have the highest frequencies of execution. Alternatively, instead of selecting unique computations based solely on their frequency of execution, the compiler could also factor in the expected execution latency to select unique computations with the highest frequency-latency products (FLP). The FLP is simply the unique computation's frequency of execution multiplied by its execution latency.

Since instruction precomputation is based on static profiling using only a single input set, the key question is: Do two input sets have a significant number of high frequency (or FLP) unique computations in common? If the highest frequency computations are simply an artifact of the specific input set that was used, then instruction precomputation cannot be used to improve the performance of the processor since the unique computations are not a function of the program. To answer this question, Table 10.2 shows the number of unique computations that are common between the top 2048 highest frequency unique computations for two different input sets. The second column shows the number of unique computations that are present in both sets while the third column shows that number as a percentage of the total number of unique computations (2048) in either set.

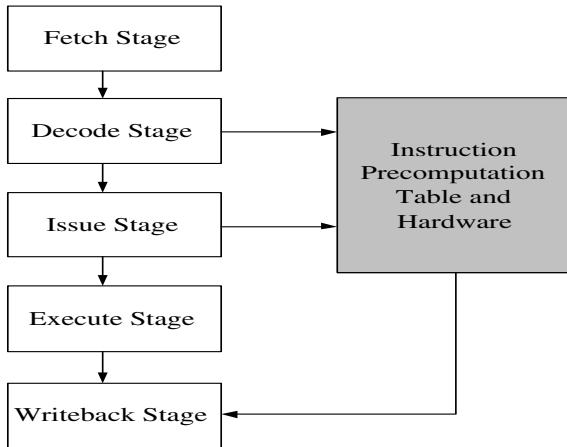


FIGURE 10.3: Operation of a superscalar pipeline with instruction precomputation.

The results in Table 10.2 show that with the exceptions of *vpr-Place*, *mesa*, and *twolf*, at least 50% of unique computations in one set are present in the other set. For *gzip*, *gcc*, and *ammp*, over 90% of the unique computations in one set are present in the other. The key conclusion from Table 10.2 is that for most benchmarks, a significant percentage of the same unique computations is present across multiple input sets. Consequently, the conclusion is that the highest frequency unique computations are primarily a function of the benchmark and less a function of the specific input set. Therefore, instruction precomputation can use static profiling to determine the highest frequency unique computations for a program.

After the compiler determines the set of the highest frequency unique computations, they are compiled into the program binary. Therefore, each set is unique only to that program.

The second step for instruction precomputation is the removal of redundant computations at run-time. Before the program begins execution, the processor initializes the Precomputation Table with the unique computations that were found in the profiling step. Then, as the program executes, for each instruction, the processor accesses the PT to see if the opcode and input operands match an entry in the PT. If a match is found, then the PT forwards the result of that computation to the instruction and the instruction needs only to commit. If a match is not found, then the instruction continues through the pipeline and executes as normal. Figure 10.3 shows how the PT is integrated into the processor's pipeline.

During the decode and issue stages, the opcode and input operands for each dynamic instruction are sent to the PT, when available. The instruction precomputation hardware then determines if there is a match between the

current opcode and input operands with the unique computations in the PT. If a match is found, the instruction precomputation hardware sends the output value for that instruction to the writeback stage, which commits that value when the instruction is retired.

Finally, unlike value reuse, instruction precomputation never updates the PT. Rather, the PT is initialized just before the program starts executing. When a unique computation is not found in the PT, the processor executes the instruction as normal. While this approach eliminates the need for hardware to dynamically update the PT – thus decreasing the complexity of and access time to the PT – it also means that the PT cannot be updated with the most current high-frequency unique computations.

10.4.1 A Comparison of Instruction Precomputation and Value Reuse

Overall, instruction precomputation and value reuse use similar approaches to reduce execution time. Both methods dynamically remove instructions that are redundant computations from the pipeline after forwarding the correct output value from the redundant computation table to that instruction. Both methods define a redundant computation to be one that is currently in the PT or VRT.

The key difference between these two approaches is how a redundant computation gets into the PT or VRT. In instruction precomputation, compiler profiling determines the set of redundant computations that are to be put into the PT. Since it is likely that, for that particular input set, the highest frequency unique computations are already in the PT, there is no need for dynamic replacement. In value reuse, in contrast, if a unique computation is not found in the VRT, then it is added to the VRT, even if it replaces a higher frequency redundant computation.

The VRT may have a lower access time than the PT, however. Instead of comparing the current instruction’s opcode and input operands against every unique computation in the VRT, using the current instruction’s PC as an index into the VRT can reduce the VRT access time by quickly selecting the matching table entry (although the input operands and the tag still need to be compared). As a result, not only does this approach require fewer comparisons than instruction precomputation, it also removes the need to compare opcodes since there can be only one opcode per PC.

10.5 Simulation Methodology

To evaluate the performance potential of instruction precomputation compared to conventional value reuse, sim-outorder, which is the superscalar processor simulator from the SimpleScalar [3] tool suite, was used. The base processor was configured to be a 4-way issue machine. Table 10.3 shows the values of the key processor and memory parameters that were used for the performance evaluations of both techniques. These parameter values are similar to those found in the Alpha 21264 [4] and the MIPS R10000 [5].

Twelve benchmarks were selected from the SPEC CPU 2000 [6] benchmark suite. These benchmarks were chosen because they were the only ones that had MinneSPEC [7] reduced input sets available at the time. MinneSPEC small, medium, and large reduced input sets and SPEC test and train reduced input sets were used to control the execution time. Benchmarks that use a reduced input set exhibit behavior similar to when the benchmark is executed using the reference input [7]. Since reduced input sets were used, the benchmarks were run to completion without any fast-forwarding. All of the benchmarks that were used in this work were compiled at optimization level -O3 using the SimpleScalar version of *gcc* (version 2.63) for the PISA instruction set, which is a MIPS-like ISA. Table 10.4 shows the benchmarks and the specific input sets that were used. The input set in the second and third columns is arbitrarily named “Input Set A” while the other input set is likewise named “Input Set B.”

10.6 Performance Results for Instruction Precomputation

10.6.1 Upper-Bound - Profile A, Run A

The first set of results presents the upper-bound performance results of instruction precomputation; the upper-bound occurs when the same input set is used for both profiling and performance simulation. To determine this upper bound, the benchmark was first profiled with Input Set A to find the highest frequency unique computations. Then the performance of instruction precomputation was evaluated with that benchmark by using Input Set A again. The shorthand notation for this experiment is Profile A, Run A.

Figure 10.4 shows the speedup due to instruction precomputation for 16 to 2048 PT entries. For comparison, the speedup due to using a L1 D-Cache that is twice as large as the L1 D-Cache of the base processor is included. This result, labeled “Big Cache,” represents the alternative of using the chip

TABLE 10.3: Key parameters for the performance evaluation of instruction precomputation

Parameter	Value
Branch Predictor	Combined
Branch History Table Entries	8192
Return Address Stack (RAS) Entries	64
Branch Misprediction Penalty	3 Cycles
Instruction Fetch Queue Entries	32
Reorder Buffer Entries	64
Number of Integer ALUs	2
Number of FP ALUs	2
Number of Integer Multipliers	1
Number of FP Multipliers	1
Load-Store Queue Entries	32
Number of Memory Ports	2
L1 D-Cache Size	32 KB
L1 D-Cache Associativity	2-Way
L1 D-Cache Block Size	32 Bytes
L1 D-Cache Latency	1 Cycle
L1 I-Cache Size	32 KB
L1 I-Cache Associativity	2-Way
L1 I-Cache Block Size	32 Bytes
L1 I-Cache Latency	1 Cycle
L2 Cache Size	256 KB
L2 Cache Associativity	4-Way
L2 Cache Block Size	64 Bytes
L2 Cache Latency	12 Cycles
Memory Latency, First Block	60 cycles
Memory Latency, Following Block	5 Cycles
Memory Bandwidth (Bytes/Cycle)	32
TLB Latency	30 Cycles

TABLE 10.4: Selected SPEC CPU 2000 benchmarks and input sets

Benchmark	Input Set A Name	Instr. (M)	Input Set B Name	Instr. (M)
gzip	Small (log)	526.4	Medium (log)	531.4
vpr-Place	Medium	216.9	Small	17.9
vpr-Route	Medium	93.7	Small	5.7
gcc	Medium	451.2	Test	1638.4
mesa	Large	1220.6	Test	3239.6
art	Large	2233.6	Test	4763.4
mcf	Medium	174.7	Small	117.0
quake	Large	715.9	Test	1461.9
ammp	Medium	244.9	Small	68.1
parser	Medium	459.3	Small	215.6
vortex	Medium	380.3	Large	1050.0
bzip2	Large (source)	1553.4	Test	8929.1
twolf	Test	214.6	Large	764.9

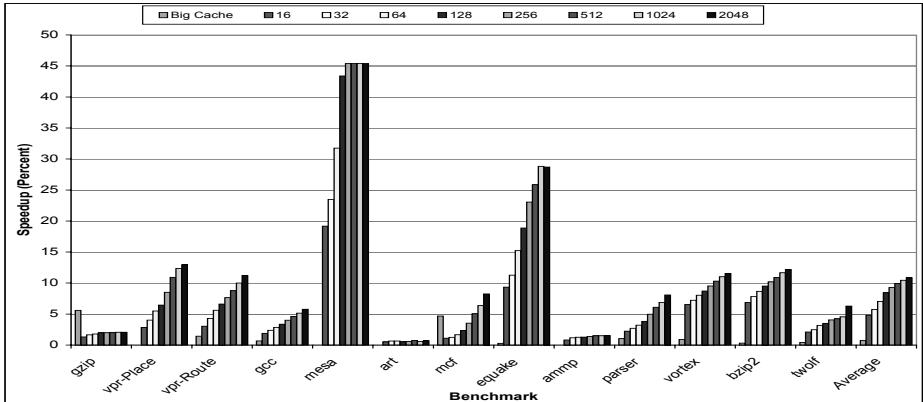


FIGURE 10.4: Instruction precomputation speedup; profile input Set A, run input set A.

area for something other than the PT. The total capacity of this cache is 64 KB. These results show that the average upper-bound speedup due to using a 16-entry PT is 4.82% for these 13 benchmarks (counting *vpr-Place* and *vpr-Route* separately) while the average speedup for a 2048-entry PT is 10.87%. Across all benchmarks, the range of speedups for a 2048-entry PT is 0.69% (*art*) to 45.05% (*mesa*). The average speedup results demonstrate that the upper-bound performance improvement due to instruction precomputation is fairly good for all table sizes.

Instruction precomputation is very effective in decreasing the execution time for two benchmarks, *mesa* and *quake*, since the Top 2048 unique computations account for a very large percentage of the total dynamic instructions for these two benchmarks. Table 10.1 shows that the 2048 highest frequency unique computations account for 44.49% and 37.87% of the total dynamic instruction count in *mesa* and *quake*, respectively.

The average speedup due to using the larger L1 D-Cache is only 0.74%. By comparison, the upper-bound speedup when using a 2048-entry Precomputation Table averages 10.87%. Therefore, using approximately the same chip area for a Precomputation Table instead of for a larger L1 D-Cache appears to be a much better use of the available chip area.

10.6.2 Different Input Sets - Profile B, Run A

Since it is unlikely that the input set that is used to profile the benchmark will also be the same input set that will be used to run the benchmark,

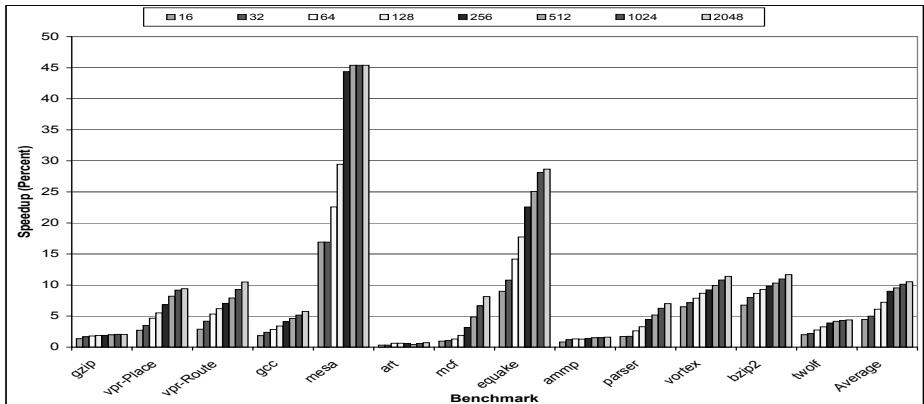


FIGURE 10.5: Instruction precomputation speedup; profile input set B, run input set A.

this section evaluates the situation when two different input sets are used for profiling and for execution. This group of results represents the typical case in which instruction precomputation is most likely to be used. Figure 10.5 shows the speedup due to instruction precomputation when using Input Set B for profiling and Input Set A for execution, i.e., Profile B, Run A.

As shown in Figure 10.5, the average speedup ranges from 4.47% for a 16-entry PT to 10.53% for a 2048-entry PT. By comparison, for the same PT sizes, the speedup for Profile A, Run A ranges from 4.82% to 10.87% for the same table sizes. These results show that the average speedups for Profile B, Run A are very close to the upper bound speedups for the endpoint PT sizes. In addition, with the exception of *mesa*, the speedups for each benchmark are similar.

Although the speedups for a 32-entry, 64-entry, and 128-entry PT are significantly lower than the upper-bound for *mesa*, those differences completely disappear for PT sizes larger than 256 entries. The reason for the speedup differences and their subsequent disappearance is that the highest frequency unique computations for Input Set B do not have as high a frequency of execution for Input Set A. Therefore, until the highest frequency unique computations for Input Set A are included in the PT (for PT sizes larger than 128), the speedup for Profile B, Run A for *mesa* will be lower than the upper-bound speedup.

In conclusion, the key result of this sub-section is that the performance of instruction precomputation is generally not affected by the specific input set since the Profile B, Run A speedups are very close to the upper-bound speedup-

s. This conclusion is not particularly surprising since Table 10.2 showed that a large number of the highest frequency unique computations are common across multiple input sets.

10.6.3 Combination of Input Sets - Profile AB, Run A

While the performance of instruction precomputation is generally not affected by the specific input set, the speedup when different input sets are used for profiling and execution affects the speedup for at least one benchmark (*mesa*). Although the difference in speedups disappeared for PT sizes larger than 256 entries, sufficient chip area may not exist to allow for a larger table. One potential solution to this problem is to combine two sets of unique computations – which are the product of two different input sets – to form a single set of unique computations that may be more representative of all input sets. Since two input sets are profiled and combined together, this approach is called Profile AB, Run A.

To form this combined set, unique computations were selected from the 2048 highest frequency unique computations from Input Set A and Input Set B. Excluding duplicates, the unique computations that were chosen for the final set were the ones that accounted for the largest percentage of dynamic instructions for their input set.

Figure 10.6 shows the speedup due to instruction precomputation for Profile AB, Run A for 16-entry to 2048-entry PT tables. These results show that the average speedup ranges from 4.53% for a 16-entry PT to 10.71% for a 2048-entry PT. By comparison, the speedup for Profile A, Run A ranges from 4.82% to 10.87% for the same table sizes, while the speedup for Profile B, Run A ranges from 4.47% to 10.53%. Therefore, while the average speedups for Profile AB, Run A are closer to the upper bound speedups, using the combined set of unique computations provides only a slight performance improvement over the Profile B, Run A speedups.

The main reason that the speedups for Profile AB are only slightly higher than the speedups for Profile B is that the highest frequency unique computations from Input Set A are very similar to their counterparts from Input Set B, for most benchmarks. Table 10.2 shows that with the exceptions of *vpr-Place*, *mesa*, and *twolf*, more than half of the highest frequency unique computations are common to both input sets. Therefore, it is not surprising to see that the speedup with this combined profile is only slightly higher.

Finally, although Profile AB yields higher speedups, the downside of this approach is that the compiler needs to profile two input sets. Therefore, from a cost-benefit point-of-view, an additional 0.29% (16 PT entries) to 0.15% (2048 PT entries) average speedup is not likely to offset the cost of profiling two input sets and combining their unique computations together.

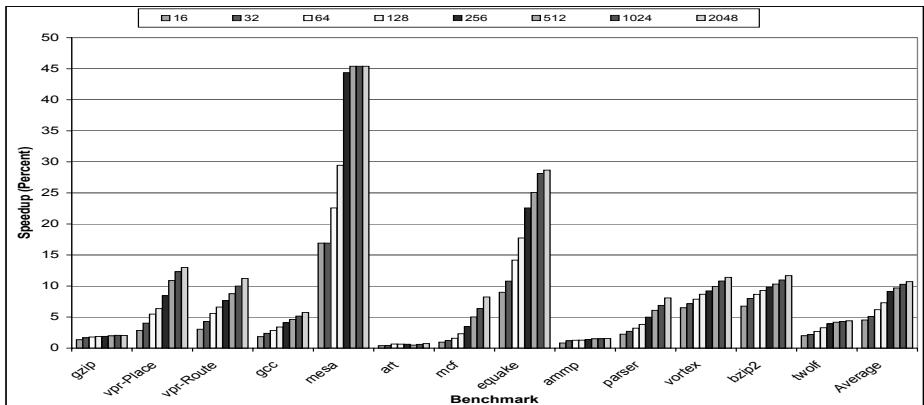


FIGURE 10.6: Instruction precomputation speedup; profile input set AB, run input set A.

10.6.4 Frequency vs. Frequency and Latency Product

Although the set of the highest frequency unique computations represents the largest percentage of dynamic instructions, those instructions could have a lower impact on the execution time than their execution frequencies would suggest since many of those dynamic instructions have a single-cycle execution latency. Therefore, instead of choosing unique computations based only on their frequency of execution, choosing the unique computations that have the highest frequency-latency product (FLP) could yield a larger performance gain. The execution latency of a unique computation is strictly determined by its opcode, except for loads. Consequently, the FLP for a unique non-load computation can be computed by multiplying the frequency of that unique computation by its execution latency.

Figure 10.7 presents the speedup due to instruction precomputation for Profile B, Run A for 16-entry to 2048-entry PT tables. As shown in Figure 10.7, the average speedup ranges from 3.85% for a 16-entry PT to 10.49% for a 2048-entry PT. In most cases, the speedup when using the highest FLP unique computations is slightly lower than the speedups when using the highest frequency unique computations. While this result may seem a little counterintuitive, the reason for this result is because the processor can issue and execute instructions out of order. This out-of-order execution allows the processor to hide the latency of high latency instructions by executing other instructions.

While the out-of-order processor is able to tolerate the effects of longer execution latencies, it is somewhat limited by the number of functional u-

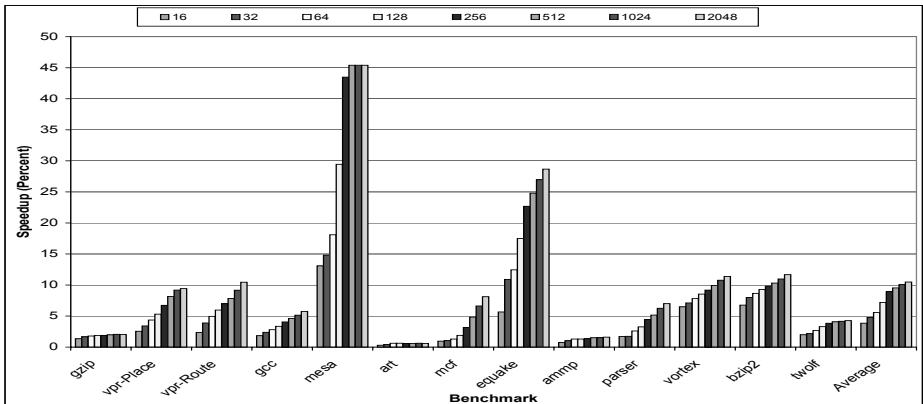


FIGURE 10.7: Instruction precomputation speedup; Profile input set B, run input set A for the highest frequency-latency products.

nits. By using the highest FLP unique computations, fewer instructions are dynamically eliminated (as compared to when using the highest frequency unique computations), thus increasing the number of instructions that require a functional unit. As a result, any performance improvements gained by using the highest FLP unique computations are partially offset by functional unit contention.

10.6.5 Performance of Instruction Precomputation versus value reuse

As described above, the key difference between value reuse and instruction precomputation is that value reuse dynamically updates the VRT while the PT is statically managed by the compiler. Since the two approaches are quite similar, this sub-section compares the speedup results of instruction precomputation with the speedup results for value reuse.

The configuration of the base processor is the same as the base processor configuration for instruction precomputation. The VRT size varies from 16 to 2048 entries. When the program begins execution, all entries of the VRT are invalid. During program execution, the opcode and input operands of each dynamic instruction are compared to the opcodes and input operands in the VRT. As with instruction precomputation, when the opcodes and input operands match, the VRT forwards the output value to that instruction and it is removed from the pipeline. Otherwise, the instruction executes normally.

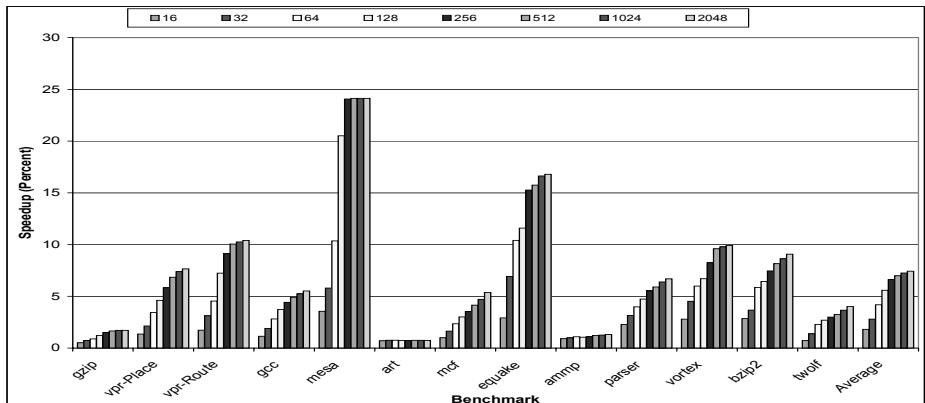


FIGURE 10.8: Speedup due to value reuse; Run A.

Entries in the VRT are replaced only when the VRT is full. In that event, the least-recently used (LRU) entry is replaced.

As shown in Figure 10.8, the average speedup ranges from 1.82% for a 16-entry VRT to 7.43% for a 2048-entry VRT while, by comparison, the speedup for instruction precomputation (Profile B, Run A) ranges from 4.47% to 10.53%. For all table sizes, instruction precomputation has a higher speedup; this difference is especially noticeable for the 16-entry tables.

Since value reuse constantly replaces the LRU entry with the opcode and input operands of the latest dynamic instruction, when the VRT is small, it can easily be filled with low frequency unique computations. By contrast, instruction precomputation is most effective when the table size is small since each entry in the PT accounts for a large percentage of dynamic instructions.

10.7 An Analytical Evaluation of Instruction Precomputation

While the speedup results in the previous section show that instruction precomputation yields significant performance improvements, the key question is: Why does instruction precomputation - and, by association, value reuse - improve the processor's performance? To answer this question, a Plackett and Burman design [8], as described in [9], was applied to instruction precompu-

TABLE 10.5: Processor performance bottlenecks, by average rank, before and after adding instruction precomputation

Component	Before	After	Before - After
ROB Entries	2.77	2.77	0.00
L2 Cache Latency	4.00	4.00	0.00
Branch Predictor Accuracy	7.69	7.92	-0.23
Number of Integer ALUs	9.08	10.54	-1.46
L1 D-Cache Latency	10.00	9.62	0.38
L1 I-Cache Size	10.23	10.15	0.08
L2 Cache Size	10.62	10.54	0.08
L1 I-Cache Block Size	11.77	11.38	0.39
Memory Latency, First	12.31	11.62	0.69
LSQ Entries	12.62	13.00	-0.38

tion. The Plackett and Burman design is a statistically-based approach that measures the effect that each variable parameter has on the output variable, e.g., execution time. By knowing which processor or memory parameters have the most effect on the execution time, the computer architect can determine which parameters are the largest performance bottlenecks. By comparing the performance bottlenecks that are present in the processor before and after instruction precomputation is applied, the effect that instruction precomputation has on relieving or exacerbating the processor's bottlenecks can be easily seen.

The advantage that a fractional multi-factorial design, such as the Plackett and Burman design, has over a full multi-factorial design, such as the Analysis of Variance (ANOVA) design, is that the number of test cases required to execute the design is proportionally related to – instead of exponentially related to – the number of variable parameters.

Table 10.5 shows the results of using a Plackett and Burman design to analyze the effect that instruction precomputation has on the processor's performance bottlenecks. The first column of Table 10.5 lists the ten most significant performance bottlenecks, out of a possible 41, while the second and third columns show the average rank of each component before and after, respectively, instruction precomputation is added to the processor. Finally, the fourth column shows the net change in the average rank. Since the components are ranked in descending order of significance, the most significant performance bottleneck is given a rank of 1, while the second most significant parameter is given a rank of 2, and so on.

The results in Table 10.5 show that instruction precomputation significantly relieves one performance bottleneck, the number of integer ALUs, since its average rank increases. Therefore, instruction precomputation has a similar, but not precisely the same, effect on the processor as adding additional integer ALUs. This result is not particularly surprising since most of the unique com-

putations that are cached in the PT are operations that would have executed on an integer ALU. Therefore, adding instruction precomputation reduces the amount of contention for the integer ALU.

On the other hand, adding instruction precomputation somewhat exacerbates another performance bottleneck, the memory latency of the first block. The result is expected since, with instruction precomputation, the processor core consumes instructions at a faster rate, which puts more stress on the memory hierarchy.

In summary, using a Plackett and Burman design to analyze the effect of instruction precomputation shows that instruction precomputation improves the processor's performance primarily by reducing the amount of pressure on the functional units. However, since the processor's performance is somewhat limited by the memory latency, further performance improvements due to instruction precomputation will come only by also diminishing the impact of the memory latency.

10.8 Extending Instruction Precomputation by Incorporating Speculation

Although the speedup results in Section 10.6 show that instruction precomputation can significantly improve the processor's performance, two problems limit additional performance gains. The first problem is that instruction precomputation is a non-speculative technique. While this characteristic eliminates the need for prediction verification hardware, the associated cost is that the instruction precomputation hardware must wait until both input operands are available. Since many of the unique computations in the PT are for operations that would execute on a low latency functional unit, dynamically removing these redundant computations does not dramatically reduce the latency of these operations; in fact, the only pipeline stages that these computations can bypass – and the total number of cycles that can be saved – are between the stage when the PT is accessed and the execute stage. Consequently, the non-speculative nature of instruction precomputation, i.e. waiting for both input operand values to become available, limits the performance gain.

The second problem is that the access time of the PT depends on the number of bits in the opcode and input operands. Since complete bit-by-bit comparisons are necessary, increasing the number of bits in the input operands from 32 to 64 bits, for example, dramatically increases the access time to the PT. Furthermore, depending on its specific implementation, the PT may be fully-associative, which further increases the PT access time. However, each additional cycle that is needed to access the PT directly decreases the latency reduction benefit of instruction precomputation.

One possible solution to these two problems that can further increase the performance of instruction precomputation is to speculatively “reuse” the output value after one input operand becomes available instead of waiting for both to become available. When one of the input operands becomes available, that value, or a hashed version, is used as an index in the PT. Then, depending on its implementation, the PT would either return the output value of the highest frequency unique computation (or multiple high frequency unique computations) that has that value as one of its input operand values. That output value can then be forwarded to any dependent instructions, which can speculatively execute based on that value. Obviously, the output value of the “reused” instruction needs to be checked and dependent instructions need to be re-executed if the speculation was incorrect.

While this approach is essentially a combination of instruction precomputation and Value Prediction [10], it has at least one advantage over either approach. As compared to instruction precomputation, speculatively reusing instructions requires the instruction precomputation hardware to wait for only a single input operand to become available. Also, fewer bits need to be compared. Furthermore, by profiling to determine the highest frequency unique computations and actually using a single input operand value to index into the PT, this approach may yield higher prediction accuracies than value prediction for more difficult-to-predict output value patterns.

10.9 Related Work

Sodani and Sohi [11] analyzed the frequency of unique computations associated with static instructions in the integer benchmarks of the SPEC 95 benchmark suite. Their results showed that 57% to 99% of the dynamic instructions produced the same result as an earlier instance of the instruction. Therefore, in typical programs, a very large percentage of the computations are redundant. Of the static instructions that execute more than once, most of the repetition is due to a small sub-set of the dynamic instructions. More specifically, with the exception of *m88ksim*, less than 20% of the static instructions that execute at least twice are responsible for over 90% of the dynamic instructions that are redundant. For *m88ksim*, those static instructions are responsible for over 50% of the instruction repetition.

González et al. [12] also analyzed the frequency of redundant computations, but in both the integer and floating-point benchmarks of the SPEC 95 benchmark suite. Their results showed that 53% to 99% of the dynamic instructions repeated and that there is not a significant difference in the frequency of redundant computation between the integer and floating-point benchmarks.

Overall, their results confirm the key conclusion from [11] that there is a significant amount of redundant computations associated with static instructions.

Sodani and Sohi [1] proposed a dynamic value reuse mechanism that exploited the value reuse that was associated with each static instruction. Each static instructions PC is used as an index to access the value reuse table. This approach produced speedups of 0% to 17%, 2% to 26%, and 6% to 43% for a 32-entry, a 128-entry, and a 1024-entry, respectively, VRT.

By contrast, Molina et al. [13] proposed a dynamic value reuse mechanism that exploits the output value that was associated with each static instruction and across all static instructions, i.e., by unique computation. The resultant speedups are correlated to the area used. For instance, when using a 221KB table, the speedups range from 3% to 25%, with an average of 10%. The speedups dropped to 2% to 15% with an average of 7% when the table area decreased to 36KB.

Citron et al. [14] proposed using distributed Value Reuse tables that are accessed in parallel with the functional units. Since this approach reduces the execution latency of the targeted instruction to a single cycle, it is best suited to bypass the execution of long latency arithmetic and logical instructions, e.g., integer and floating-point multiplication, division, and square roots. As a result, although this mechanism produces speedups up to 20%, it is best suited for benchmarks with a significant percentage of high-latency instructions, such as the MediaBench benchmark suite [16].

Huang and Lilja [15] introduced Basic Block Reuse, which is value reuse at the basic block level. This approach uses the compiler to identify basic blocks where the inputs and outputs were semi-invariant. At run-time, the inputs and outputs for that basic block are cached after the first execution of that basic block. Before a subsequent execution, the current inputs of that basic block are compared with cached versions. If a match is found, the register file and memory are updated with the correct results. This approach produced speedups of 1% to 14% with an average of 9%.

Azam et al. [17] proposed using value reuse to decrease the processor's power consumption. Value Reuse can decrease power consumption by reducing the latency of instructions to a single cycle and by removing the instruction from the pipeline. Their results showed that an eight-entry reuse buffer decreased the power consumption by up to 20% while a 128-entry reuse buffer decreased the power consumption by up to 60%, even after adding a pipeline stage to access the reuse buffer.

10.10 Conclusion

Redundant computations are computations that the processor previously executed. Instruction precomputation is a new mechanism that can improve the performance of a processor by dynamically eliminating these redundant computations. Instruction precomputation uses the compiler to determine the highest frequency unique computations, which subsequently are loaded into the Precomputation Table (PT) before the program begins execution. Instead of re-computing the results of a redundant computation, its output value is forwarded from the matching entry in the PT to the dependent instruction and then the redundant instruction is removed from the pipeline.

The results in this chapter show that a small number of unique computations account for a disproportionate number of dynamic instructions. More specifically, less than 0.2% of the total unique computations account for 14.68% to 44.49% of the total dynamic instructions. When using the highest frequency unique computations from Input Set B while running Input Set A (Profile B, Run A), a 2048-entry PT improves the performance of a base 4-way issue superscalar processor by an average of 10.53%. This speedup is very close to the upper-limit speedup of 10.87%. This speedup is higher than the average speedup of 7.43% that Value Reuse yields for the same processor configuration. More importantly, for smaller table sizes (16-entry), instruction precomputation outperforms value reuse, 4.47% to 1.82%. Finally, the results show that the speedup due to instruction precomputation is approximately the same regardless of which input set is used for profiling and regardless of how the unique computations are selected (frequency or frequency/latency product).

Overall, there are two key differences between instruction precomputation and value reuse. First of all, instruction precomputation uses the compiler to profile the program to determine the highest frequency unique computations while Value Reuse does its profiling at run-time. Since the compiler has more time to determine the highest frequency unique computations, the net result is that instruction precomputation yields a much higher speedup than value reuse for a comparable amount of chip area. Second, although using the compiler to manage the PT eliminates the need for additional hardware to dynamically update the PT, it can dramatically increase the compile time since the compiler must profile the program.

10.11 Acknowledgment

A preliminary version of this work was presented at the 2002 Euro-Par conference [2]. This work was supported in part by National Science Founda-

tion grants EIA-9971666 and CCR-9900605, the IBM Corporation, and the Minnesota Supercomputing Institute.

References

- [1] Sodani, A. and Sohi, G., Dynamic Instruction Reuse, *International Symposium on Computer Architecture*, 1997.
- [2] Yi, J., Sendag, R., and Lilja, D., Increasing Instruction-Level Parallelism with Instruction Precomputation, *Euro-Par*, 2002.
- [3] Burger, D. and Austin, T., The SimpleScalar Tool Set, Version 2.0, *University of Wisconsin-Madison Computer Sciences Department Technical Report #1342*, 1997.
- [4] Kessler, R., McLellan, E., and Webb, D., The Alpha 21264 Microprocessor Architecture, *International Conference on Computer Design*, 1998.
- [5] Yeager, K., The MIPS R10000 Superscalar Microprocessor, *IEEE Micro*, 16, 28, 1996.
- [6] Henning, J., SPEC CPU2000: Measuring CPU Performance in the New Millennium, *IEEE Computer*, 33, 28, 2000.
- [7] KleinOsowski, A. and Lilja, D., MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research, *Computer Architecture Letters*, Vol. 1, June 2002.
- [8] Plackett, R. and Burman, J., The Design of Optimum Multifactorial Experiments, *Biometrika*, 33, 305, 1946.
- [9] Yi, J., Lilja, D., and Hawkins, D., A Statistically Rigorous Approach for Improving Simulation Methodology, *International Conference on High-Performance Computer Architecture*, 2003.
- [10] Lipasti, M., Wilkerson, C., and Shen, J., Value Locality and Load Value Prediction, *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.
- [11] Sodani, A. and Sohi, G., An Empirical Analysis of Instruction Repetition, *International Symposium on Architectural Support for Programming Languages and Operating Systems*, 1998.
- [12] González, A., Tubella, J., and Molina, C., The Performance Potential of Data Value Reuse, *University of Politecenica of Catalunya Technical Report: UPC-DAC-1998- 23*, 1998.

- [13] Molina, C., González, A., and Tubella, J., Dynamic Removal of Redundant Computations, *International Conference on Supercomputing*, 1999.
- [14] Citron, D. and Feitelson, D., Accelerating Multi-Media Processing by Implementing Memoing in Multiplication and Division Units, *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998.
- [15] Huang, J. and Lilja, D., Exploiting Basic Block Locality with Block Reuse, *International Symposium on High Performance Computer Architecture*, 1999.
- [16] Lee, C., Potkonjak, M., and Mangione-Smith, W., MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems, *International Symposium on Microarchitecture*, 1997.
- [17] Azam, M., Franzon, P., and Liu, W., Low Power Data Processing by Elimination of Redundant Computations, *International Symposium on Low Power Electronics and Design*, 1997.

Chapter 11

Profile-Based Speculation

Youfeng Wu and Jesse Fang

*Intel Microprocessor Technology
Labs*

11.1	Introduction	269
11.2	Commonly Used Profiles	270
11.3	Profile Collection	272
11.4	Profile Usage Models	276
11.5	Profile-Based ILP Speculations	278
11.6	Profile-Based Data Prefetching	282
11.7	Profile-Based Thread Level Speculations	287
11.8	Issues with Profile-Based Speculation	290
11.9	Summary	292
	References	293

11.1 Introduction

Modern processors often employ speculation to improve instruction level and thread level parallelism. They predict the outcome of data and control decisions, and speculatively execute the operations and commit results only if the original predictions were correct. Mis-speculation usually causes expensive recovery overhead, and therefore processors will need some form of guidance to maximize the benefits of speculation against recovery overhead.

Profiling is an effective technique to guide highly successful speculations. A profile is the aggregate information about the behavior of a program collected from program analysis or past execution. Depending on the hardware speculation mechanism, various profile-based speculations can be used.

Block frequency profile is often used to order basic blocks or form regions to improve execution path speculation [14] [34] [68] [35]. Edge profile is similarly crucial to decide load speculation passing conditional branches to improve instruction level parallelism [46]. Cache profile can be used to identify load instructions that frequently miss caches so that they can be speculatively prefetched into the caches [47] [42] [2], or prevented from occupying cache storage if they cannot be cached profitably [67]. Value profile has been used

successfully to guide stride prefetching [60], speculative reusing [63], and speculative multi-threading [23].

In this chapter, we describe a few common profiles that are used in practice, techniques for profile collection, examples of profile-based speculations, and the issues with successful profile-based speculations.

11.2 Commonly Used Profiles

We may loosely classify the commonly used profiles into the following categories: control flow profile, memory profile, and value profile. Control flow profiles focus on program structures. Memory profiles reveal memory system activities and reference patterns. Value profiles reason about the values of computations and data locations.

11.2.1 Control Flow Profile

Historically, profile-based techniques mostly explore the control flow profiles. A control flow profile provides aggregate information about the properties of the program's structure, such as how frequently a block is executed, how likely a control flow edge takes, and how often a control flow path is executed. Examples of control flow profile include block profile, edge profile, path profile, and call graph profile.

Block profile counts the number of times each basic block is executed, and thus provides relative importance about various areas of a program to guide compiler optimizations. Edge profile captures the frequencies of control transfer between basic blocks, which are often used by compiler to connect basic blocks to enlarge optimization scope. Block profile can be used to derive edge profile although in certain cases that cannot be done accurately. Edge profile can always be used to calculate the block profile [6].

For blocks with multiple predecessors and successors, edge profile sometimes is not sufficient to find the most probable execution paths involving the blocks. Path profile directly determines the frequency that each program path is executed. A program path is usually acyclic, such as a path from a loop entry to a corresponding loop exit or from a function entry to a function return without going through a backedge of any loop [7]. Cyclic paths [69] and inter-procedural paths [3] can also be profiled and useful.

A special inter-procedural control flow profile is the call graph profile, which counts the number of times one procedure calls the others. Call graph profile is especially useful for procedural placement to improve instruction spatial locality [4] [40] [57] [33]. Notice that the frequency that a function calls another function is not simply the block frequency of the basic block containing

the function call. Complication arises when conditional call and indirect call are encountered.

11.2.2 Memory Profile

As memory stalls become more and more important in program performance, many forms of memory profiles begin to emerge. Memory profile characterizes memory reference patterns from past execution, which can be used to speculatively perform memory operations and hide memory stalls. Memory profile has many different forms, such as cache profile, affinity profile, alias profile, stride profile, and invalidation profile.

Cache profile [45] [47] [42] [2] identifies memory references (e.g., loads) that cause the most cache misses. Knowing which loads generate most of the cache misses, the speculative execution may fetch the loaded memory in advance to overlap the cache misses with other executions.

Alias (or data dependence) profiling [44] [23] calculates probabilities that memory operations may conflict with each other to support speculative optimizations that rely on accurate alias information.

Affinity profile [17] maps each pair of field references to a numeric number indicating how often they are both referenced within a pre-defined time interval. A field reordering heuristic can then use this pair-wise relationship to place fields according to their temporal affinity.

Temporal data reference profile [18] identifies hot data streams, which are data reference sequences that frequently repeat in the same order, to support runtime detection and speculatively prefetching of hot data streams.

Stride profile recognizes regular stride reference patterns for load instructions inside loops to help prefetching the loads [60]

Invalidation profiling [64] monitors memory references in candidate data speculation regions and collects invalidation ratios for guiding compile-time data speculations.

11.2.3 Value Profile

A computer operation, such as a load instruction, a subroutine invocation, etc., ultimately produces some outcome values. If the values can be predicted before the operation is executed, the program can run much faster by using the outcome value without waiting for the operation to complete. Value profile [12] [31] provides an effective method to collect the mostly likely outcome values.

Value profile may be collected for individual instructions. For example, a multiplication or divide instruction can be profiled to identify the mostly like operand values. If one of the operands is a runtime constant, the long latency operation can be speculatively replaced by a few simple instructions [58].

Value profile may also be collected at region level. A region of code can be profiled to identify the most likely live-in and live-out values. If the region,

such as a loop, has a few dominant live-in and live-out values, it is then a good candidate for speculative computation reuse [63].

11.3 Profile Collection

The simplest method to collect profile is static program analysis [8] [56] [66]. Profile accuracy can be improved significantly with instrumentation and a separate run of the program with a training input [13]. However, some profile, such as cache profile, is very expensive to collect by instrumentation. Sampling-based hardware performance monitor becomes popular to collect profiles related to the micro-architectural events, such as cache miss and branch miss prediction. Special hardware supports such as the profile buffer and the software-hardware collaborative approach can obtain control flow profiles with very little overhead to support runtime optimizations ([21], [26], [55], [65]). Some compilation system uses multiple profiles to guide its optimizations. For example, the StarJIT Java system uses instrumentation to collect control flow profile for classical optimizations, and uses performance monitoring to collect cache profile for memory optimizations [1].

11.3.1 Static Analysis

By analyzing program syntax and semantics, it is often possible to estimate how often a branch may be taken. This eliminates the drawbacks of other profiling techniques, such as programmer's intervention for a separate profiling phase, or the overhead in the dynamic profiling. A heuristics-based method is proposed in [8], which uses a set of heuristics to predict branch directions, such as "loop branch heuristic" predicts that the loop back edge will be taken, "pointer heuristic" predicts that a comparison of the pointer against null will fail, and "opcode heuristic" predicts that a comparison of an integer for less than zero will fail, etc.

The prediction of the branch directions can be converted to branch probabilities. For example, a fixed assignment of 0.8 may be assigned to be the branch probability for the branch predicted to take, and 0.2 as the branch probability for the branch predicted not to take [56]. A more sophisticated method [66] uses the Dempster-Shafer theory to systematically convert the predictions from all the heuristics to obtain the branch probability information.

One usage of the branch probability information is to derive the relative ordering of the basic blocks. Among the 20% of the hottest blocks, more than 80% of them are identified correctly by static program analysis [66]. Al-

most all commercial compilers have static analysis modules to provide profile information for optimizations when more expensive profiling is not used.

11.3.2 Instrumentation

Instrumenting a program to collect profile data involves the insertion of code to increment a set of counters that corresponds to a subset of the selected program points. For example, to collect Control Flow Graph (CFG) edge profile, a compiler first counts the number of edges in the CFG for a function and assigns a unique identification number to each edge. This identification number is then used as an index into an array of execution counters.

Straightforward instrumentation may incur significant overhead for collecting the profiles. Compilers often try hard to reduce the amount of instrumented code. For example, Knuth [25] suggests a technique to locate the minimal set of blocks to place instrumentation. From the frequencies of the minimal set of blocks, the block frequency for all the blocks can be derived. Similarly, to collect the edge profile for a function, only the edges on the cords of a minimal spanning tree of the control flow graph need to be instrumented. The edge frequency information for other edges can be derived from the instrumented edges [6]. Although the number of all possible paths in a program can be extremely large, clever instrumentation can be placed on the cords of the minimal spanning tree to efficiently collect the frequencies for all the executed paths [7]. Still programs with instrumentation for block/edge/path profiling typically run about 10% to 40% slower than without the instrumentation.

For architecture with predicated execution, the instrumentation code for edge profiling can be predicated with the appropriate condition indicating whether or not the branch was taken. In this manner, a compiler is able to avoid inserting additional basic blocks into the CFG, while inserting instructions “on the edge.” Furthermore, the inserted instructions can then be scheduled in parallel with other instructions in the basic block, thereby reducing the cost of executing the instrumentation code.

Instrumentation can also be used to collect other more detailed profiles (e.g., cache, aliasing, value, etc.). For example, to collect cache profile, code can be inserted before load and store instructions to simulate the effect of the memory reference on cache and record the cache miss statistics. The program with the instrumentation usually runs many times slower than without the instrumentation. The profiling overhead can be reduced via either sampling based instrumentation (e.g., the Burst Tracing [36]) or sampling the hardware performance monitoring unit (PMU).

11.3.3 Hardware Performance Monitoring

One way to reduce profiling overhead is to use infrequent sampling of program PCs to identify frequently executed code. In [70], a time interrupt-based approach is used to collect sampled frequency of the executed code blocks. By

accumulating the number of samples occurred at each code block, the relative order of the code blocks can be determined. The relative ordering is used to lay out basic blocks in an offline process. They reported an overhead of about 0.3% due to the interrupt and the saving of the sampled data.

More sophisticated performance monitoring uses dedicated performance monitoring unit (PMU) to record detailed microarchitectural events for supporting instruction-level profiling. For example, the IPF (Itanium Processor Family) PMU hardware [39] attributes events like branches, cache, and TLB misses to individual instructions so that software can know exactly where to optimize in the program. IPF implements three hardware structures for performance monitoring: Branch Trace Buffer (BTB), Instruction Event Address Register (I-EAR), and Data Event Address Register (D-EAR). By performing statistical sampling on these structures, instruction-level profiling can be done at a low cost.

The BTB captures information about the last 4 to 8 branches, including the branch's PC, branch target's PC, and mispredict status. By taking enough samples of the BTB, a control flow graph for frequently executed code can be constructed, and hot traces can be identified for optimizations. Also the branches that are frequently mispredicted can be identified for branch optimization (such as predication).

The I-EAR records the information about the last I-cache and I-TLB miss, including the instruction PC, miss latency in cycles, and which level serviced the I-TLB miss (L2 I-TLB, VHPT (Virtual Hashed Page Table), or software). D-EAR is similar, it records the miss instruction PC, data address, miss latency in cycles, and which level serviced the D-TLB miss (L2 D-TLB, VHPT, or software). Once the instructions that cause the most cache misses are identified, instruction or data prefetching can be used to improve program performance.

11.3.4 Special Hardware

Although PMU can be used to collect instruction-level profile information, it is primarily designed for performance analysis and tuning. Here we discuss special profiling hardware targeting special optimization needs. For example, The DAISY VLIW processor [27] contains hardware support for profiling in the form of an 8K entry 8-way set associative (hardware) array of cached counters indexed by the exit point ID of a tree region. These counters are automatically incremented upon exit from a tree region and can be inspected to see which exit points are consuming the most time. If it is found that the time spent in a tree region from the root to an exit point is greater than a threshold of the total cycles spent in the program, the tree region is extended at the exit point. Profile-guided tree region extension allows translations using larger windows and achieving a significantly higher ILP.

As another example, the JRPM system [16] uses a hardware profiler to determine dynamic inter-thread dependency behavior and buffer requirement

for identifying the best speculative thread loops (STL) to parallelize dynamically. The load dependency analysis looks for inter-thread dependencies for a STL. To collect inter-thread dependency information, a store timestamp is recorded on a memory or local variable store, and retrieved on a subsequent load to the same address. The store timestamp is checked against thread start timestamps to determine if an inter-thread dependency arc exists to the previous thread ($t - 1$), or an earlier thread ($< t - 1$). The speculative state overflow analysis checks buffer requirement for each STL. The best STL to parallelize should have minimal number of inter-thread dependencies with speculative state fitting within the L1 caches and store buffers. The profiling hardware leverages the store buffer available in the Hydra chip multiprocessor for profiling purposes and adds less than 1% to the total transistor count of Hydra. Benchmarks experience shows an average 7.8% slowdown during profiling. Without hardware support, the profiling execution may slow down the program execution over 100x when applying the analysis with software alone.

11.3.5 Software-Hardware Collaborative Profiling

The approach in [65] achieves low profiling overhead and high profile accuracy by combining two collaborative techniques: compiler analysis that inserts minimal profiling instructions and hardware that asynchronously executes profiling operations in free slots in user programs running on wide-issue processors like IA-64.

With the collaborative method, the compiler first analyzes the user program to determine the minimal set of blocks that need profiling (e.g., using the Knuth algorithm [25]). Next, it allocates profile counter space for each such block and inserts an “iniprof” instruction in the function entry block to let the hardware know the starting address of the profile counter space. Then it inserts a “profile ID” instruction in each profiled block to pass profiling requests to the hardware. The hardware derives the profile counter address from the information passed by the “iniprof” and “profile ID” instructions and generates profile update operations. These update operations are then executed when free slots become available during the user program execution.

Since most of the profiled blocks contains a branch instruction, the “profile ID” instruction can be avoided by encoding the ID into the branch instruction. By using an 8-bit ID field in branch instructions, most “profile ID” instructions can be eliminated. For big functions that require larger profile ID than the 8-bit ID field can hold, the compiler employs a graph partitioning algorithm to partition the CFG into single-entry regions, such that the ID values in each region can be encoded into the 8-bit ID field. It uses a “set offset” instruction in each region entry block to pass an offset value of the region to the hardware.

The new profiling technique enables programs with profiling to run almost as fast as without profiling. For example, a less than 0.6% cycle count overhead for SPECint95 benchmarks is observed while collecting complete block profile without sampling.

11.4 Profile Usage Models

Profile-based optimization typically operates in two phases, a profile phase and an optimization phase that uses the profile. Depending on when the profile is collected, we often see two different usage models: the compile-time profiling model, and the runtime profiling model. There are also recent proposals that suggest continuous profiling for phase detection and re-optimization in dynamic systems.

11.4.1 Compile-Time Profiling

The profiling phase in static compiles is a separate compilation pass. During the profiling pass, the program is compiled and run to collect profile information. After the profile is collected, the program is compiled the second time to use the profile. This approach is straightforward to implement and almost all optimizing compilers support it. However, this two-pass process is inconvenient to use, as it not only complicates software development but also the process for validation and performance evaluation.

Another issue with the profiling pass is the training input required for the profiling run. It is a challenge to obtain representative training input to collect profile. If the training input is not representative of the actual input of the program, the program may be optimized with incorrect profile. For example, many commercial programs have multiple functions to suit various field requirements. It may be more effective to specialize the program with real-world workload at customer sites rather than collecting profile by the program developers.

11.4.2 Runtime Profiling

Dynamic systems run real-world workload at customer sites, and are normally not allowed to have separate trial runs to collect profiles. Consequently, most dynamic optimization systems (e.g., StarJIT [1] IA32EL [9], Transmeta [22], Daisy [26], and Dynamo [5]) use a short initial profiling phase to identify frequently executed code. In the initial profiling phase, blocks of code are interpreted or translated without optimization to collect execution frequency information for the blocks. In the second phase (optimization phase), frequently executed blocks are grouped into regions and advanced optimizations are applied on them.

In dynamic systems, the time to collect the profile is counted as the execution time of the program, and any performance gain from profile-based optimizations has to amortize the profiling overhead first. Consequently the system needs to manage the profiling phase sophisticatedly so that just enough profile is collected and majority of the execution is with the optimized

code. This is especially important for instrumentation based runtime profiling. In [24], a profiling technique is proposed to identify hot paths efficiently. The key idea is to focus the profiling effort on only the potential starting points of hot paths. Once a path starting point has become hot a prediction is made by speculatively selecting the Next Executing Tail (NET) as the hot path. In an implementation of this method, only loop head blocks (targets of backward taken branches) are instrumented to detect hot path heads. If the execution frequency at a path head exceeds the prediction threshold, the head block is considered hot and the next executing path can be collected using incremental instrumentation. With incremental instrumentation, the profiler collects the next path by subsequently collecting each non-branching sequence in that path. If profiling is implemented inside an emulator, such as in a binary translator, the NET path can directly be collected during emulation of the blocks following the path head.

Supporting both the profiling and optimization phases while the real program is running involves significant bookkeeping effort. For example, the interpreter and the optimizer need to work together so the transition between interpreted execution and optimized execution occurs smoothly.

The benefit of the runtime profiling approach is that the profiling phase uses the early part of the same execution to collect profile for the late part of the execution. This requires significantly less profiling effort to obtain similar or more accurate profiles than the training input. For example, the runtime profiling approach in [62] needs to run each basic block only for a few thousands of times and achieves similar or more accurate branch probability profile than that obtained from the training input. Profiling each block for a few thousands of times represents about 0.5% of the profiling operations required for the training run (for the SPEC2000 benchmarks).

11.4.3 Continuous Profiling

Recent studies [10] [41] [54] have shown that some programs exhibit phases behavior. For those programs, a single profiling phase is clearly unable to respond to the phase changes. Continuous profiling collects profile even after the program is optimized. When the executing program changes characteristics significantly from the previous profile, the program is re-optimized with the new profile.

Since the optimized program needs to be continuously profiled, the profile must be collected without noticeably slowing down the optimized execution. The instrumentation-based method in [61] inserts a few instructions in each loop to continuously collect loop trip count information for advanced loop optimizations. The instrumented code incurs about 0.5% of performance penalty. The advanced loop optimization needs to improve the performance for more than 0.5% for the profiling effort to pay off.

PMU-based profiling may also be used for continuous profiling. Although the periodic sampling and profile processing incur overhead, PMU-based pro-

filling can avoid inserting code into optimized programs so the code quality won't suffer. Furthermore, the continuous profiling aims at detecting significant phase changes, and therefore can tolerate certain profile inaccuracy. As the result, the sampling rate can be relatively low to reduce profiling overhead.

The ADORE (ADaptive Object code Reoptimization) system [45] uses PMU to continuously detect instructions with high data cache misses. The overhead of this prefetching scheme and the dynamic optimization is very low due to the use of sampling on the Itanium processor's performance monitoring unit. Using this scheme, they can improve runtime performance by as much as 57% on some SPEC2000 benchmarks compiled at O2 for Itanium-2 processors. For programs compiled at O3 with static prefetching, the system can improve performance by as much as 20% for some SPEC2000 benchmarks.

However, for optimizations other than data prefetching, it is still open whether the continuous profiling and optimization for capturing phase changes is able to improve performance significant enough to offset the overhead of continuous profiling and re-optimization [41].

11.5 Profile-Based ILP Speculations

To improve instruction level parallelism (ILP), it is important to have a large span of program visible to the instruction scheduler. The processor uses branch prediction hardware to speculatively schedule future instructions for parallel execution. The compiler uses profile information to speculatively move instructions across branches to increase ILP. Examples of such speculative transformations are the trace scheduling [28], hot-code optimizations [19], and the superblock optimizations [38]. The compiler may also use profile to arrange code and data [52] so that the control and data flow can be easily speculated by the hardware.

11.5.1 Trace Scheduling

Trace scheduling was first implemented in the Multiflow compiler for a VLIW processor to improve instruction level parallelism for control-intensive programs. The compiler uses edge profile information to select the most likely paths, or “traces” that the code will follow during execution. Each trace is then treated as if it is free of conditional branches during instruction scheduling. The aggressive instruction scheduling may introduce code motions which could cause logical inconsistencies when branches off-trace are taken. The compiler inserts special compensation code into the program graph on the off-trace branch edges to undo those inconsistencies. Experiment [30] shows that trace scheduling improves the performance of SPEC89 suite by 30% over

basic block scheduling. Restricting trace scheduling so that no compensation code is required improves the performance by 25%.

The possibility that a trace may have side entries to the middle of trace complicates the scheduling task significantly. A superblock is essentially a trace without side entries. Any trace with multiple entries can be converted to a superblock by an optimization called tail duplication. Profile-guided superblock optimization and scheduling [38] is now widely used due to its simplicity and performance improvement.

11.5.2 Hot-Cold Optimizations

In a superblock, a loaded value before a branch may be only used after a side-exit is taken. Since the side exit is unlikely to be taken, the load instruction is often unnecessarily executed. In general, a dynamic instruction trace often contains many unnecessary instructions that are required only by the unexecuted portion of the program. The spirit of hot-code optimization is to use profile information to partition the program into frequently executed (hot) and infrequently executed (cold) parts. Unnecessary operations in the hot portion are removed, and compensation code is added on transition from hot to cold as needed. About 3-8% reduction in path length beyond conventional optimization is reported for large Windows NT applications [19].

With hardware support, this idea can be pushed further. FastForward [15] is a collaborative technique that aggressively optimizes the predicted hot paths and employs hardware and compiler collaborations to handle the possibility of mis-prediction.

A FastForward region (FFR) is a copy of a superblock (or hyperblock) with all side exit branches removed. For each removed branch, the compiler inserts an assert instruction [49] inside the FFR or an abort in the original superblock.

On an Itanium processor, each assert instruction takes the predicate of a removed branch as the sole operand. It fires when the predicate (i.e., infrequent condition) becomes true. A fired assert instruction will terminate the execution of the containing FFR, and when that occurs, the partial result computed by the FFR is discarded and the original code is executed.

After the infrequently executed side exits are removed, the FFR regions often have significantly simplified data and control flow graphs. A compiler can optimize such regions much more effectively. FastForward regions enable the compiler to explore more optimization opportunities that were not present in the original program. For example, the following are a few situations:

- More dead code removal. Since many low-probability side exits are removed, a computation that is live out only at the side exits becomes dead. Furthermore, when a low-probability branch is removed with a corresponding abort inserted in the original code, the code that defines the branch condition may become dead.

- Better scheduling. Simplified control flow removes the need for speculation and compensation code in many cases. In addition, large regions provide a large scope for scheduling and thus yield better IPC (instructions per cycle).
- Better register allocation. Simplified data and control flow graphs reduce the number of live variables and thus the register pressure.
- More predication. By removing infrequent side entries and side exits, more predication opportunities exist to explore the modern architecture features [39].

Experimental result shows about 10% reduction in path length beyond conventional optimization for SPECint95 benchmarks.

Notice that partial redundancy elimination (PRE) and partial dead-code elimination (PDE) [32] could achieve some of the benefits of FastForward. However, our experience indicates that PRE and PDE are ineffective and are very complex to implement. FastForward simplifies data and control flow to a single basic block so redundancy and dead code can be easily removed. In addition, simplified data and control flow brings many benefits for other compiler optimizations, such as register allocation, instruction scheduling, and if-conversion.

11.5.3 Code Layout

Code layout optimization [51] improves ILP by reducing pipeline stalls due to instruction cache misses. Typical basic block size is smaller than the i-cache line size. Arranging basic blocks such that the most likely execution path is along the fall-through target of the branches will help improving the icache locality. Also, many processors employ instruction prefetching so that when the current instruction is executed, the next few cache lines of code are prefetched into the instruction cache. Arranging a function close to the function that calls it will improve the chance that the called function will be already prefetched into icache when it is called. Furthermore, the average function size is significantly smaller than page size. Arranging a function close to the function that calls it will improve the chance that the called function will be in the same page as its caller. This will reduce the dynamic memory footprint and potentially reduce ITLB misses as well as bus traffic [57]. For large functions, function splitting can separate cold portion of the function from the hot portion to improve the spatial locality of frequently executed code. Most of the code layout algorithms use profile to determine block, branch, and function call frequencies.

A recent study [52] provides a detailed investigation of profile-driven compiler optimizations to improve the code layout in commercial workloads with large instruction footprints. It shows that code layout optimizations can provide a major improvement in the instruction cache behavior, providing a 55%

to 65% reduction the application misses for 64-128K caches. This improvement primarily arises from longer sequences of consecutively executed instructions and more reuse of cache lines before they are replaced. Furthermore, it shows that better code layout can also provide substantial improvements in the behavior of other memory system components, such as the instruction TLB and the unified second-level cache. The overall performance impact of the code layout optimizations is an improvement of 1.33 times in the execution time of workload.

11.5.4 Data Layout

Traditionally loop transformation (e.g., tiling) and data layout have been able to reduce data cache miss and DTLB miss via program analysis and transformation for numeric programs [48]. For general-purpose programs, however, compiler analysis usually is insufficient to make data layout decisions. As the result, compiler resorts to profile to determine data layout optimization.

Data-layout optimizations synthesize a layout with good spatial locality generally by (i) attempting to place contemporaneously accessed memory locations in physical proximity (i.e., in the same cache block or main-memory page), while (ii) ensuring that frequently accessed memory cells do not evict each other from caches. Finding an optimal memory layout is intractable [50]. Most techniques use profile information to guide the heuristic data layout optimizations. For example, field reordering optimization [17] first collects field pair-wise affinity profile to map each pair of fields to a frequency value of the number of times the two fields are accessed in close time intervals. The profile information is then used to construct an affinity graph of fields, where nodes are fields and arcs are marked with frequency values. Based on graph clustering algorithms, the optimization clusters fields with high temporal affinity into the same cache block. This approach can also be used to reorder global variables by treating all the global variables as the fields of a single data structure [11].

Similar to function splitting, large data structure may be split into hot and cold sub-structures [17]. Hot fields can be identified using the field access frequency profile. If the size of hot fields in the structure is smaller than the cache line size while the original structure is much larger than the cache line size, splitting the hot fields into a separate structure can help reduce cache misses.

A profile analysis framework is proposed in [53] for data-layout optimizations. The framework finds a good layout by searching the space of possible layout, with the help of profile feedback. The search process iteratively prototypes candidate data layouts, evaluating them by simulating the program on a representative trace of memory accesses.

11.6 Profile-Based Data Prefetching

Data prefetching is a speculation technique to overlap cache miss latency with useful computation. Hardware schemes require a significantly large silicon budget. Compile-time automatic data prefetch however is inefficient for general purpose programs. A recent trend is to use profile to guide effective data prefetching.

11.6.1 Stride Prefetching

Irregular programs such as the SPECint2000 benchmarks contain many irregular data references caused by pointer-chasing code. Irregular data references are difficult to prefetch, as the future address of a memory location is hard to anticipate by a compiler or hardware. However, many SPECint2000 programs contain important references with stride patterns. For example, the SPECint2000 197.parser benchmark has code segments as shown in Figure 11.1. The first load at S1 chases a linked list and the second load at S2 references the string pointed to by the current list element. The program maintains its own memory allocation. The linked elements and the strings are allocated in the order that is referenced. Consequently, the address strides for both loads remain the same 94% of the time.

```
for (; string_list != NULL; string_list = sn) {  
    S1: sn = string_list->next;  
    S2: use string_list->string;  
    other operations;  
}
```

FIGURE 11.1: Pointer-chasing code with stride patterns.

SPECint2000 benchmark 254.gap also contains pointer reference loads with stride patterns. An important loop in the benchmark performs garbage collection. A simplified version of the loop is shown in Figure 11.2.

The variable `s` is a handle. The first load at the statement S1 accesses `*s` and it has four dominant strides, which remain the same for 29%, 28%, 21%, and 5% of the time, respectively. One of the dominant stride occurs because of the increment at S4. The other three stride values depend on the values in `(*s & ~3)->size` added to `s` at S3. The second load at the statement S2 accesses `(*s & ~3)->ptr`. This access has two dominant strides, which remain constant for 48% and 47% of the time, respectively. These strides are mostly affected by the values in `(*s & ~3)->size` and by the allocation of the memory pointed to by `*s`.

```

while ( s < bound ) {
    S1: if ((*s & 3 == 0) {           // 71% time is true
        S2:     access (*s & ~3)->ptr
        S3:     s = s + ((*s & ~3)->size)+values;
                  other operations;
        } else if ((*s & 3 == 2) {   // 29% time is true
        S4:     s = s + *s;
        } else {
                  // never come here
        }
}

```

FIGURE 11.2: Irregular code with multiple stride patterns.

Static compiler techniques, however, cannot easily discover the stride patterns in irregular programs. Pointer references make it hard for a compiler to understand the stride patterns of the load addresses. Also, the stride patterns in many cases are the results of memory allocation and compiler has limited ability to analyze memory allocation patterns. Without knowing that a load has stride patterns, it would be futile to insert stride prefetching, as doing so will penalize the references not exhibiting the prescribed strides.

An efficient profiling method to discover loads with stride patterns is developed in [60]. It integrates the stride profiling into the traditional edge profiling pass. The combined profiling pass is only slightly slower than the edge profiling alone, as only loads in loops with high trip counts (e.g., > 128)

are profiled. The resulting stride profile is used to guide compiler prefetching during the profile feedback pass.

The example in Figure 11.3 illustrates the profiling and prefetching techniques. Figure 11.3 (a) shows a typical pointer-chasing loop. Assume that the data address of the load reference $P \rightarrow \text{data}$ at L is P . The compiler instruments the loop for both block frequency and stride profiling as shown in Figure 11.3 (b). The operations `freq[b]++` and `freq[b2]++` collect block frequency information for the loop pre-head block $b1$ and the loop entry block $b2$. The conditional assignment "`pr = (r2/r1) > 128`" sets the predicate `pr` to true when the loop trip count is greater than 128, and false otherwise. The profiling runtime routine `strideProf` is guarded by the predicate `pr` and it is actually invoked only when the predicate `pr` is true. After the instrumented program is run, the stride profile is fed back and analyzed. The profile could indicate that the load at L has the same stride, e.g., 60 bytes, 80% of the time. In this case, the compiler can insert prefetching instructions as shown in Figure 11.3 (c), where the inserted instruction prefetches the load two strides ahead ($120=2*60$). The compiler decides the number of iterations ahead using information such as the size of the loop and the stride value. In case the profile indicates that the load has multiple dominant strides, e.g., 30 bytes 40% of the time and 120 bytes 50% of the time, the compiler may insert prefetching instructions as shown in Figure 11.3 (d) to compute the strides before prefetching. Furthermore, the profile may suggest that a load has a constant stride, e.g., 60, sometimes and no stride behavior in the rest of the execution; the compiler may insert a conditional prefetch as shown in Figure 11.3 (e). The conditional prefetch can be implemented on Itanium using predication.

The stride profile guided compiler prefetching obtains significant performance improvement for the SPECint2000 programs running on real Itanium machines [60]. For example, it achieves a 1.59x speedup for "181.mcf", 1.14x for "254.gap," 1.08x for "197.parser". These performance improvements, with an average of 7% for the entire benchmark suite, are significant for highly optimized SPECint2000 programs running on real machines.

11.6.2 Hot Data Stream Prefetching

A hot data stream is a data reference sequence that frequently repeats in the same order. A hot data stream prefetching scheme is proposed in [18] to dynamically prefetch for general programs. It operates in three phases while the application program is running. First, the profiling phase gathers a temporal data reference profile with sampling. Second, the profiling is turned off and a fast analysis algorithm extracts hot data streams from the temporal data reference profile. The system dynamically injects code at appropriate program points to detect and prefetch these hot data streams. For example, given the hot data stream "a b a c a d a e," the optimizer inserts code to detect the data references "a b a," which is called the prefix of the hot data stream,

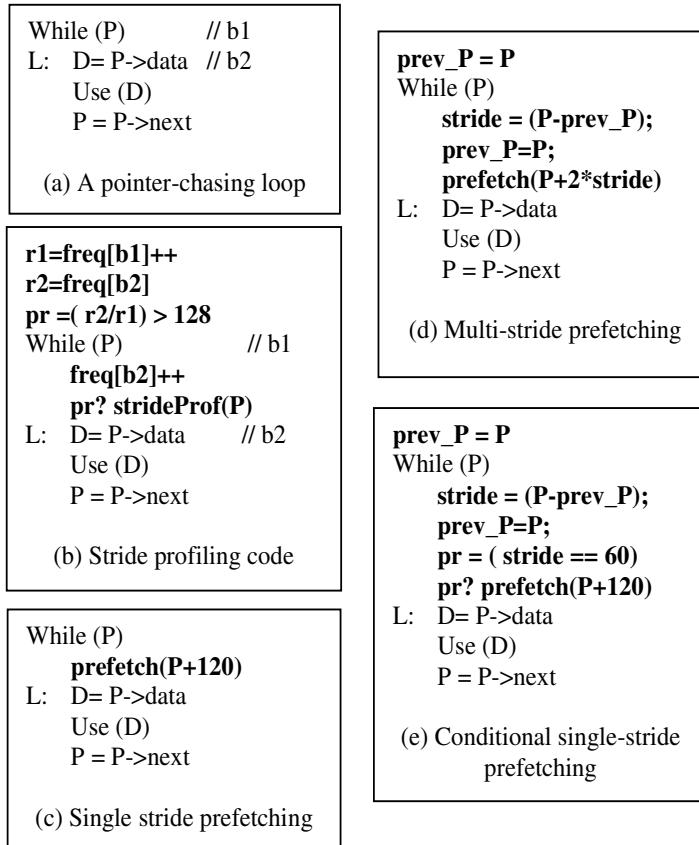


FIGURE 11.3: Example of stride profile guided prefetching.

and prefetches from the addresses “c a d a e” when the prefix is detected. Ideally, the data from these prefetched addresses will be cache resident by the time the data references take place, avoiding cache misses and speeding up the program. Finally, the process enters the hibernation phase where no profiling or analysis is performed, and the program continues to execute with the added prefetch instructions. At the end of the hibernation phase, the program is deoptimized to remove the inserted check and prefetch instructions, and control returns to the profiling phase. For long-running programs, this profile, analyze and optimize, and hibernate cycle will repeat multiple times. Experimental results demonstrate that the prefetching technique is effective,

providing execution time improvements of 5-19% for several SPECint2000 benchmarks running their largest (ref) inputs.

Notice that loads that possess a stride pattern will generate temporal data references without repeating address. For example, a load starting at address x with a stride 1 will generate a long string of temporal data references $x, x+1, x+2, \dots$, and so on without a repeating pattern. Therefore, hot data stream prefetching and stride prefetching usually prefetch different sets of loads so that they can complement each other.

11.6.3 Mississippi Delta Prefetching

Mississippi Delta prefetching (MS Delta) [2] is a novel technique for prefetching linked data structures. It closely integrates the hardware performance monitoring unit (PMU), garbage collector's global knowledge of heap and object layout, and JIT compiler analysis. The garbage collector uses the PMU's data cache miss information to first identify cache-miss intensive traversal paths through linked data structures, and then to discover regular distances between objects along these linked data structures. Coupling this information with JIT compiler analysis, MS Delta materializes prefetch targets and injects prefetches that improve memory subsystem performance.

Figure 11.4 shows the high-level flow of the system as it abstracts from raw PMU samples up to prefetches. First, the IPF PMU hardware provides samples of high latency cache miss. Each sample includes the instruction pointer address (IP) and the reference effective address (EA) of the memory access. MS Delta abstracts these raw samples first into the objects that caused the misses and then into a high level metadata graph, whose nodes represent object types annotated with instruction addresses, and whose edges represent relations induced by fields and array elements containing references. During heap traversal, the garbage collector uses object samples to discover edges in the metadata graph that approximate the high latency traversals between linked data. It then composes these edges into paths representing linked data structure traversals that cause high latency cache misses. Taking advantage of object placement, the garbage collector determines regular deltas between objects along the paths. Knowing the deltas and the paths simplifies the JIT compiler analysis needed to schedule prefetches along a traversal path: The JIT combines the address of the first object in the path with the deltas to materialize prefetch targets. This means that the miss latency experienced by the first object in a traversal path hides the miss latency of subsequent objects along the path. MS Delta prefetching achieves a speedup of 12% to SPEC JBB2000 benchmark over a baseline system that performs dynamic profile-guided optimizations but no data prefetching.

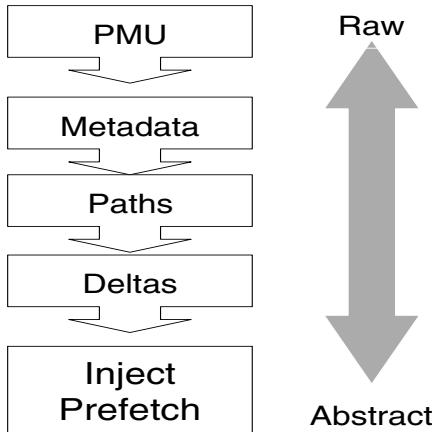


FIGURE 11.4: High-level algorithm flow.

11.7 Profile-Based Thread Level Speculations

Thread level speculation explores parallelism beyond the instruction level. It may speculatively execute multiple loop iterations or multiple regions in parallel. Java Runtime Parallelizing Machine [16] speculate loop iterations within a dynamic compilation system. Speculative Parallel Threading [23] speculates loop iterations in a two-pass static compiler. Computation reuse [67] speculates regions of code. A helper thread [43] may execute code covering multiple functions, trying to overlap cache miss latency with useful computations. In all these techniques, profile significantly improves the effectiveness of the speculations.

11.7.1 Java Runtime Parallelizing Machine

Java runtime parallelizing machine (JRPM) [16] is a dynamic compilation system supported by a special hardware profiler, see Section 11.3.4. The system is based on a chip multiprocessor (CMP) with thread-level speculation (TLS) support. CMPs have low sharing and communication costs relative to traditional multiprocessors, and thread-level speculation (TLS) simplifies program parallelization by allowing users to parallelize optimistically and relies on hardware to preserve correct sequential program behavior.

When the input (e.g., bytecode) is first translated, loops without obvious serializing dependency are annotated so the hardware profiler will analyze the speculative buffer requirements and inter-thread dependencies of the prospective speculative thread loops (STLs) in real-time to identify the best loops

to parallelize. When sufficient data have been collected for a potential STL (e.g., at least 1000 iterations have been executed), the estimated speedup for a STL is computed using the dependency arc frequency, thread sizes, critical arc length, overflow frequency, and speculation overheads. Only loops with predicted speedups > 1.2 are recompiled into speculative threads. Experimental results demonstrate that JRPM can exploit thread-level parallelism with minimal effort from the programmer. On four processors, it achieved speedups of 3 to 4 for floating point applications, 2 to 3 on multimedia applications, and between 1.5 and 2.5 on integer applications.

11.7.2 Speculative Parallel Threading

Selecting profitable loops for speculative parallel threading (SPT) can also be explored in a static compiler with the support of profiling. The cost-driven compilation framework [23] for speculative parallelization of sequential programs employs both dependency profile and value profile.

The cost-driven compilation framework uses a two-pass compilation process to select and transform all and only “good” SPT loops in a program. In the first pass compilation, the initial loop selection selects all loops that meet simple selection criteria (such as the loop body size requirements) as SPT loop candidates. Loop preprocessing such as loop unrolling and privatization is then applied to transform the loops into better forms for speculative parallelization. For each loop candidate, the SPT compilation framework core is invoked to determine its best SPT loop partition (e.g., which portion of the loop body to run sequentially and which part to run in parallel). The result of the first pass is a list of SPT loop candidates with their optimal SPT loop partition result. All loop transformations are performed in a tentative manner and do not alter the compiled program. No actual SPT code is generated. This pass allows the compiler to measure the amount of speculative parallelism in all loop candidates (including each level of a loop nest) and their potential benefits/costs.

The second pass takes the result from the first pass and performs the final SPT loop selection. It evaluates all loop candidates together (not individually as in the case in the first pass) and selects only those good SPT loops. Then the selected loops are again preprocessed and partitioned. Finally the compiler applies the SPT transformations to generate the SPT loop code.

The SPT compilation includes a data dependence profiling tool. The profiling is done offline, and the results are used during pass 1 compilation. The data dependence profile information together with the reaching probability information of the control flow graph is used to annotate the cost graph for best SPT loop partition.

The SPT compilation also uses a technique called software value prediction to reduce misspeculation overhead. For critical dependencies that introduce unacceptably high misspeculation costs, the compiler instruments the program (during pass 1) to profile the value patterns of the corresponding variables. If

the values are found to be predictable, and the total value-prediction overhead and the mis-speculation cost are acceptably low, the compiler inserts the appropriate software value prediction code to generate the predicted value in the selected STP loop code. It also generates the software check and recovery code to detect and correct potential value mis-prediction.

With the sophisticated cost-driven compilation and profiling support, this technique may achieve an average 15% speedup for SPECint2000 benchmarks.

11.7.3 Speculative Computation Reuse

Computation Reuse [20] caches the inputs and results of previous execution instances of a computation region. When the same computation is encountered later with previously seen inputs, the cached results are retrieved and used directly without actually executing the computation. Performance of programs improves due to skipping the execution of the reusable computation.

Computation Reuse uses a buffer (called the computation reuse buffer or CRB) to cache the inputs and output for each computation region. If the inputs or outputs changes frequently, CRB will overflow and the computation may not be able to be reused. For example, if the input parameter to the region changes with a constant stride, such as $x+1, x+2, \dots, x+i$, it will be unlikely that the current input matches any of the previously seen input. Furthermore the compiler needs to insert an invalidation instruction after each store instruction that may access the same address as a memory operation (either a load or a store) inside a computation region. When an invalidation instruction is executed, the hardware marks the CRB entry for computation region specified in the invalidation instruction as invalid. An invalid region will not be reused. However, the invalidation instructions often have to be inserted conservatively as alias analysis is a known hard problem. In addition, even when a store accesses the same memory location as that accessed by a memory operation in a computation region, the stored value may remain unchanged and the region may still be reused.

Speculative computation reuse [63] uses value profile to help computation reuse. It includes a loop or an instruction into a region only if the loop or the instruction has up to 5 result values that account more than 60% of all possible result values for the loop or the instruction. To prevent the unnecessary invalidation, speculative computation reuse takes advantage of the speculative multithreading hardware. In speculative reuse, there is no need to insert invalidation instructions. Instead, the output in the matching instance can be “predicted” to be correct and is speculatively reused, even though some store executed outside the region may have accessed the same address as a memory operation inside the region. The speculative multithreading hardware is used to overlap the speculative execution of the code after the region with the execution of the computation region that verifies the predicted output. The speculative reuse will not commit result to memory or register file until

the verification thread confirms that the predicted output is correct. If the verification thread finds out that the reused output is incorrect, the speculative execution will be squashed. At that time, the output of the computation region is available from the verification thread and can be used in the code following the region non-speculatively.

Experimental result shows that speculative computation reuse with 4 inputs/outputs per region can achieve up to 40% performance gain.

11.7.4 Software-Based Speculative Precomputation

Software-based speculative precomputation [43] is a profile-guided technique that aims at improving the performance of single-threaded applications by leveraging multithreading resources to perform memory prefetching via speculative prefetch threads. SSP first collects cache profiles from the simulator or PMU to identify the delinquent loads that contribute to at least 90% of the cache misses. It then analyzes the single threaded application to generate prefetch threads and identify and embed trigger points for the prefetching threads. The execution of the resultant program spawns the speculative prefetch threads, which are executed concurrently with the main thread. Experiment with Itanium-like in-order processor shows that for a set of pointer-intensive benchmarks, software-based speculative precomputation achieves an average of 87% performance improvement.

11.8 Issues with Profile-Based Speculation

Although profiling is effective to improve speculation accuracy and is widely used, several issues with profile-based speculation remain to be solved or need further investigations. For example, static profiling via program analysis usually can not produce profile that is accurate enough for aggressive speculations. Some compilers select optimization levels based on which kinds profiles (static or dynamic) are available. Even instrumentation-based dynamic profiles may face the following issues.

11.8.1 Stability across Multiple Workloads

Profile-based optimizations are by nature speculative. Its usefulness depends on how accurate the profile collected from early run captures the future behavior of the program. Block and edge profiles are relatively stable across different workloads [29]. However, lower level profiles, such as cache profile, are more sensitive to workloads and the underlining machines [37].

11.8.2 Update when Program Changes

To support profile-based optimization, a data base of profile for the input program is stored. Large scale commercial program involves millions of lines of code and many third party modules. Even a small change in one part of the program may affect the profile globally. Recompiling the whole program to regenerate the profile could be overwhelming. Program analysis may help to derive new profile for the changed portion. If this is not possible, program analysis may help identify the portion of the program to re-profile and recompile locally.

11.8.3 Maintenance during Optimizations

A profile is usually used by multiple optimizations. For example, an edge profile may be used in both the function inlining optimization to decide the profitable call sites, and in region formation to expand optimization scope. However, when one optimization changes the control flow graph, the edge profile may change and the optimization must maintain the profile. Otherwise the profile may become out-dated quickly.

For example, the control flow profile is usually collected early so the profile can be used by all the optimization phases. In this case, each optimization needs to update the profile to reflect the program changes in the optimized code. This profile maintenance issue is studied in [59].

If the profile is collected after the optimizations, such as the profile collected from performance monitoring, the binary level profile needs to be mapped to the intermediate representation of the IR before it is used by the optimization. This mapping usually requires compiler annotation about how the binary code is optimized and generated, and sometimes the precise mapping is impossible.

11.8.4 Perturbation by Profiling Code

Profiling may perturb the application program in several ways. For memory profiling (such as the temporary reference profiling), instrumented code and profiling runtime may change the memory reference addresses. This may prevent techniques such as hot data stream prefetching [18] from being employed by the static compilers. Similarly, profiling operations usually increase the execution time of the user programs. This poses a serious challenge to the dynamic optimization systems to obtain sufficient performance gain to overcome the profiling overhead. For real-time and reactive systems the increased execution time may affect the program behavior or even cause the program fail to run correctly.

11.9 Summary

Modern processors with speculation support can benefit greatly from profile information and profile-guided optimizations. The commonly used profiles include the control-flow profile, memory profile, and value profile. Most of the commercial compilers support static analysis to obtain an estimate of the control flow profile. Program instrumentation sometimes is used to collect more accurate profile with a training run. Hardware performance monitoring enables sampling based profiling without modifying the user programs. There is also special profiling hardware in systems where normal profiling technique is insufficient or too slow. A recent trend is to use software and hardware collaboration to achieve accurate profile at low cost.

Profiles can be used by a static compile to boost program performance. Dynamic optimizers typically have an initial profiling phase to collect profile at runtime for identifying and optimizing frequently executed code. For programs with significant phase changes, continuous profiling and optimizations may be necessary to achieve the best performance.

Compilers have been very successful at improving instruction level parallelism with profile-guided optimizations. Profile-guided superblock optimization is widely used in both static and dynamic compilation systems. Code and data layout as well as hot-cold optimization all rely on accurate profile, and have shown good performance potentials. Recent progress demonstrates outstanding performance improvement with profile guided data prefetching.

Thread-level speculation can benefit from profile information even more as mis-speculation overhead increases with the increase in speculation granularity. Profile information allows the compiler to avoid speculation that is not profitable. Profile information is also instrumental in new thread-level speculation paradigm, such as speculative computation reuse and speculative helper threads.

There are, however, many issues remaining to be resolved to successfully deploy instrumentation-based profile-guided speculation. The stability of a profile across multiple workloads is still affecting the usefulness of profiles. Profile collection remains cumbersome and the profile generation phase needs to be repeated even when only a small portion of a program changes. Using profile information within a compiler also poses challenges to compiler writers as maintaining profile accurately during optimization is both subtle and error-prone. Furthermore, profiling increases execution time, which can affect the behavior of real-time and reactive systems.

References

- [1] A. Adl-Tabatabai, J. Bharadwaj, D-Y. Chen, A. Ghuloum, V. Menon, B. Murphy, M. Serrano, and T. Shpeisman. The starjit compiler: A dynamic compiler for managed runtime environments. *Intel Technology Journal*, Feb. 2003.
- [2] A. Adl-Tabatabai, R. L. Hudson, M. J. Serrano, and S. Subramoney. Prefetch injection based on hardware monitoring and object metadata. In *Proceedings of the ACM Programming Languages, Design, and Implementation*, Jun. 2004.
- [3] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, May 1997.
- [4] Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. Call graph prefetching for database applications. *ACM Transactions on Computer Systems (TOCS)*, 21(4), Nov. 2003.
- [5] V Bala, E. Duesterwald, and S. Banerjia. Transparent dynamic optimization. Technical report, Hewlett Packard Labs, Jun. 1999. HPL-1999-77.
- [6] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4), 1994.
- [7] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 46–57, Dec. 2-4 1996.
- [8] T. Ball and J. R. Larus. Branch prediction for free. In *PLDI03*, 1993.
- [9] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2003.
- [10] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W.W. Hwu. Compilation and run-time systems: Vacuum packing: extracting hardware-detected program phases for post-link optimization. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, Nov. 2002.

- [11] B. Calder, K. Chandra, S. John, and T. Austin. Cache conscious data placement. In *Proceedings of ASPLOS'98, Conference on Architectural Support for Programming Languages and Operating Systems*, 1998. (San Jose CA, 1998).
- [12] B. Calder, P. Feller, and A. Eustace. Value profiling. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 259–269, Dec. 1997.
- [13] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic code optimizations. *Software-Practice & Experience*, 21(12):1301–1321, Dec. 1991.
- [14] P. P. Chang and W. W. Hwu. Trace selection for compiling large c application programs to microcode. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 1998.
- [15] L.-L. Chen and Y. Wu. Aggressive compiler optimization and parallelism with thread-level speculation. In *International Conference on Parallel Processing*, Kaohsiung, Taiwan, October 2003.
- [16] M. K. Chen and K. Olukotun. The JPRM system for dynamically parallelizing Java programs. *Proceedings of the 30th Annual ACM International Symposium on Computer Architecture*, May 2003.
- [17] T.M. Chilimbi, B. Davidson, and J. R. Larus. Cache conscious structure definition. In *Proceedings of the ACM Conference on Programming Languages, Design, and Implementation*, pages 13–24, May 1999. Atlanta, GA.
- [18] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. *Proceedings of the ACM Conference on Programming Language, Design, and Implementation*, May 2002.
- [19] R. Cohn and P.G. Lowney. Hot cold optimization of large windows/nt applications. In *MICRO-29. Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 80–89, Dec. 2-4 1996.
- [20] D. Connors and W.-M. Hwu. Compiler-directed dynamic computation reuse: Rationale and initial results. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, Nov. 1999.
- [21] T.M. Conte, K.N. Menezes, and M.A. Hirsch. Accurate and practical profile-driven compilation using the profile buffer. In *MICRO-29. Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 36–45, Dec. 2-4 1996.
- [22] J.C. Dehnert, B.K. Grant, J.P. Banning, R. Johnson, T. Kistler, A. K-laiber, and J. Mattson. The transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life

- challenges. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 15–24, Mar. 2003.
- [23] Z.-H. Du, C. Lim, X. Li, C. Yang, Q. Zhao, and N. Tin-fook. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proceedings of the ACM Conference on Programming Languages, Design, and Implementation*, Jun. 2004.
 - [24] E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2000.
 - [25] Knuth D. E. and F. R. Stevenson. Optimal measurement of points for program frequency counts. *BIT*, 13:313–322, 1973.
 - [26] K. Ebcioğlu and E. R. Altman. Daisy: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th International Symposium on Computer Architecture*, Jun. 1997.
 - [27] K. Ebcioğlu, E. R. Altman, M. Gschwind, and S. Sathaye. Dynamic binary translation and optimization. *IEEE Transactions on Computers*, 50(6), Jun. 2001.
 - [28] J. A. Fisher. Trace scheduling: A technique of global microcode compaction. *IEEE Transactions on Computers*, C-30(7), Jul. 1981.
 - [29] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, Oct. 1992. Boston, MA.
 - [30] S. M. Freudenberger, T. R. Gross, and P. G. Lowney. Avoidance and suppression of compensation code in a trace scheduling compiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4), Jul. 1994.
 - [31] F. Gabbay and A. Mendelson. Can program profiling support value prediction? In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, Dec. 1997.
 - [32] R. Gupta, D.A. Benson, and J.Z. Fang. Path profile guided partial dead code elimination using predication. In *Proceedings of Parallel Architectures and Compilation Techniques*, 1997.
 - [33] R.J. Hall. Call path profiling. In *Proceedings of the 14th International Conference on Software Engineering*, Jun. 1992.
 - [34] R.E. Hank, S.A. Mahlke, R.A. Bringmann, J.C. Gyllenhaal, and W.W. Hwu. Superblock formation using static program analysis. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 247–255, Dec. 1993.

- [35] R. E. Hank, W. W. Hwu, and B. Ramakrishna Rau. Region-based compilation: an introduction and motivation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, Dec. 1995.
- [36] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *Proceedings of the Workshop on Feedback-Directed and Dynamic Optimizations (FDDO)*, 2001.
- [37] W. C. Hsu, H. Chen, P. C. Yew, and D.-Y. Chen. On the predictability of program behavior using different input data sets. In *Proceedings of the 6th Annual Workshop on Interaction between Compilers and Computer Architectures*, pages 45–53, Feb. 2002.
- [38] W.M. Hwu and et al. The superblock: An effective technique for vliw and superscalar compilation. In *The Journal of Supercomputing*, 1993.
- [39] Intel Corporation. *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*, May 2002.
- [40] J. Kalamatianos, A. Khalafi, D.R. Kaeli, and W. Meleis. Analysis of temporal-based program behavior for improved instruction cache performance. *IEEE Transactions on Computers*, 48(2), 1999.
- [41] T. Kistler and M. Franz. Continuous program optimization: A case study. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(4):500–548, 2003.
- [42] A.R. Lebeck and D.A. Wood. Cache profiling and the spec benchmarks: A case study. *Computer*, 27(10):15–26, Oct. 1994.
- [43] S. W. Liao, P. H. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. P. Shen. Post-pass binary adaptation for software-based speculative pre-computation. In *Proceedings of the ACM Conference on Programming Language, Design and Implementation*, May 2002.
- [44] J. Lin, T. Chen, W.-C. Hsu, P.-C. Yew, R. D. Ju, T.-F. Ngai, and S. Chan. A compiler framework for speculative analysis and optimizations. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, May 2003.
- [45] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2003.
- [46] S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W.-M. W. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling: A model for compiler-controlled speculative execution. *ACM Transactions on Computer Systems (TOCS)*, 11(4), Nov. 1993.

- [47] T.C. Mowry and C.-K. Luk. Understanding why correlation profiling improves the predictability of data cache misses in nonnumeric applications. *IEEE Transactions on Computers*, 49(4):369–384, Apr. 2000.
- [48] N. Park, B. Hong, and V.K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Transactions on Parallel and Distributed Systems*, 14(7):640–654, 2003.
- [49] S. J. Patel and S. S. Lumetta. Replay: A hardware framework for dynamic program optimization. Technical report, CRHC, Draft, Dec. 1999.
- [50] E. Petrank and D. Rawitz. The hardness of cache conscious data placement. In *Proceedings of the Conference on Principles of Programming Languages*, Jan. 2002. Portland, OR.
- [51] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM Conference on Programming Language, Design, and Implementation*, pages 16–27, Jun. 1990. White Plains, NY.
- [52] A. Ramirez, L. Barroso, K. Gharachorloo, R. Cohn, J. L. Pey, P. G. Lowney, and M. Valero. Code layout optimizations for transaction processing workloads. *Proceedings of the 28th Annual International Symposium on Computer Architecture*, May 2001.
- [53] Shai Rubin, Rastislav Bodik, and Trishul Chilimbi. An efficient profile-analysis framework for data-layout optimizations. *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 37(1), Jan. 2002.
- [54] Timoth Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, Jun. 2003.
- [55] S Subramanya Sastry, Rastislav Bodik, and James E. Smith. Rapid profiling via stratified sampling. *Proceedings of the 28th Annual International Symposium on Computer Architecture*, May 2001.
- [56] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison. Accurate static estimators for program optimization. In *Proceedings of the ACM Conference on Programming Language, Design, and Implementation*, Jun. 1994.
- [57] Y. Wu. Ordering functions for improving memory reference locality in shared memory multiprocessor systems. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, Dec. 1992.
- [58] Y. Wu. Strength reduction of multiplications by integer constants. *SIGPLAN Notices*, 30(2):42–48, 1995.

- [59] Y. Wu. Accuracy of profile maintenance in optimizing compilers. In *Proceedings of the 6th Annual Workshop on the Interaction Between Interaction between Compilers and Computer Architectures*, Feb. 2002.
- [60] Y. Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Proceedings of the ACM Conference on Programming Language, Design, and Implementation*, May 2002.
- [61] Y. Wu, M. Breternitz, and T. Devor. Continuous trip count profiling for loop optimizations in two-phase dynamic binary translators. In *Proceedings of the 8th Annual Workshop on the Interaction Between Compilers and Computer Architectures*, Feb. 2004, Madrid, Spain.
- [62] Y. Wu, M. Breternitz, J. Quek, O. Etzion, and J. Fang. The accuracy of initial prediction in two-phase dynamic binary translators. In *Proceedings of the International Symposium on Code Generation and Optimization*, Mar. 2004, Palo Alto, CA.
- [63] Y. Wu, D.-Y. Chen, and J. Fang. Better exploration of region-level value locality with integrated computation reuse and value prediction. In *Proceedings 28th Annual International Symposium on Computer Architecture*, pages 98–108, Jun. 2001.
- [64] Y. Wu and Y. F. Lee. Accurate invalidation profiling for effective data speculation on epic processors. In *Proceedings of the 13th International Conference on Parallel and Distributed Computing Systems*, Aug. 2000. Vegas, NV.
- [65] Y. Wu and Y. F. Lee. Explore free execution slots for accurate and efficient profiling on epic processors. In *Proceedings of Asia-Pacific Computer Systems Architecture Conference*, Sep. 7-9 2004. Beijing, China.
- [66] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 1994.
- [67] Y. Wu, R. Rakvic, L.-L. Chen, C.-C. Miao, G. Chrysos, and J. Fang. Compiler managed micro-cache bypassing for high performance epic processors. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 2002.
- [68] C. Young, D. S. Johnson, M. D. Smith, and D. R. Karger. Near-optimal intraprocedural branch alignment. In *Proceedings of the ACM Conference on Programming Language, Design, and Implementation*, pages 183–193, Jun. 1997. Las Vegas, NV.
- [69] C. Young and M. D. Smith. Better global scheduling using path profiles. In *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture*, Nov. 1998.

- [70] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith. System support for automatic profiling and optimization. *Proceedings of the 16th ACM Symposium on Operating systems Principles*, 31(5), 1997.

Chapter 12

Compilation and Speculation

Jin Lin, Wei-Chung Hsu, and Pen-Chung Yew

*University of
Minnesota*

12.1	Introduction	301
12.2	Runtime Profiling	303
12.3	Speculative Analysis Framework	310
12.4	General Speculative Optimizations	314
12.5	Recovery Code Generation	321
12.6	Conclusion	329
	References	330

12.1 Introduction

The performance of application programs has been steadily improved by exploiting more instruction scheduling parallelism (ILP). Recently, Explicitly Parallel Instruction Computing (EPIC) architectures provide a number of novel architectural features that support software pipelining, predication and speculation to improve ILP even further. Those hardware features enable compilers to exploit more parallelization and optimization effectively.

Speculative execution, such as control speculation and data speculation, is an effective way to improve program performance. Control speculation refers to the execution of instructions *before* it has been determined that they would be executed in the normal flow of execution. Data speculation refers to the execution of instructions *on most likely correct* (but potentially incorrect) operand values.

Control speculation and data speculation have been used effectively in instruction scheduling. Consider the program in Table 12.1(a): *the frequency* (or the *probability*) of the execution paths can be collected using edge/path profiling at runtime and represented in the control flow graph. If the branch-taken path (i.e., the condition being true) has a high probability, the compiler can move the *load* instruction up and execute it speculatively (*ld.s*) before the branch instruction. A *check* instruction (*chk.s*) is inserted at its original (or home) location to catch and to recover from any invalid speculation. The *ld.s* and *chk.s* are IA64 instructions that support control speculation [8]. Since

the execution of the speculative load may overlap with the execution of other instructions, the critical path can be shortened along the speculated path. This example shows that instruction scheduling using control speculation can be used to hide memory latency.

TABLE 12.1: Using control and data speculation to hide memory latency

	ld.s r30 =[r31]	... *x = ld.a r30 = [r40]
(a) original program	if (cond) { ... = *p ... } ... next: recover: ld r30 = [r31] br next	if (cond) { chk.s r30, recover } = *y ... ld.c r30 = [r40]	st ld.c r30 = [r40]
(b) control speculative version			
(c) original program			
(d) data speculative version			

Similarly, data speculation can be also used to hide memory latency. In Table 12.1(c), we assume $*x$ is potentially aliased with $*y$. The compiler moves the load instruction up speculatively and it is executed using *ld.a* instruction in the *IA64* architecture. In the original (or home) location, a check load instruction *ld.c* is inserted so that it can recover by reloading the data when mis-speculation occurs.

In fact, there are speculation opportunities in other compiler optimizations, such as redundancy elimination. Here is one example in Figure 12.1(a). Assume that the leftmost flow path of an *if* statement is executed much more frequently than the rightmost path. The compiler can insert a speculative load in the form of *ld.s* on the rightmost path and a *chk.s* instruction to replace the redundant load of $*x$. Similarly, data speculation can be used for redundancy elimination. For example, elimination of redundant loads can sometimes be inhibited by an intervening aliasing store. Considering the program in Figure 12.1(c), the traditional redundancy elimination cannot remove the second load of $*x$ unless the compiler analysis proves that the expressions $*x$ and $*y$ do not access the same location. However, through profiling or simple heuristic rules, if we know that there is only a small probability that $*x$ and $*y$ will access the same memory location, the second load of $*x$ can be speculatively removed as shown in Figure 12.1(d). The first load $*x$ is replaced with a speculative load instruction (*ld.a*), and a *ld.c* instruction is added to replace the second load instruction. If the store of $*y$ does not access the same location

as the load of $*x$, the value in register $r42$ is used directly without re-loading $*x$.

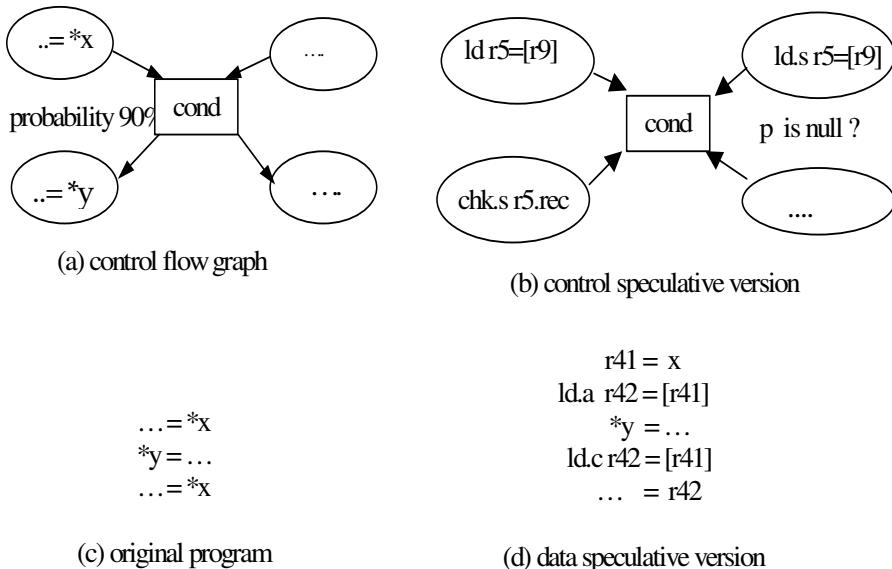


FIGURE 12.1: Redundancy elimination using data and control speculation.

12.2 Runtime Profiling

In order to build an effective framework for speculative analyses and optimizations, a comprehensive instrumentation-based profiling environment is required to generate a set of edge/path profile [20], alias profile [3] and dependence profile [4]. Such profiles augment compiler analyses with probabilistic information, and can be used to guide effective speculative optimizations.

Figure 12.2 illustrates a possible profiling process in a profile-based optimizing compiler environment. It usually includes two major components: an instrumentation tool and a set of library routines that supports profiling. Application programs are first instrumented with calls to the profiling library routines. Branch probability, alias and/or data dependence information are collected during the execution of the instrumented program with some train-

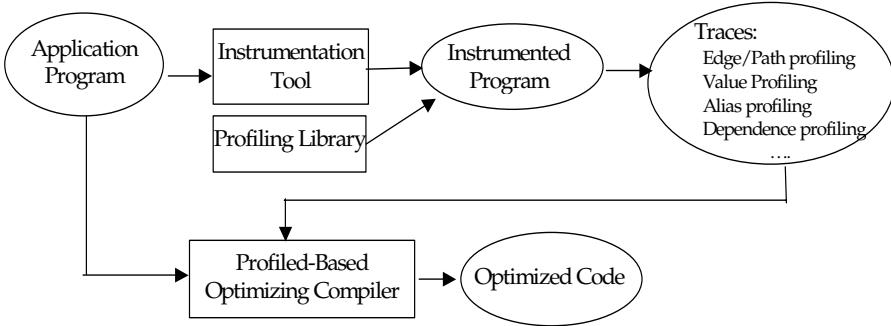


FIGURE 12.2: A typical instrumentation-based profiling process in a profile-based optimizing compiler.

ing inputs. For example, the alias profile can collect the dynamic points-to set of every memory operation. Such profile information is then fed back to the compiler for more advanced compiler optimizations.

12.2.1 Alias Profiling

In general, there are two types of points-to targets: the *local* and the *global* variables defined in programs, and the *heap* objects allocated at runtime. The local and global variables usually have explicit variable names given in a program. Unlike the named variables, the dynamically allocated heap locations are anonymous in the program [3]. In order to remedy such a situation, a variable name could be assigned according to the calling path at the invocation site of the *malloc()* function in addition to the line number. To control the complexity of such a naming scheme, the last n procedure calls of a calling path could be used in the naming scheme. Different n will thus give different levels of granularity to the named memory objects. Different granularities can affect the quality of the alias profiling results [3].

The selected naming scheme can be implemented by setting up a mapping at runtime from the address of a memory object to its name according to the naming scheme. At runtime, the address of an indirect memory reference (such as a pointer) is known. Targets of a pointer are identified by looking up the mapping with the address of the reference to its name. The target set thus obtained represents approximately the best result that a naming scheme can be expected to achieve [3].

However, there are some limitations to the alias profiling using such naming schemes. For example, it will require a lot of memory space to distinguish field accesses in structures and individual elements in arrays. Hence, such detailed profiling may not be always feasible. Furthermore, the naming scheme used for dynamically allocated objects could also make a difference if a program uses

its own memory allocator/manager (e.g., *gap* and *perlrbmk* in the SPECint2000 benchmarks do not use *malloc* directly). Thus, it often needs more accurate profiling information such as those obtained from data dependence profiling [4] to avoid the constraints imposed by an imprecise, but more efficient, alias profile implementation. The dependence profiling often uses an address-based approach.

Unlike the alias profiling which provides the dynamic points-to information, the data dependence profiling provides a pair-wise relation among the memory references. Such information includes the data dependence type, the data dependence distance, and the frequency (i.e., the probability) that the data dependence may occur between the pair of memory references. Although the data dependence profiling can provide much more accurate information, such profiling requires a lot of memory space and is very time consuming because every memory reference needs to be monitored and compared pair-wise. It could slow down the program execution by orders of magnitude during the profiling phase unless schemes such as sampling are used [4].

12.2.2 Data Dependence Profiling

In this section, the issues related to data dependence profiling that include detection of data dependences, handling of function calls and nested loops, and the definition of the probability for a data dependence edge are discussed.

12.2.2.1 Data Dependences Using Shadow Variables

In the data dependence profiling, the instrumented profiling instructions collect the *address value* and the *reference ID* of each instance of memory references at runtime. The reference ID is used to identify each static reference. It is assigned during the instrumentation so that profiling results could be mapped back to the compiler.

A data dependence occurs when two memory references access the *same memory location* in the data space. The occurrence of data dependence can be represented as a *dependence edge* from the *source* to the *sink*. There are four data dependence types: *flow dependence* (or true dependence), *anti-dependence*, *output dependence*, and *input dependence* (or reference dependence).

It is very expensive to check data dependence based on pair-wise address comparison, especially when the program has a large number of memory references. The number of dynamic instances of all memory references could be very large when there are nested loops in the program. A special data structure, called *shadow memory*, is used to efficiently detect data dependences (see Figure 12.3). This is a typical tradeoff between *time* and *space*.

Shadow is used to store memory access information during profiling. A simple hash function using the *address value* maps each memory reference in the data space to its corresponding shadow entry in the shadow memory

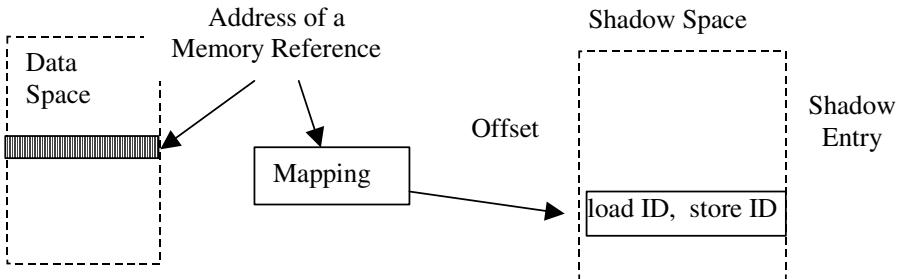


FIGURE 12.3: Detect data dependence using shadow.

(illustrated in Figure 12.3). The shadow memory is allocated on demand, and can be freed when the profiling of a particular region such as a particular nested loop or a procedure of interests has completed.

To detect data dependences during the profiling process, first each memory reference locates its shadow entry using the hash function on the address value. The shadow entry contains the reference ID of the *latest load* and the *latest store* operations to this memory location, or is *empty* when this location is accessed for the first time. If the memory reference is a *load* operation, there could be a *flow* dependence edge from the latest store to this load operation, and also an *input* dependence edge from the latest load to this load operation. If the memory reference is a *store* operation, there could be an *anti* dependence edge from the latest load to this store operation, and also an *output* dependence edge from the latest store to this store operation. Finally, the reference ID of the current memory reference is stored into the shadow entry, and the reference ID of the previous load or store in the shadow entry is overwritten.

Using this scheme, data dependences can be quickly detected without expensive pair-wise address comparisons among all memory references. Using shadow space is equivalent to a software implementation of the associative memory [3]. The number of pair-wise comparisons could also be reduced by focusing only on the execution path of a program, i.e., only the latest load and the latest store are compared. A separate linked list could be used for each type of data dependence edges to record the dependence edges.

12.2.2.2 Procedure Calls

When procedure calls occur in a profiled program, a dependence edge detected between two memory references in two different procedures needs to be mapped into the common procedure that contains both procedures. It is because, to be useful to a compiler, a data dependence graph of a procedure should contain only the dependences among the memory references and procedure calls within the same invocation of the procedure. However, profiling

information at runtime may contain a lot of dependences between different invocations of procedures at different call sites. We need to sort through these data dependences and extract relevant data dependence information. It requires the calling path information of all memory references to avoid creating *false* dependence edges.

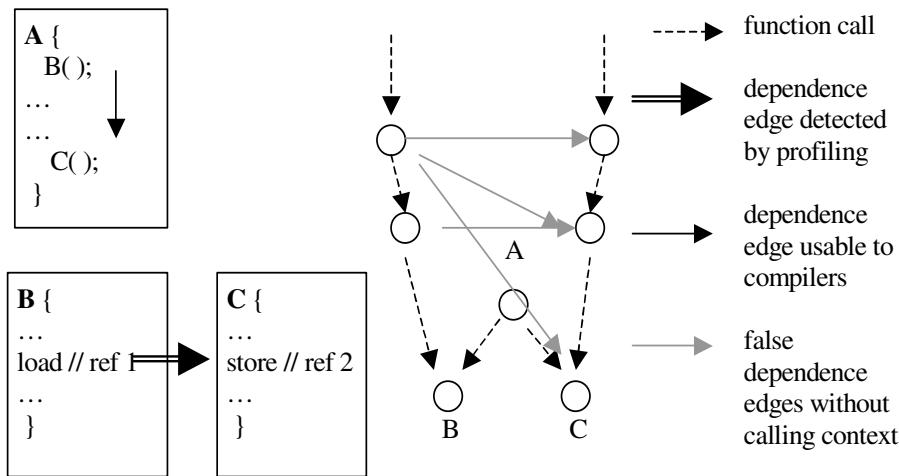


FIGURE 12.4: False dependences may be introduced by function calls.

In Figure 12.4, for example, a flow dependence from reference 2 to reference 1 occurs only when procedure *B* and *C* are called within procedure *A*. Therefore, only one dependence edge from the corresponding call site of *B* to the call site of *C* in the procedure *A* should be generated. If we do not have calling path information for each memory reference, we have to assume that there is a dependence edge from any call site that may call *B* to any call site that may call *C*. Here, we include all the procedures calling *B* or *C* directly or indirectly. They are shown as *false* dependence edges in Figure 12.4. The example illustrates the need for calling path information in data dependence profiling.

To reduce the number of false dependence edges caused by procedure calls, the calling context for each memory reference is recorded in its shadow entry. When a dependence edge is detected, their calling contexts help to locate the common procedure. Different kinds of calling context can be used in the dependence profiling, resulting in different precision and efficiency.

- Fully extended call paths. Whenever a procedure is called, a new call node is assigned for this invocation. The call path made up with such call nodes is able to distinguish both the different call sites of a procedure

and the different invocations of a call site. It is easy to maintain the fully extended calling path at runtime: push a new node to the call stack when a procedure is invoked, and pop the top node from the call stack when a procedure call returns. The nodes in the call stack can be linked backwardly and only the top node in the shadow entry is kept. When a dependence edge occurs, the common part in their calling context can be identified. A dependence edge should be added in the farthest common procedure from the main procedure between the corresponding references or call sites.

- Compacted call paths. The space requirement of the fully expanded call paths could be very high because a call site may appear in a nested loop or in a recursive call. The size of a call path could be reduced by using only one node to represent multiple invocations of a call site in a nested loop or in recursive calls to reduce the space requirement. Other compression techniques can also be used. However, compacted call paths may result in some false dependence edges.

For SPEC CPU2000 benchmarks, even with the train input set, it is still too costly to fully extend all call paths.

12.2.2.3 Loops

When a dependence edge occurs in a loop nest, a *dependence distance* tells how far the sink is away from the source of a dependence edge in terms of the number of iterations. Optimizations may treat dependences with different distances differently. It is important for a profiling tool to be able to generate distance vectors.

In order to generate a distance vector for a dependence edge, each loop is assigned an iteration counter. This counter is incremented when the beginning of the loop body is encountered. The values in the iteration counters of the loops nested outside of the current reference form an iteration vector. The iteration vector could be stored in the shadow. When a dependence edge is detected, it needs to determine how the two references nested in loops, with the consideration of the calling context discussed in the previous section. The loop nesting information is not recorded in the shadow. The calling context information in the shadow is used to help identify the commonly nested loops of the two references. The iteration vectors of the two references are aligned and the iteration counters of their commonly nested loops are identified. The distance vector is computed by subtracting the iteration vector in the shadow from the current iteration vector. Dependences together with their distance vectors can then be recorded. Depending on the compiler optimizations that use the profile, limited distances or the distance directions, e.g., $=$, $<$ or $>$, can be recorded instead for better efficiency.

12.2.2.4 Dependence Probability

A static dependence analysis in a compiler usually could only tell whether an edge between two memory references exists or not. However, with the dependence profiling, we can further observe whether the data dependence *rarely*, *frequently*, or *always* occurs between the two memory references. Such additional information allows a compiler optimization to deal with a data dependence edge accordingly for a better performance. In general, a compiler can speculate only on those dependences that *rarely* occur in order to avoid costly mis-speculations. Hence, such information is especially important to speculative optimizations in a compiler.

The *dependence probability* depicts such dynamic behavior of a dependence. There are several ways to define the *dependence probability*. It depends on how such information is to be used by later speculative optimizations. Here, only two possible definitions are used: the *reference-based probability* and the *iteration-based probability*.

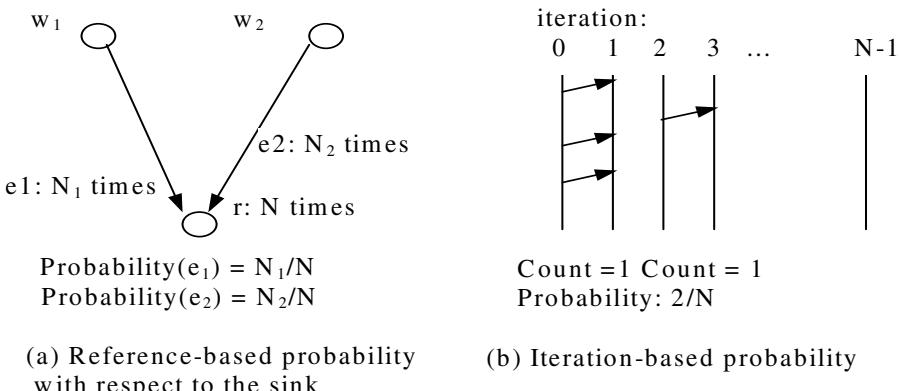


FIGURE 12.5: Different definitions of dependence probability for dependence profiling.

The *reference-based probability* is defined as the number of the occurrences of a *dependence edge* over the total number of the occurrence of the *reference*. The *reference* may be either the *source* or the *sink* of the *dependence edge*. This information gives an indication of how often a dependence edge occurs when either the source or the sink of the dependence is referenced during the program execution. In Figure 12.5(a), an example of the reference-based probability for the sink reference *r* is illustrated. Assume *e*₁ and *e*₂ are two flow dependent edges to the sink reference *r*. The reference *r* is accessed *N* times while the edges *e*₁ and *e*₂ occur *N*₁ and *N*₂ times respectively. The probability of the value coming from reference *w*₁ is defined as $\frac{N_1}{N}$ and from

reference w_2 as $\frac{N_2}{N}$. Such probability indicates which definition is more likely to provide the value to the reference r .

The *iteration-based probability* is defined as *the number of iterations* in which the dependence edge occurs over the total number of iterations of the loop nest, as shown in Figure 12.5(b). Again, the *iteration-based probability* may be *sink-based* or *source-based*. This information is often related to the dependence distance and the control flow within the loop.

12.3 Speculative Analysis Framework

Alias profiling has led to the development of many speculative analyses and optimization techniques. Figure 12.6 shows a framework for speculative analyses and optimizations. It is built on a Static Single Assignment (SSA) framework [7]. While this general framework could include many commonly used optimizations in compilers, *partial redundancy elimination (PRE)* [6, 18] is presented in this section as an example to show the potential use of the framework. Many optimizations, such as redundancy elimination, strength reduction, and register promotion, can be modeled and resolved as a partial redundancy elimination (PRE) problem.

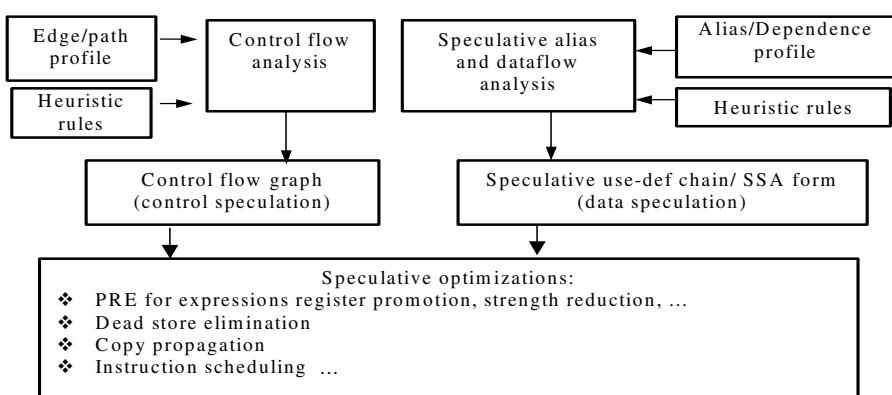


FIGURE 12.6: A framework of speculative analyses and optimizations [16].

12.3.1 Speculative Alias and Dataflow Analysis

The extended speculative SSA form is based on the Hashed Static Single Assignment (HSSA) form proposed by Chow et al. [5]. The traditional SSA form [7] only provides *use-def factored chain* for the scalar variables. In order to accommodate pointers, Chow et al. proposed the HSSA form which integrates the alias information directly into the intermediate representation using explicit *may modify operator* (χ) and *may reference operator* (μ). In the HSSA form, *virtual variables* are first created to represent indirect memory references (e.g., pointers). The rule that governs the assignment of virtual variables is that all indirect memory references that have similar alias behavior in the program are assigned a unique virtual variable (see Section 12.3.2). Thus, an alias relation could only exist between *real variables* (i.e., original program variables) and *virtual variables*. In order to characterize the effect of such alias relations, the χ assignment operator and the μ assignment operator are introduced to model the *may modify* and the *may reference* relations, respectively.

To facilitate data speculative optimizations, the notion of *likeliness* is further introduced to such alias relations by attaching a speculation flag to the χ and μ assignment operators according to the following rules:

DEFINITION 12.1 *Speculative update χ_s :* A speculation flag is attached to a χ assignment operator if the χ assignment operator is highly likely to be substantiated at runtime. It indicates that this update is highly likely and can't be ignored.

DEFINITION 12.2 *Speculative use μ_s :* A speculation flag is attached to a μ assignment operator if the operator is highly likely to be substantiated at runtime. It indicates that the variable in the μ assignment operator is highly likely to be referenced during the program execution.

The compiler can use the profiling information and/or heuristic rules to specify the *degree of likeliness* for an alias relation. For example, the compiler can regard an alias relation as *highly likely* if it exists during alias profiling, and attach speculation flags to the χ and μ assignment operators accordingly. These speculation flags can help to expose opportunities for data speculation.

Example 12.2 in Table 12.2 shows how to build a use-def chain speculatively by taking such information into consideration. In this example, v is a virtual variable that represents $*q$, and the numerical subscript of each variable indicates the version number of the variable. Assume variables x and y are potential aliases of $*q$. The fact that the variables x and y could be potentially updated by the $*q$ store reference in $s1$ is represented by the χ operations on x and y after the store statement.

Let us further assume that, according to the profiling information, the indirect memory reference $*q$ is *highly likely* to be an alias of the variable y , but not of the variable x , at runtime. Hence, a speculation flag is attached to $\chi(y_1)$ in $s3$, denoted as $\chi_s(y_1)$, because the potential update to y caused by the store $*q$ is also *highly likely*. Similarly, $*q$ in $s8$ will also be *highly likely* to reference y , and a speculation flag is attached to $\mu(y_2)$ in $s7$, denoted as $\mu_s(y_2)$ in $s7$.

TABLE 12.2: Example 12.2

s0: $x_1 = \dots$	s0: $x_1 = \dots$
s1: $*q_1 = 4$	s1: $*q_1 = 4$
s2: $x_2 \leftarrow \chi(x_1)$	s2: $x_2 \leftarrow \chi(x_1)$
s3: $y_2 \leftarrow \chi(y_1)$	s3: $y_2 \leftarrow \chi_s(y_1)$
s4: $v_2 \leftarrow \chi(v_1)$	s4: $v_2 \leftarrow \chi(v_1)$
s5: $\dots = x_2$	s5: $\dots = x_2$
s6: $x_3 = 4$	s6: $x_3 = 4$
s7: $\dots \mu(x_3), \mu(y_2), \mu(v_2)$	s7: $\dots \mu(x_3), \mu_s(y_2), \mu(v_2)$
s8: $\dots = *q_1$	s8: $\dots = *q_1$

The advantage of having such likeliness information is that those updates that do not carry the speculation flag, such as the update to x in $s2$, could be speculatively ignored and be considered as *speculative weak updates*. When the update to x in $s2$ is ignored, the reference of x_2 in $s5$ becomes *highly likely* to use the value defined by x_1 in $s0$. Similarly, because $*q$ is *highly likely* to reference y in $s8$ (from $\mu_s(y_2)$ in $s7$), the use of x_3 and v_3 in $s7$ can be ignored. The definition of $*q$ in $s1$ thus becomes *highly likely* to reach the use of $*q$ in $s8$.

From this example, it shows that the speculative SSA form could be made to contain both traditional compiler analysis information and speculation information. The compiler can use the speculation flags to conduct speculative optimizations.

12.3.2 Framework for Speculative Alias and Dataflow Analysis

Figure 12.7 shows a basic framework of the alias analysis and the dataflow analysis with the proposed extension to incorporate speculation flags to the χ and μ assignment operators using profiling information and/or compiler heuristic rules.

In this framework, the *equivalence class* based alias analysis proposed by Steensgard [19] could be used to generate the *alias equivalence classes* for the memory references within a procedure. Each alias class represents a set of *real*

- ◆ Equivalence class based alias analysis
- ◆ Create χ and μ list
 - ❖ Generate the χ_s and μ_s list based on alias profile
 - ❖ In the absence of alias profile, generate the χ_s and μ_s list based on heuristic rules
- ◆ Construct speculative SSA form
- ◆ Flow sensitive pointer alias analysis

FIGURE 12.7: A framework for speculative alias and dataflow analysis.

program variables. Next, a unique *virtual variable* is assigned to each alias class. The initial μ list and χ list are also created for the indirect memory references and the procedure call statements.

The rules of the construction of μ and χ lists are as follows: (1) For an indirect memory *store* reference or an indirect memory *load* reference, its corresponding μ list or χ list is initialized with all the variables in its alias class and its virtual variable. (2) For a procedure call statement, the μ list and the χ list represent the *ref* and *mod* information of the procedure call, respectively.

Using alias profiling information and/or heuristic rules, the χ_s and μ_s lists can be constructed. In the next step, all program variables and virtual variables are renamed according to the standard SSA algorithm [7]. Finally, a flow sensitive pointer analysis using factored use-def chain [7] is performed to refine the μ_s list and the χ_s list. The SSA form is also updated if the μ_s and χ_s lists have any change.

In the following sections, a more detailed description on how to construct speculative SSA form using alias profile or compiler heuristic rules is presented.

12.3.2.1 Speculative SSA Using Alias Profile

The concept of *abstract memory locations* (LOCs) [9] can be used to represent the points-to targets in the alias profile. LOCs are storage locations that include local variables, global variables, and heap objects. Since heap objects are allocated at runtime, they do not have explicit variable names in the programs. Before alias profiling, the heap objects need to be assigned a unique name according to a naming scheme. Different naming schemes may assume different storage granularities [3].

For each indirect memory reference, there is a LOC set to represent the collection of memory locations accessed by the indirect memory reference at runtime. In addition, there are two LOC sets at each procedure call site to represent the side effect information that includes *modified* and *referenced* locations, respectively.

The rules of assigning a speculation flag for χ and μ list are as follows:

DEFINITION 12.3 χ_s : Given an indirect memory store reference and its profiled LOC set, if any of the members in its profiled LOC set is not in its χ list, add the member to the χ list using the speculation update χ_s . If the member is in its χ list, then a speculation flag is attached to its χ operator (thus becoming a speculative update χ_s).

DEFINITION 12.4 μ_s : Given an indirect memory load reference and its profiled LOC set, if any of the members in its profiled LOC set is not in its μ list, add the member to the μ list using the speculative use μ_s . If the member is in its μ list, then a speculation flag is attached to its μ operator (thus becoming a speculative use μ_s).

12.3.2.2 Speculative SSA Using Heuristic Rules

In the absence of alias profile, compiler can also use heuristic rules to assign the speculation flags. The heuristic rules discussed here are based on the pattern matching of syntax tree. Here are three possible heuristic rules that could be used in this approach:

1. If two *indirect* memory references have an identical address expression, they are assumed to be *highly likely* to hold the same value.
2. Two *direct* memory references of the same variable are assumed *highly likely* to hold the same value.
3. Since it is quite difficult to perform speculative optimization across procedure calls, the side effects of procedure calls obtained from compiler analysis are all assumed *highly likely*. Hence, all definitions in the procedure call are changed into χ_s . The μ list of the procedure call remains unchanged.

The above three heuristic rules imply that all updates caused by statements *other than* call statements between two memory references with the same syntax tree can be speculatively ignored. Using a trace analysis on SPEC2000 integer benchmark, it has been found that these three heuristic rules are quite satisfactory with surprisingly few mis-speculations.

12.4 General Speculative Optimizations

In this section, partial redundancy elimination (PRE) [18] is used as an example to show how to perform speculative optimizations in a speculative analysis framework. PRE [18] is chosen because it includes a set of optimizations that are important in most compilers. The set of optimizations in PRE

include: *partial redundancy elimination for expressions, register promotion, strength reduction, and linear function test replacement*. A quick overview on the PRE algorithm is given first. Then, an extension to incorporate both data and control speculation is presented in more details.

12.4.1 Overview

Most of the work in PRE is focused on inserting additional computations in the *least likely* execution paths. These additional computations cause *partial* redundant computations in *most likely* execution paths to become *fully* redundant. By eliminating such *fully* redundant computations, the overall performance can be improved.

All expressions are assumed to be represented as trees with leaves being either constants or SSA renamed variables. For indirect loads, the indirect variables have to be in SSA form. Using an extended HSSA form presented in [5], it can uniformly handle indirect loads together with other variables in the program.

PRE is performed one expression at a time, so it suffices to describe the algorithm with respect to a given expression. In addition, the operations in an expression tree are processed in a bottom-up order. The algorithm consists of six separate steps [18]. The first two steps, *Φ -Insertion* and *Rename*, construct an expression SSA form using a temporary variable h to represent the *value* of an *expression*. In the next two steps, *Downsafety* and *WillBeAvailable*, an appropriate set of merge points for h is selected that allow computations to be inserted. In the fifth step, *Finalize*, additional computations are inserted in the *least likely* paths, and redundant computations are marked after such additional computations are inserted. The last step, *CodeMotion*, transforms the code and updates the SSA form in the program.

Control speculation is suppressed in order to ensure the safety of code placement. Control speculation is realized by inserting computations at the incoming paths of a control merge point Φ whose value is not *downsafe* (e.g., its value is not used before it is killed) [20]. The symbol Φ is used to distinguish the merge point in the *expression* SSA form that is different from the merge point ϕ in the *original* SSA form. Since control speculation may or may not be beneficial to overall program performance, depending on which execution paths are taken frequently, the edge profile of the program can be used to select the appropriate merge points for insertion. The changes to support control speculation lie in the *Downsafety* step.

The *Rename* step plays an important role in facilitating the identification of redundant computations in the later steps. In the original PRE without data speculation as shown in the example in Figure 12.8(a), the two occurrences of an *expression* x have the same value. Hence, the temporary variable h that represents the *expression* x is assigned the *same* version number for both references. Since they have the *same* value, the second occurrence is

redundant to the first. Thus, the second load can be replaced with a register access. “ $h_1 \leftarrow$ ” in Figure 12.8(a) means a value is to be *stored* into h_1 .

However, if there is a store of $*q$ that may modify the value of the expression x , the second occurrence of x is not redundant and should be assigned a different version number, as shown in Figure 12.8(b). In Figure 12.8(b), the traditional alias analysis will report that the assignment to $*q$ *may kill* the value of the first occurrence of x .

Now, as in Figure 12.8(c), if the speculative SSA form indicates that the alias relation between the expression of x and $*q$ is *not likely*, it can be speculatively assumed that the potential update to x due to the alias relationship to $*q$ can be ignored. The second occurrence of x is regarded as *speculatively redundant* to the first one, and a check instruction is inserted to check whether the value of x is changed before the second occurrence of x . (This can be done, for example, by inserting a *ld.c* instruction on the IA64 architecture [8].) The register that contains the value in the first occurrence can be used in the second occurrence, instead of reloading it. By *speculatively* ignoring those updates, *speculative redundancy* can be exposed between those two occurrences of the expression x .

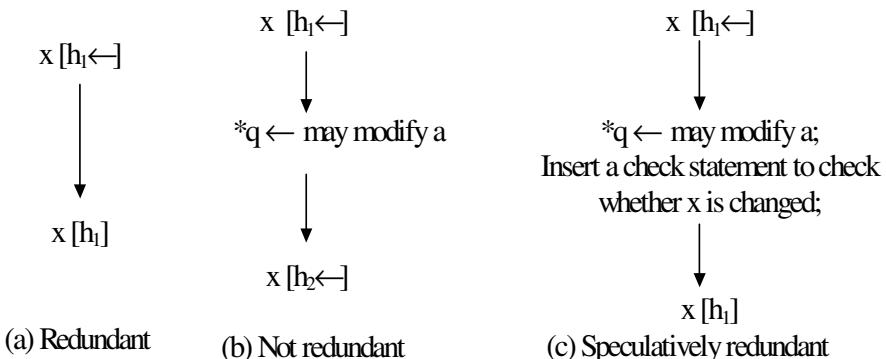


FIGURE 12.8: Types of occurrence relationships (h is temporary variable for a).

Thus, the SSA form built for the variable x by the Φ -Insertion and Rename steps can exhibit more opportunities for redundancy elimination if it is enhanced to allow data speculation. The generation of check statements is performed in *CodeMotion*. The *CodeMotion* step also generates the speculative load flags for those occurrences whose value can reach the check statements along the control flow paths.

12.4.2 Φ -Insertion Step

One purpose of inserting Φ 's for the temporary variable h of an expression is to capture all possible insertion points for the expression. Inserting too few Φ 's will miss some PRE opportunities. On the other hand, inserting too many Φ 's will have an unnecessarily large SSA graph to be dealt with.

As described in [18], Φ 's are inserted according to two criteria. First, Φ 's are inserted at the *Iterated Dominance Frontiers* (DF+) of each occurrence of an expression. Secondly, a Φ can be inserted where there is a ϕ for a *variable* contained in the *expression*, because it indicates a change of value for the expression that reaches the merge point. This type of Φ insertion is performed in a demand-driven way. An expression at a certain merge point is defined to be *not anticipated* if the value of the expression is never used before it is killed, or reaches an exit. A Φ is inserted at a merge point only if its expression is *partially anticipated* [15], i.e., the value of the expression is used along one control flow path before it is killed.

For a *not-anticipated* expression at a merge point, its killing definition can be recognized as a *speculative weak update*. The expression can become *partially anticipated speculatively*. Thus its merge point could potentially be a candidate for inserting computations to allow a *partial redundancy* to become a *speculative full redundancy*.

TABLE 12.3: Enhanced Φ insertion allows data speculation.

s0: $\cdots = x_1$	s0: $\cdots = x_1[h]$	s0: $\cdots = x_1[h]$
s1: $if(\dots)\{$	s1: $if(\dots)\{$	s1: $if(\dots)\{$
s2: $*q_1 = \dots$	s2: $*q_1 = \dots$	s2: $*q_1 = \dots$
s3: $x_2 \leftarrow \chi_s(x_1)$	s3: $x_2 \leftarrow \chi_s(x_1)$	s3: $x_2 \leftarrow \chi_s(x_1)$
s4: $y_2 \leftarrow \chi_s(y_1)$	s4: $y_2 \leftarrow \chi_s(y_1)$	s4: $y_2 \leftarrow \chi_s(y_1)$
s5: $v_2 \leftarrow \chi_s(v_1)$	s5: $v_2 \leftarrow \chi_s(v_1)$	s5: $v_2 \leftarrow \chi_s(v_1)$
}	}	}
s6: $x_3 \leftarrow \phi(x_1, x_2)$	s6: $x_3 \leftarrow \phi(x_1, x_2)$	s6: $h \leftarrow \phi(h, h)$
s7: $y_3 \leftarrow \phi(y_1, y_2)$	s7: $y_3 \leftarrow \phi(y_1, y_2)$	s7: $x_3 \leftarrow \phi(x_1, x_2)$
s8: $v_3 \leftarrow \phi(v_1, v_2)$	s8: $v_3 \leftarrow \phi(v_1, v_2)$	s8: $y_3 \leftarrow \phi(y_1, y_2)$
s9: $*q_1 = \dots$	s9: $*q_1 = \dots$	s9: $v_3 \leftarrow \phi(v_1, v_2)$
s10: $x_4 \leftarrow \chi(x_3)$	s10: $x_4 \leftarrow \chi(x_3)$	s10: $*q_1 = \dots$
s11: $y_4 \leftarrow \chi_s(y_3)$	s11: $y_4 \leftarrow \chi_s(y_3)$	s11: $x_4 \leftarrow \chi(x_3)$
s12: $v_4 \leftarrow \chi_s(v_3)$	s12: $v_4 \leftarrow \chi_s(v_3)$	s12: $y_4 \leftarrow \chi_s(y_3)$
s13: $\cdots = x_4$	s13: $\cdots = x_4[h]$	s13: $v_4 \leftarrow \chi_s(v_3)$
s14:	s14:	s14: $\cdots = x_4[h]$
(a) original program	(b) after traditional ϕ insertion	(c) after enhanced ϕ insertion

Table 12.3 gives an example of this situation. In this example, x and y are *may* alias to $*q$. However, y is *highly likely* to be an alias of $*q$, but x

is *not likely* to be an alias of $*q$. Hence, without any data speculation in Table 12.3(a), the value of x_3 in $s6$ cannot reach x_4 in $s13$ because of the potential $*q$ update in $s9$, i.e., x_3 is *not anticipated* at the merge point in $s6$. Hence, the merge point in $s6$ is no longer considered as a candidate to insert computations along the incoming paths as shown in Table 12.3(b).

Since x is *not likely* to be an alias of $*q$, the update of x_4 in $s10$ can be *speculatively ignored*, and the expression x_3 can now reach x_4 in $s13$. Hence, x_3 in $s6$ becomes speculatively *anticipated*, and a Φ can be inserted for temporary variable h as shown in Table 12.3(c).

12.4.3 Rename Step

In the Φ -insertion step, it inserts more Φ 's at the presence of *may-alias* stores creating more opportunities for inserting more computations. In contrast, the Rename step assigns more occurrences of an expression to the *same* version of temporary variable h and allows more redundancies to be identified. The enhancement to the Rename step is to deal with *speculative weak updates* and *speculative uses*.

Like traditional renaming algorithms, the renaming step keeps track of the current version of the expression by maintaining rename stack while conducting a pre-order traversal of the dominator tree of the program. Upon encountering a new expression occurrence q , its use-def chain is traced to determine whether the value of the expression p on top of the rename stack can reach this new occurrence. If so, q is assigned the same version number as that of the expression p . Otherwise, q is checked to see whether it is speculative redundant to p by ignoring the *speculative weak update* and continuing tracing upward along the use-def chain. If it eventually reaches the expression p , q is speculatively assigned the same version number as given by the top of the rename stack, and q is then annotated with a speculation flag in order to enforce the generation of a check instruction for expression q later in the code motion step. If the value of p cannot reach q , the process is stopped and q is assigned a new version number. Finally, q is pushed onto the rename stack and the process continues.

TABLE 12.4: Enhanced renaming allows data speculation

$\dots = x_1[h_1]$	$\dots = x_1[h_1]$
$*q_1 = \dots$	$*q_1 = \dots$
$v_2 \leftarrow \chi(v_1), x_2 \leftarrow \chi(x_1),$	$v_4 \leftarrow \chi(v_3), x_2 \leftarrow \chi(x_1),$
$y_2 \leftarrow \chi(y_1)$	$y_2 \leftarrow \chi_s(y_1)$
$\dots = x_2[h_2]$	$\dots = x_2[h_1 < \text{speculation} >]$
(a) traditional renaming	(b) speculative renaming

Table 12.4 gives an example that shows the effect of the enhanced renaming. In this example, there are two occurrences of an expression x that are represented by the temporary variable h . The alias analysis shows that expression $*q$ and x *may* be aliases. Variable x may be updated after the store of $*q$, and is represented by the χ operation in the SSA form. These two occurrences of x are assigned with different version numbers in the original Rename step. However, if q does not point to x (either by alias profile and/or heuristic rules), the χ operation with x is *not* marked with χ_s , so this update can be ignored in the Rename step. In Table 12.4(b), the second occurrence of x is *speculatively* assigned with the *same* version number as the first one. In order to generate the check instruction in the CodeMotion step, the second occurrence of x is annotated with a *speculation flag*. Hence, the algorithm successfully recognizes that the first and the second real occurrences of x are of the *same* version by ignoring the *speculative weak update* caused by the indirect reference $*q$.

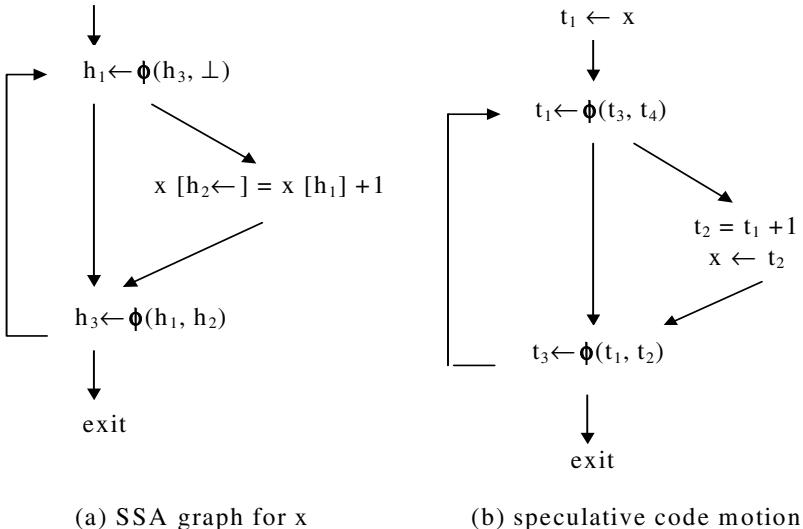
12.4.4 Downsafety Step

It is not allowed to insert any computation at a point in the program where the computation is not *downsafe*. This is necessary to ensure the safety of the code placement. Speculation corresponds to inserting computation at Φ 's where the computation is not down-safe. It can be achieved by selectively marking non-down-safe Φ 's *downsafe* in *Downsafety* step. In the extreme case, all Φ are marked as *downsafe*, which is referred to as full speculation. Since control speculation is supported by the hardware in IA64, both direct loads and indirect loads can be speculated.

12.4.4.1 Conservative Speculation

The conservative speculation strategy is used when profile data are not available. In this case, speculative code motion is restricted to moving loop-invariant computations out of single-entry loops.

The analysis is based on the ϕ located at the start of the loop body. Speculative insertion is performed at the loop header only if no other insertion inside the loop is needed to make the computation fully available at the start of the loop body. For the example in Figure 12.9(a), the Φ corresponding to $h1$ is marked as not *downsafe* by traditional *Downsafety* algorithm. Under the new criteria, the Φ is marked as *downsafe*. Figure 12.9(b) shows the subsequent code motion moves the load out of the loop speculatively. Another approach proposed by Bodik et al. [2] used control flow restructuring technique to enable control speculation in PRE. The reader can refer to [2] for more details.

**FIGURE 12.9:** Speculative code motion.

12.4.4.2 Profile-Driven Speculation

Without the knowledge of execution frequency, control speculation may potentially hurt performance. However, when an execution profile is available, it is possible that the run-time performance could be maximized by control speculation using the given profile. Lo et al. [15] proposed a profile-driven speculation algorithm that compares the performance of a program with and without speculation using the basic block frequency provided by the execution profile. In this approach, the granularity for deciding whether to speculate or not is based on each connected component of the SSA graph. For more details, please refer to Lo et al. [15].

12.4.5 CodeMotion Step

The CodeMotion step introduces a new temporary variable t , which is used to realize the generation of *assignment statements* and *uses* of temporary variable h [18]. With data speculation, this step is also responsible for generating *speculative check statements*.

The speculative check statements can only occur at places where the occurrences of an expression are *partially anticipated speculatively*. At the same time, multiple speculative check statements to the same temporary variable should be combined into as few check statements as possible.

The speculative check statements are generated in the main pass of Code-Motion. Starting from an occurrence x with a speculation flag in a use-def chain (shown as “ $x_2[h1 < \text{speculation flag} >]$ ” in Table 12.5(a)), the first spec-

ulatively weak update (i.e., “ $x_2 \leftarrow \chi(x_1)$ ” in Table 12.5(a)) can be reached. A speculative check statement is generated if it has not been generated yet. In a real implementation, an *advance load check flag* is actually attached to the statement first as shown in Table 12.5(b), and the real speculative check instruction, i.e., *ld.c*, is generated later in the code generation phase.

The occurrences of the temporary variable h that are marked with “ \leftarrow ” are annotated with an *advanced load flag* (as shown in Table 12.5(b)) if the value of those occurrences can reach their speculative check statements. An actual *ld.a* instruction will then be generated in the later code generation phase.

TABLE 12.5: An example of speculative load and check generation

$\cdots = x_1[h_1]$ $*q_1 = \dots$ $v_4 \leftarrow \chi(v_3)$ $x_2 \leftarrow \chi(x_1)$ $y_4 \leftarrow \chi_s(y_3)$ $\cdots = x_2[h_1 <\text{speculation flag}>]$	$t_1 = x_1$ (advance load flag) $\cdots = t_1$ $*q_1 = \dots$ $v_4 \leftarrow \chi(v_3)$ $x_2 \leftarrow \chi(x_1)$ $y_4 \leftarrow \chi_s(y_3)$ $t_4 = x_2$ (advance load check flag) $\cdots = t_4$
(a) Before Code Motion	(b) Final Output

12.5 Recovery Code Generation

12.5.1 Recovery Code Generation for General Speculative Optimizations

In order to support general speculative optimizations, a framework that uses an *if-block* structure [14] is used to facilitate check instructions and recovery code generation. It allows speculative and recovery code generated early in the compiler phases to be integrated effectively into the subsequent compiler optimization phases. It also allows *multi-level speculation* for multi-level pointers and multi-level expression trees to be handled with no additional complexity.

12.5.1.1 Check Instructions and Recovery Code

The semantics of the *check* instruction and its corresponding *recovery block* can be *explicitly* represented as a conditional *if-block* as shown in Table 12.6(c).

The *if-block* checks whether the speculation is successful or not; if not, the recovery block is executed. Initially, only the *original load* instruction associated with the *speculative load* is included in the recovery block, shown as s_5 in Table 12.6(c). A Φ -function is inserted after the *if-block* for r . Since mis-speculation needs to be *rare* in order to make speculative execution worthwhile, the *if-condition* is set to be *highly unlikely to happen*. Based on such an *explicit* representation, later analyses and optimizations could treat the recovery code as any other *highly biased if-block*. There will be no need to distinguish *speculative* code from *non-speculative* code during the analyses and the optimizations, as will be discussed in Section 12.5.3.

TABLE 12.6: Different representations used to model check instructions and recovery code

(a) source	(b) check instruction is modeled as an assignment program	(c) check instruction is modeled as explicit control flow structure statement
$s_1: \dots = *x$ $s_2: *y = \dots$ (may update $*x$) $s_3: \dots = *x + 10$	$s_1: r_1 = *x / \text{ld.a flag}/$ $s_1': \dots = r_1$ $s_2: *y = \dots$ $s_3: r_2 = *x$ $s_3: \dots = *x + 10$ <i>/*check flag*/</i>	$s_1: r_1 = *x / \text{ld.a flag}/$ $s_1': \dots = r_1$ $s_2: *y = \dots$ $s_4: \text{if}(r_1 \text{ is invalid})\{$ <i>/*check flag*/</i> $s_5: r_2 = *x$ $s_6: \}$ $s_7: r_3 \leftarrow \phi(r_1, r_2)$ $s_3: \dots = *x + 10$

12.5.2 Check Instructions and Recovery Code Representation for Multi-Level Speculation

Multi-level speculation refers to the data speculation that occurs in a multi-level expression tree as in the Table 12.7(a) and Table 12.7(d). In Table 12.7(d), the expression tree $*x + *y$ in s_3 could be speculatively redundant to $*x + *y$ in s_1 , if $*a$ in s_2 is a potential weak update to $*p$ and/or $*q$. In this case, two speculative loads and two check instructions for $*x$ and $*y$ need to be generated as shown in Table 12.7(e). Each check instruction will have a different recovery block. Another example of a typical multi-level speculation is as shown in Table 12.7(a). It is also referred to as a *cascaded speculation* in [18]. In Table 12.7(a), $* * y$ could be a potential weak update to $*x$ and/or $* * x$. Two check instructions for both $*x$ and $* * x$ need to be generated

as shown in Table 12.7(b), each with a different recovery block. If $* * y$ is a potential weak update only to $*x$, then the check instruction and recovery block will be as shown in Table 12.7(c). According to [3], there are many multi-level indirect references (e.g., multi-level field accesses) in the SPEC2000 C programs. Those references present opportunities for multi-level speculative PRE.

TABLE 12.7: Two examples of multi-level speculation

s1 : $\cdots = * * x$	ld .a r = [x]	ld .a r = [x]
...	ld .a f = [r]	ld .a f = [y]
s2 : $* * y = \dots$...	add s = r, f
s3 : $\cdots = * * x$	st ...	$*a = \dots$
(a) source program	chk.a r, <i>recovery</i> ₁	chk.a r, <i>recovery</i> ₁
	<i>label</i> ₁ :	<i>label</i> ₁ :
	...	chk.a f, <i>recovery</i> ₂
ld .a r = [x]	<i>recovery</i> ₁ :	<i>label</i> ₂ :
ld .a f = [r]	ld r = [x]	recovery ₁ :
...	ld f = [r]	ld r = [x]
st ...	br <i>label</i> ₁	add s = r, f
chk.a r, <i>recovery</i> ₁		br <i>label</i> ₁
<i>label</i> ₁ :	(c) only x could	
chk.a f, <i>recovery</i> ₂	potentially be	
<i>label</i> ₂ :	updated by	<i>recovery</i> ₂ :
...	intervening	ld f = [y]
<i>recovery</i> ₁ :	store * * y	add s = r, f
ld r = [x]		br <i>label</i> ₂
ld f = [r]	s1 : $\cdots = *x + *y$	
br <i>label</i> ₁	s2 : $*a = \dots$	
<i>recovery</i> ₂ :	(may update $*x$ or $*y$)	(e) speculation for
ld f = [r]	s3: $\cdots = *x + *y$	both the variables
br <i>label</i> ₂	(d) source program	and the value
(b) x and *x could		in a computation
potentially be		expression
updated by intervening		
store * * y		

The recovery code representation discussed in Section 12.5.1.1 can be applied directly to multi-level speculation without any change. In Table 12.8(b), both the address expression x and the value expression $*x$ are candidates for speculative register promotion. Hence, two *if-blocks* are generated. In Table 12.8(c), only the address x may be modified by the potential weak update. It

TABLE 12.8: Examples of check instructions and their recovery blocks in multi-level speculation

$s1 : \dots = **x$ \dots $s2 : **y = \dots$ $s3 : \dots = **x$ (a) source program	$r_1 = *x /*ld.a flag*/$ $f_1 = *r_1$ $\dots = *f_1$ $**y = \dots$ $\text{if}(r_1 \text{ is invalid}) \{$ $\quad /*\text{check flag}*/$ $r_2 = *x$ $f_2 = *r_2$ $\}$ $r_3 \leftarrow \phi(r_1, r_2)$ $f_3 \leftarrow \phi(f_1, f_2)$ $\dots = f_3$ $\text{if}(r_1 \text{ is invalid}) \{$ $\quad /*\text{check flag}*/$ $r_2 = *x$ $f_2 = r_2$ $r_3 \leftarrow \phi(r_1, r_2)$ $f_3 \leftarrow \phi(f_1, f_2)$ $\text{if}(f_3 \text{ is invalid}) \{$ $\quad /*\text{check flag}*/$ $f_4 = *r_1$ $\}$ $f_5 \leftarrow \phi(f_3, f_4)$ $\dots = f_5$ (b) x and *x could potentially be updated by intervening store **y	$r_1 = *x /*ld.a flag*/$ $f_1 = *y /*ld.a flag*/$ $s_1 = r_1 + f_1$ $**a = \dots$ $\text{if}(r_1 \text{ is invalid}) \{$ $\quad /*\text{check flag}*/$ $r_2 = *x$ $s_2 = *r_2 + f_1$ $\}$ $r_3 \leftarrow \phi(r_1, r_2)$ $s_3 \leftarrow \phi(s_1, s_2)$ $\text{if}(f_1 \text{ is invalid}) \{$ $\quad /*\text{check flag}*/$ $f_2 = *y$ $s_4 = r_3 + f_2$ $\}$ $f_3 \leftarrow \phi(f_1, f_2)$ $s_5 \leftarrow \phi(s_3, s_4)$ $\dots = s_5$ (c) only x could potentially be updated by intervening store $* * y$ $s1: \dots = *x + *y$ $s2: *a = \dots (\text{may update } *x \text{ or } *y)$ $s3: \dots = *x + *y$ (d) source program	(e) speculation for both the variables and the value in a computation
--	---	--	--

TABLE 12.9: Examples of check instructions and their recovery blocks in multi-level speculation

$s1: \dots = x_1$ $s2: **y = \dots$ (may update $\dots *x$ and $*x$) $s3: \dots = **x + 10$	$s1: \dots = x_1[h_1]$ $s2: **y = \dots$ $s3: \dots = **x + 10$ $[h_1 <\text{speculative}>]$	$s1: r_1 = *x / \text{ld.a flag/}$ $s2: \dots = r_1$ $**y = \dots$ $s4: \text{if}(r_1 \text{ is invalid}) \{$ $\quad /*\text{check flag}*/$ $\quad s5: \quad r_2 = *x$ $\}$ $s7: r_3 \leftarrow \phi(r_1, r_3)$ $s8: \dots = *r_3 + 10$
(a) source program	(b) output of the renaming step for speculative PRE for expression $*x$, where h is hypothetically temporary for the load of x	(c) output for the code motion step for speculative PRE for expression $*x$

is obvious that the representation in Table 12.8 matches very well with final assembly code shown in Table 12.7.

12.5.2.1 Recovery Code Generation for Speculative PRE Supporting Both Single-Level and Multi-Level Speculation

To generate check instructions and recovery code, the strategy is to generate the *check instruction* at the point of the *last weak update* but before its next *use*. Starting from a speculative redundant candidate, it reaches at its latest weak update. There, if the check instruction has yet to be generated, an *if-then statement* is inserted to model the check instruction and the corresponding recovery block, which initially will include just one single load instruction (i.e., the original load) in the *then* part of the *if-then statement*.

Table 12.9(a) gives an example to show the effect of the algorithm. It assumes that $**y$ in $s2$ could potentially update the loads $*x$ and $**x$ with a very low probability. The second occurrence of the expression $*x$ in $s3$ has been determined to be speculatively redundant to the first one in $s1$, as shown in Table 12.9(b). The compiler inserts an *if-then statement* after the weak update $**y$ in $s2$ for the speculative redundant expression $*x$ in $s3$, as shown in Table 12.9(c). The *chk.a* instruction can help later code generation to generate a *chk.a* instruction, or an *ld.c* instruction if there is only one load instruction in the recovery block.

Using an *if-block* to represent a check instruction and its recovery block, all previous discussions on recovery code generation for speculative PRE can be applied directly to multi-level speculation without any modification. It only requires an expression tree to be processed in a bottom-up order. For example,

given an expression $* * *x$, it begins by processing the sub-expression $*x$ for the first-level speculation, then the sub-expression $* * x$ for the second-level speculation, and then $* * *x$ for the third level speculation. This processing order also guarantees that the placement of check instructions and the recovery code generation are optimized level-by-level.

Therefore, when a sub-expression is processed for the n^{th} level speculation, every sub-expression at the $(n - 1)^{th}$ level speculation has been processed, and its corresponding recovery block (i.e., *if-block*) generated. According to the placement scheme described in Section 12.5.1.1, an *if-block* is generated at the last weak update of the sub-expression. A re-load instruction for the value of the sub-expression will be included initially in the *if-block*.

Table 12.10(b) shows the result after the first-level speculation on $*x$ for the program in Table 12.10(a). It assumes $* * y$ is an alias of both $*x$ and $* * x$ with a very *low* probability. Hence, $* * y$ in $s2$ is a speculative weak update for both $*x$ and $* * x$. During the first-level speculation, the sub-expression $*x$ is speculatively promoted to the register r . The subscript of r represents the version number of r . As can be seen in Table 12.10(b), the value of $*x$ in $s1'$ becomes unavailable to the use in $s3$ along the *true* path of the *if-block* because of the re-load instruction in $s5$. The value of an expression is unavailable to a later *use* of the expression if the value of the expression is redefined before the *use*. This can be seen by the version number of r that changes from r_1 to r_3 due to the $\Phi(r_1, r_2)$ in $s6$.

In the second-level speculation for $* * x$, $* * x$ can also be speculatively promoted to the register s . A hypothetical variable h is created for $*r$ (i.e., $* * p$) in $s1'$. Table 12.10(c) shows the result after the Φ -insertion step [5]. A Φ -function (h, h) in $s6'$ is inserted because of the *if-block* in $s4$. The first operand of $\Phi(h, h)$, which corresponds to the false path of the *if-block* in $s4$, will have the same version number as that in $s1'$ because of the speculative weak update in $s2$ after the rename step [18]. However, the second operand of $\Phi(h, h)$, which corresponds to the *true* path, is replaced by \perp . It is because the value of h (i.e., $*r$) becomes unavailable due to the re-load (i.e., a re-definition) of r in $s5$. The result is shown in Table 12.10(d).

The *if-block* in $s8$ is inserted after the speculative weak update of $s2$ with a re-load of $*r$ in $s9$ (Table 12.10(e)). Due to the second operand of Φ -function (h_1, \perp) in $s6'$ of Table 12.10(d), the algorithm will insert a reload of $*r$ (in $s5'$) along the true path of the *if-block* (in $s4$) to make the loading of $*r_3$ in $s3$ fully redundant. That is, it will automatically update the recovery block from the first level (i.e., $*p$) without additional effort.

This advantage of the *if-block* representation for the recovery block can be better appreciated by the difficulty it will incur if the check instructions are represented as assignment statements as mentioned in Section 12.5.1.1. Assume the expression $*x$ and $* * x$ are speculative redundant candidates. Using an assignment statement to represent a check instruction, the recovery block generated for the expression $*x$ is shown in Table 12.10(f). Since the value r is redefined at $s4$, it is no longer possible to detect the expression $*r$

(i.e., $\ast \ast x$) in $s3$ as speculative redundant (because r has been redefined). In order to support multi-level speculation, the algorithm has to be significantly modified.

12.5.3 Interaction of the Early Introduced Recovery Code with Later Optimizations

The recovery blocks, represented by *highly biased* if-then statements, can be easily maintained in later optimizations such as instruction scheduling. In this section, *partially ready* (P -ready) code motion algorithm [1] during the instruction-scheduling phase is used as an example to illustrate this case. Consider the example in Figure 12.10(a) and assume that the *right* branch is *very rarely* taken. The instruction $b = w$ in the basic block D can be identified as a P -ready candidate [1]. It will then be scheduled along the most likely path $A \rightarrow B \rightarrow D$, instead of the unlikely path $A \rightarrow C \rightarrow D$. Figure 12.10(b) shows the result of such a code motion, with this instruction being hoisted to the basic block A and a compensation copy is placed in the basic block C .

The recovery code represented by an *explicit* and *highly biased* if-then statement is a good candidate for P -ready code motion. If there exists a dependent instruction that could be hoisted before the recovery block, this instruction will be duplicated as a compensation instruction in the recovery block. At the end of instruction scheduling, the recovery block is already well formed. After instruction scheduling, these if-then blocks can be easily converted into checks and recovery code.

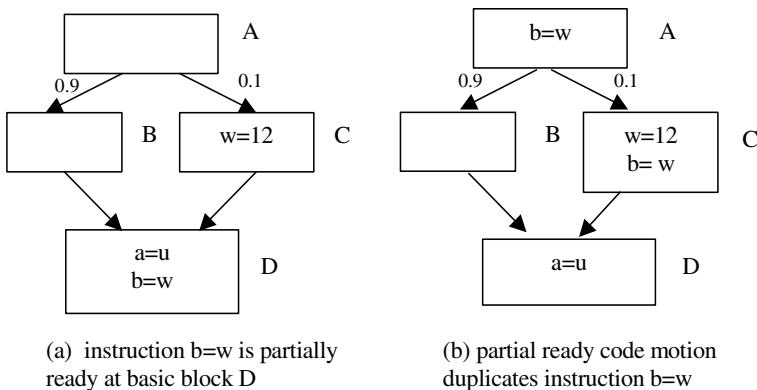


FIGURE 12.10: Example of partial ready code motion for latency hiding.

Using the example in Table 12.6(a), the output of the speculative PRE (shown in Table 12.6(c)) is further optimized with instruction scheduling.

TABLE 12.10: Example of recovery code generation in speculative PRE

<p>s1: $\dots = **x$</p> <p>s2: $**y = \dots$ (may update $**x$ and $*x$)</p> <p>s3: $\dots = **x + 10;$</p> <p>(a) source program</p> <p>s1: $r_1 = *x /ld.a\ flag/$</p> <p>s1': $\dots = *r_1[h]$</p> <p>s2: $**y = \dots$</p> <p>s4: if (r_1 is invalid) { /chk flag/</p> <p>s5: $r_2 = *x$</p> <p style="text-align: right;">}</p> <p>s6: $r_3 \leftarrow \phi(r_1, r_2)$</p> <p>s6': $h \leftarrow \phi(h, h)$</p> <p>s3: $\dots = *r_3[h] + 10$ /* h is hypothetical temporary for the load of $**x$ */</p> <p>(c) output of insertion</p> <p>step for 2nd level</p> <p>speculation</p> <p>s1: $r_1 = *x /ld.a\ flag/$</p> <p>s1': $\dots = *r_1[h_1]$</p> <p>s2: $**y = \dots$</p> <p>s4: if (r_1 is invalid) { /chk flag/</p> <p>s5: $r_2 = *x$</p> <p style="text-align: right;">}</p> <p>s6: $r_3 \leftarrow \phi(r_1, r_2)$</p> <p>s6': $h_2 \leftarrow \phi(h_1 <sp^1>, \perp)$</p> <p>s3: $\dots = *r_3[h_2] + 10$</p> <p>(d) output of rename</p> <p>step for 2nd level</p> <p>speculation</p>	<p>s1: $r_1 = *x /ld.a\ flag/$</p> <p>s1': $\dots = *r_1[h]$</p> <p>s2: $**y = \dots$</p> <p>s4: if (r_1 is invalid) { /chk flag/</p> <p>s5: $r_2 = *x$</p> <p style="text-align: right;">}</p> <p>s6: $r_3 \leftarrow \phi(r_1, r_2)$</p> <p>s6': $s_2 = *r_2$</p> <p style="text-align: right;">}</p> <p>s3: $r_3 \leftarrow \phi(r_1, r_2)$</p> <p>s7: $s_3 \leftarrow \phi(s_1, s_2)$</p> <p>s8: if ($s_1$ is invalid) { /chk flag/</p> <p>s9: $s_4 = *r_3$</p> <p style="text-align: right;">}</p> <p>s10: $s_5 \leftarrow \phi(s_3, s_4)$</p> <p>s3: $\dots = s_5 + 10$</p> <p>(e) output of recovery</p> <p>code for 2nd level</p> <p>speculation</p> <p>s1: $r_1 = *x /ld.a\ flag/$</p> <p>s1': $\dots = *r_1$</p> <p>s2: $**y = \dots$</p> <p>s4: $r_2 = *x /chk\ flag/$</p> <p>s3: $\dots = *r_2 + 10$</p> <p>(f) output of recovery</p> <p>code for 1st level</p> <p>speculation using</p> <p>assignment statement</p> <p>representation</p>
---	---

¹speculative

As can be seen in Table 12.11(c), if the compiler determines the instruction $s = r + 10$ can be speculatively hoisted across the store $*y$ statement, it will duplicate the instruction in the recovery block. The generated check and recovery code is shown in Table 12.11(d).

TABLE 12.11: The recovery code introduced in the speculative PRE interacts with the instruction scheduling

$\dots = *x$ $*y = \dots$ (may update $*x$) $\dots = *x + 10;$ (a) source program	$r = *x / \text{ld.a flag/}$ $\dots = r$ $s = r + 10$ $*y = \dots$ $\text{if } (r \text{ is invalid) \{ }$ $/\text{chk flag/}$
$r = *x /*\text{ld.a flag*/}$ $\dots = r$	$r = *x$ $s = r + 10$ $\}$ (c) output after instruction scheduling
$*y = \dots$ $\text{if } (r \text{ is invalid) \{ }$ $/* \text{chk flag */}$ $\quad r = *y$ $\quad \}$ $s = r + 10$ \dots	$\text{ld.a } r = [x]$ $\text{add } s = r, 10$ $\text{st } [y] = \dots$ $\text{chk.a } r, \text{ recovery}$ label : ... (b) output after speculative PRE
	recovery: $\text{ld } r = [x]$ $\text{add } s = r, 10$ $\text{br } \text{label}$
	(d) final output

12.6 Conclusion

Data and control speculation can be incorporated in the compiler for most of the compiler optimizations. Using profiling information and/or compiler heuristic rules, a compiler could very aggressively exploit the likeliness (or unlikeliness) of certain control and data dependence relations for more op-

timization opportunities speculatively. Some recent processor architectures such as Intel's IA64 platforms provide hardware support to detect potential mis-speculation at runtime and allow recovery action to be taken for correct execution when mis-speculation occurs. However, the compiler has to identify the opportunities in application programs for potential speculative execution that can improve overall performance, and to generate recovery code that supports correct execution when mis-speculation occurs.

In this chapter, a compiler framework based on a speculative SSA form that incorporates speculative information for both data and control speculation is used as an example to show what is needed in such a framework. It shows that simple extensions to existing compiler frameworks could effectively support data and control speculation for most common compiler optimizations. The issues related to alias profiling and data dependence profiling are also presented. Such profiling information provided needed likeliness information for control and data dependences in application programs. Compiler heuristics are also shown to be very effective for alias information.

A simple *if-block* representation that could facilitate the generation of check instructions and recovery code in case of mis-speculation is also described in detail. Such a representation allows speculative recovery code generated early on during program optimizations to be easily integrated into the subsequent optimizations such as instruction scheduling. It also allows multi-level speculation to be implemented easily.

Compiler support for data and control speculation is still in a very early stage of development. More work is needed in this area to exploit their full potential of improving overall performance in application programs. The application of such work on compilers that support speculative multi-core, multi-threaded processors will also be a very fruitful area for future research.

References

- [1] J. Bharadwaj, K. Menezes, and C. McKinsey. Wavefront scheduling: Path based data representation and scheduling of subgraphs. In Proc. of 32nd Ann. Int'l Symp. on Microarchitecture, December, 1999.
- [2] R. Bodik, R. Gupta, and M. Soffa. Complete removal of redundant expressions. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 1-14, Montreal, Canada, 17-19 June 1998.
- [3] T. Chen, J. Lin, W. Hsu, and P.-C. Yew. An empirical study on the granularity of pointer analysis in C programs. In 15th Workshop on

- Languages and Compilers for Parallel Computing, pages 151-160, College Park, Maryland, July 2002.
- [4] T. Chen, J. Lin, X. Dai, W.-C. Hsu, and P.-C. Yew. Data dependence profiling for speculative optimizations. In Proceedings of the 13th International Conference on Compiler Construction, Barcelona, Spain, March 2004.
 - [5] F. Chow, S. Chan, S. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In Proceedings of the Sixth International Conference on Compiler Construction, pages 253–267, April 1996.
 - [6] F. Chow, S. Chan, R. Kennedy, S. Liu, R. Lo, and P. Tu. A new algorithm for partial redundancy elimination based on SSA form. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 273–286, Las Vegas, Nevada, May 1997.
 - [7] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4): 451–490, 1991.
 - [8] C. Dulong. The IA-64 Architecture at Work, *IEEE Computer*, Vol. 31, No. 7, pages 24-32, July 1998.
 - [9] R. Ghiya, D. Lavery, and D. Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, pages 47-58, Snowbird, Utah, June 2001.
 - [10] K. Ishizaki, T. Inagaki, H. Komatsu, T. Nakatani. Eliminating exception constraints of Java programs for IA-64. In Proceedings of The Eleventh International Conference on Parallel Architectures and Compilation Techniques (PACT-2002), pp. 259-268, September 22-26, 2002
 - [11] R. D.-C. Ju, K. Nomura, U. Mahadevan, and L.-C. Wu. A unified compiler framework for control and data speculation. In Proc. of 2000 Int'l Conf. on Parallel Architectures and Compilation Techniques, pages 157 - 168, Oct. 2000.
 - [12] M. Kawahito, H. Komatsu, and T. Nakatani. Effective null pointer check elimination utilizing hardware trap. In Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), November 12-15, 2000.
 - [13] J. Lin, T. Chen, W.-C. Hsu, P.-C. Yew, R. D-C Ju, T-F. Ngai, and S. Chan. A compiler framework for speculative analysis and optimiza-

- tions, In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2003.
- [14] J. Lin, W.-C. Hsu, P.-C. Yew, R. D.-C. Ju, and T.-F. Ngai. A compiler framework for recovery code generation in general speculative optimizations. In Proceedings of Int'l Conf. on Parallel Architectures and Compilation Techniques, pages 17 - 28, Oct. 2004.
 - [15] R. Lo, F. Chow, R. Kennedy, S. Liu, and P. Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation, pages 26–37, Montreal, 1998
 - [16] U. Mahadevan, K. Nomura, R. D.-C. Ju, and R. Hank. Applying data speculation in modulo scheduled loops. Proc. of 2000 Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT), pp. 169 - 176, Oct. 2000.
 - [17] R. Kennedy, F. Chow, P. Dahl, S.-M. Liu, R. Lo, and M. Streich. Strength reduction via SSAPRE. In Proceedings of the Seventh International Conference on Compiler Construction, pages 144–158, Lisbon, Portugal, Apr. 1998.
 - [18] R. Kennedy, S. Chan, S. Liu, R. Lo, P. Tu, and F. Chow. Partial redundancy elimination in SSA form. ACM Trans. on Programming Languages and Systems, v.21 n.3, pages 627-676, May 1999.
 - [19] B. Steensgaard. Points-to analysis in almost linear time. In Proceedings of ACM Symposium on Principles of Programming Languages, pages 32–41, Jan. 1996.
 - [20] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-27), pages 1–11, Nov 1994.

Chapter 13

Multithreading and Speculation

Pedro Marcuello, Jesus Sanchez, and Antonio González

Intel-UPC Barcelona Research Center; Intel Labs; Universitat Politecnica de Catalunya; Barcelona (Spain)

13.1	Introduction	333
13.2	Speculative Multithreaded Architectures	336
13.3	Helper Threads	339
13.4	Speculative Architectural Threads	341
13.5	Some Speculative Multithreaded Architectures	349
13.6	Concluding Remarks	350
	References	350

13.1 Introduction

Moore's law states that the number of available transistors per chip is doubled at each process generation. Such continuous advance in technology has allowed processor designers to incorporate new and more powerful features in each processor generation to run the applications faster. Processor microarchitecture has evolved from single-issue in-order pipelined processors to current superscalar architectures that can run simultaneously multiple threads in the same processor core. These microarchitectures that can exploit both instruction-level parallelism (ILP) and thread-level parallelism (TLP) are known as *multithreaded processors*.

In previous chapters of this book, speculation techniques to extract higher degrees of ILP in dynamically-scheduled superscalar processors have been discussed. Branch prediction and complex fetch engines provide a continuous flow of instructions to the back-end; data speculation diminishes the penalties due to data dependences; and prefetching and precomputation techniques reduce the cost of high-latency operations such as memory instructions that miss in cache. We can conclude that the use of speculative instruction-level parallelism has become one of the main assets for the design of current microprocessors. However, the improvement in performance achieved by scaling up current superscalar organizations, even with the most complex speculation mechanisms, is decreasing and approaching a point of diminishing returns.

The evolution of the workloads that run in most computers together with the difficulties to further increase the exploitation of ILP have motivated researchers to look for alternative techniques to increase the performance of the processors. An approach that has been reflected in some recent processors is based on the exploitation of different sources of parallelism, namely, fine-grain (instruction) and coarse-grain (thread) level parallelism. This has given rise to the so called multithreaded and multi-core processors.

Exploiting thread-level parallelism is an old idea and it has been widely studied in the past, especially in the context of multiprocessor architectures. Traditionally, thread-level parallelism has been exploited in a non-speculative manner, that is, parallel threads always commit, even though speculative instruction-level parallelism is exploited for each thread.

The main sources for thread-level parallelism have been basically two: 1) different applications that are run in parallel, and 2) parallel threads generated from a single program through the compiler/programmer support.

In the former case, threads run independently one from each other and no communication among them is required. The main benefit of this scheme is an increase in throughput (number of works finished per time unit). However, the individual execution time of a single application may increase when it is executed simultaneously with some others. This is due to the fact that the instructions of each application have to compete for some shared resources (e.g., cache memories, branch predictors, etc.) with instructions from other applications.

On the other hand, in the latter case, threads correspond to different pieces of the same program that are executed in parallel. In this case, partitioning the application into small parts and running them concurrently can significantly reduce the execution time of the application with respect to a single-threaded execution. However, unlike the previous model, threads are highly coupled and the processor has to provide support for communicating and synchronizing the parallel threads.

Partitioning applications into parallel threads may be straightforward for some programs, such as some numeric and multimedia applications, but it is very hard for many others.

Compilers often fail to find thread-level parallelism because they are conservative when partitioning a program into parallel threads, since they have to guarantee that their parallel execution will not violate the semantics of the program. This significantly constrains the amount of thread-level parallelism that the compiler can discover.

By allowing threads to execute in a speculative way and being able to squash them in case of a misspeculation, the scope for exploiting thread-level parallelism is significantly broadened. Speculative threads are not allowed to modify the architectural state of the processor until their correctness is verified. Then, if a thread's verification fails, the work done by the speculative thread is discarded and a roll-back mechanism is used to return the processor to a correct state. On the other hand, if the verification succeeds, the

work performed by the thread is allowed to be committed. This parallelism exploited by speculative threads is referred to as *speculative thread-level parallelism*.

In order to exploit speculative thread-level parallelism, some requirements are needed. First, hardware and/or software support for executing speculative threads is necessary. This hardware/software should include mechanisms to hold the speculative state until it can be safely committed, mechanisms for communicating and synchronizing concurrent threads, and verification and recovery schemes. In particular, inter-thread data dependence management has a significant impact on the design and performance of these processors.

In addition to that, a strategy for deciding which speculative threads are to be spawned is also necessary. What instructions speculative threads are to execute, at which point such threads are to be spawned and conclude their work, how a misspeculation on these threads is to affect the processor, and the steps required for recovery dramatically impact on the performance of the processor too. Besides, the hardware requirements may be different depending on the speculation strategy. The combination of the hardware/software support and the speculation strategy defines a *speculative multithreaded architecture*.

In this chapter, different proposals for speculative multithreaded architectures are discussed and their main hardware/software requirements are analyzed. We distinguish two main families of architectures. In the first family, speculative threads do not alter the architectural state of the processor but they just try to reduce the cost of high-latency instructions through the side-effects caused by speculative threads on some subsystems of the processor, such as prefetching on caches or warming up branch predictors. The speculative threads executed by this model are referred to as *Helper Threads* (HT). In the second family, each speculative thread executes a different part of the code and computes its architectural state that is only committed if the speculation is correct. In this execution model, the speculative threads are referred to as *Speculative Architectural Threads* (SAT).

The rest of the chapter is organized as follows. Section 13.2 introduces basic concepts on speculative thread-level parallelism and outlines the main features of both families for exploiting it. Section 13.3 presents the Helper Thread family and analyzes its main features. The Speculative Architectural Threads family is presented in Section 13.4 and the main requirements for this model are analyzed. Some examples of proposed speculative multithreaded architectures are presented in Section 13.5. Finally, Section 13.6 summarizes the chapter and points out some future issues for this execution paradigm.

13.2 Speculative Multithreaded Architectures

Speculative Multithreading is a technique based on speculation that tries to reduce the execution time of an application by means of running several speculative threads in parallel. Threads are speculative in the sense that they can be data and control dependent on other threads and their correct execution and commitment are not guaranteed, unlike the traditional non-speculative multithreading exploited conventionally. Speculative thread-level parallelism is the additional parallelism obtained by these speculative threads on a speculative multithreaded processor.

To reduce the execution time of applications, several approaches based on speculative threads can be considered. The main differences among them reside on the strategy to decide which speculative threads are spawned, the requirements to concurrently execute such speculative threads (i.e., if they require to communicate or to synchronize among them), and how speculative threads are spawned and committed or squashed. These features can be combined to exploit speculative thread-level parallelism in different ways.

There are two main approaches to exploiting speculative thread-level parallelism that have been widely studied: Helper Threads and Speculative Architectural Threads.

The Helper Thread paradigm ([8, 20, 28, 38, 39] among others) is based on the observation that the cost of some instructions severely hurt the performance of the processor, such as loads that miss in cache or branches that are incorrectly predicted. This paradigm tries to reduce the execution time of the applications by means of using speculative threads that cut down the latency of these costly operations.

The Helper Thread paradigm consists of a main non-speculative thread that executes all the instructions of the code and some speculative threads that help to decrease the execution time of the main thread by means of some side effects. For instance, these speculative threads may be used to prefetch memory values in order to reduce cache misses when they are required by the main thread, or to precompute some branches that are hard to predict.

There are several manners to build the speculative threads as it is described in the next section of the chapter, but basically a speculative thread contains previous dynamic instructions in the control-flow and data-dependence graph of a long-latency instruction. The speculative thread executes these instructions before the non-speculative thread does. It does not change the architectural state of the processor but modifies the microarchitectural state (e.g., warms up the cache or the branch predictor) so that when the main thread executes the critical instruction, it is executed faster.

Figure 13.1 shows an example of the Helper Thread paradigm. Assume that the bold load always misses in cache and then the dependent instructions (the add) is delayed until this instruction is satisfied. Below the code, the helper

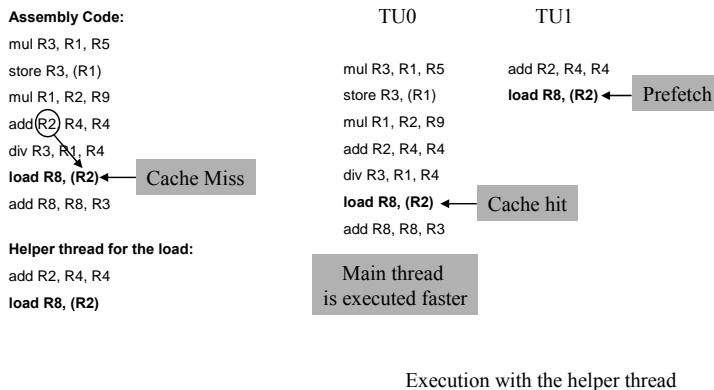


FIGURE 13.1: Helper thread example.

thread for that load is presented. The speculative thread is executed in a different context (thread unit 1) and performs the load before, so that when it is executed by the main thread, it does not miss in cache since it has been prefetched. To do that, the speculative thread consists of the instructions that the load depends on. The performance of this strategy will depend on the ability to spawn the helper thread with enough anticipation to perform the prefetch and keep the requested line in cache until the load is executed.

In this execution model, the communication among threads is straightforward since speculative threads do not modify any architectural state of the processor. They may reuse values from the non-speculative thread and modify microarchitectural structures also used by the non-speculative thread as pointed out above.

In the Speculative Architectural Thread paradigm ([1, 6, 22, 31] among others) there is also one non-speculative thread at any point in time and speculative threads, but unlike the previous approach, all threads contribute to compute architectural state. In other words, the program is parallelized in the conventional sense, but in a speculative manner. The speculative nature of these threads may cause that the work done by a speculative thread is useless since it may execute with incorrect inputs. In this case, this thread is squashed and is not allowed to modify the architectural state of the processor.

When the non-speculative thread reaches the first instruction of a running or terminated speculative thread, a verification process checks whether such speculative thread is correct (that is, it has not violated any data nor control dependence). If so, the non-speculative thread finishes, its resources are freed for future use by another thread, and such speculative thread becomes the

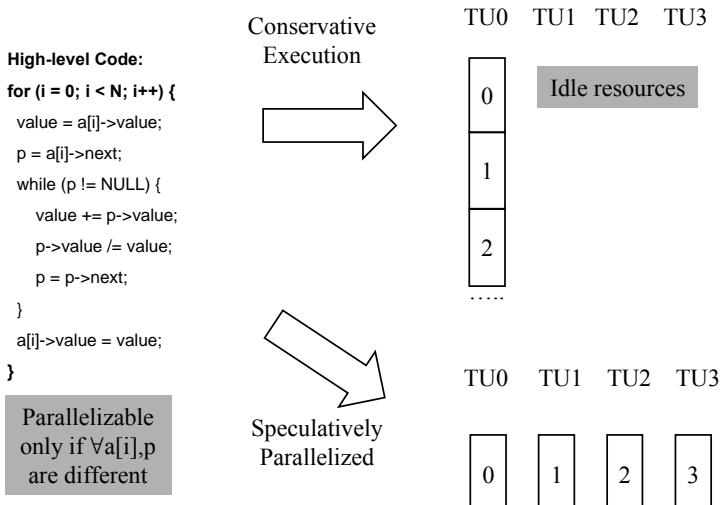


FIGURE 13.2: Speculative architectural thread example.

new non-speculative one. If the verification fails, then the speculative thread is squashed and the current non-speculative thread continues executing instructions in a normal way. Note that the verification has not been totally performed at the point when the non-speculative thread reaches a speculative one. Part of the verification can be performed before, as the threads are executed, to anticipate the detection of misspeculations.

Figure 13.2 shows an example of the Speculative Architectural Thread approach. The outer loop cannot be parallelized since the compiler cannot assure that the pointers in vector a and the corresponding linked lists are always different. This fact may be critical since the inner loop modifies the field value of the linked elements. However, the outer loop can be speculatively parallelized by spawning a speculative thread for every iteration assuming there will not be dependences among iterations. If a misspeculation occurs due to a data dependence violation, then the work done by the speculative thread is discarded. Note that, if this loop is executed on a conventional processor, the iterations will be executed serially as it is shown in the figure and some resources will be wasted. However, if the speculation is correct, several iterations can be performed in parallel and the execution time of the application is reduced.

In both speculative multithreaded models, there are two main factors that have a significant impact on the performance of the processor. One of them

is the thread spawning scheme. Speculative threads are obtained from applications using different techniques (compiler support with/without profiling, run-time mechanisms, etc.). The features of the selected threads will dramatically impact on the performance of the processor. For Helper Threads, speculative threads should alter the architectural state of the processor before the main thread executes the target instruction. On the other hand, for Speculative Architectural Threads, features such as control and data independence, data predictability, load balance, and coverage are key for performance.

In addition to that, hardware/software support for executing speculative threads is necessary. The requirements for spawning, verifying, and committing threads as well as providing support for storing the speculative state have a significant impact on the cost and performance of these systems. In particular, the way inter-thread data dependences are managed strongly affects the required hardware/software and the resulting performance.

13.3 Helper Threads

A helper thread consists of a set of instructions such that, when executed in parallel with the main thread, they are expected to produce a beneficial side effect on the latter. A helper thread does not compute a part of a program, but executes a piece of code that may improve the execution of the main thread. All instructions of the program must still be executed by the main thread. The main objective of helper threads is to speedup some long latency instructions such as loads and branches. In the case of loads, a helper thread prefetches the data to avoid a long latency memory access in the main thread. For branches, a helper thread may help by precomputing the branch outcome. Other possible uses of helper threads are feasible but loads and branches have been the main target of the proposals so far.

The benefit of such technique depends on which loads or branches are selected to be affected by helper threads. Trying to help any load or branch in a program in a blind manner would usually mean a too high overhead that may even offset the potential benefits, in addition to significant increase in power budget. That results in a reduction in the efficiency of the technique and in a waste of resources and power. Fortunately, caches and branch predictors work quite well for the majority of loads and branches respectively. Thus, helper threads should be only used to help those cases where the standard mechanisms are inefficient. We will refer to these instructions as critical instructions.

There are several ways to identify critical instructions: based on heuristics [20], based on profiling [8], or based on dynamic history of the instructions [39]. Though it is not necessary, the first two approaches are usually

employed when helper threads are constructed statically by either the compiler or an assembly/binary optimizer, whereas dynamic detection of critical instructions is used when helper threads are built at run time.

The typical execution model for helper threads consists of triggering the execution of the helper thread from the main thread some time before the critical instruction is actually executed. This trigger may be an explicit instruction inserted statically or an event detected by a hardware mechanism. In any case, the time ahead at which the trigger is launched is an important decision since the goal is to solve the critical instruction before its actual execution. If that is done earlier than needed, the result must be kept in some place, wasting storage, or may be lost. On the other hand, if the trigger is performed too late, then the benefit may be reduced or even disappear.

13.3.1 Building Helper Threads

Another important issue is how helper threads are built. There are two main approaches proposed in the literature. The first one is based on identifying the backward slice of a critical instruction [37] and selecting a subset of it that is referred to as a pre-computation slice. This subset usually represents the minimum dependence sub-graph that is needed to correctly execute the critical instruction most of the times (to compute the effective address and access memory in case of a load, or to compute the register/s implied in a conditional branch in case of the branch). There are also different ways to identify the static instructions that make up the speculative thread. The two most common cases consist of: 1) marking in the original binary the instructions that correspond to the helper thread (this means the addition of a field to each instruction to indicate whether it belongs to a given helper thread or not), or 2) copying instructions from the main thread to a new section in the program. In any case, the length of the pre-computation slice will, among other issues, determine how ahead the helper thread trigger must be inserted. As the execution of the helper thread will not affect the correctness of the main thread (it may just speed up its execution), the pre-computation slice could be speculatively optimized. Memory disambiguation and branch pruning have been proposed among other optimizations.

A different approach to build a helper thread can be found in the Subordinate Multithreading framework [3], in which the instructions that form a helper thread do not necessarily belong to the program. A subordinate thread code consists of micro-code in the internal ISA of the machine. This micro-code may contain several routines, each of them implementing a certain algorithm to be applied on specific events (for instance, cache misses or branch missprediction). This allows the application to use powerful and adaptable algorithms to deal with such events that would be very costly to implement in hardware.

13.3.2 Microarchitectural Support for Helper Threads

Since the execution of a helper thread does not modify the architectural state of the processor, no verification or recovery mechanisms are needed, no matter if the work done by the helper thread is successful or not.

Regarding implementation issues, the processor must include some mechanism to pass data between the main thread and the helper threads. For instance, in the case of helper threads for data prefetching, the data cache can be shared between the main and the helper threads. Simultaneous multithreaded architectures [34] (e.g., Intel's Hyperthreading [26]) are good candidates for implementing such technique.

13.4 Speculative Architectural Threads

In the Speculative Architectural Thread paradigm, the main idea is to execute different sections of a program in parallel and speculatively, in such a way that if the execution is correct, the values computed by speculative threads can be committed and the code executed by them does not need to be re-executed by the main thread. Thus, the objective of this technique is to parallelize a program as opposed to speeding up the main thread. In any case, this technique is orthogonal to that of helper threads and they can be combined.

The problem of parallelizing a program (or a section of it) has been extensively studied in the past. Traditional parallelizing compilers try to find independent sections of code (typically loop iterations) or sections of code with few dependences, and add the corresponding synchronization and communication operations to honor such dependences. Though this technique has shown to be effective for scientific codes, in which most of the time the program is executing loops that manipulate matrix-like structures, it is much less effective for other types of codes. For example, the abundance of pointers and the use of irregular memory access patterns make it very difficult for the compiler to disambiguate most memory dependences and many opportunities to parallelize code may be lost.

If we relax this constraint of generating code that honors any possible dependence, the opportunities to parallelize a code greatly increase. In front of a possible dependence, one way to speculate is to ignore the dependence if it is likely not to occur. Another way to speculate is to try to predict or precompute the values that will be produced through these dependences, if they are likely to occur.

The potential of Speculative Architectural Threads is greater than that of Helper Threads since side effects that can be achieved through the latter can also be achieved by the former, and only the former can parallelize a code.

On the other hand, Speculative Architectural Threads require a more complex hardware.

In a Speculative Architectural Thread architecture there is a non-speculative (or main) thread always running, whereas the rest of the threads (if any) are speculative. Threads have a predecessor-successor relationship among them that reflect the order in which their instructions would be executed in a single-threaded processor. Only the non-speculative thread, which corresponds to the oldest thread, is allowed to commit its state. In some implementations, the only thread allowed to spawn speculative threads is the non-speculative one [40], whereas in other implementations, any thread can spawn others [1, 22].

During its execution, a speculative thread goes through different stages. First, it is created and some time is spent to allocate the resources for the new thread and initialize the different structures needed (e.g., register file, etc.). Afterwards, depending on the type of speculative multithreading, the next stage consists of identifying the inter-thread dependences and computing the thread live-ins. After that, the thread starts the execution of the body code until the end of the thread is found. After that, the thread is validated and, if the results are found to be correct, the values are committed.

In the next subsections, different schemes for partitioning the applications into speculative threads are presented as well the basic support for executing speculative architectural threads.

13.4.1 Thread Spawning Schemes

A thread spawning mechanism tries to identify which parts of the programs are the most suitable to be executed by speculative threads. That is, at which points of the program speculative threads should be spawned, which part of the code is to be executed by speculative threads, and when the speculation should be validated. The different ways these issues are handled result in different spawning schemes.

For any speculative thread, two basic points in the program stream must be determined: when the spawn of it is triggered, and where the new thread starts. Hereinafter, we will refer to these two points as *spawning point* and *thread start point* respectively. A *spawning pair* is the spawning point and the thread start point of a given thread.

Figure 13.3 illustrates the execution model. Assume that two spawning pairs have been found in this application. These pairs are identified as (SP1,TSP1) and (SP2,TSP2). The program starts executing in Thread Unit 0. When this thread finds a spawning point, then a new speculative thread is allocated in a free thread unit, in this case, thread unit 2. The spawned thread will execute instructions starting from the corresponding thread start point, that is, TSP1. However, there is an initialization overhead that includes the set-up of all the structures of the thread unit. A discussion of this initialization is presented in following subsections. Meanwhile, the non-

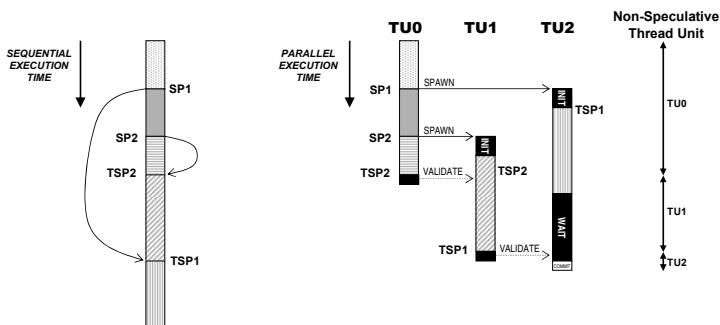


FIGURE 13.3: Example of execution in a speculative multithreaded processor.

speculative thread continues executing instructions and both threads finally proceed in parallel. When the non-speculative thread reaches a new spawning point (SP2), a new speculative thread is spawned at thread unit 1 and these three threads run in parallel. When the non-speculative thread reaches the thread start point of any active thread, in that case TSP2, it stops fetching instructions. Then, a validation process starts to verify that the speculation was correct. If so, the non-speculative thread is committed and the thread unit is freed for future use. Afterwards, the speculative thread on thread unit 2 reaches the end of the program. As it is not the non-speculative thread, it has to wait until all previous threads finish and validate their speculation. Then, when the non-speculative thread (running on thread unit 1) reaches TSP1, it verifies the speculation and commits, and the thread that is stalled at thread unit 2 is allowed to commit.

Which sections of code are executed by speculative threads can be determined either at compile time or run time. In the former case, the compiler performs an analysis of the code and inserts special instructions in the program that determine the spawning pair of each speculative thread. In the latter, a specialized hardware is responsible for analyzing run-time statistics and detect effective spawning pairs.

The effectiveness of a thread spawning scheme may be computed in different ways: coverage, ratio of successful speculations, average active threads per cycle, etc. However, the metric that really determines the goodness of a spawning scheme is the execution time. If an application with a given spawning policy is executed faster than with others, the former is better. It is possible that a policy with lower coverage and less active threads per cycle outperforms others in execution time. For instance, if speculative threads are

data dependent on previous ones, they may have to wait for the computation of the dependent values whereas if speculative threads are almost independent they may proceed in parallel most of the time. Certain properties that may improve the quality of the speculative threads are:

- Control independence: The probability of reaching the thread start point from the spawning point must be very high. Otherwise, this thread will be cancelled at some time in the future. It will not contribute to increase performance but it will consume energy and may prevent other useful threads from being executed.
- Thread size: The distance between the spawning point and the thread start point should not be too small nor too large to keep the thread size into a certain limit. Small threads result in too much spawning overhead and large threads may require large storage for speculative state.
- Data independence: Threads may have data dependences among them. Instructions after the thread start point (the instructions that will correspond to the speculative thread) should have few dependences with instructions between the spawning and the thread starting point. The more dependences, the more difficult to predict/precompute the thread input values, and the more likely to fail due to a misspeculation.
- Workload balance: The ideal scenario is when the maximum number of threads are always active and doing useful work. This may not happen due to several reasons: a spawning point is not found immediately after finishing a thread, a thread is waiting for some event (for instance, to be validated), a speculative thread is doing some initializations, or a speculative thread is misspecified and squashed.

The previous list is not exhaustive. Depending on the particular speculative multithreaded architecture, some other criteria can be considered such as data predictability if we use data value speculation to deal with data dependences.

Some thread spawning schemes are based on simple heuristics. In some proposals speculative threads are assigned to well-known program constructs that may provide certain of the desired features mentioned above. The three most studied schemes are loop iterations, loop continuation, and call continuation [1, 9, 27, 23, 24, 33]. Spawning on loop iterations tries to parallelize different iterations of the same loop. For spawn on loop continuation the spawning point is the beginning of a loop and the thread start point is the exit of the loop. Similarly, for spawn on call continuation the spawning point is the beginning of a subroutine and the thread start point is the instruction after the return. A similar approach to this is spawning threads at module-level instead of subroutine-level and it was studied in [36]. The common feature of these schemes is that they are very likely to reach the thread start point after the spawning point is reached. There are some other heuristics are also based on cache misses such as presented in [6].

Other thread spawning schemes are based on a more sophisticated analysis of the code in order to quantify some relevant metrics, as the one described above, that can help to identify the sections of the code that can most benefit from being speculatively parallelized. These schemes can be implemented in the compiler [25, 35] and generate special code for speculative multithreaded processors or at run-time [4]. This last technique includes the re-compilation of Java applications.

13.4.2 Microarchitectural Support for Speculative Architectural Threads

The design space for Speculative Architectural Threads architecture is huge. The main required feature for supporting this execution model is to provide mechanisms to deal with multiple threads simultaneously and some special extensions to deal with the special nature of the speculative threads. These additional features are related to the communication and synchronization among speculative threads, how speculative threads are spawned and committed, and how the speculative state is kept.

Speculative Architectural Threads have been studied in different platforms such as centralized or monolithic processors and clustered processors. Centralized speculative multithreaded processors are similar to simultaneous multi-threaded processors and almost all the subsystems of the processor are shared among concurrent threads. Sharing has the benefit of low communication latencies and better resource usage, but it may hurt the performance due to resource contention or degradation of the performance of several systems such as cache memories.

On the other hand, clustered processors reduce resource contention by physically distributing contention. However, clustering increases the amount of communication and synchronization latency. Larger clustered structures, such as register files and caches, can help to limit the amount of communications.

In the next subsections, some insights into the most important features of a speculative multithreaded processor, such as the spawning and the committing process, the storage of the speculative state and the management of inter-thread data dependences, are discussed.

13.4.2.1 Spawning Process

A speculative thread may be spawned through a special instruction inserted in the binary by the compiler or when some hardware or run-time system detects a certain event (e.g., a call to a subroutine or a cache miss).

The spawning process includes several non-negligible tasks such as looking for an idle context, computing the logical position of the thread among all the active threads, and initializing the values for the spawned thread.

Assigning a context for a spawned thread is an easy task if there are available contexts. In that case, the new speculative thread is spawned at one of

them. If there are no free contexts, different solutions can be taken: 1) not to spawn the thread, 2) wait for a free context, and 3) squash one thread and spawn the new one at the recently freed context.

Since there may be data dependences among speculative threads and it is possible that values flow from one speculative thread to another, the logical order among threads must be known by the processor. When a speculative thread is created, it is placed in the correct order in the list of threads. The approach to identify the order may depend on the spawning scheme. If we consider a processor where only the most speculative thread is allowed to spawn threads, the placement function is straightforward since the new spawned thread is always the most speculative one. However, in a processor where all active threads are allowed to spawn new threads, it is not as easy. In that case, the processor may implement a thread order predictor based on previous history.

Finally, in order to perform useful work, speculative threads have to be executed with the correct input values. Thus, it is necessary to initialize the register and memory values that are to be used by the spawned thread. If such values are available at the spawning point, they can be directly copied from the spawner thread to the new spawned thread. If the values have to be produced by a preceding thread, they may be predicted as described in later subsections or a synchronization may be required.

13.4.2.2 Committing Process

A thread may finish when it finds a special instruction or when it reaches the thread start point of any other active thread. When this occurs, this thread stops fetching instructions and stalls until its control and data speculation is verified. That is, it waits until this thread becomes the non-speculative.

Thus, when the non-speculative thread reaches the thread start point of any other active thread, it verifies that it corresponds to the next thread in sequential program order. If this is not the case, an order misspeculation has occurred. This situation may have been caused by a value that was produced by a younger thread being incorrectly forwarded to an older one. In addition to the thread order, all the input values used by a thread have to be validated (i.e., those consumed but not produced by itself). The verification of the input values can be done either when the non-speculative thread has reached the thread start point, or can be done on-the-fly as values are produced.

The validation process should include the comparison of all register and memory values of the speculative thread when it was spawned and the final values of the non-speculative thread. In fact, it is only necessary to check that the input values that have been consumed by the speculative thread match with the final values.

In case of misspeculations, the speculative thread is cancelled and the produced state must be discarded. A recovery mechanism is needed in order to roll back the processor to a safe state, which can be implemented through the

same techniques used to recover from other types of speculation such as branch or value missprediction. In general, either full squash of the misspecified thread and its successors or a selective reissue scheme can be implemented.

Note that, when a misspeculation is detected, all threads more speculative than the offending one may also incur in misspeculations. Then, recovery actions in all threads more speculative than the offending one are required.

13.4.2.3 Storage for Speculative State

The fact that all threads work on the same register and memory space implies that each variable may have potentially a different value for each of the threads running concurrently. This requires a special organization of the register file [2, 17] and the memory hierarchy [11, 13, 17, 32, 29], which represents the main added complexity of this paradigm.

In both cases, two solutions may be considered: 1) separate structures for each thread (physical partitioning), that is, different register files or different caches for the speculative threads; and 2) a large shared structure but with separate locations for the different versions (logical partitioning), that is, a large register file and separate map tables or a large cache that supports multiple values for every location. In the first case, the access time is usually lower but communicating values is more costly. On the other hand, sharing reduces the communication latency but affects the scalability and increases the access time.

13.4.2.4 Dealing with Inter-Thread Dependences

Ideally, speculative threads should be both control and data independent. In this way, the concurrent execution of these speculative threads would not require any communication/synchronization. However, as it is expected, such kind of speculative threads are hard to find for many applications. Therefore, speculative multithreaded processors have to provide mechanisms to deal with inter-thread dependences. The way such dependences are managed will strongly affect the performance of such processors.

A speculative thread is data dependent on a previous thread if it consumes any value produced by it. Thus, an inter-thread data misspeculation may occur when a speculative thread reads a variable before it is produced by the previous thread.

The values that are consumed by a speculative thread are normally referred to as thread inputs or thread live-ins. Those thread live-ins that are not available when the thread is spawned are those that are critical for performance.

Dealing with inter-thread data dependences includes several activities: 1) identifying the thread live-ins of a speculative thread; 2) providing the dependent value to the consumer thread; and 3) detecting when a speculative thread has consumed a wrong value.

The identification of the thread live-ins is relatively straightforward for register values and can be easily performed with compiler or run-time techniques.

However, it is more difficult for memory values. Compilers usually are unable to statically determine the memory locations accessed and often take a conservative approach. To predict thread memory live-ins, data dependence prediction techniques such as those explained in Chapter 9 can be used.

Regarding how speculative threads obtain the dependent values, we can distinguish two different families of solutions: 1) synchronization mechanisms, and 2) value prediction schemes. These families will also impact on how misspeculations are detected. Obviously, a hybrid scheme that combines both can also be considered.

Synchronization techniques stall the execution of dependent instruction of a speculative thread until the value is forwarded from the producer thread. This scheme requires identification of the producer instruction of each thread live-in and hardware support for forwarding the dependent values from the producer to the consumer.

The simplest synchronization scheme is to stall the execution of a speculative thread until all live-ins are produced and validated. These schemes have been shown to significantly constrain the performance that can be obtained through a Speculative Architectural Thread architecture since, in most cases, a thread cannot know all the values of its inputs until all previous threads have practically completed [24]. Code reorganization may help by moving up the producer instructions in the producer thread and moving down the consumer instructions in the consumer thread [33].

In order to obtain some benefit, synchronization schemes should accurately identify when dependent values can be forwarded in order to avoid unnecessarily stalls in the speculative threads. Similarly to the identification of the thread live-ins, identifying when a register value is available to be forwarded is relatively easy to do statically, but harder for memory values. In that case, data dependence prediction techniques may also help.

In the case of inter-thread memory dependences, synchronization mechanisms based on data dependence speculation have been widely used. Hardware examples of that for speculative architectural threads are Address Resolution Buffer for Multiscalar architectures [11], the Speculative Versioning Cache [13], the Memory Disambiguation Table [17], the Thread-Level Data Speculation [32] among others. On the other hand, there are some software approaches to deal with inter-thread data dependences [29] that consists of associating to each shared variable a data structure to allow multiple threads to access it.

For synchronization mechanisms the performance we can expect from Speculative Architectural Threads is limited by the critical path of the data dependence graph. In order to further boost performance, a mechanism that breaks the serialization imposed by data dependences is required. Thus, the second family of techniques to deal with data dependences is based on value prediction [19, 30]. Data value speculation is a technique that can relieve the cost of serialization caused by data dependences and boost the performance of superscalar processors as it has been shown in Chapter 9. Predicting the

input/output operands of instructions before they are available allows the processor to start the execution of those instructions and their dependent ones speculatively. The study in [12] concludes that the potential of that technique in platforms such as multithreaded processors is very promising.

Both hardware and software data predictors have been used so far for speculative multithreaded processors. Hardware schemes use similar type of predictors to those proposed for superscalar (see Chapter 9), though predictors specialized for multithreading have also been proposed [23]. A different approach is to perform the prediction in software. Distilled programs are an example on this type of value prediction [40]. The idea is to include in the binary some code that quickly computes the live-ins. This code is executed before the speculative thread starts.

The detection of misspecified predicted values requires storing the value obtained by the prediction in order to compare them with the current ones once they are available.

13.5 Some Speculative Multithreaded Architectures

There are several proposals in the literature for speculative multithreaded architectures:

For instance, examples of Helper Threads that build slices from the applications are Data-Driven Multithreaded Architecture and the Backward Slices [28, 38, 39], the Delinquent loads [8, 18], and Luk's work [20] among others. On the other hand proposals where helper threads are not built from the applications are the Simultaneous Subordinate Microthreading [3].

For Speculative Architectural threads, pioneer work was the Multiscalar [10, 31]. After that, other schemes have been proposed such the Superthreaded [33], the SPSM architecture [9], the Dynamic Multithreaded Processor [1], and the Clustered Speculative Multithreaded processors [21, 23] among others. Moreover, schemes for exploiting speculative thread-level parallelism on a on-chip multiprocessor such as the Atlas [6, 7], the Stanford's Hydra [4, 14, 15, 27], the IACOMA project [17, 5], the Agassiz project [33, 16], the STAMPede project [32], and the Chalmers University project [29, 36] among others have been proposed.

13.6 Concluding Remarks

A shift on the main paradigm exploited by microprocessors has occurred in the past about every decade. During the 80's, RISC was the main paradigm, followed by superscalar in the 90's, and multithreaded in the current decade. Speculative Multithreaded processors may become the main paradigm for the next decade. Speculative multithreading is targeted to exploit thread-level parallelism in order to increase performance. Thread-level parallelism is not new, but the novelty comes from exploiting it in a speculative way. This opens a lot of new opportunities in terms of performance potential and challenges in terms of complexity and power. In this chapter we have reviewed some of the most relevant schemes proposed so far. However, speculative multithreading still requires significant research efforts before becoming a mainstream technology.

Two main families of speculative multithreaded processors have been investigated so far: Helper Threads and Speculative Architectural Threads. The former requires a much simpler support at the microarchitectural level but the latter has a much higher potential to increase performance. Each one represents a different trade-off between cost and performance that may be attractive for different workloads and/or market segments.

A main challenge to be further investigated is the scheme to identify and spawn speculative threads, which can rely on compiler technology, run-time schemes, or a combination of both. Another important aspect to further investigate is the approach to dealing with inter-thread dependences. In the Speculative Architectural Threads model, another important challenge is the design of a complexity-effective memory organization for managing the multiple versions of each variable, some of them speculative, that are alive simultaneously. Solutions to these challenges already exist but these are the key areas where more innovation is required in order for this paradigm to become a mainstream technology.

Overall, the various schemes proposed so far suggest that speculative multithreading may be the next paradigm to keep performance growth on the Moore's curve.

References

- [1] Akkary, H. and Driscoll, M.A. A dynamic multithreading processor. In Proc. of the 31st Int. Symp. on Microarchitecture, 1998.

- [2] Breach, S., Vijaykumar, T.N. and Sohi, G.S. The anatomy of the register file in a multiscalar processor. In Proc. of the 25th Int. Symp. on Computer Architecture, pp. 181-190, 1994.
- [3] Chappel, R.S. et al. Simultaneous subordinate microthreading (SSMT). In Proc. of the 26th Int. Symp. on Computer Architecture, pp. 186-195, June 1999.
- [4] Chen, M. and Olukotun, K. TEST: A tracer for extracting speculative threads. In Proc. of the Int. Symp on Code Generation and Optimizations, pp. 301-312, 2003.
- [5] Cintra, M., Martinez, J.F. and Torrellas, J. Architectural support for scalable speculative parallelization in shared-memory systems. In Proc. of the 27th Int. Symp. on Computer Architecture, 2000.
- [6] Codrescu, L. and Wills, D. On dynamic speculative thread partitioning and the MEM-slicing algorithm. In Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques, pp. 40-46, 1999.
- [7] Codrescu, L., Wills, D. and Meindl, J. Architecture of atlas chip-multiprocessor: dynamically parallelizing irregular applications. IEEE Transaction on Computers, vol 50(1), pp. 67-82, January 2001.
- [8] Collins, J.D. et al. speculative precomputation: Long range prefetching of delinquent loads. In Proc. of the 28th Int. Symp. on Computer Architecture, 2001.
- [9] Dubey, P.K. et al. Single-program speculative multithreading (SPSM) architecture: Compiler-assisted fine-grain multithreading. In Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques, pp. 109-121, 1995.
- [10] Franklin, M. and Sohi, G.S. The expandable split window paradigm for exploiting fine grain parallelism. In Proc. of the 19th Int. Symp. on Computer Architecture, pp. 58-67, 1992.
- [11] Franklin, M and Sohi, G.S. ARB: A hardware mechanism for dynamic reordering of memory references. IEEE Transaction on Computers, vol 45(6), pp. 552-571, May 1996.
- [12] González, J. and González, A. The potential of data value speculation to boost ILP. In Proc. of the 12th Int. Conf. on Supercomputing, pp. 21-28, 1998.
- [13] Gopal, S. et al. Speculative versioning cache. In Proc. of the 4th Int. Symp. on High Performance Computer Architecture, 1998.
- [14] Hammond, L. et al. The Stanford Hydra CMP. Micro IEEE, vol. 20 (2), pp. 6-13, March 2000.

- [15] Hammond, L., Willey, M. and Olukotun, K. Data speculation support for a chip multiprocessor. Proc. of the 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, 1998.
- [16] Kazi, I.H. and Lilja, D.J. Coarse-grained speculative execution in shared-memory multiprocessors. In Proc. of the 12th Int. Conf. on Supercomputing, pp. 93-100, 1998.
- [17] Krishnan, V. and Torrellas, J. Hardware and software support for speculative execution of sequential binaries on a chip multiprocessor. In Proc. of the 12th Int. Conf. on Supercomputing, pp. 85-92, 1998.
- [18] Liao, S.S. et al. Post-pass binary adaptation for software-based speculative precomputation. In Proc. of the Int. Symp. of Programming Languages, Design and Implementation, 2002.
- [19] Lipasti, M.H., Wilkerson, C.B and Shen, J.P. Value locality and load value prediction. In Proc. of the 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, pp. 138-147, 1996.
- [20] Luk, C. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In Proc. of the 28th Int. Symp. on Computer Architecture, pp. 40-51, 2001.
- [21] Marcuello, P., Tubella, J. and González, A. Speculative multithreaded processors. In Proc. of the 12th Int. Conf. on Supercomputing, pp. 76-84, 1998.
- [22] Marcuello, P. and González, A. Clustered speculative multithreaded processors. In Proc. of the 13th Int. Conf. on Supercomputing, pp. 365-372, 1999.
- [23] Marcuello, P. and González, A. Value prediction for speculative multithreaded architectures. In Proc. of the 32nd Int. Symp. on Microarchitecture, pp. 230-236, November 1999.
- [24] Marcuello, P. and González, A. A quantitative assessment of thread-level speculation techniques. In Proc. of the 15th Int. Symp. on Parallel and Distributed Processing, 2000.
- [25] Marcuello, P. and González, A. Thread spawning schemes for speculative multithreaded architectures. In Proc. of the 8th Int. Conf. on High Performance Computer Architecture, 2002.
- [26] Marr, T. et al. Hyper-threading technology architecture and microarchitecture. Intel Technology Journal, vol. 6(1), 2002.
- [27] Oplinger, J., Heine, D. and Lam, M. In search of speculative thread-level parallelism. In Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques, pp. 303-313, 1999.

- [28] Roth, A. and Sohi, G.S. Speculative data-driven multithreading. In Proc. of the 7th Int. Symp. on High Performance Computer Architecture, pp.37-48, 2001.
- [29] Rundberg, P. and Stenstrom, P. A low overhead software approach to thread-level data dependence on multiprocessors. Chalmers University, Sweden, TR-00-13, July 2000.
- [30] Sazeides, Y. and Smith, J.E. The predictability of data values. In Proc. of the 30th Int. Symp. on Microarchitecture, 1996.
- [31] Sohi, G.S., Breach, S. and Vijaykumar, T.N. Multiscalar processors. In Proc. of the 22nd Int. Symp. on Computer Architecture, pp. 414-425, 1995.
- [32] Steffan, J. and Mowry, T. The potential of using thread-level data speculation to facilitate automatic parallelization. In Proc. of the 4th Int. Symp. on High Performance Computer Architecture, pp. 2-13, 1998.
- [33] Tsai, J.Y. and Yew, P-C. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques, pp. 35-46, 1996.
- [34] Tullsen, D.M., Eggers, S.J. and Levy, H.M. Simultaneous multithreading: Maximizing on-chip parallelism. In Proc. of the 22nd Int. Symp. on Computer Architecture, pp. 392-403, 1995.
- [35] Vijaykumar, T.N. Compiling for the multiscalar architecture. Ph.D. Thesis, University of Wisconsin at Madison, 1998.
- [36] Warg, F. and Stenstrom, P. Limits on speculative module-level parallelism in imperative and object-oriented programs on CMP platforms. In Proc. of the Int. Conf. on Parallel Architectures and Compilation Techniques, pp. 221-230, 2001.
- [37] Weiser, M. Program slicing. IEEE Transaction on Software Engineering, vol. 10(4), pp. 352-357, 1984.
- [38] Zilles, C.B. and Sohi, G.S. Understanding the backward slices of performance degrading instructions. In Proc. of the 27th Int. Symp. on Computer Architecture, pp. 172-181, 2000.
- [39] Zilles, C.B. and Sohi, G.S. Execution-based prediction using speculative slices. In Proc. of the 28th Int. Symp. on Computer Architecture, pp. 2-13, 2001.
- [40] Zilles, C.B. and Sohi, G.S. Master/slave speculative parallelization. In Proc. of the 35th Int. Symp. on Microarchitecture, pp. 85-96, 2002.

Chapter 14

Exploiting Load/Store Parallelism via Memory Dependence Prediction

Andreas Moshovos

University of Toronto

14.1	Exploiting Load/Store Parallelism	357
14.2	Memory Dependence Speculation	361
14.3	Memory Dependence Speculation Policies	363
14.4	Mimicking Ideal Memory Dependence Speculation	365
14.5	Implementation Framework	369
14.6	Related Work	374
14.7	Experimental Results	374
14.8	Summary	387
	References	389

Since memory reads or loads are very frequent, memory latency, that is the time it takes for memory to respond to requests, can impact performance significantly. Today, reading data from main memory requires more than 100 processor cycles while in “typical” programs about one in five instructions reads from memory. A naively built multi-GHz processor that executes instructions sequentially would thus have to spend most of its time simply waiting for memory to respond. The overall performance of such a processor would not be noticeably better than that of a processor that operated with a much slower clock (in the order of a few hundred MHz). Clearly, increasing processor speeds alone without at the same time finding a way to make memories respond faster makes no sense. Ideally, the memory latency problem would be attacked directly. Unfortunately, it is practically impossible to build a large, fast and cost effective memory. While it is presently impossible to make memory respond fast to all requests, it is possible to make memory respond faster to some requests. The more requests it can process faster, the higher the overall performance. This is the goal of traditional memory hierarchies where a collection of faster but smaller memory devices (commonly referred to as caches) is used to provide faster access to a dynamically changing subset of memory data. Given the limited size of caches and imperfections in the caching policies, memory hierarchies provide only a partial solution to

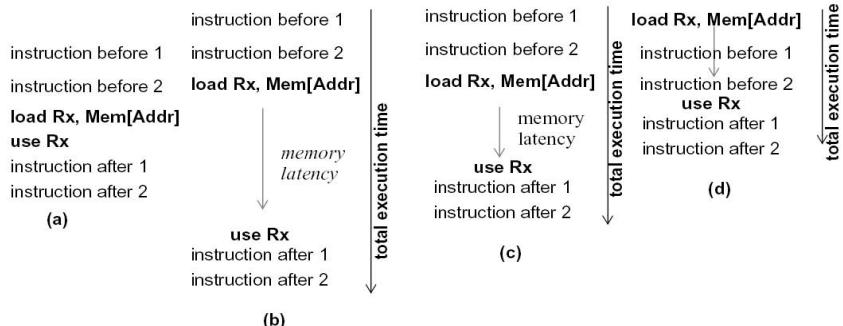


FIGURE 14.1: (a) Code sequence that reads and uses memory data. (b) Execution timing showing how memory latency impacts execution time with simple, in-order execution. (c) Making memory latency shorter reduces execution time. (d) Sending the load request earlier reduces wait time: memory processing is overlapped with other useful work.

the memory latency problem. Figure 14.1 parts (a) through (c) illustrate how memory latency impacts performance.

An alternative, yet orthogonal way of attacking the memory latency problem seeks to tolerate memory latency. The goal here is to send loads to memory earlier, as far in advance from the instructions that need the memory data in effect overlapping the memory request with other useful work. With this technique, memory still takes the same amount of time to respond, but because the data is needed later, to the consuming instructions it appears that memory is faster (i.e., they have to wait for less time or not at all). Sending loads to memory as early as possible requires moving loads up in the execution order, placing them in a position that might be different than the one implied by the program. Figure 14.1 part (d) shows an example of this technique. Because memory writes (stores) are also fairly frequent (about one in ten instructions is a store) moving loads up the execution order often requires moving them before otherwise preceding stores. To do so we have to be able to extract and exploit *load/store parallelism* and to execute instructions *out-of-order*. A load can execute before a preceding store if they access different memory locations. Techniques for exploiting instruction level parallelism have been developed since the 60s. However, these techniques work only for instructions that access registers and not memory. Extracting load/store parallelism is much harder since the memory locations accessed are not fixed at compile time as are the register accesses per instruction.

In this chapter we review previously proposed static and dynamic methods of executing memory operations out-of-order, and demonstrate that higher performance is possible if *memory dependence speculation* is used. In memory dependence speculation, a load may execute before a preceding store on which

it may be data dependent (i.e., the store may be writing the data needed by the load). We discuss the trade-offs involved in using memory dependence speculation and explain that care must be taken to balance the benefits of correct speculation against the net penalty incurred by erroneous speculation. We demonstrate that as dynamically scheduled ILP processors are able to schedule instructions over larger regions, the net performance loss of erroneous memory dependence speculation (misspeculation) can become significant.

The rest of this chapter is organized as follows: in Section 14.1 we motivate the need for exploiting load/store parallelism and discuss the challenges raised by ambiguous (i.e., temporarily unknown) memory dependences. We briefly discuss static and hybrid methods for exploiting load/store parallelism in Sections 14.1.1 and 14.1.2 respectively. We explain the difficulties raised in exploiting load/store parallelism dynamically in Section 14.1.3. We use this discussion to motivate memory dependence speculation which we discuss in Section 14.2. Here we review how memory dependence speculation is being used today and provide qualitative arguments on why techniques to improve the accuracy of memory dependence speculation might be useful. In Section 14.3 we discuss a number of memory dependence speculation policies and argue for memory dependence speculation and synchronization (speculation/synchronization for sort), a policy that aims at mimicking what is ideally possible. In Section 14.4, we discuss the requirements of speculation/synchronization. In Section 14.5 we present an implementation framework for our proposed technique. We review related work in Section 14.6. We provide experimental evidence in support of the utility of our proposed technique and of our observations in Section 14.7. Finally, we summarize our findings in Section 14.8.

14.1 Exploiting Load/Store Parallelism

The most widespread programming model in use today assumes sequential execution semantics. In sequential execution programs are written with an implied, total order where instructions are meant to execute one after the other and in the order specified by the program. Even parallel programs assume sequential execution semantics within individual threads. However, closer inspection of sequential execution semantics reveals that there are many execution orders that produce exactly the same results. Key in determining whether a particular execution order will produce valid results is the discovery of *dependences* (see [16] for detailed explanation of dependence types). Of particular importance are true (read-after-write or RAW) dependences. A true dependence exists between two instructions A and B, if B reads the data produced by A. Whenever a true dependence exists between two instructions

they must execute in the implied order;¹ otherwise they can execute in any order, possibly in parallel. This ability is also useful in tolerating slower memory devices by overlapping the processing of load requests with other useful computation. Moving loads as far ahead of the instructions that read their data, and in general exploiting instruction-level parallelism can be done statically, dynamically or via a combination of both static and dynamic methods.

14.1.1 Static Methods

The first class of methods for exploiting load/store parallelism relies on static rearrangement or scheduling of memory accessing instructions during compile time. The key difficulty here is knowing whether it is safe to move a load earlier in the schedule and specifically before an initially preceding store. A load should not be scheduled before a store if it is going to access the same address. At the core of all software-based load scheduling techniques are *static disambiguation* or *alias analysis* techniques. The goal of these methods is to determine whether a given load and store will be data independent during execution. Many techniques have been proposed. Initially research focused primarily on array variables [3],[9],[4], while recently methods have been proposed for dynamically allocated data types [10], [42]. With these methods, a load would be scheduled before an initially preceding store only if it can be proven that the two instructions will always be independent. Pointers and the inconvenience of multipass compiling (alias analysis across functions declared in separate files requires whole program analysis) are the two most important limitations of such static methods.

It was observed that while many loads are independent of preceding stores it was not possible to prove (with existing methods) this independence during compile time. To allow such loads to execute early Nicolau proposed *runtime disambiguation* [33], a software only approach to memory dependence speculation. In his technique, loads can be speculatively scheduled before a preceding store without knowing for sure that no dependence exists. Code is inserted after the store to detect whether a true dependence is violated (this is done by comparing the addresses accessed by the store and the load), and repair code is also inserted to recover from memory dependence violations. Another software only approach was proposed by Moudgil and Moreno [32]. Their approach differs from Nicolau's in that they compare values rather than addresses to detect violation of program semantics. Code bloat and the execution overhead of checking and repairing in case of a violation are important considerations with these techniques. Figure 14.2 shows an example of runtime disambiguation.

¹Strictly speaking, program semantics are maintained so long as instructions read the same value as they would in the original program implied order. This does not necessarily imply that a dependent pair of instructions executes in the program implied order. We avoid making this distinction in the discussion of this chapter for clarity.

```

...
store Ry, Mem[Addr1]
...
load Rx, Mem[Addr2]
use Rx
...
(a)                                     (b)

load Rx, Mem[Addr2]
...
store Ry, Mem[Addr1]
...
if (Addr2 != Addr1) goto OK
load Rx, Mem[Addr2]
OK: use Rx
...

```

FIGURE 14.2: Run-time disambiguation. (a) Original code with load following a store. (b) Compiled code where the load is scheduled before the store. Additional code (shown in bold) is needed to check whether a dependence was violated. If so the load is re-executed to read the correct data (that stored by the store).

14.1.2 Hybrid Static/Dynamic Methods

In purely static speculation, any benefits gained by scheduling loads early had to be balanced against the execution overhead of checking for memory dependence violations. A major advantage of static speculation techniques however is that they require no hardware support. The next class of memory dependence speculation techniques introduces minimal additional hardware in order to reduce the checking code overhead. Specifically, Gallagher, Chen, Mahlke, Gyllenhaal and Hwu [13], [6] proposed the *Memory Conflict Buffer* (MCB), a software-hardware hybrid approach. In their technique, load motion and misspeculation recovery are done in software while misspeculation detection is done in hardware. Two copies of each speculated load are executed, one at the original program order (non-speculative) and the other as early as desired (speculative). Speculative loads record their addresses in the MCB. Intervening stores also post their addresses to the MCB, so that dependence violations can be detected. The non-speculative load checks the appropriate entry in the MCB (the target register of the load is used as a handle), and if any dependence was violated, control is transferred to recovery code. Figure 14.3 illustrates how the MCB method works. MCB-like solutions have been implemented in Transmeta’s Crusoe processors [8] and in the Intel/HP EPIC architecture [28]. Huang, Slavenburg and Shen proposed *speculative disambiguation* [19], another hybrid approach to memory dependence speculation. In their technique multiple versions of the same code are generated, one with speculation enabled and another with speculation disabled. These versions are then scheduled together using predication. Hardware similar to that used for *boosting* [37] is used to invalidate all but the appropriate path during execution.

14.1.3 Dynamic Methods

Typical modern dynamically scheduled ILP processors exploit instruction-level parallelism by forging ahead into the execution stream, building an in-

```

load.speculative Rx, Mem[Addr2]
...
store Ry, Mem[Addr1]
...
check.speculative Rx, OK
load Rx, Mem[Addr2]
OK: use Rx
...

```

FIGURE 14.3: The memory conflict buffer approach to memory dependence speculation. A load is scheduled for execution before an originally preceding store. A special load instruction, `load.speculative`, is used. Upon its execution, and using its target register Rx as a handle an entry is allocated in the memory conflict buffer. The entry records the address accessed by the `load.speculative` (Addr1 in our example). When the store executes, it posts its address to the memory conflict buffer. If a matching entry is found (that is if Addr1 equals Addr2) a flag is set in the corresponding entry. Later, a `check.speculative` instruction is executed where one of the arguments is the same register used as the target by the preceding `load.speculative`. The `check.speculative` instruction is essentially a conditional branch that succeeds if the corresponding memory conflict buffer entry has its flag clear. This means that no store after the `load.speculative` has accessed the same address and thus the `load.speculative` did not violate a memory dependence. If the flag is set, then a memory dependence violation occurred and the memory read must be repeated.

struction window, a set of instructions to execute. These processors then attempt to convert the total, program implied order within this set into a partial order. The shape of the partial order and for that the performance improvements so obtained are heavily influenced by the processors' ability to uncover the true data dependences among the instructions currently under consideration. In the case of loads, the performance improvements obtained are determined by the processors' ability to send load requests to memory as early as possible without, however, allowing a load to access memory before a preceding store with which a true data dependence exists. One way of doing so is to first determine the true dependences a load has and then use that information to schedule its execution. With this approach, we ensure that no true dependences are violated in the resulting execution order. In the case of loads and stores the process of determining the data dependences they have is commonly referred to as *disambiguation*.

Determining the data dependences among the instructions in the instruction window requires inspection of the addresses they access. Unfortunately, these addresses are not necessarily available immediately. This is typical for stores and loads which have to perform an address calculation. As a result, at any point during execution, memory dependences may be *unambiguous* (i.e., a load consumes a value that is known to be created by a store preceding it in the total order) or *ambiguous* (i.e., a load consumes a value that *may* be

produced by a store preceding it in the total order). During execution, an ambiguous dependence gets eventually resolved to either a true dependence, or to no dependence. We will use the term *false dependence* to refer to an ambiguous dependence that eventually gets resolved to no dependence. As we explain next, false dependences present a challenge to the out-of-order execution of load instructions.

Ambiguous memory dependences may obscure some of the parallelism that is present. To maintain program semantics a load has to wait for a store with which an ambiguous dependence exist only if a dependence really exists. If the ambiguous dependence is a false dependence, any execution order is permissible, including ones that allow the load to execute before the store. This latter case represents an opportunity for parallelism and for higher performance. Unfortunately, the mere classification of a dependence as ambiguous implies the inability to determine whether a true dependence exists without actually waiting for the addresses accessed by both instructions to be calculated. Worse, in the absence of any explicit memory dependence information (the common case today), a dynamically scheduled ILP processor has to assume that ambiguous dependences exist among a load and any preceding store that has yet to calculate its address (provided that no intervening store accesses the same address and has calculated its address).

As we will demonstrate in the evaluation section, significantly higher performance is possible if we could make loads wait only for those ambiguous dependences that get resolved to true dependences. Moreover, we demonstrate that this performance difference widens as the size of the instruction window increases. To expose some of the parallelism that is hindered by ambiguous memory dependences, memory dependence speculation can be used. This technique is the topic of the next section.

14.2 Memory Dependence Speculation

Memory dependence speculation aims at exposing the parallelism that is hindered by ambiguous memory dependences. Under *memory dependence speculation*, we do not delay executing a load until *all* its ambiguous dependences are resolved. Instead, we guess whether the load has any true dependences. As a result, a load may be allowed to obtain memory data speculatively before a store on which it is ambiguously dependent executes. Eventually, when the ambiguous dependences of the load get resolved, a decision is made on whether the resulting execution order was valid or not. If no true dependence has been violated, speculation was successful. In this case, performance may have improved as the load executed earlier than it would had it had to wait for its ambiguous dependences to be resolved. Howev-

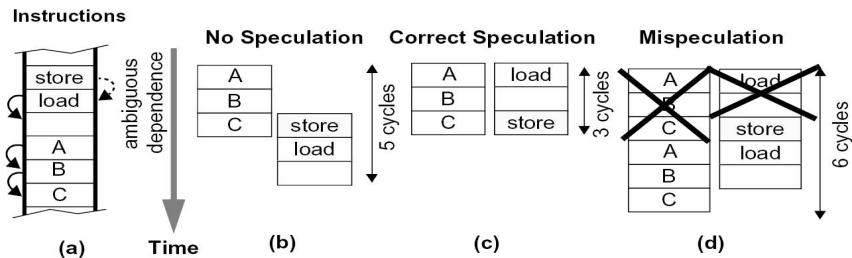


FIGURE 14.4: Using memory dependence speculation may affect performance either way. (a) Code with an ambiguous memory dependence. Continuous arrows indicate register dependences. Parts (b) through (d) show how this code may execute in a dynamically scheduled ILP processor capable of executing two instructions per cycle. We assume that due to other dependences, the store may execute only after two cycles have passed. (b) Execution order when no memory dependence speculation is used. (c) Memory dependence speculation is used and the ambiguous dependence gets resolved to no dependence. (d) Memory dependence speculation is used, and the ambiguous dependence gets resolved to a true dependence.

er, if a true dependence was violated, the speculation was erroneous (i.e., a misspeculation). In the latter case, the effects of the speculation must be undone. Consequently, some means are required for detecting erroneous speculation and for ensuring correct behavior. Several mechanisms that provide this functionality, in either software and/or hardware, have been proposed [38],[11],[12],[13],[20],[26],[33],[36]. The hardware techniques used today work by invalidating and re-executing all instructions following the misspecified load. We will use the term *squash invalidation* to refer to this recovery method.

Though memory dependence speculation may improve performance when it is successful, it may as well lead to performance degradation when it is wrong. We demonstrate either possibility with the example of Figure 14.4. The reason is that a penalty is typically incurred on misspeculation. The penalty includes the following three components: (1) the work thrown away to recover from the misspeculation, which, in the case of squash invalidation, may include unrelated computations, (2) the time, if any, required to perform the invalidation, and finally (3) the opportunity cost associated with not executing some other instructions instead of the misspecified load and the instructions that used erroneous data. Consequently, in using memory dependence speculation care must be taken to balance the performance benefits obtained when speculation is correct against the net penalty incurred by erroneous speculation. To gain the most out of memory dependence speculation we would like to use it as aggressively as possible while keeping the net cost

of misspeculation as low as possible. Ideally, loads would execute as early as possible while misspeculations would be completely avoided.

The first dynamically scheduled processors did not use memory dependence speculation because in their relatively small instruction windows (10-20 instructions) there was often little to be gained from extracting load/store parallelism. When instruction windows became larger (few tens of instructions) *naive memory dependence speculation* was used where a load was executed immediately after its address was calculated. Naive memory dependence speculation was very appropriate since the probability of a true memory dependence being violated within the relatively small instruction windows was very small. In most cases, naive memory dependence speculation offers superior performance compared to having to wait until ambiguous dependences are resolved (i.e., no speculation). Moreover, the benefits of memory dependence speculation increase as the size of the instruction window also increases. More importantly, further performance improvements are possible if we could avoid misspeculations. The latter is possible via more accurate memory dependence speculation methods.

14.3 Memory Dependence Speculation Policies

The ideal memory dependence speculation mechanism not only avoids misspeculations completely, but also allows loads to execute as early as possible. That is, loads with no true dependences (within the instruction window) execute without delay, while loads that have true dependences are allowed to execute only after the store (or the stores) that produces the necessary data has executed. It is implied that the ideal memory dependence speculation mechanism has perfect knowledge of all the relevant memory dependences.

An example of how the ideal memory dependence speculation mechanism affects execution is shown in Figure 14.5. In part (b), we show how the code sequence of part (a) may execute under ideal memory dependence speculation and in part (c) we show how the execution may progress under naive memory dependence speculation. The example code sequence includes two store instructions, ST-1 and ST-2, that are followed by two load instructions, LD-1 and LD-2. Ambiguous dependences exist among these four instructions as indicated by the dotted arrows. During execution, however, only the dependence between ST-1 and LD-1 is resolved to a true dependence (as indicated by the continuous arrow). Under ideal dependence speculation, LD-2 is executed without delay, while LD-1 is forced to synchronize with ST-1.

In contrast to what is ideally possible, in a real implementation, the relevant memory dependences are often unknown. Therefore, if we are to mimic the ideal memory dependence speculation mechanism, we have to attempt: (1)

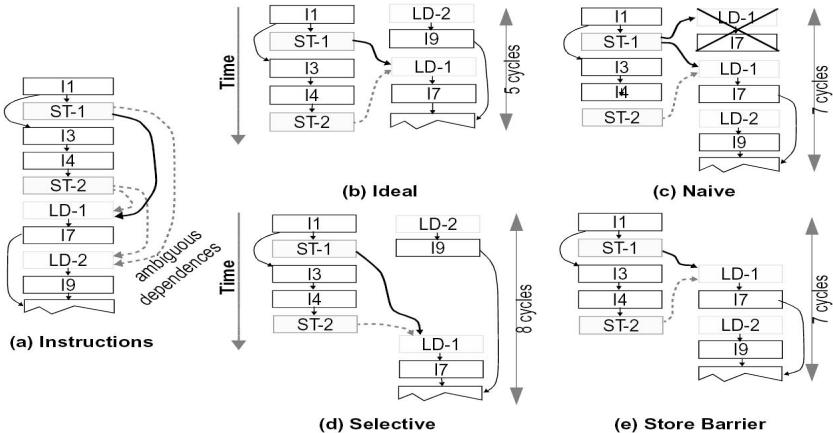


FIGURE 14.5: Example illustrating various memory dependence speculation policies. Arrows indicate dependences. Dependences through memory are indicated by thicker lines. Dotted arrows indicate ambiguous dependences that are resolved to no-dependence during execution.

to predict whether the immediate execution of a load is likely to violate a true memory dependence, and if so, (2) to predict the store (or stores) the load depends upon, and (3) to enforce synchronization between the dependent instructions.

However, since this scheme seems elaborate, it is only natural to attempt to simplify it. One possible simplification is to use *selective memory dependence speculation*, i.e., carry out only the first part of the ideal three-part operation. In this scheme the loads that are likely to cause misspeculation are not speculated. Instead, they wait until all their ambiguous dependences are resolved; explicit synchronization is not performed. We use the term selective memory dependence speculation (or selective speculation for short) to signify that we make a decision on whether a load should be speculated or not. In contrast, in ideal dependence speculation, we make a decision on when is the right time to speculate a load. While selective memory dependence speculation may avoid misspeculations, due to the lack of explicit synchronization, this prediction policy may as well make loads wait longer than they should and for this reason may negatively impact performance. This case we illustrate with the example shown in part (d) of Figure 14.5. In this example, LD-2 is speculated, whereas LD-1 is not, since prediction correctly indicates that LD-2 has no true dependences while LD-1 does. However, as shown LD-1 is delayed more than necessary as it has to wait not only for ST-1 but also for ST-2. In practice, selective data dependence speculation can lead to inferior performance when compared to naive speculation (part (c) of Figure 14.5) even when perfect prediction of dependences is assumed. While this

policy avoids misspeculations it often fails to delay loads only as long as it is necessary.

Another possible simplification, the store barrier has been proposed by Hesson, LeBlanc and Ciavaglia [17],[2]. In this technique a prediction is made on whether a store has a true dependence that would normally get misspecified. If it does, all loads following the store in question are made to wait until the store has posted its address for disambiguation purposes. While the store barrier policy can be successful in (1) eliminating misspeculations, and (2) delaying loads that should wait only as long as it is necessary, it may as well lead to inferior performance since it may unnecessarily delay other unrelated loads that have no true dependences that can be misspecified. While, in the example of Figure 14.5, the store barrier policy is shown to perform better than selective speculation, the opposite can also be true (for example, if other loads following LD-1 existed they would too get delayed under the store barrier policy, while they would not under the selective policy). The store barrier method has been shown to perform worse than other memory dependence speculation methods we present in the next section [7]. However, it is an attractive alternative due to its simplicity.

In the next section, we present dynamic memory dependence speculation/synchronization, a technique that utilizes memory dependence prediction to identify those store-load pairs that ought to be synchronized in order to avoid memory dependence violations while delaying load execution only as long as it is necessary.

14.4 Mimicking Ideal Memory Dependence Speculation

To mimic the ideal data dependence speculation system, we need to implement all the three components of the ideal system as described in the previous section. That is, we must: (1) dynamically identify the store-load pairs that are likely to be dependent and whose normal execution will result in a memory dependence violation, (2) assign a synchronization mechanism to dynamic instances of these dependences, and (3) use this mechanism to synchronize the store and the load instructions.

To identify the store-load pairs that need to be synchronized we may use history-based memory dependence prediction. With this scheme, naive memory dependence speculation is initially used for all loads. A load is initially allowed to execute as soon as its address is calculated and memory resources are available. With this policy, as execution progresses, misspeculations will be encountered. Instead of discarding the information available when a misspeculation occurs (as we would under naive memory dependence speculation), we collect information about the instructions involved. For example,

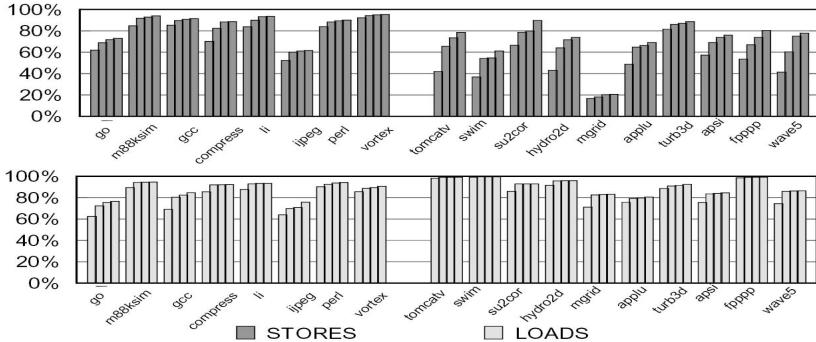


FIGURE 14.6: Memory dependence set locality of read-after-write dependences. Locality range shown is 1 to 4 (left to right).

we may record the static dependence that was violated, that is a (store PC, load PC) pair. The next time a load or a store that has previously incurred a misspeculation is encountered, we can use the recorded information to predict whether synchronization has to take place in order to avoid a misspeculation. In the predictors we present misspeculation history is associated with the static loads and stores using their PC. For history-based memory dependence prediction to be possible, it is imperative that past misspeculation behavior to be indicative of future dependences that ought to be synchronized. Fortunately, this is true of typical programs. This can be seen in Figure 14.6 where we report the dependence set locality for loads and stores in parts (a) and (b) respectively (see Section 14.7.1 for our methodology). We define *dependence set locality (n)* of a load instruction to be the probability that once it experiences a memory dependence then this dependence is within the last n unique dependences it experienced. Dependence set locality is defined similarly for stores. A high dependence set locality for small values of n suggests that loads experience repeatedly the same dependences. In turn this is direct indication that past dependence behavior is indeed strongly correlated to future dependence behavior and thus history-based prediction should be possible. We report locality measurements in the range of 1 to 4 and as a fraction over all executed loads or stores that experience a RAW dependence. Memory dependence locality is very strong. In most programs, nearly 80% or more of all loads with RAW dependences experience the same exact dependence as the last time they were encountered. Locality is also strong for stores but not as strong as it is for loads. This is because store values are typically read more than once; hence stores tend to have more dependences than loads.

With a mechanism to predict whether a load or a store needs to be synchronized we next need: (1) a synchronization mechanism, and (2) a method of having the appropriate dynamic instances of loads and stores locate each other through the synchronization mechanism. In the rest of this section we

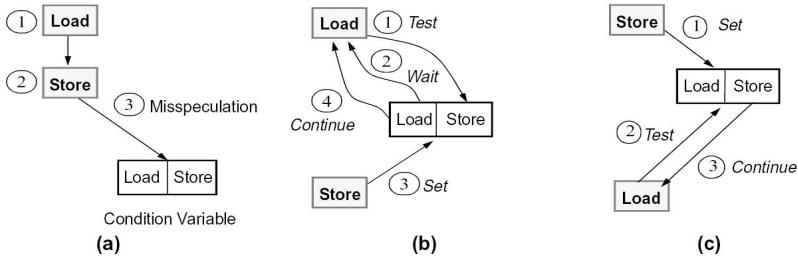


FIGURE 14.7: Dynamic synchronization example.

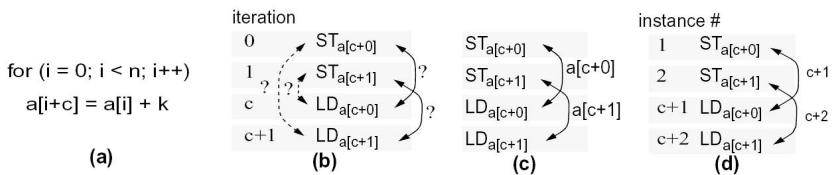


FIGURE 14.8: Example code sequence that illustrates that multiple instances of the same static dependence can be active in the current instruction window. In parts (b), (c), and (d), the relevant store and load instructions from four iterations of the loop of part (a) are shown.

first discuss a synchronization mechanism. Then, we consider how load and store instances locate each other through this synchronization mechanism.

An apt method of providing the required synchronization dynamically is to build an association between the store-load instruction pair. Figure 14.7 illustrates this process. Initially, loads and stores are allowed to execute freely. The first time a misspeculation occurs we build an association between the offending instructions. This dynamic association is a condition variable on which only two operations are defined: *wait* and *signal*, which test and set the condition variable respectively. These operations are logically incorporated into the dynamic actions of the dependent load and store instructions to achieve the necessary synchronization. Next time the same instructions are executed, the previously built association prevents them from executing independently and enforces synchronization.

Some means are required to assign a condition variable to a dynamic instance of a store-load instruction pair that has to be synchronized. If synchronization is to occur as planned, the mapping of condition variables to dynamic dependences has to be unique at any given point of time. One approach is to use just the address of the memory location accessed by the store-load pair as a handle. This method provides an indirect means of identifying the store and load instructions that are to be synchronized. Unless the store location is accessed only by the corresponding store-load pair, the assignment will not be unique.

Alternatively, we can use the dependence edge as a handle. The static dependence edge may be specified using the (full or part of) instruction addresses (PCs) of the store-load pair in question. (Compared to using addresses, a potential advantage of this approach is that PC information is available earlier in the pipeline. This property could be exploited to reduce the effective latency of synchronization by having stores initiate synchronization in parallel or prior to the completion or initiation of their memory access.) Unfortunately, as exemplified by the code sequence of Figure 14.8 part (b), using this information may not be sufficient to capture the actual behavior of the dependence during execution; the pair (PCST, PCLD) matches against all four edges shown even though the ones marked with dotted arrows should not be synchronized. A static dependence between a given store-load pair may correspond to multiple dynamic dependences, which need to be tracked simultaneously.

To distinguish between the different dynamic instances of the same static dependence edge, a tag (preferably unique) could be assigned to each instance. This tag, in addition to the instruction addresses of the store-load pair, can be used to specify the dynamic dependence edge. In order to be of practical use, the tag must be derived from information available during execution of the corresponding instructions. A possible source of the tag for the dependent store and load instructions is the address of the memory location to be accessed, as shown in Figure 14.8 part (c). An alternate way of generating tags is to have a load synchronize with the closest preceding instance of the store identified by the static dependence. While this scheme may delay a load more than it should (as in our example, where LDa[$c+0$] will wait for STa[$0+1$]), the performance impact of this delay may not be large.

For clarity in the discussion that follows we assume the scheme shown in part (d) of Figure 14.8, where dynamic store and load instruction instances are numbered based on their PCs. The difference in the instance numbers of the instructions which are dependent, referred to as the *dependence distance*, is used to tag dynamic instances of the static dependence edge (as may be seen for the example code, a dependence edge between STi and LD $i+distance$ is tagged - in addition to the instruction PCs - with the value $i+distance$). We note that from a practical perspective, several inconveniences exist in the scheme we have just described (for example, how to track and predict multiple dependences per store or load). In the discussion that follows and for clarity, we initially ignore these issues and present an implementation framework in Section 14.5.

14.5 Implementation Framework

In the discussion that follows, we first describe the support structures and then proceed to explain their operation by means of an example. We present the support structures as separate, distinct components of the processor. Other implementations may be possible and desirable. We assume a centralized implementation, ignore the possibility of multiple dependences per load or store, and assume fully-associative structures. In Section 14.5.2, we address these issues. We partition the support structures into two interdependent tables: a *memory dependence prediction table* (MDPT) and a *memory dependence synchronization table* (MDST). The MDPT is used to identify, through prediction, those instruction pairs that ought to be synchronized. The MDST provides a dynamic pool of condition variables and the mechanisms necessary to associate them with dynamic store-load instruction pairs to be synchronized.

MDPT: An entry of the MDPT identifies a static dependence and provides a prediction as to whether or not subsequent dynamic instances of the corresponding static store-load pair will result in a misspeculation (i.e., should the store and load instructions be synchronized). In particular, each entry of the MDPT consists of the following fields: (1) valid flag (V), (2) load instruction address (LDPC), (3) store instruction address (STPC), (4) dependence distance (DIST), and (5) optional prediction (not shown in any of the working examples). The valid flag indicates if the entry is currently in use. The load and store instruction address fields hold the program counter values of a pair of load and store instructions. This combination of fields uniquely identifies the *static* instruction pair for which it has been allocated. The dependence distance records the difference of the instance numbers of the store and load instructions whose misspeculation caused the allocation of the entry (if we were to use a memory address to tag dependence instances this field would not have been necessary). The purpose of the prediction field is to capture, in a reasonable way, the past behavior of misspeculations for the instruction pair in order to aid in avoiding future misspeculations or unnecessary delays. Many options are possible for the prediction field (for example an up-down counter or dependence history based schemes). The prediction field is optional since, if omitted, we can always predict that synchronization should take place. However, we note that in our experimentation we found that it is better if synchronization is enforced only after a load has been misspecified a couple of times (e.g., three times).

MDST: An entry of the MDST supplies a condition variable and the mechanism necessary to synchronize a dynamic instance of a static instruction pair (as predicted by the MDPT). In particular, each entry of the MDST consists of the following fields: (1) valid flag (V), (2) load instruction address (LDPC), (3) store instruction address (STPC), (4) load identifier (LDID),

(5) store identifier (STID), (6) instance tag (INSTANCE), and (7) full/empty flag (F/E). The valid flag indicates whether the entry is, or is not, in use. The load and store instruction address fields serve the same purpose as in the MDPT. The load and store identifiers have to uniquely identify, within the current instruction window, the dynamic instance of the load or the store instruction respectively. These identifiers are used to allow proper communication between the instruction scheduler and the speculation/synchronization structures. The exact encoding of these fields depends on the implementation of the OoO (out-of-order) execution engine (for example, in a superscalar machine that uses reservation stations we can use the index of the reservation station that holds the instruction as its LDID or STID, or if we want to support multiple loads per store, a level of indirection may be used to represent all loads waiting for a particular store). The instance tag field is used to distinguish between different dynamic instances of the same static dependence edge (in the working example that follows we show how to derive the value for this field). The full/empty flag provides the function of a condition variable.

14.5.1 Working Example

The exact function and use of the fields in the MDPT and the MDST is best understood with an example. In the discussion that follows we are using the working example of Figure 14.9. For the working example, assume that execution takes place on a processor which: (1) issues multiple memory accesses per cycle from a pool of load and store instructions and (2) provides a mechanism to detect and correct misspeculations due to memory dependence speculation. For the sake of clarity, we assume that once an entry is allocated in the MDPT it will always cause a synchronization to be predicted.

Consider the memory operations for three iterations of the loop, which constitute the active pool of load and store instructions as shown in part (a) of the figure. Further, assume that `child->parent` points to the same memory location for all values `child` takes. The dynamic instances of the load and store instructions are shown numbered, and the true dependences are indicated as dashed arrows connecting the corresponding instructions in part (a). The sequence of events that leads to the synchronization of the ST2-LD3 dependence is shown in parts (b) through (d) of the figure. Initially, both tables are empty. As soon as a misspeculation (ST1-LD2 dependence) is detected, a MDPT entry is allocated, and the addresses of the load and the store instructions are recorded (action 1, part (b)). The DIST field of the newly allocated entry is set to 1, which is the difference of the instance numbers of ST1 and LD2 (1 and 2 respectively). As a result of the misspeculation, instructions following the load are squashed and must be re-issued. We do not show the re-execution of LD2.

As execution continues, assume that the address of LD3 is calculated before the address of ST2. At this point, LD3 may speculatively access the memory hierarchy. Before LD3 is allowed to do so, its instruction address, its

instance number (which is three), and its assigned load identifier (the exact value of LDID is immaterial) are sent to the MDPT (action 2, part (c)). The instruction address of LD3 is matched against the contents of all load instruction address fields of the MDPT (shown in grey). Since a match is found, the MDPT inspects the entry predictor to determine if a synchronization is warranted. Assuming the predictor indicates a synchronization, the MDPT allocates an entry in the MDST using the load instruction address, the store instruction address, the instance number of LD3, and the LDID assigned to LD3 by the OoO core (action 3, part (c)). At the same time, the full/empty flag of the newly allocated entry is set to empty. Finally, the MDST returns the load identifier to the load/store pool indicating that the load must wait (action 4, part (c)).

When ST2 is ready to access the memory hierarchy, its instruction address and its instance number (which is 2) are sent to the MDPT (action 5, part (d)). The instruction address of ST2 is matched against the contents of all store instruction address fields of the MDPT (shown in grey). Since a match is found, the MDPT inspects the contents of the entry and initiates a synchronization in the MDST. As a result, the MDPT adds the contents of the DIST field to the instance number of the store (that is, $2 + 1$) to determine the instance number of the load that should be synchronized. It then uses this result, in combination with the load instruction address and the store instruction address, to search through the MDST (action 6, part (d)), where it finds the allocated synchronization entry. Consequently, the full/empty field is set to full, and the MDST returns the load identifier to the load/store pool to signal the waiting load (action 7, part (d)). At this point, LD3 is free to continue execution. Furthermore, since the synchronization is complete, the entry in the MDST is not needed and may be freed (action 8, part (d)).

If ST2 accesses the memory hierarchy before LD3, it is unnecessary for LD3 to be delayed. Accordingly, the synchronization scheme allows LD3 to issue and execute without any delays. Consider the sequence of relevant events shown in parts (e) and (f) of Figure 14.9. When ST2 is ready to access the memory hierarchy, it passes through the MDPT as before with a match found (action 2, part (e)). Since a match is found, the MDPT inspects the contents of the entry and initiates a synchronization in the MDST. However, no matching entry is found there since LD3 has yet to be seen. Consequently, a new entry is allocated, and its full/empty flag is set to full (action 3, part (e)). Later, when LD3 is ready to access the memory hierarchy, it passes through the MDPT and determines that a synchronization is warranted as before (action 4, part (f)). The MDPT searches the MDST, where it now finds an allocated entry with the full/empty flag set to full (action 5, part (f)). At this point, the MDST returns the load identifier to the load/store pool so the load may continue execution immediately (action 6, part (f)). It also frees the MDST entry (action 7, part (f)).

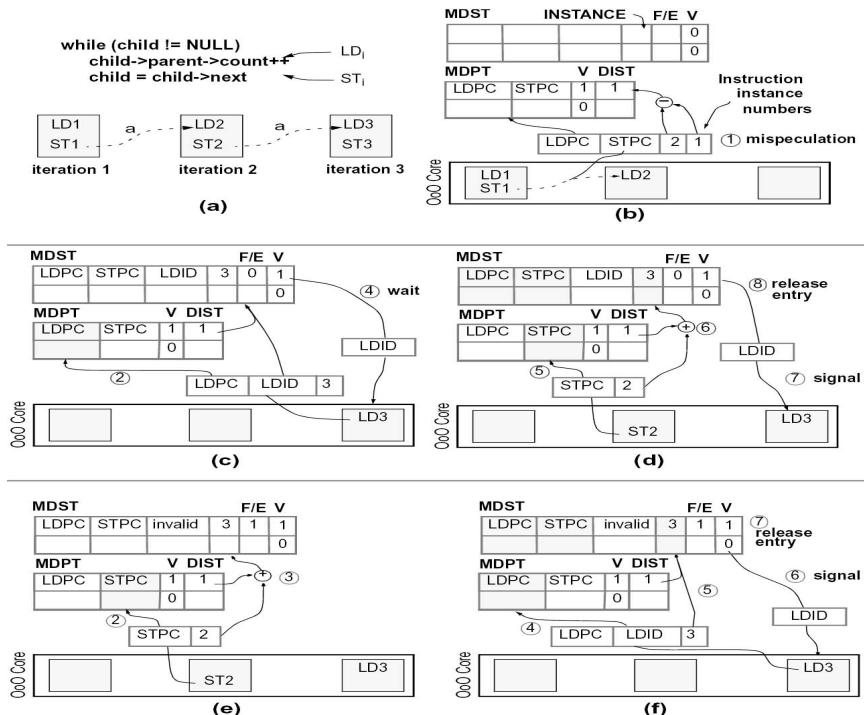


FIGURE 14.9: A mechanism for synchronizing memory dependences.



FIGURE 14.10: Supporting multiple static dependences per load or store: Split MDPT/MDST using a level of indirection to represent dependence sets.

14.5.2 Multiple Dependences per Static Load or Store

Although not illustrated in the example, it is possible for a load or a store to match multiple entries of the MDPT and/or of the MDST. This case represents multiple memory dependences involving the same static load and/or store instructions (for example in the code `if (cond) store1 M else store2 M; load M`, there are two dependences (`store1, load`) and (`store2, load`) which may alternate in the dynamic execution stream. There are three challenges from a practical perspective: (1) how to predict multiple dependences per load, (2) how to allocate multiple MDST entries when multiple dependences are predicted on a single load, and (3) how to wake-up a load forced to wait on multiple MDST entries. One solution would be to define the problem away by tracking only a single dependence per store or load. However, support for multiple dependences per static instruction is very important.

The solution is to use level of indirection to represent the set of all dependences that have a common store or load (for example in the code `if (cond) store1 M; else store2 M; load M`); both the (`store1, load`) and the (`store2, load`) dependences will be represented using a common tag). This approach was suggested in [30],[7]. In this scheme, separate entries for loads and stores are allocated in the MDPT. The format of these entries is shown in part (b) of Figure 14.10. (Note that a split MDPT and MDST is illustrated in the figure.) As shown, in these entries, we do not record the dependences the corresponding instructions have. Instead we use a tag, to which we will refer to as a *synonym*. Synonyms are assigned using a global counter when misspeculations occur. If no synonym has been assigned to either the static load or the static store, a new synonym is generated using the global counter and is assigned to both instructions. If a synonym has been already assigned to only one of the instructions (as the result of misspeculating a different static dependence involving that instruction), the same synonym is assigned to the other instruction. If both instructions already have synonyms assigned to them which are different, the smallest one is assigned to both instructions [7].

14.6 Related Work

Naive memory dependence speculation was proposed for the Multiscalar architecture [11]. Support for dependence misspeculation detection and recovery was proposed in the form of the *Address Resolution Buffer* (ARB) [12] which also implements memory renaming. Other recently proposed techniques to support speculation of memory dependences, memory renaming and memory dependence misspeculation detection are presented in [14], [15]. Naive memory dependence speculation was used in the PA8000 processor [21] and in the Power 620 processor [1],[25]. Knight also proposed using memory dependence speculation along with a hardware-based misspeculation detection mechanism in the context of speculative, parallel execution of otherwise sequential Lisp programs [24].

Finally, several hardware-based techniques have been proposed that aim at improving accuracy over naive memory dependence speculation. There are two closely related proposals. In the first proposal by Steely, Sager and Fite [39], misspecified loads and stores are given tags derived from the addresses via which the misspeculation occur. These tags are used by the out-of-order scheduler to restrict load execution. Hesson, LeBlanc and Ciavaglia [17] describe the *store barrier cache* and the *store barrier* approach. An implementation of the store barrier cache was also presented [2]. The techniques we describe in this chapter were also reported in [29],[31].

Selective speculation is implemented in the Alpha 21264 processor [23] where an independence predictor is used to predict whether a load can execute freely. Finally, Chrysos and Emer proposed using a level of indirection for the purposes of memory dependence speculation/synchronization. In their store set approach, a tag is used to represent the set of all stores associated with a load that encountered a memory dependence misspeculation. They proposed the incremental approach we also utilize to build memory dependence sets. Synchronization takes place through a separate table, and moreover, to preclude ordering problems on dependences with non-unit distances and to attain a simple synchronization table design, stores that have been assigned to the same store set (i.e., synonym) are executed in-order.

14.7 Experimental Results

In this section, we study various methods of extracting load/store parallelism and their interaction with memory dependence speculation under a centralized, continuous instruction window execution model. Specifically: (1) we demonstrate that higher performance is possible if we could extrac-

t load/store parallelism and that the performance improvements are higher when the instruction window is larger (Section 14.7.2). (2) We demonstrate that naive memory dependence speculation can be used to attain some of the performance benefits possible. However, we also demonstrate that the net penalty of misspeculation is significant (Section 14.7.3). (3) We consider using an address-based scheduler to extract this parallelism and show that higher performance is possible compared to naive memory dependence speculation (Section 14.7.4). In an address-based scheduler, loads and stores post their addresses as soon as possible, and loads are allowed to inspect the addresses of preceding stores before obtaining a memory value. (4) We show that performance drops rapidly when inspecting the preceding store addresses due to increases in load execution latency (Section 14.7.4). (5) We demonstrate that using memory dependence prediction to schedule load/store execution instead of address-based scheduling offers performance similar to that possible had we had perfect in-advance knowledge of all memory dependences (Section 14.7.5).

We note that the various load/store execution models we consider in this section are derived from meaningful combinations of the following parameters: (1) whether loads are allowed to inspect preceding store addresses before obtaining a value from memory, (2) whether stores wait for both data and base registers to become available before posting their addresses for loads to inspect, (3) whether loads with ambiguous dependences can issue (i.e., whether memory dependence speculation is used).

14.7.1 Methodology

We used the SPEC95 benchmark suite [40] with the input data sets shown in Table 14.1. All programs were compiled using the GNU gcc compiler version 2.7.2. The base instruction set architecture is the MIPS-I [22] but with no architectural delay slots of any kind. Fortran sources were compiled by first converting them to C the AT&T f2c compiler. All programs were compiled using the -O2 optimization level and with loop unrolling and function inlining enabled. We employ two simulation techniques: (1) trace-driven simulation and (2) detailed, execution-driven timing simulation. Traces were generated using a functional simulator for the MIPS-I ISA. The functional simulator executes all user-level instructions. System calls are serviced by the OS of the host machine. We also make use of detailed, execution-driven timing simulation. For this purpose we utilized a simulator of a dynamically scheduled superscalar processor. The simulator executes all user-level instructions including those on control speculative paths. Systems calls are redirected to the OS of the host machine.

The default superscalar configuration we used is detailed in Table 14.2. We used a 32K data cache to compensate for the relatively small working sets of the SPEC95 programs. For some experiments we use a 64-entry reorder buffer model. That model has four copies of all functional units, a 2-port load/store queue and memory system, and can fetch up to four instructions per cycle.

TABLE 14.1: Program input data sets

Program	Input Data Set.
SPECint95	
099.go	play level = 9, board size = 9
124.m88ksim	modified test input: 370 iterations of Dhrystone Benchmark
126.gcc	reference input file recog.i
129.compress	modified train input: maximum file size increased to 50,000
130.li	modified test input: (queens 7)
132.jpeg	test input
134.perl	modified train input: jumble.pl with dictionary reduced by retaining every other 15th word
147.vortex	modified train input: persons.250k database PART_COUNT 250, LOOKUPS 20, DELETES 20, STUFF_PARTS 100, PCT_NEWPARTS 10, PCT_LOOKUPS 10, PCT_DELETES 10, PCT_STUFFPARTS 10
SPECfp95	
101.tomcatv	modified train input: N = 41
102.swim	modified train input: X = 256, Y = 256
103.su2cor	modified test input: LSIZE = 4 4 4 8 16
104.hydro2d	modified test input: MPROW = 200
107.mgrid	modified test input: LMI = 4
110.applu	modified train input: itmax = 25, nx = 10, ny = 10, nz = 10
125.turb3d	modified train input: nsteps = 1, itest = 0
141.apsi	modified train input: grid points x = 32, grid points z= 8, time steps = 130
145.fpppp	modified reference input: natoms = 4
146.wave5	modified train input: particle distribution 1000 20, grid size 625x20

TABLE 14.2: Default configuration for superscalar timing simulations

Fetch Interface	8 instructions fetched per cycle 4 fetch requests active Fetch up to 4 non-continuous blocks
Branch Predictor	64K-entry predictor [27] 2K BTB. Selector uses 2-bit counters. 1st predictor: 2-bit counter based. 2nd predictor: Gselect w/5-bit global history. Up to 4 branches resolved per cycle.
Instruction Cache	64K, 2-WSA, 8 banks, block interleaved, 256 sets per bank, 32 bytes per block 2 cycles hit, 10 cycle miss to unified. 50 cycle miss to main memory. Lockup free, 2 primary misses per bank, 1 secondary miss per primary. LRU.
OoO Core	128-entry ROB, max 8 ops per cycle, 128-entry combined L/S queue, with 4 inputs and 4 output ports. Loads can execute as soon as their address becomes available. Stores check for memory dependence violations by comparing addresses and data. Takes a combined 4 cycles for an instruction to be fetched and placed into the ROB.
Registers Functional Units	64 int, 64 fp, HI, LO and FSR. 8 copies of all functional units, all are fully pipelined. 4 memory ports.
Functional Unit Latencies	Integer: 1 cycle except for mult (4 cycles) and division (12 cycles), FF: 2 cycles for add/sub and comparison (single and double precision or SP/DP). 4 cycles SP mult, 5 cycles DP mult, 12 cycles SP div, 15 cycles DP div.
Store Buffer	128-entry. Does not combine stores. Combines store requests for load forwarding.
Data Cache	32K, 2WSA, 4 banks, 256 sets per bank, 32 bytes per block, 2 cycle hit, 10 cycle miss 50 cycle miss to main memory. Lockup-free, 8 primary misses per bank, 8 secondary miss per primary. LRU
Unified Cache	4M-byte, 2WSA, 4 banks, 128-byte block, 8 cycle + # 4 word transfer * 1 cycle hit, 50 cycles miss to main memory. Lockup-free, 4 primary miss per bank, 3 secondary per primary.
Main Memory	Infinite 34 cycles + #4 word transfers * 2 cycles.

Finally, to attain reasonable simulation times we utilized sampling for the timing simulations. In this technique which was also employed, for example, in [41],[35],[5], the simulator switches between functional and timing simulation. The mode of simulation is changed once a predefined number of instructions have been simulated. In all sampling simulations the observation size is 50,000 instructions. We chose sampling ratios that resulted in roughly 100M instructions being simulated in timing mode (i.e., sample size). We did not use sampling for 099.go, 107.mgrid, 132.jpeg and 141.apsi. We used a 1:1 timing to functional simulation ratio (i.e., once 50000 instructions are simulated in timing mode, we switch to functional mode and simulate 50000 instructions before switching back to timing mode, and so on) for: 110.applu, 124.m88ksim, 130.li, 134.perl and 145.fpppp. We used a 1:2 timing to functional simulation ratio for: 101.tomcatv, 102.swim, 126.gcc, 129.compress, 146.wave5 and 147.vortex. We used a 1:3 timing to functional simulation ratio for 103.su2cor. And finally, we used a 1:10 timing to functional simulation ratio for: 104.hydro2d and 125.turb3d. During the functional portion of the simulation the following structures were simulated: I-cache, D-cache, and branch prediction.

14.7.2 Performance Potential of Load/Store Parallelism

An initial consideration with the techniques we proposed is whether exploiting load-store parallelism can yield significant performance improvements. The reason is that in order to determine or predict memory dependences we need additional functionality: (1) To determine memory dependences we need a mechanism where loads and stores can post their addresses and execute accordingly to the dependences detected, that is, we need an address-based scheduler. (2) To predict memory dependences we need a memory dependence predictor coupled with a synchronization mechanism. For this reason an important consideration is whether this additional functionality is justified. Accordingly, we motivate the importance of exploiting load-store parallelism by comparing a model of a typical dynamically-scheduled ILP processor that does not attempt to determine and exploit load-store parallelism with one that is identical except in that it includes an oracle load-store disambiguation mechanism. Under this model, execution proceeds as follows: After an instruction is fetched, it is decoded and entered into the instruction window where its register dependences and availability are determined. If the instruction is a store, an entry is also allocated in a store buffer to support memory renaming and speculative execution. All instructions except loads can execute (i.e., issue) as soon as their register inputs become available. Stores wait for both data and address calculation operands before issuing. Loads wait in addition for all preceding stores to issue. As a result, loads may execute out-of-order only with respect to other loads and non-store instructions. The second configuration includes an oracle disambiguation mechanism that identifies load-store dependences as soon as instructions are entered into the

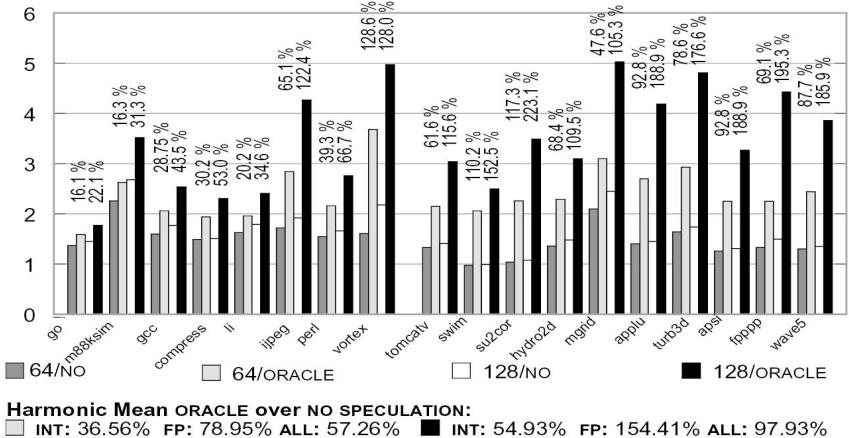


FIGURE 14.11: Performance (as IPC) with and without exploiting load/store parallelism. Notation used is instruction window size/load/store execution model. Speedups of ORACLE speculation over NO speculation are given on top of each bar.

instruction window. In this configuration, loads may execute as soon as their register and memory dependences (RAW) are satisfied. Since an oracle disambiguator is used, a load may execute out-of-order with respect to stores and does not need to wait for all preceding stores to calculate their addresses or write their data.

Figure 14.11 reports the performance improvements possible when perfect memory dependence information is available. We consider two base configurations, one with a 64-entry instruction window and one with a 128-entry window. For all programs, exploiting load/store parallelism has the potential for significant performance improvements. Furthermore, we can observe that when loads wait for all preceding store (NO bars) increasing the window size from 64 to 128 results in very small improvements. However, when the oracle disambiguator is used, performance increases sharply. This observation suggests that the ability to extract load/store parallelism becomes increasingly important performance wise as the instruction window increases.

14.7.3 Performance with Naive Memory Dependence Speculation

As we have seen, extracting load/store parallelism can result in significant performance improvements. In this section we measure what fraction of these performance improvements naive memory dependence speculation can offer. For this purpose, we allow loads to speculatively access memory as soon as their address operands become available. All speculative load accesses are

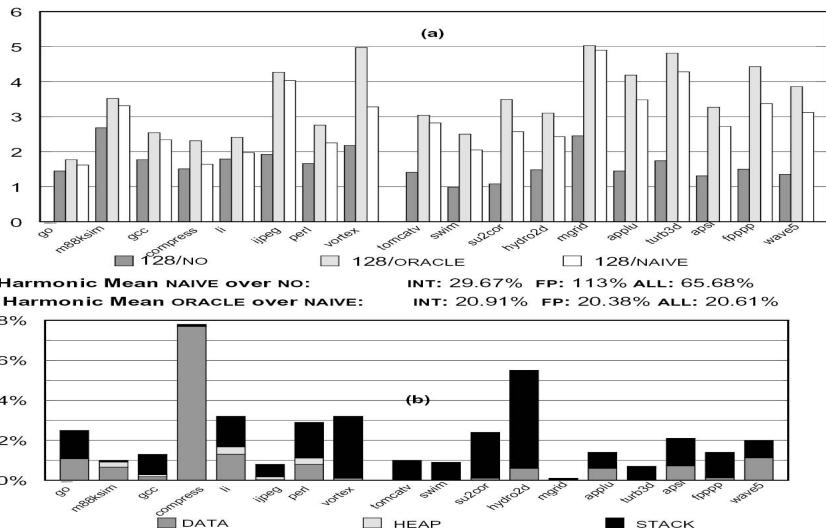


FIGURE 14.12: Naive memory dependence speculation. (a) Performance results (IPC). (b) Memory dependence misspeculation frequency.

recorded in a separate structure, so that preceding stores can detect whether a true memory dependence was violated by a speculatively issued load. Figure 14.12, part (a) reports performance (as IPC) for the 128-entry processor model when, from left to right, no speculation is used, when oracle dependence information is available and when naive memory dependence speculation is used. We can observe that for all programs naive memory dependence speculation results in higher performance compared to no speculation. However, the performance difference between naive memory dependence speculation and the oracle mechanism is significant, supporting our claim that the net penalty of misspeculation can become significant. Memory dependence misspeculations are at fault. The frequency of memory dependence misspeculations is shown in part (b) of Figure 14.12. We measure misspeculation frequency as a percentage over all committed loads. A breakdown in terms of the address-space through which the misspeculation occurs is also shown. We can observe that though loads cannot inspect preceding store addresses, misspeculations are rare. Nevertheless, the impact misspeculations have on performance is large.

In this context, the memory dependence speculation/synchronization methods we proposed could be used to reduce the net performance penalty due to memory dependence misspeculations. However, before we consider this possibility (which we do in Section 14.7.5) we first investigate using an address-based scheduler to extract load/store parallelism and its interaction with memory dependence speculation.

14.7.4 Using Address-Based Scheduling to Extract Load/Store Parallelism

We have seen that using oracle memory dependence information to schedule load/store execution has the potential for significant performance improvements. In this section we consider using address-based dependence information to exploit load/store parallelism. In particular we consider an organization where an address-based scheduler is used to compare the addresses of loads and stores and to guide load execution. We confirm that even in this context, memory dependence speculation offers superior performance compared to not speculating loads. However, we also demonstrate that when having to inspect addresses increases load execution latency, performance drops compared to the organization where oracle dependence information is available in advance (which we evaluated in the preceding section).

In the processor models we assume, stores and loads are allowed to post their addresses for disambiguation purposes as soon as possible. That is, stores *do not wait* for their data before calculating an address. Furthermore, loads are allowed to inspect preceding store addresses before accessing memory. If a true dependence is found the load always waits. When naive memory dependence speculation is used, a misspeculation is signaled only when: (1) a load has read a value from memory, (2) the value has been propagated to other instructions, and (3) the value is different than the one written by the preceding store that signals the misspeculation. As we noted earlier, under this processor model missspeculations are virtually non-existent. There are three reasons why this is so: (1) loads get either delayed because they can detect that a true dependence exists with a preceding store, (2) loads with unresolved dependences that correspond to true dependences are allowed to access memory but before they have a chance of propagating the value read from memory they receive a value from a preceding store, or (3) loads are delayed because preceding stores consume resources to have their addresses calculated and posted for disambiguation purposes.

Figure 14.13 reports how performance varies when naive memory dependence speculation is used compared to the same configuration that performs no speculation of memory dependences. For these experiments we use an 128-entry window processor model. We also measure how performance varies in terms of the time it takes for loads and stores to go through the address-based scheduler. We vary this delay from 0 to up to 2 cycles. In the calculation of the relative performance with naive speculation of part (a) of the figure, we should note that the base configuration is different for each bar. The absolute performance (i.e., the IPC) of the corresponding base configuration is reported in part (b). It can be seen that, for most programs, naive memory dependence speculation is still a win. Performance differences are not as drastic as they were for the model where loads could not inspect preceding store addresses, yet they are still significant. More importantly the performance difference between no speculation and memory dependence speculation

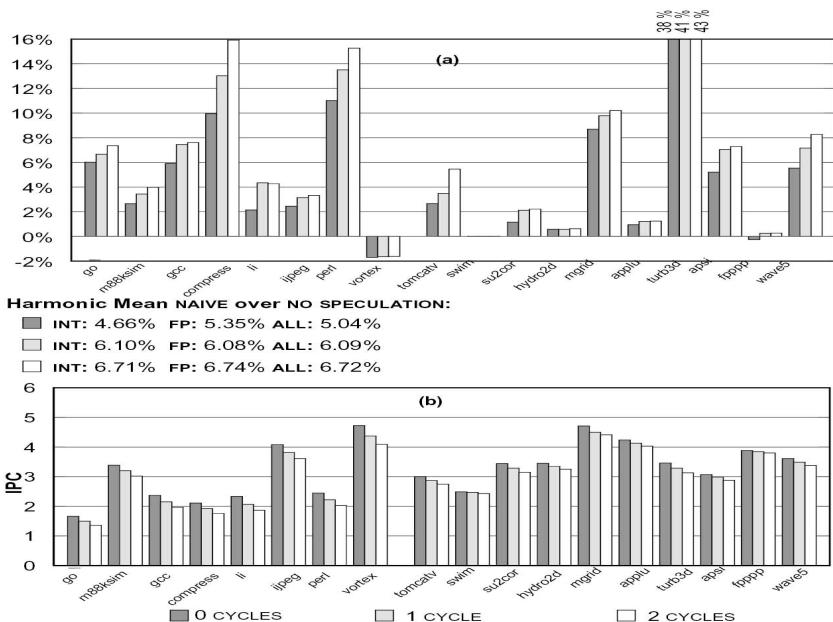


FIGURE 14.13: (a) Relative performance of naive memory dependence speculation as a function of the address-based scheduler latency. Performance variation is reported with respect to the same processor model that does not use memory dependence speculation. Base performance (IPC) is shown in part (b).

increases as the latency through the load/store scheduler also increases. For some programs, naive memory dependence speculation results in performance degradation. These programs are 147.vortex for all scheduling latencies and 145.fpppp when the scheduling latency is 0. It is not misspeculations that cause this degradation. This phenomenon can be attributed to loads with ambiguous dependences that get to access memory speculatively only to receive a new value from a preceding store before they had a chance to propagate the value they read from memory. These loads consume memory resources that could otherwise be used more productively. This phenomenon supports our earlier claim that there is an opportunity cost associated with erroneous speculation.

While including an address-based scheduler does help in exploiting some of the load/store parallelism, a load may still be delayed even when naive memory dependence speculation is used. The reason is that in a real implementation for preceding stores to calculate their addresses and post them for scheduling purposes, they must consume resources. These resources include issue bandwidth, address calculation adders and load/store scheduler ports. The same applies to loads that should wait for preceding stores. If perfect knowledge of dependences was available in advance, stores would consume resources only when both their data and address calculation operands become available. For this reason, we next compare the absolute performance of the processor models that use an address-based scheduler to that of the processor model that utilizes oracle dependence information to schedule load execution (Section 14.7.2).

Figure 14.14 reports relative performance compared to the configuration that uses no speculation but utilizes an address-based scheduler with 0 cycle latency (the IPC of this configuration was reported in Figure 14.12, part (b), 0-cycle configuration). From left to right, the four bars per program report performance with: (1) oracle disambiguation and no address-based scheduler (Section 14.7.2), (2) through (4) naive memory dependence speculation and address-based scheduler with a latency of 0, 1 and 2 cycles respectively (which we evaluated earlier in this section). We can observe that, with few exceptions, the 0-cycle address-based scheduler that uses naive speculation and the oracle mechanism perform equally well. Interestingly, the oracle configuration performance is significantly better for 147.vortex and 145.fpppp supporting our earlier claim about resource contention and the opportunity cost associated with erroneous speculation. We can also observe that once it takes one or more cycles to go through address-based disambiguation the oracle configuration has a clear advantage. The only exception is 104.hydro2d where the oracle configuration does significantly worse. This result may come as a surprise; however it is an artifact of our euphemistic use of the term oracle. In the oracle model we assume a store is allowed to issue only after both its data and address operands become available. As a result, dependent loads always observe the latency associated with store address calculation, which in this case is one cycle to fetch register operands and one cycle to do the

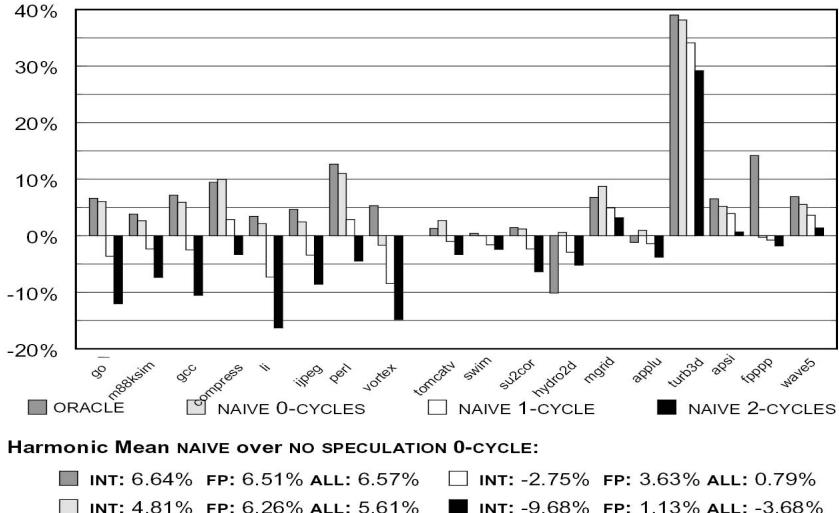


FIGURE 14.14: Comparing oracle disambiguation and address-based scheduling plus naive memory dependence speculation.

addition. Under these conditions, dependent loads can access the store value only after 3 cycles the store has issued. When the address-based scheduler is in place, a store may calculate its address long before its data is available and dependent loads can access the store’s value immediately. In an actual implementation, it may be possible to overlap store address calculation and store data reception without using an address-based scheduler (e.g., [18], [39]).

14.7.5 Speculation/Synchronization

In this section we consider using an implementation of our speculation/synchronization approach to improve accuracy over naive memory dependence speculation. In a continuous window processor that utilizes an address-based load/store scheduler along with naive memory dependence speculation, mis-speculations are virtually non-existent. In such an environment there is no need for our speculation/synchronization method. However, as we observed in Section 14.7.3, if an address-based scheduler is not present then the net penalty of misspeculation resulting from naive memory dependence speculation is significant. Moreover, and as we have seen in the previous section, the performance potential of a method such as speculation/synchronization (oracle configuration of Figure 14.13) is often close or exceeds the performance possible with an address-based scheduler with 0 cycle latency. Accordingly, in this section we restrict our attention to a configuration that does not use an address-based load/store scheduler. In the rest of this section we first provide

the details of the speculation/synchronization mechanism we simulated and then proceed to evaluate its performance.

The speculation/synchronization mechanism we used in these experiments comprises a 4K, 2-way set associative MDPT in which separate entries are allocated for stores and loads. Dependences are represented using synonyms, i.e., a level of indirection. The algorithm used to generate synonyms is the one described in Section 14.5.2 with the only difference being that no distance is associated with each synonym. It is implied that a load is synchronized with the last preceding store instance that has been assigned the same synonym. MDPT entries are allocated if they don't exist already upon the detection of a memory dependence violation for both offending instructions. No confidence mechanism is associated with each MDPT entry; once an entry is allocated, synchronization is always enforced. However, we flush the MDPT every million cycles to reduce the frequency of false dependences (this method was proposed in [7]). The functionality of the MDST is incorporated into the register-scheduler, which we assume to follow the RUU model [38]. This is done as follows: an additional register identifier is introduced per RUU entry to allow the introduction of speculative dependences for the purposes of speculation/synchronization. Stores that have dependences predicted use that register identifier to mark themselves as producers of the MDPT supplied synonym. Loads that have dependences predicted by the MDPT use that register identifier to mark themselves as consumers of the MDPT supplied synonym. Synchronization is achieved by: (1) making loads wait for the immediately preceding store (if there is any) that is marked as the producer of the same synonym, and (2) having stores broadcast their synonym once they issue, releasing any waiting loads. A waiting load is free to issue one cycle after the store it speculatively depends upon issues.

Figure 14.15, part (a) reports performance results relative to naive memory dependence speculation (Section 14.7.3). As it can be seen our speculation/synchronization mechanism offers most of the performance improvements that are possible had we had perfect in advance knowledge of all memory dependences (oracle). This is more clearly shown in part (b) of the same figure, where we report the relative over our speculation/synchronization mechanism performance of oracle speculation. On the average, the performance obtained by the use of our mechanism is within 1.001% of that possible with the oracle mechanism. For four programs (126.gcc, 101.tomcatv, 102.swim and 107.mgrid) our mechanism results in performance that is virtually identical to that possible with oracle speculation. For the rest of the programs the differences are relatively minor (3% in the worst case) when compared to the performance improvements obtained over naive speculation. To help in interpreting these differences we also present the misspeculation rates exhibited when our mechanism is in place. These results are shown in Table 14.3 (reported is the number of misspeculations over all committed loads). As it can be seen, misspeculations are virtually non-existent. This observation suggests that for the most part the performance differences compared to oracle specu-

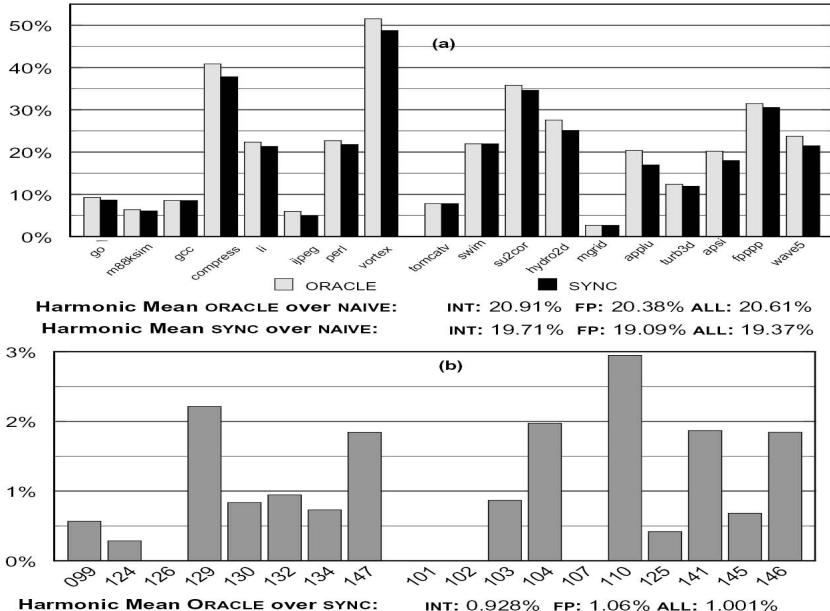


FIGURE 14.15: Performance of an implementation of speculation/synchronization. (a) Performance improvements over naive speculation. (b) Relative performance of oracle speculation over our speculation/synchronization.

lation are the result of either (1) false dependences, or (2) of failing to identify the appropriate store instance with which a load has to synchronize. False dependences are experienced when our mechanism incorrectly predicts that a load should wait although no store is actually going to write to the same memory location. In some cases and even when prediction correctly indicates that a dependence exists, our mechanism may fail to properly identify the appropriate store instance. This is the case for memory dependences that exhibit non-unit instance distances (e.g., $a[i] = a[i - 2]$). In such cases, a load is delayed unnecessarily as it is forced to wait for the very last preceding store instance that has been assigned the same synonym.

The results of this section suggest that our speculation/synchronization method can offer performance that is very close to that possible had we had perfect, in advance knowledge of all memory dependences. However, further investigation is required to determine how selective speculation and the store barrier policy would perform under this processor model.

TABLE 14.3: Memory dependence misspeculation rate with our speculation/synchronization mechanism (SYNC columns) and with naive speculation (NAIVE columns).

	NAIVE	SYNC		NAIVE	SYNC
go	2.5%	0.0301%	tomcatv	1.0%	0.0001%
m88ksim	1.0%	0.0030%	swim	0.9%	0.0017%
gcc	1.3%	0.0028%	su3cor	2.4%	0.0741%
compress	7.8%	0.0034%	hydro2d	5.5%	0.0740%
li	3.2%	0.0035%	mgrid	0.1%	0.0019%
jpeg	0.8%	0.0090%	applu	1.4%	0.0039%
perl	2.9%	0.0029%	turb3d	0.7%	0.0009%
vortex	3.2%	0.0286%	apsi	2.1%	0.0148%
			fppp	1.4%	0.0096%
			wave5	2.0%	0.0034%

14.8 Summary

Techniques to exploit instruction-level parallelism (ILP) are an integral part of virtually all modern high-performance processors. With these techniques, instructions do not necessarily execute one after the other and in the order they appear in the program. Rather, instructions are executed in any order convenient provided that program semantics are maintained (i.e., the same results are produced). This ability is useful in reducing execution time by executing instructions in-parallel (many at the same time) and by avoiding stalling execution while an instruction takes its time to execute (e.g., it performs a relatively time consuming calculation or accesses a relatively slow storage device for its operands). An area where ILP techniques are particularly useful is that of tolerating memory latency where these techniques are used to send load requests as early as possible, overlapping memory processing time with other useful computation.

One approach to exploiting ILP is to first make sure that executing an instruction will not violate program semantics before the instruction is allowed to execute. In the case of a load, this action amounts to waiting to determine if a preceding, yet to be executed store writes to the same memory location, that is whether a *true dependence* with a preceding store exists. However, waiting to determine whether a preceding store writes to the same memory location is not the best option. Higher performance is possible if *memory dependence speculation* is used, that is, if a load is allowed to execute speculatively before a preceding store on which it may be data dependent. Later on, and after the preceding store has calculated its address, we can check whether program semantics were violated. If no true memory dependence is violated in the resulting execution order, speculation was successful. Otherwise, spec-

ulation was erroneous and corrective action is necessary to undo the effects of erroneous execution. A penalty is typically incurred in the latter case.

In this chapter we focused on dynamic memory dependence speculation techniques and studied how existing methods of applying memory dependence speculation will scale for future generation processors. We demonstrated that as processors attempt to extract higher-levels of ILP by establishing larger instruction windows: (1) memory dependence speculation becomes increasingly important, and (2) the net penalty of memory dependence misspeculation can become significant. The latter observation suggests that further performance improvements are possible if misspeculations could be avoided. In the conventional centralized, continuous window processor, the net penalty of misspeculation becomes significant when loads cannot inspect the addresses of preceding stores either because a mechanism is not provided (to simplify the design) or because of the latency required to inspect store addresses. Moreover, we demonstrated that the potential benefits increase as the size of the instruction window also increases.

Motivated by the aforementioned observations we studied the trade-offs involved in memory dependence speculations and presented techniques to improve the accuracy of memory dependence speculation. Specifically, we presented techniques to: (1) identify via memory dependence prediction those loads and stores that would otherwise be misspecified, and (2) delay load execution only *as long as it is necessary* to avoid a memory dependence misspeculation. The best performing technique we propose is *memory dependence speculation and synchronization*, or *speculation/synchronization*. With this technique, initially loads are speculated whenever the opportunity exists (as it is common today). However, when misspeculations are encountered, information about the violated dependence is recorded in a memory dependence prediction structure. This information is subsequently used to predict whether the immediate execution of a load will result in a memory dependence violation, and (2) if so, which is the store this load should wait for.

We studied memory dependence speculation under a conventional centralized, continuous window processor (typical current superscalar) that utilizes fetch and execution units of equal bandwidth, and a program order priority scheduler (i.e., when there are many instructions ready to execute, the older ones in program order are given precedence). For this processor model we made two observations. The first is that using an address-based load/store scheduler (i.e., a structure where loads can inspect preceding store addresses to decide whether memory dependences exist) coupled with naive memory dependence speculation offers performance very close to that possible with perfect, in advance knowledge of all memory dependences, provided that going through the address-based scheduler does not increase load latency. The second is that if building an address-based load/store scheduler is not an option (clock cycle) or not a desirable option (complexity), naive memory dependence speculation can still offer most of the performance benefits possible by exploiting load/store parallelism. However, under this set of constraints

the net penalty of misspeculation is significant suggesting that our memory dependence speculation and synchronization technique might be useful in improving performance. Specifically, timing simulations show that an implementation of our techniques results in performance improvements of 19.7% (integer) and 19.1% (floating-point) which are very close to those ideally possible: 20.9% (integer) and 20.4% (floating-point).

While most modern processors utilize a centralized, instruction window it is not unlikely that future processors will have to resort to distributed, split-window organizations. As many argue, the reason is that the existing, centralized methods may not scale well for future technologies and larger instruction windows (e.g., [34]). An example where techniques similar to those we presented have been put to use already exists: the Alpha 21264 processor utilizes both a split-instruction window approach and selective memory dependence speculation to reduce the net penalty of dependence misspeculations [23]. As our results indicate, techniques to predict and synchronize memory dependences can greatly improve the performance of future split-window processors, allowing them to tolerate slower memory devices by aggressively moving loads up in time while avoiding the negative effects of memory dependence misspeculations. Moreover, speculation/synchronization may be used as a potentially lower complexity/faster clock cycle alternative to using an address-based load/store scheduler for exploiting load/store parallelism.

References

- [1] ——. *PowerPC 620 RISC Microprocessor Technical Summary*. IBM Order number MPR620TSU-01, Motorola Order Number MPC620/D, October 1994.
- [2] D. Adams, A. Allen, R. Flaker J. Bergkvist, J. Hesson, and J. LeBlanc. A 5ns store barrier cache with dynamic prediction of load/store conflicts in superscalar processors. In *Proc. IEEE International Solid-State Circuits Conference*, February 1997.
- [3] R. Allen and K. Kennedy. Automatic translation for FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.
- [4] U. Banerjee. *Dependence Analysis for Supercomputing*. Boston, MA: Kluwer Academic Publishers, 1988.
- [5] S. E. Breach. *Design and Evaluation of a Multiscalar Processor*. Ph.D. thesis, University of Wisconsin-Madison, Madison, WI 53706, December

1998.

- [6] W. Y. Chen. *Data Preload for Superscalar and VLIW Processors*. Ph.D. thesis, University of Illinois, Urbana, IL, 1993.
- [7] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proc. 25th International Symposium on Computer Architecture*, pages 142–153, June-July 1998.
- [8] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta code morphing software: Using speculation, recovery, and adaptive re-translation to address real-life challenges. In *Proc. International Symposium on Code Generation and Optimization*, March 2003.
- [9] J. R. Ellis. Bulldog: A compiler for VLIW architectures. Research Report YALE/DCS/RR-364, Department of Computer Science, Yale University, February 1985.
- [10] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.
- [11] M. Franklin. *The Multiscalar Architecture*. Ph.D. thesis, University of Wisconsin-Madison, Madison, WI 53706, November 1993.
- [12] M. Franklin and G. S. Sohi. ARB: A Hardware Mechanism for Dynamic Memory Disambiguation. *IEEE Transactions on Computers*, 45(5):552–571, May 1996.
- [13] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W.-M. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–193, October 1994.
- [14] S. Gopal, T.N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. In *Proc. 4th International Symposium on High-Performance Computer Architecture*, pages 195–205, February 1998.
- [15] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proc. VIII International Symposium on Architectural Support for Programming Languages and Computer Architecture*, October 1998.
- [16] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, 2nd edition, 1996.
- [17] J. Hesson, J. LeBlanc, and S. Ciavaglia. Apparatus to dynamically control the out-of-order execution of load-store instructions. *U.S. Patent 5,615,350*, March 1997.

- [18] G. J. Hinton, R. W. Martell, M. A. Fetterman, D. B. Papworth, and J. L. Schwartz. Circuit and method for scheduling instructions by predicting future availability of resources required for execution, *U.S Patent 5,555,432*, filed on Aug. 19, 1994, September 1996.
- [19] A. S. Huang and J. P. Shen. The intrinsic bandwidth requirements of ordinary programs. In *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–114, October 1996.
- [20] A. S. Huang, G. Slavenburg, and J. P. Shen. Speculative disambiguation: A compilation technique for dynamic memory disambiguation. In *Proc. 21st Annual International Symposium on Computer Architecture*, pages 200–210, April 1994.
- [21] D. Hunt. Advanced performance features of the 64-bit PA-8000. In *Proc. IEEE International Computer Conference COMPCON'95*, pages 123–128, March 1995.
- [22] G. Kane. *MIPS R2000 RISC Architecture*. Prentice Hall, Englewood Cliffs, New Jersey, 1987.
- [23] R. E. Kessler, E. J. McLellan, and D. A. Webb. The Alpha 21264 architecture. In *Proc. Intl. Conference on Computer Design*, pages 90–95, December 1998.
- [24] T. Knight. An architecture for mostly functional languages. In *Proc. ACM Conference on Lisp and Functional Programming*, August 1986.
- [25] D. Levitan, T. Thomas, and P. Tu. The PowerPC 620 Microprocessor: A High Performance Superscalar RISC Processor. In *Proc. IEEE International Computer Conference COMPCON'95*, March 1995.
- [26] S. A. Mahlke, W. Y. Chen, W.-C. Hwu, B. R. Rau, and M. S. Schlansker. Sentinel scheduling for VLIW and superscalar processors. In *Proc. 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 238–247, October 1992.
- [27] S. McFarling. Combining branch predictors. *WRL Technical Note, TN-36*, June 1993.
- [28] B. R. Rau and Michael S. Schlansker. EPIC: Explicitly Parallel Instruction Computing. *IEEE Computer Magazine*, 33, February 2000.
- [29] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. A dynamic approach to improve the accuracy of data speculation. Technical Report 1316, Computer Sciences Dept., University of Wisconsin-Madison, March 1996.
- [30] A. Moshovos and G.S. Sohi. Streamlining inter-operation communication via data dependence prediction. In *Proc. 30th Annual International Symposium on Microarchitecture*, December 1997.

- [31] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proc. 24th Annual International Symposium on Computer Architecture*, pages 181–193, June 1997.
- [32] M. Moudgil and J. H. Moreno. Run-time detection and recovery from incorrectly reordered memory operations. In *IBM research report RC 20857 (91318)*, May 1997.
- [33] A. Nicolau. Run-time disambiguation: Coping with statically unpredictable dependencies. *IEEE Transactions on Computers*, 38(5):663–678, May 1989.
- [34] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proc. 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.
- [35] M. Reilly and J. Edmondson. Performance simulation of an Alpha microprocessor. In *IEEE Computer*, 31(5), May 1998.
- [36] A. Rogers and K. Li. Software support for speculative loads. In *Proc. 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, October 1992.
- [37] M. D. Smith, M. Horowitz, and M. S. Lam. Efficient superscalar performance through boosting. In *Proc. 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 248–259, October 1992.
- [38] G. S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Trans. on Computers*, 39(3):349–359, March 1990.
- [39] S. Steely, D. Sager, and D. Fite. Memory reference tagging. *U.S. Patent 5,619,662*, April 1997.
- [40] The Standard Performance Evaluation Corporation. SPEC CPU95 Benchmarks. *SPEC Newsletter*, available on-line from <http://www.specbench.org/osg/cpu95/news/cpu95descr.html>, September 1995.
- [41] K. M. Wilson, K. Olukotun, and M. Rosenblum. Increasing cache port efficiency for dynamic superscalar microprocessors. In *Proc. 23rd Annual International Symposium on Computer Architecture*, May 1996.
- [42] R. P. Wilson and M. S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proc. International Symposium on Programming Language Design and Implementation*, June 1995.

Chapter 15

Resource Flow Microarchitectures

David A. Morano,¹ David R. Kaeli,¹ and Augustus K. Uht²

¹Northeastern University; ²University of Rhode Island

15.1	Introduction	393
15.2	Motivation for More Parallelism	395
15.3	Resource Flow Basic Concepts	396
15.4	Representative Microarchitectures	406
15.5	Summary	416
	References	417

15.1 Introduction

Speculative execution has proven to be enormously valuable for increasing execution-time performance in recent and current processors. The use of speculative execution provides a powerful latency-hiding mechanism for those microarchitectural operations that would otherwise cause unacceptable stalls within the processor, such as waiting for conditional branches to resolve or for memory reads to be fulfilled from the memory hierarchy. Further, in order to extract ever larger amounts of instruction level parallelism from existing programs (generally quite sequential in nature) over many basic blocks, much more speculative execution is usually required. However, the complexity of implementing speculative execution is substantial and has been a limiting factor in its evolution to more aggressive forms beyond control-flow speculation.

Most existing implementations of speculative execution focus on conditional branch prediction and the subsequent speculative execution of the instructions following those branches. Generally, only one path following a branch is followed although multiple successive branches can be predicted. Speculative instruction results are stored in microarchitectural structures that hold those results as being tentative until they can be determined to constitute the committed state of the program being executed. If a predicted branch is determined (resolved) to have been mis-predicted, any speculatively executed instructions need to be squashed. This generally entails the abandonment of any speculatively generated results as well as the purging of all currently executing speculative instructions from the machine. The management

and sequencing of existing speculative execution is already moderately complex. This complexity has limited or discouraged the use of more advanced speculation techniques such as value prediction, multipath execution, and the retention of speculative instructions that may still be correct after a misprediction. Further, as clock speeds get higher and pipeline depths get larger the performance penalty of squashing speculative instructions and any associated correct results get undesirably larger also.

A microarchitectural approach oriented towards handling speculative execution in a more general and uniform way is presented. The approach attempts to generalize the management and operational issues associated with control-flow prediction, value prediction, and the possible re-execution of those instructions that have already been fetched and dispatched. Moreover, to accommodate the much larger number of speculatively executed instructions needed in order to extract more instruction level parallelism from the program, a strategy for scaling a microarchitecture in terms of its component resources needs to also be formulated. The microarchitectural approach presented also lends itself towards resource scalability through manageable spatial distribution of machine components. This last objective is also realized through the rather general and uniform way in which instructions and operands are handled.

This microarchitectural approach is termed *resource flow computing* and centralizes around the idea that speculative execution is not constrained by either the control flow graph or the data flow graph of the program, but rather by the available resources within a representative microarchitecture. Further, the importance of instruction operands (whether they be control or data) is elevated to almost the level of importance of an instruction itself. Operands are enhanced with additional state that allows for a more uniform management of their flow through the machine. This philosophy of execution allows for a large number of simultaneous instruction executions and re-executions as can be sustained on the available machine resources since executions are allowed to proceed with any available feasible source operands whether predicted or not. The idea is to first speculatively execute any pending instruction whenever any suitable machine resource is available and then to perform successive re-executions as needed, as control and data dependency relationships are determined dynamically during execution.

The basic concepts employed in existing Resource Flow microarchitectures are presented along with the various structures needed to manage this aggressive form of speculative execution in a generalized way. Also described is how a variety of aggressive speculative schemes and ideas fit into this new microarchitectural approach more easily than if they were considered and implemented without this generalized instruction and operand management. Finally, two representative example microarchitectures, based on the resource flow concepts, are briefly presented. The first of these is oriented for a silicon area size somewhere between previous or existing processors such as the Alpha EV8 [15], the larger Pentium-4 [6] variants, or the Power-4 (in the 100 to 200

million transistor range) and next generation processors (maybe 200 to 300 million transistors). The second microarchitecture presented illustrates where the resource flow ideas could lead with much larger machine sizes and component resources (approaching 1 billion transistors and beyond). As will be seen, the physical scalability of this microarchitecture is substantially facilitated by the basic resource flow design concepts.

15.2 Motivation for More Parallelism

Although many high performance applications today can be parallelized at the source code level and executed on symmetric multiprocessors or clustered systems, there are and will continue to be requirements for achieving the highest performance on single threaded program codes. An attempt is made to target this application problem space through the extraction of instruction level parallelism (ILP). However, high execution performance through ILP extraction has not generally been achieved even though large amounts of ILP are present in integer sequential programs codes. Several studies into the limits of instruction level parallelism have shown that there is a significant amount of parallelism within typical sequentially oriented single-threaded programs (e.g., SpecInt-2000). The work of researchers including Lam and Wilson [10], Uht and Sindagi [26], González and González [5] have shown that there exists a great amount of instruction level parallelism that is not being exploited by any existing computer designs.

A basic challenge is to find program parallelism and then allow execution to occur speculatively, in parallel, and out of order over a large number of instructions. Generally, this is achieved by introducing multiple execution units into the microarchitecture where each unit can operate as independently as possible and in parallel, thus achieving increased executed instructions per clock (IPC). Since it is also usually very desirable (if not critical) to support legacy instruction set architectures (ISAs), this should also be a goal when pursuing enhanced high IPC mechanisms. For this reason, a microarchitecture that is suitable for implementing any ISA is explored.

Microarchitectures that have employed the use of multiple execution units are the Multiscalar-like processors [20, 21], the SuperThreaded processor model [24], and the Parallel Execution Window processor model [9]. Other microarchitecture proposals such as the MultiCluster machine model by Farkas et al. [2] are also in this category. In particular, the Multiscalar processors have realized substantial IPC speedups over conventional superscalar processors, but they rely on compiler participation in their approach.

Another attempt at realizing high IPC was done by Lipasti and Shen on their Superspeculative architecture [11]. They achieved an IPC of about 7

with conventional hardware assumptions but also by using value prediction. Nagarajan proposed a *Grid Architecture* of ALUs connected by an operand network. [14] This has some similarities to the present work. However, unlike the present work, their microarchitecture relies on the coordinated use of the compiler along with a new ISA to obtain higher IPCs.

The present goal is to combine many of the design features of proposed speculative microarchitectures into a more generalized form. These include: control-flow prediction, value prediction, dynamic microarchitectural instruction predication, dynamic operand dependency determination, and multipath execution. Another objective is to try to facilitate a substantially increased sized processor along with many more resources for large-scale simultaneous speculative instruction execution, and therefore greater possible ILP extraction. This last objective is illustrated with a discussion of the second representative microarchitecture below, which shows how large physical processor scalability might possibly be achieved.

15.3 Resource Flow Basic Concepts

In the following sections several of the basic concepts embodied in the resource flow execution model are presented. In this model, the management of operands, including dependency ordering and their transfer among instructions, dominates the microarchitecture. Another primary element of the execution process is the dynamic determination of the control and data dependencies among the instructions and operands. Also, the establishment of a machine component used to hold an instruction during its entire life-cycle after it has been dispatched for possible execution is presented in some detail. This component will generalize how instructions are handled during both initial execution and subsequent executions as well as eliminate the need for any additional re-order components. This component is termed an *active station* (AS). Additionally, some details about how operands are passed from one instruction to another are discussed.

15.3.1 The Operand as a First Class Entity

Within the presented microarchitectural framework, three types of instruction operands that need special handling can be identified. These are termed *register*, *memory*, and *predicate*. Although register and memory operands are both data operands, they are different enough in their use and characteristics that they need to be treated somewhat differently at the microarchitectural level of the machine. For example, register operands (generally) have fixed addresses associated with them (their address is not usually dynamically de-

terminated by the instruction during its execution) while it is very common for memory operands to have their addresses determined by a memory-accessing instruction during the execution of the instruction. Often the address of a memory operand is calculated at execution time through some manipulation of a constant and one or more register operands. Older ISAs (such as the Digital Equipment VAX) could even determine a memory address from a memory location. Further, the number of registers within a given ISA tends to be very small (like 32 to 256 or so) while the number of memory locations is rapidly being standardized on two to the power 64. These differences therefore warrant different hardware treatment within the machine.

For predicate operands, two types need to be distinguished. One type is a part of the ISA of the machine and is therefore visible to programmer. A predicate operand of this type is present in an ISA such as the iA-64, for example. [7] For the present purpose, this sort of explicit predicate operand is identical to a register operand (discussed above) and is simply treated as such. Not as clear is the handling of microarchitectural predicate operands that are not a part of the ISA and are therefore not visible to the programmer. This latter type of predication is entirely maintained by the microarchitecture itself and essentially forms an additional input operand for each instruction. This single bit operand is what is used to predicate the instruction's committed execution, the same as if it was explicit in the ISA. This input predicate thus enables or disables its associated instruction from producing its own new output operand. It should be noted that, at any time, any instruction can always still be allowed to execute. The only issue is whether the output result can become part of the program committed state. In the resource flow model, microarchitecture-only predicate operands share some similarities to register and memory (data) operands but are still different enough that they warrant some additional special treatment. Dynamic microarchitectural instruction predication is employed, whether or not the ISA defines explicit predicate registers.

15.3.2 Dynamic Dependency Ordering

Rather than calculating instruction dependencies at instruction dispatch or issue time, instructions are allowed to begin the process of executing (possibly with incorrect operands) while also providing for instructions to dynamically determine their own correct operand dependencies. The mechanism used for this dynamic determination of operand dependencies is to provide a special tag that conveys the program-ordered relative time of the origin of each operand. This time ordering tag is associated with the operational state of both instructions and operands and is a small integer that uniquely identifies an instruction or an operand with respect to each other in program ordered time. Typically, time-tags take on small positive values with a range approximately equal to the number of instructions that can be held in-flight within an implementation of a machine. The Warp Engine [1] also used time-tags

to manage large amounts of speculative execution, but the present use of time-tags is substantially simpler than theirs.

A time-tag value of zero (in the simplest case) is associated with the in-flight instruction that is next ready to retire (the one that was dispatched the furthest in the past). Later dispatched instructions take on successively higher valued tags. As instructions retire, the time-tag registers associated with all instructions and operands are decremented by the number of instructions being retired. Committed operands can thus be thought of as taking on negative valued time-tags. Negative valued tags can also have microarchitectural applications such as when memory operands are committed but still need to be snooped in a store queue. Output operands, created as a result of an instruction's execution, take on the same time-tag value as its originating instruction. By comparing time-tag values with each other, the relative program-ordered time relationship is determined. This use of time-tags for dependency ordering obviates the need for more complicated microarchitectural structures such as dependency matrices. [22]

15.3.3 Handling Multipath Execution

Multipath execution is a means to speculatively execute down more than one future path of a program simultaneously (see Chapter 6). In order to provide proper dependency ordering for multipath execution an additional register tag (termed a *path ID*) is introduced that will be associated with both instructions and operands. Generally, a new speculative path may be formed after each conditional branch is encountered within the instruction stream. Execution is speculative for all instructions after the first unresolved conditional branch.

At least two options are available for assigning time-tags to the instructions following a conditional branch (on both the taken and not-taken paths). One option is to dynamically follow each output path and assign successively higher time-tag values to succeeding instructions on each path independently of the other. The second option is to try to determine if the *taken* output path of the branch joins with the *not-taken* output path instruction stream. If a join is determined, the instruction following the *not-taken* output path is assigned a time value one higher than the branch itself, while the first instruction on the *taken* path is assigned the same value as the instruction at the join as it exists on the not-taken path of the branch.

Note that both options may be employed simultaneously in a microarchitecture. Details on when each of these choices might be made depends on whether it is desirable to specially handle speculative path joins in order to take further advantage of control-independent instructions beyond the join point. This and many other details concerning multipath execution is an open area of ongoing research. In either case, there may exist instructions in flight on separate speculative paths that possess the same value for their time-tag. Likewise, operands resulting from these instructions would also share the

same time-tag value. This ambiguity is resolved through the introduction of the path ID. Through the introduction of the program-ordered time-tag and the path ID tag, a fully unique time-ordered execution name space is now possible for all instructions and operands that may be in flight.

15.3.4 Names and Renaming

For instructions, names for them can be uniquely created with the concatenation of the following elements:

- a path ID
- the time-tag assigned to a dispatched instruction

For all operands, unique names consist of :

- type of operand
- a path ID
- time-tag
- address

Generally, the type of the operand would be *register*, *memory*, or *predicate*. The address of the operand would differ depending on the type of the operand. For register operands, the address would be the name of the architected register. All ISA architected registers are typically provided a unique numerical address. These would include the general purpose registers, any status or other non-general purpose registers, and any possible ISA predicate registers. For memory operands, the address is just the programmer visible architected memory address of the corresponding memory value. Finally, for predicate operands, the address may be absent entirely for some implementations, or might be some address that is used within the microarchitecture to further identify the particular predicate register in question. Microarchitecture predication schemes that have used both approaches have been devised.

Through this naming scheme, all instances of an operand in the machine are now uniquely identified, effectively providing full renaming. All false dependencies are now avoided. There is no need to limit instruction dispatch or to limit speculative instruction execution due to a limit on the number of non-architected registers available for holding temporary results. Since every operand has a unique name defined by a time-tag, all necessary ordering information is provided for. This information can be used to eliminate the need for physical register renaming, register update units, or reorder buffers.

15.3.5 The Active Station Idea

The introduction of the active station component and its operation is what most distinguishes the present execution model from that of most others. The active station and its operational philosophy has been previously introduced by Uht et al. [28] These active stations can be thought of as being reservation stations [23] but with additional state and logic added that allows for dynamic operand dependency determination as well as for holding a dispatched

instruction (its decoded form) until it is ready to be retired. This differs from conventional reservation stations or issue window slots in that the instruction does not free the station once it is issued to a function unit (FU). Also, unlike reservation stations, but like an issue window slot, the instruction operand from an active station may be issued to different function units (not just the one that is strictly associated with a reservation station). With a conventional reservation station, the outputs of the execution are looped back to both waiting reservation stations and to the architected register file. That does not occur with action stations. Rather, output results from executions return to the originating active station and they are then further distributed from there with additional identifying information added to them. Having execution units forward results directly is possible but has not been explored due to the added complexity of the bus interconnections needed. In any event, the active must still acquire output results in the event that those results need to be broadcast to other ASes subsequently due to control or data flow dependency changes.

The state associated with an active station can be grouped into two main categories. There is state that is relevant to the instruction itself, and secondly there is state that is related to the operands of that instruction (both source and destination operands). The instruction specific state is used to guide the instruction through its life cycle until retirement, while the remaining state consists of zero or more input source operands and one or more output destination operands. All operands regardless of type and whether source or destination occupy a similar structure within an active station, termed an *operand block*. The operand blocks all have connectivity to both the operand request and forwarding buses as well as to the FU issue and result buses. More detail on these operand blocks and operand management is provided in the next section.

The state that is primarily associated with the instruction itself consists of the following:

- instruction address
- instruction operation
- execution state
- path ID
- time ordering tag
- predication information

The *instruction operation* is derived from the decoded instruction and specifies the instruction class and other details needed for the execution of the instruction. This information may consist of subfields and is generally ISA specific. The *instruction address* and *predicate information* is used as part of the dynamic predication scheme used. [12] The *path ID* value is used when dynamic multipath execution is done. The *time-tag* value is used to order this instruction with respect to all others that are currently within the execution window of the machine. The additional *execution state* of an AS consists of

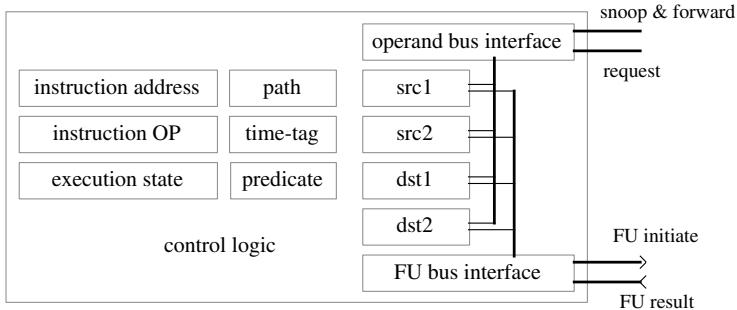


FIGURE 15.1: High-level block diagram of the active station. The major state associated with an active station is shown: four operand blocks (two source and two destination) and its four bus interfaces, grouped according to bus function into two major logic blocks.

information used to guide the AS through operand acquisition, the handling of requested operands by other ASes, determining when execution is possible or when re-executions are needed, and when commitment of this instruction is possible. Many state machines operate in parallel to manage the complex AS operational life-cycle and there is generally miscellaneous state associated with each.

A simplified block diagram of the active station is shown in Figure 15.1. The state associated primarily with just the instruction is shown on the left of the block diagram while the operand blocks and their connectivity to the various buses is shown on the right. In this example, a total of four operand blocks are shown, labeled: *src1*, *src2*, *dst1*, and *dst2*. The number of source and destination operand blocks that are used for any given machine is ISA dependent. For example, some ISAs require up to five source operands and up to three destination operands (usually for handling double precision floating point instructions). Generally, any operand in the ISA that can cause a separate and independent data dependency (a unique dependency name) requires a separate operand block for it. No additional operand block is needed to accommodate dynamic predication since its state is included as part of the instruction-specific active station state mentioned previously.

Execution of the instruction can occur either entirely within the AS (like for the case of simple control-flow instructions and other simple instructions) or needs to be carried out with an execution unit of some sort. Most instructions generally require the use of an execution unit since the silicon size of the required execution pipeline is not only too large to fit within each AS but is something that benefits from being shared among many ASes. The complexity of the instructions for the ISA being implemented would be used to determine what type of instructions might be able to be executed within the AS itself. The nature and degree of sharing of execution resources among ASes is an implementation variation.

When execution is needed by a function unit, either a first execution or a re-execution, the AS contends for the use of an execution resource and upon granting of such it sends the specific operation to be performed along with the necessary input operands and an identifying tag indicating the originating AS to the execution unit. The execution proceeds over one or more pipeline stages and the results are sent back to the originating AS. The AS then contends for outbound operand forwarding bus bandwidth and sends its output operands to future program-ordered ASes for their operand snooping and acquisition (discussed more below). Whether an AS is allowed to initiate a subsequent execution request before an existing one is completed (allowing for multiple overlapped executions of the same instruction) is an implementation variation. This is a performance issue that is still being researched.

The conditions determining when an AS can retire (either commitment or abandonment) vary somewhat according to specific implementations, but the more restraining condition of an AS retiring in a committed state generally must meet several criteria. Some of these are:

- all program-ordered past ASes are ready to retire
- the present AS is predicated to be enabled
- the present AS has executed at least once
- all input data operands have been snooped at least once (to verify correct value predictions)
- all of the latest created output operands have been forwarded

Specific implementations may add some other conditions for retirement (either commitment or abandonment) but when a retirement does occur, the AS is free to accept another dispatched instruction in the next clock.

15.3.6 Register and Memory Operand Storage

Register and memory operands are similar enough that they share an identical operand block within the active station. The state within an operand block consists of:

- type of operand
- path ID
- time ordering tag
- address
- size
- previous value
- value

The operand *type*, *path ID*, and *time ordering tag* serve an analogous purpose as those fields do within an active station, except that these fields now apply specifically to this particular operand rather than to the instruction as a whole.

The *address* field differs depending on the type of the operand. For register operands, the address would be the name of the architected register. All ISA architected registers are typically provided a unique numerical address. These

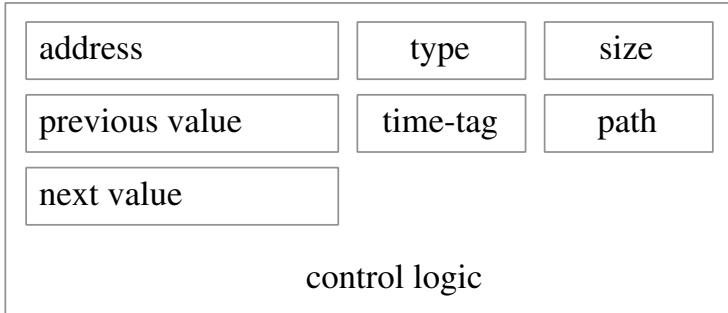


FIGURE 15.2: Block diagram of an operand block. Each operand block holds an effectively renamed operand within the active stations. Several operand blocks are employed within each active station depending on the needs of the ISA being implemented. The state information maintained for each operand is shown.

would include the general purpose registers, any status or other non-general purpose registers, and any possible ISA (architected) predicate registers (like those in the iA-64 ISA [8, 18]. For memory operands, the identifying address is just the programmer-visible architected memory address of the corresponding memory value.

The *size* is only used for memory operands and holds the size of the memory reference in bytes. The *value* holds the present value of the operand, while the *previous value* is only used for destination operands and holds the value that the operand had before it may have been changed by the present instruction. The previous value is used in two possible circumstances. First, when the effects of the present instruction need to be squashed (if and when its enabling predicate becomes false), the previous value is broadcast forward. It is also used when a forwarded operand with a specific address was incorrect and there is no expectation that a later instance of that operand with the same address will be forwarded. This situation occurs when addresses for memory operands are calculated but are later determined to be incorrect. An operand with the old address is forwarded with the previous value to correct the situation. Figure 15.2 shows a simplified block diagram of an operand block.

15.3.7 Operand Forwarding and Snooping

Similar to all microarchitectures, operands resulting from the execution of instructions are broadcast forward for use by waiting instructions. The time-tag and path ID accompany the operand's identifying address and value when it is forwarded. The time-tag will be used by subsequent instructions (later in

program order time) already dispatched to determine if the operand should be *snarfed*¹ as an input that will trigger its execution or re-execution.

The information associated with each operand that is broadcast from one instruction to subsequent ones is referred to as a *transaction*, and generally consists of:

- transaction type
- operand type
- path ID
- time-tag of the originating instruction
- identifying address
- data value for this operand

This above information is typical of all operand transactions with the exception of predicate transactions (which generally contain more information). True flow dependencies are enforced through the continuous snooping of these transactions by each dispatched instruction residing in an issue slot that receives the transaction. Each instruction will snoop all operands that are broadcast to it but an operand forwarding interconnect fabric may be devised so that transactions are only sent to those instructions that primarily lie in future program-ordered time from the originating instruction. More information on operand forwarding interconnection networks is presented in a later section.

Figure 15.3 shows the registers inside an AS used for the snooping and snarfing of one of its input operands. The *time-tag*, *address*, and *value* registers are reloaded with new values on each snarf, while the *path* and *instr-time-tag* are only loaded when an instruction is dispatched.

If the path ID and the identifying address of the operand matches any of its current input operands, the instruction then checks if the time-tag value is less than its own assigned time-tag, and greater than or equal to the time-tag value of the last operand that it snarfed, if any. If the snooped data value is different than the input operand data value that the instruction already has, a re-execution of the instruction is initiated. This above evaluation is common for all operand snooping and provides for the dynamic discovery of feasible input operands.

15.3.8 Result Forwarding Buses and Operand Filtering

The basic requirement of the operand interconnection fabric is that it must be able to transport operand results from any instruction to those instructions in future program ordered time (those with higher valued time-tags). A variety of operand forwarding fabrics are possible but generally some combination of one or more buses have been the preferred choice so far. One choice

¹Snarfing entails snooping address/data buses, and when the desired address value is detected, the associated data value is read

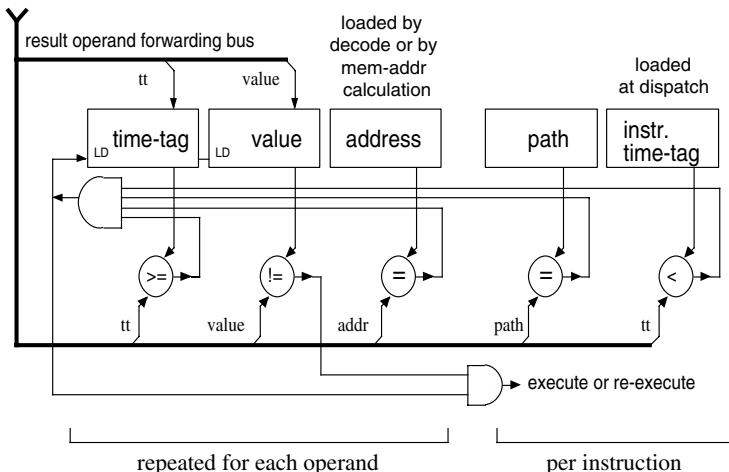


FIGURE 15.3: *Operand snooping.* The registers and snooping operation of one of several possible source operands is shown. Just one operand forwarding bus is shown being snooped but typically several operand forwarding buses are snooped simultaneously.

is just to have one or more buses just directly interconnect all of the ASes, LSQ, and any register files (if needed). Although appropriate for smaller sized microarchitectures, this arrangement does not scale as well as some other alternatives. A more appropriate interconnection fabric that would allow for the physical scaling of the microarchitecture is one in which forwarding buses are segmented with active repeaters between the stages. Instructions lying close in future program-ordered time will get their operands quickly (since they are either on the same physical bus segment or only one or two hops away) while those instructions lying farther away will incur additional delays. This arrangement exploits the fact that register lifetimes are fairly short. [4, 20] Since these active repeaters forward operands from one bus segment to another, they have been termed *forwarding units*. A minimal forwarding unit would consist largely of a small sized FIFO. When the FIFO is full, the bus being snooped by that forwarding unit is stalled to prevent overflow. A variety of forwarding arrangements are possible: taking into account different buses, different operand types, and different forwarding units for each operand type.

In addition to allowing for physical scaling, the use of forwarding units makes possible a range of new microarchitectural ideas not easily accomplished without the use of time-tags and dynamic dependency determination. While some forwarding units can just provide a store-and-forward function, others can store recently forwarded operands and filter out (not forward) newly snooped operands that have previously been forwarded (termed *silent forwards*). This latter type of forwarding unit is therefore more specifically termed a *filter unit*. Filter units snoop and snarf operands in a way analogous

to how ASes do but with slight differences to account for their forwarding operation. Specifically, they also are assigned time-tag values identical to the ASes they lie next to in relation to the forwarding fabric. This filtering technique can be used to reduce the operand traffic on the forwarding fabric and thus reduce implementation costs for forwarding bandwidth. Two types of filter units have been explored so far. For registers, the *register filter unit* (RFU) can maintain either a small cache or a copy of all architecture registers within it and only forward those register operands whose values have not yet been seen by program-ordered future ASes. The ability to hold all of the architected registers also opens up the possibility for eliminating the need for a centralized register files (architected or not). Similarly, for memory operands, a cache (call it a *L0 cache*) can be situated inside a *memory filter unit* (MFU) and can hold recently requested memory operands. This can reduce the demand on the L1 data cache and the rest of the memory hierarchy by satisfying some percent of memory operand loads from these caches alone. The use of both types of these filter units is shown in the second of example microarchitecture presented in the next section.

15.4 Representative Microarchitectures

Two representative microarchitectures are now presented that both employ the resource flow execution model. The first is a relatively simple example that is oriented towards a machine with approximately the same physical silicon size (numbers of transistors) and complexity of a current or next-generation state-of-art processor. The second example shows how resource flow computing can be scaled to much larger physical sizes and numbers of machine resources. The possible machine size, in terms of resource components, with this example is currently substantially beyond those that can be achieved using any conventional microarchitectural models.

Both of these microarchitectures are similar to conventional microarchitectures in many respects. The common aspects of both microarchitectures are discussed first and then the distinctives of each is presented in subsequent sections. The L2 cache (unified in the present case), and the L1 instruction cache are both rather similar to those in common use. Except for the fact that the main memory, L2 cache, and L1 data cache are generally all address-interleaved, there is nothing further unique about these components. The interleaving is simply used as a bandwidth enhancing technique and is not functionally necessary for the designs to work.

The i-fetch unit first fetches instructions from i-cache along one or more predicted program paths. Due to the relatively large instruction fetch bandwidth requirement, the fetching of several i-cache lines in a single clock may be gen-

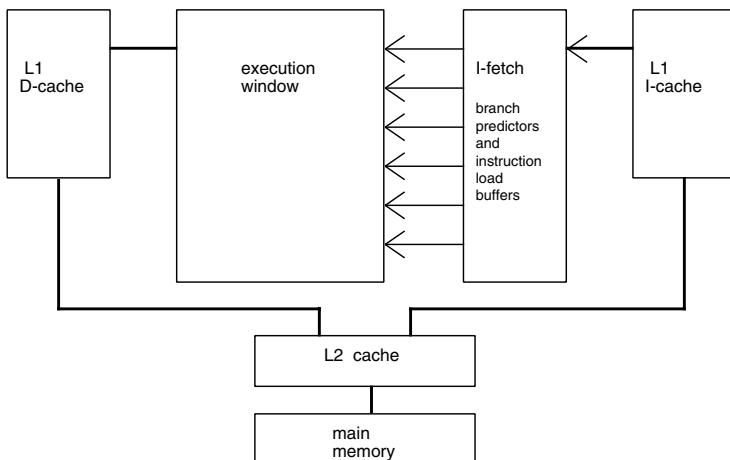


FIGURE 15.4: High-level view of a resource flow microarchitecture. Shown are the major hardware components of the microarchitecture. With the exception of the execution window block, this is similar to most conventional microarchitectures.

erally required. Instructions are immediately decoded after being fetched and any further handling of the instructions is done in their decoded form. Decoded instructions are then staged into an *instruction dispatch buffer* so that they are available to be dispatched into the *execution window* when needed. The execution window is where these microarchitectures differ substantially from existing machines. This term describes that part of the microarchitecture where the active stations and processing units are grouped. The instruction dispatch buffer is organized so that a large number of instructions can be broadside loaded (dispatched) into the active stations within the execution window in a single clock.

Figure 15.4 provides a high-level view of both microarchitectures. Multiple branch predictors have been used in the i-fetch stage so that several conditional branches can be predicted in parallel in order to retain a high fetch and dispatch rate. Research on the multiple simultaneous branch prediction and dispatch is ongoing. Instructions can be dispatched to active stations with or without initial input source operands. Instructions can also be dispatched with predicted input operands using value prediction. Once a new instruction is dispatched to an active station, once it has acquired its input operands (if it was not dispatched with predicted ones) it begins the process of contending for execution resources that are available to it (depending on microarchitectural implementation). Operand dependency determination is not done before either instruction dispatch (to an active station) or before instruction operation issue to an execution unit. All operand dependencies are determined dynamically through snooping after instruction dispatch. As

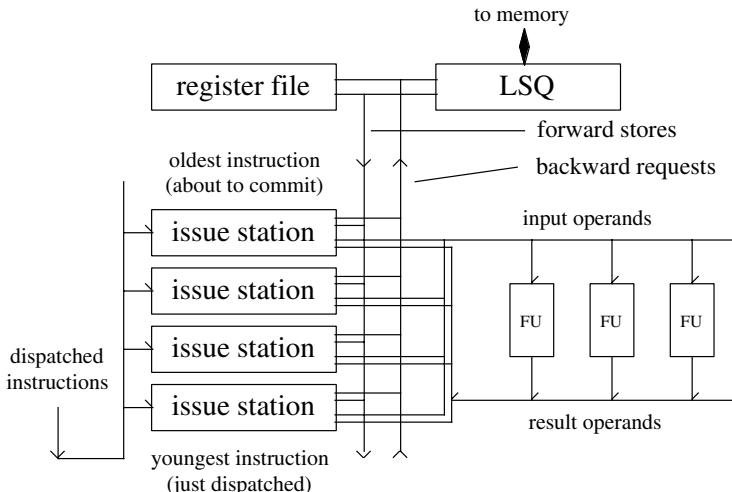


FIGURE 15.5: *High-level block diagram of a representative microarchitecture.* Active stations are shown on the left and various function units on the right. An architected register file and a load-store-queue is shown at the top. Bidirectional operand request and forwarding buses are shown vertically oriented (to the right of the active stations). Buses to transport an instruction operation and its source operands to the function units are also shown. Likewise buses to return result operands are present.

stated previously, instructions remain in their AS executing and possibly re-executing until they are ready to be retired. All retirement occurs in-order.

15.4.1 A Small Resource-Flow Microarchitecture

In addition to a fairly conventional memory hierarchy and fetch unit, this microarchitecture also has a load-store-queue (LSQ) component, an architected register file, execution function units, and the active station components discussed previously. Decoded instructions are dispatched from the fetch unit to one or more active stations when one or more of them are empty (available for dispatch) or becoming empty on the next clock cycle. Instructions are dispatched in-order. The number of instructions dispatched in any given clock cycle is the lesser of the number of ASes available and the dispatch width of the machine.

Figure 15.5 shows a block diagram of the execution window for this microarchitecture. In the top left of the figure is the architected register file. Since all operand renaming is done within the active stations, only architected registers are stored here. In the top right is the load-store-queue. The lower right shows (laid out horizontally) a set of function units. Each function unit (three

are shown in this example) is responsible for executing a class of instructions and generally has independent pipeline depths. Instructions not executed by a function unit are executed within the active station itself. This currently includes all control-flow change instructions as well as load-store instructions but this is largely ISA dependent. The lower left shows (laid out vertically) the set of active stations (four are shown in this example) that may be configured into an implementation of the machine. The ASes are all identical without regard to instruction type. This allows for all instructions to be dispatched in-order to the next available stations without further resource restrictions or management.

In the center of Figure 15.5, running vertically, two buses are shown. These bidirectional and multi-master buses form the means to request and forward operands among the ASes, register file, and LSQ. Each of these buses is actually a parallel set of identical buses that are statistically multiplexed to increase operand transfer bandwidth. Other bus arrangements are possible (some fairly complicated by comparison). In the present case, all buses can carry all operand types; however an implementation where separate bus fabrics for handling different types of operands is also possible. One of these buses is used by ASes for requesting source operands and has been termed a *backwarding request* bus. The name is derived from the fact that requested operands should only be satisfied by those instructions that lie in the program-ordered past from the instruction requesting the operand. The other bus is used for forwarding operands to younger instructions and is often termed the *forwarding* bus. Operands need to be forwarded from older dispatched instructions to younger dispatched instructions. The arrows on the buses show the direction of intended travel for operands (or operand requests) that are placed on each bus respectively. Although the register file and LSQ only receive operand requests on the backwarding request bus, the ASes both receive and transmit requests from their connections to that bus. Likewise, although the register file and LSQ only transmit operands on the operand forwarding bus, the ASes both receive and transmit on their connections to that bus.

Finally, unidirectional buses are provided to interconnect the ASes with the function units. One bus serves to bring instruction operations along with their source operands from an issue station to a function unit. The other bus returns function unit results back to its originating active station. Again these buses are generally multiple identical buses in parallel to allow for increased transfer bandwidth. It is assumed that all buses carry out transfers at the same clock rate as the rest of the machine including the execution function units. The number of ASes in any given machine implementation roughly corresponds to the number of elements of a reorder buffer or a register update unit in a more conventional machine. The number and types of function units can vary in the same manner as in conventional machines.

This particular microarchitecture is oriented towards implementing the resource flow ideas while having an overall machine size similar to existing high-end processors or to that of next generation processors. Specifically, large

scalability of the machine was not a goal with this design. However, the second representative microarchitecture introduced next is designed to explicitly address the issue of physical machine scalability.

15.4.2 A Distributed Scalable Resource-Flow // Microarchitecture

This microarchitecture is very aggressive in terms of the amount of speculative execution it performs. This is realized through a large amount of scalable execution resources. Resource scalability of the microarchitecture is achieved through its distributed nature along with repeater-like components that limit the maximum bus spans. Contention for major centralized structures is avoided. Conventional centralized resources like a register file, reorder buffer, and centralized execution units are eliminated.

The microarchitecture also addresses several issues associated with conditional branches. Alternative speculative paths are spawned when encountering conditional branches to avoid branch misprediction penalties. Exploitation of control and data independent instructions beyond the join of a hammock branch [3, 25] is also capitalized upon where possible. Choosing which paths in multipath execution should be given priority for machine resources is also addressed by the machine. The predicted program path is referred to as the *mainline* path. Execution resource priority is given to the mainline path over other possible alternative paths that are also being executed simultaneously. Since alternative paths have lower priority for resource allocation than the mainline path, they are referred to as *disjoint* paths. This sort of strategy for the spawning of disjoint paths results in what is termed *disjoint eager execution* (DEE). Disjoint alternative execution paths are often simply termed *DEE paths*. These terms are taken from Uht [26]. Early forms of this microarchitecture were first presented by Uht [28] and then Morano. [13] A more detailed explanation of this microarchitecture can be found in [27]. Figure 15.6 shows a more detailed view of the execution window of this microarchitecture and its subcomponents.

Several ASes may share the use of one or more execution units. The execution units that are dispersed among the ASes are termed *processing elements* (PEs). Each PE may consist of an unified all-purpose execution unit capable of executing any of the possible machine instructions or, more likely, consist of several functionally clustered units for specific classes of instructions (integer ALU, FP, or other). As part of the strategy to allow for a scalable microarchitecture (more on scalability is presented below), the ASes are laid out in silicon on a two-dimensional grid whereby sequentially dispatched instructions will go to sequential ASes down a column of the two-dimensional grid of ASes. The use of a two-dimensional grid allows for a design implementation in either a single silicon IC or through several suitable ICs on a multi-chip module. The number of ASes in the height dimension of the grid

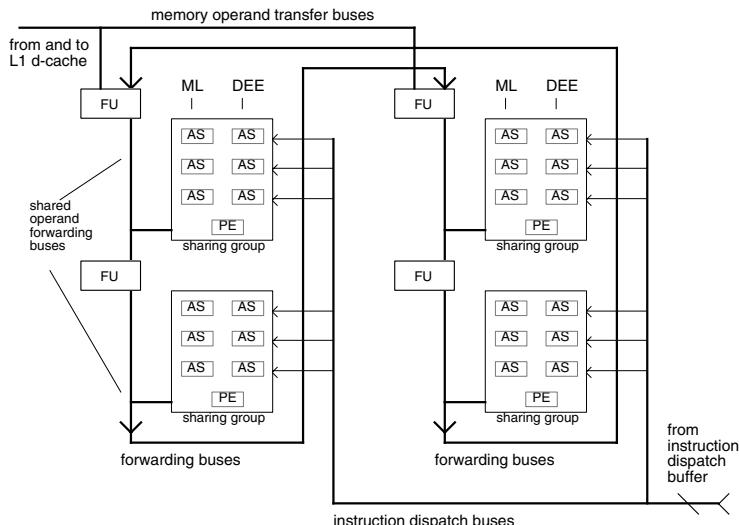


FIGURE 15.6: *The Execution Window of a distributed microarchitecture.* Shown is a layout of the Active Stations (AS) and Processing Elements (PE) along with some bus interconnections to implement a large, distributed microarchitecture. Groups of ASes share a PE; a group is called a *sharing group*.

is termed the *column height* of the machine and also is the upper limit of the number of instructions that can be dispatched to ASes in a single clock.

Groups of active stations, along with their associated PE, are called a *sharing group* (SG), since they share execution resources with the set of ASes in the group. Sharing groups somewhat resemble the relationship between the register file, reorder buffer, reservation stations, and function units of more conventional microarchitectures. They have a relatively high degree of bus interconnectivity amongst them, as conventional microarchitectures do. The transfer of a decoded instruction, along with its associated operands, from an AS to its PE is isolated to within the SG they belong to. The use of this execution resource sharing arrangement also allows for reduced interconnections between adjacent SGs. Basically, only operand results need to flow from one SG to subsequent ones. As part of the ability to support multipath execution, there are two columns of ASes within each SG. The first AS-column is reserved for the mainline path of the program and is labeled *ML* in the figure. The second column of ASes is reserved for the possible execution of a *DEE* path and is labeled *DEE* in the figure. Other arrangements for handling multipath execution are possible but have not been explored.

The example machine of Figure 15.6 has a column height of six, and is shown with six instruction load buses feeding up to six ASes in the height dimension. Each SG contains three rows of ASes (for a total of six) and a single PE. Overall this example machine consists of two columns of SGs, each

with two SG rows. A particular machine is generally characterized using the 4-tuple:

- sharing group rows
- active station rows per sharing group
- sharing group columns
- number of DEE paths allowed

These four characteristic parameters of a given machine are greatly influential to its performance, as expected, and the 4-tuple is termed the *geometry* of the machine. These four numbers are usually concatenated so that the geometry of the machine in Figure 15.6 would be abbreviated 2-3-2-2. The set of mainline AS columns and DEE AS columns should be viewed as two sets of columns (one overlaid onto the other) that are really only sharing common execution resources and interconnection fabric. Each set may be operated on rather independently based on the strategy (or strategies) employed for the management of alternative DEE paths. Also shown in the figure are buses to dispatch instructions from the i-fetch unit to the ASes and the more pervasive bus fabric to forward operands from ASes to program-ordered future ASes. The bus fabric and the units (labeled FU in the figure) are discussed in more detail below.

When an entire column of mainline ASes is free to accept new instructions, up to an entire column worth of instructions are dispatched in a single clock to the free column of ASes within that SG column. Newly dispatched instructions are always assigned to the column of ASes reserved for mainline execution. The same column of instructions may also get dispatched to the DEE column if it is determined to be advantageous to spawn an alternative execution path at dispatch time. Different methods for dispatching alternative path instructions to DEE columns of ASes are possible (including dynamically after initial dispatch) but these are beyond the scope of the present discussion. An attempt is made by the i-fetch unit to always have an entire column's worth of decoded instructions ready for dispatch, but this is not always possible. If less than a whole column of instructions are dispatched, those ASes not receiving an instruction are essentially unused and represent a wasted machine resource. Conditional branches are predicted just before they are entered into the instruction dispatch buffer and the prediction accompanies the decoded instruction when it is dispatched to an AS. Again, instructions remain with their AS during execution and possible re-execution until they are ready to be retired. Unlike the previous example microarchitecture, an entire column of ASes gets retired at once (not at the granularity of a single AS). As a column of ASes gets retired, that column becomes available for newly decoded instructions to be dispatched to it. It is also possible for more than a single column to retire within a single clock and therefore for more than a single column to be available for dispatch within a single clock. However this is not likely and research efforts so far have not yielded a machine capable of such high execution rates.

All of the ASes within an AS column (either a mainline column or a DEE column) are assigned a contiguous set of time-tag values. The AS at the top of the column has the lowest valued time-tag within it, and the value is increment by one for successive ASes down the column. The topmost AS of an adjacent AS mainline column will contain a time-tag value one higher than the previous column's AS having the highest valued time-tag (which will be located at the bottom of the previous column). The time-tag value assigned to ASes within a row and jumping from one SG column to an adjacent one (one mainline column of ASes to another in an adjacent SG column) have time-tag values differing by the height of a column as counted in ASes. At any one point in the machine operation, one column serves as the *leading* column with the AS having the lowest valued time-tag (generally zero) within it, while another column serves as the *trailing* column with the AS having the highest valued time-tag. The lead AS column contains those instructions that were dispatched earliest while the trailing AS column contains those instructions most recently dispatched.

It will be the leading mainline AS column that is always the one to be retired next. Upon a retirement of that column, the time-tags within ASes and operands within the machine are effectively decremented by an amount equal to the height of the machine in ASes (or column height). All time-tags can be decomposed into row and column parts. The column part of the time-tag serves as a name for the logical column itself. When the leading column is retired, all other columns effectively decrement their column name by one. This effectively decrements all time-tag within each column by the proper amount and serves to rename all of the operands within that column. With this column-row time-tag decomposition, the row part of the tag is actually fixed to be the same as the physical row number in the physical component layout in silicon. This means that upon column retirement, the column part of any time-tag is decremented by the number of columns being retired (generally just one). The next AS column in the machine (with the next higher time-tag name) becomes the new leading column and the next to get retired. The operation of decrementing column time-tags in the execution window is termed a *column shift*. The term was derived from the fact that when the whole of the execution window (composed of ASes and PEs) is viewed as being logically made up of columns with the lowest time-tag value one always at the far left, the columns appear to shift horizontally (leftward) on column retirement as if the components made up a large shift register, with each column of machine components being a single stage, and shifting right to left. In reality, columns are simply renamed when the column part of the time-tags are decremented, and the logical columns can be thought of as being rotated left.

15.4.2.1 Bus Interconnect and Machine Scalability

As with the previous example microarchitecture, some sort of interconnection fabric is used to allow for the backward requesting of operands among ASes as well as for the forwarding of result operands. Generally, several buses are used in parallel to handle the operand forwarding bandwidth requirement. All operand buses are multiplexed for use by all operand types. Multiplexing of all operands onto a single set of buses provides the best bandwidth efficiency but represents just one possible arrangement. The set of operand forwarding buses, partitioned into segments and separated by forwarding units (FUs in the figure), are shown vertically (primarily) and to left of each SG column. For clarity, the backwarding operand request buses are not shown in the figure.

In order to make the machine scalable to large sizes, all operand buses are of a fixed (constant) length, independent of the size of the machine (in numbers of components, ASes and PEs). This is accomplished through the use of forwarding units (previously discussed). This present microarchitecture actually uses three different types of forwarding units (one each for predicates, registers, and memory operands) but they are all shown combined in Figure 15.6 as the boxes labeled FU. Note how the operand forwarding bus segment bundles loop around from the bottom of one SG column to the top of the adjacent SG column to its right, with the forwarding bus bundle from the bottom of the rightmost SG column looped around to the top of the leftmost SG column. This is a logical arrangement only and does not necessarily represent how all components might be laid out in silicon. This looping arrangement for operand forwarding follows the familiar pattern characteristic of many proposed larger-scaled microarchitectures. [16]

For the register and memory operands, the forwarding units also provide filtering and are therefore RFUs and MFUs respectively. The register filter units (RFUs) used here contain a set of registers equal in number to that in the set of architected registers for the ISA being implemented. This provides a sort of 100% caching capability for registers. Of course, this may only be possible when the set of architected registers in the ISA (including status registers and any explicit predicate registers) is rather small – maybe less than 512 or 1024). Requests by ASes for register operands therefore will always “hit” in the preceding RFU (in program-ordered time) unless the register being requested has been invalidated for some reason due to the forwarding strategy employed. Generally different forwarding strategies are used for register, memory, and predicate operands but full details on this are not the present interest. Likewise, the memory filter units employ an internal L0 cache so that requests for memory operands have a chance of hitting in the MFU situated immediately preceding the AS making the request. Operand hits in either the RFUs or the MFUs also serve to reduce operand bus traffic. Operand misses (either RFU or MFU) are propagated backwards until a hit is achieved. For registers, a hit will always eventually be achieved even if the request has to

travel back to that RFU with the time-tag value of zero (which always has a valid value). Similarly for memory requests, misses propagate backwards; however a hit is not always guaranteed as with registers. A miss that travels back through that MFU that has a time-tag value of zero continues backward onto the L1 data case, and possibly into lower memory levels as well. A memory request, that misses in an MFU at the top of an SG column, switches from using the backwarding bus fabric to the bidirectional and multimaster *memory operand transfer buses* (also a duplicated bundle of buses) shown at the top of the figure. Memory requests returning from the L1 data cache enter an MFU at the top of the machine and get forwarded normally on the forwarding fabric until snarfed by a requesting AS. For this microarchitecture, MFUs also replace the LSQ function in more conventional microarchitectures.

15.4.2.2 Fetch Heuristics

If a conditional backward branch is predicted taken, the i-fetch unit will speculatively follow it and continue dispatching instructions into the execution window for the mainline path from the target of the branch. For a backward branch that is predicted not-taken, instructions following the not-taken output path are continued to be dispatched. If a forward branch has a near target such that it and its originating branch instruction will both fit within the execution window at the same time, then instructions following the not-taken output path of the branch are dispatched, whether or not that is the predicted path. This represents the fetching of instructions in the memory or *static* order rather than the program dynamic order. The fetching and dispatching of instructions following the not-taken output path (static program order) of a conditional branch is also advantageous for capturing hammock styled branch constructs. Since simple single-sided hammock branches generally have near targets, they are captured within the execution window and control flow is automatically managed by the dynamic instruction predicates. This is a more efficient way to handle simple single-sided hammocks than other some other methods [17].

15.4.2.3 Persistent Architected Register State

Persistent register, predicate, and some persistent memory state is stored in the forwarding units. Persistent state is not stored indefinitely in any single forwarding unit but is rather stored in different units as the machine executes column shift operations (columns of ASes get retired). However, this is all quite invisible to the ISA. This microarchitecture also implements precise exceptions [19] similarly to how they are handled in most speculative machines. A speculative exception (whether on the mainline path or a DEE path) is held pending (not signaled in the ISA) in the AS that contains the generating instruction until it would be committed. No action is needed for pending exceptions in ASes that eventually get abandoned (squashed). When an AS with a pending exception does commit, the machine directs the archi-

tected control flow off to an exception handler through the defined exception behavior for the given ISA.

15.4.2.4 Multipath Execution

DEE paths are created by dispatching instructions to a free column of ASes that is designated for holding DEE paths. All alternative paths are distinguished within the machine by a path ID value greater than one (mainline paths are always designated with a path ID value of zero). A variety of strategies can be employed for determining when spawning a DEE path is possibly advantageous for performance reasons, but generally it is when the outcome of a forward branch is either weakly predicted or if the target of the branch is both close in numbers of instructions to the branch and current DEE resources (a free column of DEE designated ASes) are abundant (can be possibly wasted). The spawning of DEE paths on the occurrence of backward conditional branches introduces some added complexity that has not been well explored. A more detailed and thorough discussion of multipath execution and disjoint eager execution was covered in Chapter 6.

15.5 Summary

A set of concepts and some basic microarchitectural components, along with an unrestrained model of speculative execution that has been termed resource flow computing, have been briefly presented. Also briefly presented was the design outline for two example microarchitectures where each embodies many of the resource flow ideas (although each to a different degree). These microarchitectures show how this model of computing might be implemented both for more modest sized machines (as compared with the next generation processors) and future large sized machines having approximately one billion or more transistors. Using the silicon technology design rules that were used for the Alpha EV8 processor, the second microarchitecture presented was estimated to have a transistor budget of approximately 600 million transistors for an 8-4-8-8 geometry machine configuration (256 ASes, issue width and number of PEs of 64, instruction dispatch and commit width of 32). Simulated performance of this configuration using existing memory performance specifications achieved a harmonic mean IPC across a mix of SpecInt benchmarks of about 4. Whether this level of performance is worth the amount of machine resources (as compared to other simpler alternatives) remains to be seen, but research is ongoing.

The basic resource flow technique as well as both microarchitectures presented can be applied to existing legacy ISAs (usually an important requirement in the commercial market). The design ideas presented are not oriented

towards all future processor needs but rather for those application needs where the maximum performance is required for a single thread of control.

References

- [1] Cleary J.G, Pearson M.W and Kinawi H. The Architecture of an Optimistic CPU: The Warp Engine. In *Proceedings of the Hawaii International Conference on System Science*, pages 163–172, Jan. 1995.
- [2] Farkas K.I., Chow P., Jouppi N.P. and Vranesic Z. The multicluster architecture: Reducing cycle time through partitioning. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 149–159, 1997.
- [3] Ferrante J., Ottenstein K., and Warren J. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.
- [4] Franklin M. and Sohi G.S. Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grained Parallel Processors. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 236–245, New York, NY, Dec 1992. ACM Press.
- [5] González J. and González A. Limits on Instruction-Level Parallelism with Data Speculation. Technical Report UPC-DAC-1997-34, UPC, Barcelona Spain, 1997.
- [6] Hinton G., Sager D., Upton M., Boggs D. Carmean D., Kyker A., and Roussel P. The microarchitecture of the Pentium-4 processor. *Intel Technical Journal - Q1*, 2001.
- [7] Intel. *Intel Itanium2 Processor Reference Manual for Software Development and Optimization*, June 2002.
- [8] Intel Corp. *iA-64 Application Developer's Architecture Guide*, 1999.
- [9] Kemp G.A. and Franklin M. PEWs: A decentralized dynamic scheduler for ILP processing. In *Proceedings of the 24th International Conference on Parallel Computing*, pages 239–246, 1996.
- [10] Lam M.S. and Wilson R.P. Limits of Control Flow on Parallelism. In *Proc. of ISCA-19*, pages 46–57. ACM, May 1992.
- [11] Lipasti M.H. and Shen J.P. Superspeculative Microarchitecture for Beyond AD 2000. *IEEE Computer*, 30(9), Sep 1997.

- [12] Morano D.A. Execution-time Instruction Predication. Technical Report TR 032002-0100, Dept. of ECE, URI, Mar 2002.
- [13] Morano D.A., Khalafi A., Kaeli D.R. and Uht A.K. Realizing high IPC through a scalable memory-latency tolerant multipath microarchitecture. In *Proceedings of MEDEA Workshop (held in conjunction with PACT'02)*, 2002.
- [14] Nagarajan R., Sankaralingam K., Burger D. and Keckler S.W. A design space evaluation of grid processor architectures. In *Proceedings of the 34th International Symposium on Microarchitecture*, New York, NY, Nov 2001. ACM Press.
- [15] Preston R.P., Badeau R.W., Bailey D.W., Bell S.L., Biro L.L., Bowhill W.J., Dever D.E., Felix S., Gammack R., Germini V., Gowan M.K., Gronowski P., Jackson D.B., Mehta S., Morton S.V., Pickholtz J.D., Reilly N.H., and Smith M.J. Design of an 8-wide superscalar RISC microprocessor with simultaneous multithreading. In *Proceedings of the International Solid State Circuits Conference*, Jan 2002.
- [16] Ranganathan N. and Franklin M. An empirical study of decentralized ILP execution models. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 272–281, New York, NY, October 1998. ACM Press.
- [17] Sankaranarayanan K. and Skadron K. A Scheme for Selective Squash and Re-issue for Single-Sided Branch Hammocks. IEEE Press, Oct 2001.
- [18] Schlansker M.S. and Rau B.R. EPIC: Explicitly parallel instruction computing. *Computer*, 33(2):37–45, Feb 2000.
- [19] Smith J.E. and Pleszkun A.R. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, Sep 1988.
- [20] Sohi G.S., Breach S., and Vijaykumar T.N. Multiscalar Processors. In *Proceedings of the 22th International Symposium on Computer Architecture*, New York, NY, Jun 1995. ACM Press.
- [21] Sundararaman K.K. and Franklin M. Multiscalar execution along a single flow of control. In *Proceedings of the International Conference on Parallel Computing*, pages 106–113, 1997.
- [22] Tjaden G.S. and Flynn M.J. Representation of concurrency with ordering matrices. In *Proceedings of COMPCON*, volume C-22, pages 752–761, Aug 1973.
- [23] Tomasulo R.M. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, Jan 1967.

- [24] Tsai J-Y. and Yew P-C. The superthreaded architecture: Thread pipelining with run-time data dependence checking and control speculation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 35–46, 1996.
- [25] Uht, A. K. An Efficient Hardware Algorithm to Extract Concurrency From General-Purpose Code. In *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, pages 41–50. University of Hawaii, in cooperation with the ACM and the IEEE Computer Society, January 1986. Second Place for Best Paper in Computer Architecture Track.
- [26] Uht A. K. and Sindagi V. Disjoint Eager Execution: An Optimal Form of Speculative Execution. In *Proc. MICRO-28*, pages 313–325. ACM, Nov 1995.
- [27] Uht A.K., Morano D., Khalafi A., Alba M., and Kaeli D. Levo – A Scalable Processor With High IPC. *Journal of Instruction Level Parallelism*, 5, Aug 2003.
- [28] Uht A.K., Morano D.A., Khalafi A., de Alba M., and Kaeli D. Realizing high IPC using time-tagged resource-flow computing. In *Proceedings of the the EUROPAR Conference*, Aug 2002.

Index

Φ-Insertion step, 315, 317, 317–318
801 processor, 42–43, 68–69

A

Abstract memory locations (LOCs), 313
Accuracy, 66–67, 71, 161
ACPC, *see* Actual collide, predicted collide
(ACPC) load promotion
ACPNC, *see* Actual collide, predicted non-collide
(ACPNC) load promotion
Active station (AS), 396
Active station idea, 399–402, 401
Actual collide, predicted collide (ACPC) load
promotion, 205
Actual collide, predicted non-collide (ACPNC)
load promotion, 205
Actual non-collide, predicted collide (ANCPNC)
load promotion, 205
Actual non-collide, predicted non-collide
(ANCPNC) load promotion, 205
Adaptive branch tree (ADT) machine, 151
Adaptive Object Code Reoptimization (ADORE),
278
Add instruction, 217
Address-based scheduling, 381–384, 382–383
Address calculation
basics, 190, 194
nonspeculative techniques, 189–192
speculative techniques, 192–194
taxonomy, 189–190
Address prediction
address calculation techniques, 189–194
basics, 187–190, 194–200, 211–212
characterization, 195–196, 195–198
compiler based speculation load promotion,
209–211
data dependence prediction, 228
definitions, 189–190
load promotion, 204–207, 209–211
memory bypassing, 207–209
nonspeculative address calculation
techniques, 191–192
predictability, 195–199, 195–201
prefetching, 193–194

prefetching mechanism combination,
199–200, 202
speculative address calculation techniques,
192–194
speculative memory disambiguation, 200–211
taxonomy, 189–194
terminology, 189–190
value predictability, 197–199, 199–201
Address Prediction and Data Prefetching (APDP),
200, 228
Address Reorder Buffer (ARB), 226, 235
Address Resolution Buffer (ARB)
data speculation, 226
inter-thread dependencies, 348
memory dependence prediction, 374
Princeton multiPath machine, 150
Address value, 305
Addresses
basics, 31
disambiguation, 225
fall-through, 95
resource flow microarchitectures, 400, 402,
404
target, 95
ADORE, *see* Adaptive Object Code
Reoptimization (ADORE)
ADT, *see* Adaptive branch tree (ADT) machine
Advance load check flag, 321
Advance load flag, 321
Advanced Load Address Table (ALAT), 210, 226,
235
Advanced RISC Machines (ARM), 122
Agassiz project, 349
ALAT, *see* Advanced Load Address Table
(ALAT)
Algorithms, branch prediction, 71
Aliases
analysis, 358
dataflow analysis, 311–314, 312–313
runtime profiling, 304–305
Align bits, 180
Alignment logic, 94
Alpha 21264 processor, 253, 374, 389
Alpha EV8, 394, 416
Alternative value predictors, 220–222
Ambiguous memory dependences, 360

- Amdahl's Law, 155
 Ammp benchmark, *see* Instruction precomputation
 Analysis, branch predication, 117–119
 Analysis of Variance (ANOVA) design, 261
 ANCPC, *see* Actual non-collide, predicted collide (ANCPC) load promotion
 ANCPCN, *see* Actual non-collide, predicted non-collide (ANCPCN) load promotion
 Anomalous behaviors, 67
 ANOVA, *see* Analysis of Variance (ANOVA) design
 Anti-dependence, 305–306
 APDP, *see* Address Prediction and Data Prefetching (APDP)
 Applications, data speculation, 234–235
 ARB, *see* Address Reorder Buffer (ARB); Address Resolution Buffer (ARB)
 Architectures
 branch predication, 119–123
 data cache prefetching, 162–163
 speculative, threads, 341–349
 speculative multithreaded, 336–339, 337–338, 349
 Argawal studies, 10
 Array prefetching, 163–167, 165
 AS, *see* Active station (AS)
 Assignment statements, 320
 Atlas architecture/computer, 33, 349
 Autonomous Prefetch Engine, 57
 Autonomy, branch prediction, 40–42, 41
 Availability of branch
 counters, 49, 49–50
 Decode History Table, 51, 51–52
 discriminators, 52
 dynamic branch prediction, 48
 group behaviors, 50–51
 hybrid predictors, 53
 implementation, 51, 52–53
 individual branch prediction, 50–51
 last branch actions, 51, 53
 multiple discriminators, 52
 profiling branch actions, 50
 static branch prediction, 48
 Average group behavior, 50
 Avid execution, 138–139, 151–152
- B**
- Babbage, Charles, 31
 Backward Slices, 349
 BAL, *see* Branch And Link (BAL)
 Balanced multipath tree, 140
 Bandwidth, 67, 156
- Base correlation, 178
 Basic Block/Basic Block Reuse, 137, 264
 BDD, *see* Binary decision diagram (BDD)
 Binary decision diagram (BDD), 118
 Binary path tag, 144, 144–145
 Boosting, memory dependence prediction, 359
 BPA, *see* Branch Prediction Accuracy (BPA)
 BPL, *see* Branch Path Length (BPL)
 Branch And Link (BAL), 60
 Branch flags, 94
 Branch flow, 43–45, 44
 Branch History Table
 basics, 55–56, 56
 branch prediction, 51
 entry size, 63–64
 operation, 56–57
 structural advantages, 60–61
 Branch instruction, 31, 34
 Branch mask, 94
 Branch on Count, 48
 Branch path, 137, 144–145
 Branch Path Length (BPL), 155–156
 Branch predication
 Advanced RISC Machines, 122
 analysis, 117–119
 architectures with predication, 119–123
 basics, 3, 109–112, 111
 compilation, 115–119
 Cydrome Cydra 5, 121, 121–122
 future directions, 129
 hardware design, 127–128, 127–129
 Hewlett-Packard PD, 119–120
 if-conversion, 116–117
 intermediate representation, 119
 Itanium processor (IA-64), 123–126, 125–126
 limited support systems, 122–123
 model, 112–115, 113
 optimization, 117–119
 Texas Instruments C6X, 122
 Branch prediction
 anomalous behaviors, 67
 autonomy, 40–42, 41
 availability of branch, 48–53
 bandwidth, 67
 basics, 70–72
 branch flow, 43–45, 44
 Branch History Table, 55–57, 56, 60–61, 63–64
 branch instruction, 31, 34
 complexity, 71
 contexts, 62–63
 control flow, 30–31
 counters, 49, 49–50
 dataflow, 30–31
 Decode History Table, 51, 51–52

- decode-time, 44, 46
delayed branch instruction, 43
design space, trace caches, 99
dimensions of branch instruction, 42–47
discriminators, 52
dynamic branch prediction, 48
economies of size, 64
ENIAC, 29–30
exotic prediction, 64
extensions, ISA, 34–35
fast prediction mechanisms, 67–68
fetch width, 58
group behaviors, 50–51
hybrid predictors, 53
implementation, 51, 52–53, 71–72
individual branches, 50–51
instruction prefetching, 40–42, 44, 46–47, 54
instruction-set architectures, 32–35
last actions caveat, 51, 53
limitations, 70
memory consistency, 36
mispredictions, 58
monolithic prediction, 68–69
multiple discriminators, 52
multithreading and multithreaded environments, 39–40, 69–70
observable order, 36
operand-address space, 65
operand space, 65
operating environment, 62–63
path-based approach, 52
performance importance, 36–42
pipelining, 37–39, 38, 43–45, 44
profiling branch actions, 50
returns, 58–60, 59
semantics, 34
simplicity, 70
stacks, 59, 59–62
static branch prediction, 48
subroutine calls, 58–60, 59
superscalar processing, 68–69
table updates, 66–67
tandem branch prediction, 66
target addresses, 54–62
trace cache coupling, 99
very high ILP environments, 68–70
virtuality, 33–34
von Neumann programming, 29–30
when to predict, 43–45, 44
working sets, 62–63
- Branch Prediction Accuracy (BPA), 140, 155–156
Branch resolution buses, 149
Branch Target Buffer (BTB)
accuracy, 66–67
anomalous behaviors, 67
basics, 55
branch mispredictions, 58
core fetch unit, 91–95, 102
exotic prediction, 64
fast prediction mechanisms, 67
fetch width, 58
instruction cache, 64
multithreaded environment, 69
operand space, 65
operation, 56, 56–57
partial matching, 98
predictor bandwidth, 67
size of entry, 63–64
stacks, 59–62
structure, 60–61
subroutine calls and returns, 58–60
superscalar processing, 69
tables update, 66–67
tandem branch prediction, 66
working sets and contexts, 62–63
- Branch Trace Buffer (BTB), 274
Branch tree, 136
Branch tree geometry, 138, 140–144
Breadth-first execution, 154
Brtop operation, 121
BTB, *see* Branch Target Buffer (BTB); Branch Trace Buffer (BTB)
Burman, Plackett and, design, 260–262
Burst Tracing, 273
Bus interconnect, 414–415
Bzip2 benchmark, *see* Instruction precomputation
- ## C
- Cache-conscious heap allocation, 172
Cache consistency table (CCT), 23–24
Cache design, 10, 10–14, 12
Cache miss isolation, 164
Call instruction, 59
Cascaded speculation, 322
CCT, *see* Cache consistency table (CCT)
CFG, *see* Control Flow Graph (CFG)
Challenges, data speculation, 235–237
Chalmer’s University project, 349
Check instructions
compilation and speculation, 301
Recovery Code, 321–322
speculative optimizations, 325
Chip multiprocessors (CMP), 147, 152–153, 157
CHT, *see* Collision History Table (CHT)
CISC architectures, 187
Citron studies, 264
Classically based hardware, 148–150

- Clustered Speculative Multithreaded architecture, 349
- CMP, *see* Chip multiprocessors (CMP)
- Coarse-grain multiprocessors, 153–154
- Coarse-procedure, 147
- Code layout, 280–281
- CodeMotion step, 315–316, 320–321, 321
- Collection, profile-based speculation, 272–275
- Collided loads, 202
- Collision History Table (CHT), 206–207
- Collision of address, 190
- Column height, 411
- Column shift, 413
- Committing process, 346–347
- Commonly used profiles, 270–271
- Compacted call paths, 308
- Compare bits, 180
- Compares, predicate defines, 124
- Compilation, branch predication, 115–119
- Compilation and speculation
 - Φ-Insertion step, 317, 317–318
 - aliases, 304–305, 311–314, 312–313
 - basics, 301–303, 302–303, 329–330
 - CodeMotion step, 320–321, 321
 - data dependence profiling, 305–310
 - dataflow analysis, 311–314, 312–313
 - Downsafety step, 319–320
 - multi-level speculation, 322–323, 323–325, 325
 - optimizations, 314, 321–322, 327, 327, 329, 329
 - Recovery Code, 321–329
 - Rename step, 318, 318–319
 - runtime profiling, 303–310, 304
 - speculative analysis framework, 310, 310–314
- Compilers
 - instruction cache prefetching, 26
 - multipath execution, 148
 - optimizations, 191
 - speculation, 4–5
 - speculation load promotion, 209–211
 - speculative optimizations, 194
- Compile-time profiling, 276
- Complexity, branch prediction, 71
- Computation reuse, 289–290
- Computation reuse buffer (CRB), 289
- CondéL2-based machine, 150
- Conditional branches, 34
- Confidence estimation, 137, 155, 222–223
- Confidence hits, 148
- Conflict loads, 202
- Conflict of address, 190
- Conjunctive predicate type, 114
- Conservative speculation, 319, 320
- Content-based prefetching, 179, 179–180
- Content-directed prefetching, 174
- Context-based predictor, 219, 221
- Context tag, 149
- Contexts, branch prediction, 62–63
- Continuous profiling, 277–278
- Control flow, 5, 30–31
- Control Flow Graph (CFG), 273
- Control flow profile, 270–271
- Control hazards, 2
- Control independence, threads, 344
- Core fetch unit, 91–92, 92–93, 94, 102
- Correlation Prefetching, 174, 176–179, 178
- Correlation Table, 176
- Counters, branch prediction, 49, 49–50
- Coverage, 161, 177
- CP, *see* Cumulative probability (CP)
- CPI, *see* Cycles Per Instruction (CPI)
- CRB, *see* Computation reuse buffer (CRB)
- Cumulative probability (CP), 140
- Cycles Per Instruction (CPI), 39
- Cydrome Cydra 5, 112, 121, 121–122

D

- DAISY systems, 148, 274, 276, 278
- DanSoft machine, 148
- DASP engine, 173
- Data, 32
- Data accesses, 156
- Data bypassing, 190
- Data cache misses, 11
- Data cache prefetching
 - architectural support, 162–163
 - array prefetching, 163–167
 - basics, 161–162
 - content-based prefetching, 179, 179–180
 - correlation prefetching, 176–179, 178
 - data locality optimizations, 172–173
 - hardware prefetching, 173–180
 - pointer prefetching, 168–172
 - software prefetching, 162–173
 - stride and sequential prefetching, 174–176, 175
- Data dependence, 4, 224–229, 227, 235
- Data dependence profiling, 305–310
- Data Driven Multithreaded Architecture, 349
- Data Event Address Register (D-EAR), 274
- Data Flow Graph (DFG), 188
- Dataflow, 30–31
- Dataflow analysis, 311–314, 312–313
- Data independence, 344
- Data layout, 281
- Data linearization prefetching, 169–170
- Data loads, 179

- Data locality optimizations, 172–173
 Data mining, 34
 Data prefetching, 282–286
 Data speculation
 alternative value predictors, 220–222
 applications, 234–235
 basics, 215–216
 challenges, 235–237
 confidence estimation, 222–223
 data dependence, 224–229, 227, 235
 data value speculation, 216–224, 218,
 233–235, 234
 future trends, 235–237
 implementation, 223–224
 multipath execution, 148
 predictors, data dependence, 226, 228–229
 recovery, 229, 232–233
 speculation, data dependence, 224–229, 227,
 235
 value predictors, 218–222, 221
 verification, 230–232
 Data stream speculation, 3–4
 Data value, 4
 Data value speculation, 216–224, 218, 233–235,
 234
 DCE, *see* Dynamic conditional execution (DCE)
 DDT, *see* Dependence Detection Table (DDT)
 Dead code removal, 279
 D-EAR, *see* Data Event Address Register (D-EAR)
 DEC Alpha, 122
 DEC GEM compiler, 123
 Decode History Table, 51, 51–52, 55
 Decode-time, 44, 46
 Decoupled architecture, 191
 DEE, *see* Disjoint eager execution (DEE)
 Degree of likeliness, 311
 Delayed branch instruction, 43
 Demand misses, 14
 Dempster-Shafer theory, 272
 Dependence Detection Table (DDT), 209
 Dependence distance, 308, 368
 Dependence edge, 305, 309
 Dependence history, 228–229, 230
 Dependence Prediction and Naming Table
 (DPNT), 209
 Dependence probability, 309, 309–310
 Dependence set locality, 366
 Dependencies, 357
 Dependency prediction, 190, 193–194
 Depth of branches, 136
 Design space, trace caches
 branch prediction coupling, 99
 core fetch unit, 102
 indexing strategy, 98
 instruction cache access, 102
 instruction cache/trace cache overlap, 101
 loop caches, 103
 L1 vs. L2 instruction cache, 103
 multi-phase trace construction, 100–101
 nonspeculative updates, 101
 parallel instruction cache access, 102
 partial matching, 98–99
 path associativity, 97
 serial instruction cache access, 102
 speculative updates, 101
 trace selection policy, 99–100
 Design tradeoffs, load promotion, 204–205
 DFG, *see* Data Flow Graph (DFG)
 Digital Equipment VAX, 397
 Dimensions of branch instruction, 42–47
 Direct mapped cache, 11
 Direct memory references, 314
 Disambiguation, 360
 Discriminators, 52
 Disjoint eager execution (DEE), 410–413, 416
 Disjoint splitting, 146
 Disjunctive predicate type, 114
 Distributed scalable resource-flow
 microarchitecture
 basics, 410–413, 411
 bus interconnect, 414–415
 fetch heuristics, 415
 machine scalability, 414–415
 multipath execution, 416
 persistent architected register state, 415
 Downsafety step, 315, 319–320
 DPNT, *see* Dependence Prediction and Naming
 Table (DPNT)
 Dynamic branch prediction, 48
 Dynamic conditional execution (DCE), 152
 Dynamic DEE tree, 141
 Dynamic dependency ordering, 397–398
 Dynamic methods, 359–361
 Dynamic Multithreaded Processor, 349
 Dynamic program order, 88
 Dynamo systems, 276

E

- Eager evaluation, 147, 154
 Eager execution, 3, 135
 Economies of size, 64
 Effective misrepresentation, 155
 Engines, 32
 ENIAC, 29–30
 Environmental instructions, 34
 EPIC, *see* Explicitly Parallel Instruction
 Computing (EPIC)

Epilogue loop, 167
Equake benchmark, 236, *see also* Instruction precomputation
 Equivalence class, 312
EU, *see* Execution Unit (EU)
 Evaluation, instruction precomputation, 260–262,
 261
 Execution state, 400
 Execution Unit (EU), 32
 Execution window, 407
 Exotic prediction, 64
 Explicitly Parallel Instruction Computing (EPIC),
 see also IMPACT EPIC
 architectures with predication, 120
 branch predication, 110
 compilation and speculation, 301
 memory dependence prediction, 359
 Extensions, ISA, 34–35

F

Fall-through address, 95, 97
 False dependence, 307, 361
 Fast prediction mechanisms, 67–68
 FastForward region (FFR), 279
 Fetch directed prefetching, 19, 21–22, 21–23
 Fetch heuristics, 415
 Fetch-load-to-use delay, 3, 188–189, 198, 211
 Fetch-load-to-use latency, 200
 Fetch width, 58
 FFR, *see* FastForward region (FFR)
 Figures list, xi–xiii
 Filter bits, 180
 Filter units, 405
 Filtering, operands, 404–406
 Fine instruction, granularity, 146
 Firing tables, 30
 First class entity, 396–397
 Flattened hierarchical verification, 231–232
 Flow dependence, 305–306
 Forked branches, 140
 Forked execution, 137
 Forwarding, operands, 403–404, 405
 Forwarding units, 405, 414
 Frequency, 258–259, 259, 301
 Frequency-latency product (FLP), 250, 258–259,
 259
 Fully extended call paths, 307–308
 Fully redundant, compilation and speculation, 315
 Function, granularity, 147
 Functional language machines, 154
 Future directions
 branch predication, 129
 data speculation, 235–237

instruction cache prefetching, 26–27
 multipath execution, 157

G

Gcc benchmark, *see* Instruction precomputation
 Geometry, machine, 412
 Global Actions, 65
 Global Branch Actions, 53
 Global Branch Predictions, 53
 Global variables, 304
 Gonzalez, A., studies
 data speculation, 215–237
 instruction precomputation, 263
 multithreading and speculation, 333–350
 Granularity, 146–147
 Greedy prefetching, 168–169
 Green instructions, 35
 Grid Architecture, 396
 Group average, 50
 Group behaviors, 50–51
 Guesses, 50
 Gzip benchmark, *see* Instruction precomputation

H

The Halting Problem, 31
 Handling multipath execution, 398–399
 Hardware
 branch predication, 127–128, 127–129
 multipath execution, 148–152
 profile-based speculation, 273–274
 Hardware prefetching
 basics, 161, 173–174
 content-based prefetching, 179, 179–180
 correlation prefetching, 176–179, 178
 stride and sequential prefetching, 174–176,
 175
 Heap objects, 304
 Hedge-BTBlookups, 63–64
 Hedge fetching, 45
 Helper threads
 basics, 192, 339–340
 building, 340
 microarchitectural support, 341
 multithreading and speculation, 335–336
 Heuristic dynamic skewed trees, 142, 148
 Heuristic skewed trees, 138, 143
 Heuristic static DEE tree, 138, 141–143
 Hewlett-Packard PA8000 processors, 226, 235
 Hewlett-Packard PD, 119–120, 124
 Hierarchical verification, 231
 Highly likely, *see* Degree of likeliness

History pointer arrays, 171
 Hit-miss prediction, 192
 Hits, 26, 57
 Hot-cold optimizations, 279–280
 Hot data stream prefetching, 284–286
 Hybrid predictors, 53
 Hybrid retirement based verification, 231
 Hybrid static/dynamic methods, 359, 360
 Hydra chip multiprocessor, 275, 349
 Hyperblock compilation framework, 116

I

IACOMA project, 349
 IA32EL system, 276
 IAS machine, 29, 33–34
 IBM 360 machines, 43, 136, 148, 157
 IBM Power 4, 173, 176, 394
I-EAR, *see* Instruction Event Address Register (I-EAR)
 If-block structure, 321, 323, 325–326, 330
 If-condition, 322
 If-conversion, 111, 116–117
 If-then statement, 325
 ILP, *see* Instruction-Level Parallelism (ILP)
 Immediate successor misses, 177
 IMPACT EPIC, 112–115, 113, 119, 123–124
 Implementation
 data speculation, 223–224
 lookups, 51, 52–53
 memory dependence prediction, 369–373
 prediction mechanisms, 71–72
 Implied path, 145
 Indexed arrays, 167
 Indexing strategy, 98
 Indirect array reference, 167
 Indirect memory references, 314
 Individual branches, 50–51
 In-flight operations, 189, 201
 In-order architecture, 193
 Input dependence, 305–306
 Input sets, 255–257, 256, 258
 Instruction address, 400
 Instruction cache, 101–103
 Instruction cache prefetching
 basics, 9
 cache design, 10, 10–14, 12
 compiler strategies, 26
 direct mapped cache, 11
 fetched directed prefetching, 19, 21–22, 21–23
 future challenges, 26–27
 integrated prefetching, 23–24, 25
 next line prefetching, 14–15, 15
 nonblocking caching, 18, 18–19, 20

out-of-order fetch, 18, 18–19, 20
 prefetching, 14–26
 pseudo associative cache, 13
 scaling trends, 9–10
 set associative cache, 11, 13
 stream buffers, 15–18, 17
 target prefetching, 15, 15
 way prediction cache, 13–14
 wrong-path prefetching, 24, 26
 Instruction cache/trace cache overlap, 101
 Instruction cross data (IXD), 65
 Instruction dispatch buffer, 407
 Instruction Event Address Register (I-EAR), 274
 Instruction fetch, 33, 156
 Instruction fetch unit, 89–97
 Instruction-Level Parallelism (ILP)
 basics, 5, 37
 branch predication, 109–110, 117–119
 branch prediction, 68
 code layout, 280
 eager execution, 135
 memory dependence prediction, 387–388
 multipath execution, 137
 multithreading, 40, 42, 333–334
 profile-based speculation, 278–281
 resource flow microarchitectures, 395
 Instruction operation, 400
 Instruction precomputation
 basics, 245–246, 250–251, 250–252, 265
 combination of input sets, 257, 258
 different input sets, 255–257, 256
 evaluation, analytical, 260–262, 261
 frequency vs. frequency-latency product, 258–259, 259
 performance, 253, 255–260, 260
 Profile A, Run A, 253, 255, 255
 Profile AB, Run A, 257, 258
 Profile B, Run A, 255–257, 256
 redundant computations, 247–248, 248–249
 related work, 263–264
 simulation methodology, 253, 254
 speculation incorporation, 262–263
 upper-bound performance, 253, 255, 255
 value reuse, 246–247, 252, 259–260, 260
 Instruction prefetching, 40–42, 44, 46–47, 54
 Instruction re-issue, 233
 Instruction-Set Architectures (ISA)
 address prediction, 187
 branch prediction, 32–35, 42
 subroutine routine, 58
 Instruction stream speculation, 2–3
 Instruction Unit (IU), 32–33
 Instructions, 29, 32
 Instructions Per Cycle (IPC), 37, 137
 Instrumentation, 273

Integrated prefetching, 23–24, 25
 Intel Itanium processor (IA-64)
 address prediction, 193, 209–210
 branch predication, 119, 123–126, 125–126
 compilation and speculation, 301, 330
 data speculation, 226, 235–236
 profile-based speculation, 275, 278
 resource flow microarchitectures, 403
 software prefetch instructions, 26
 speculative optimizations, 319
 Intel Itanium processor family (IPF), 274
 Intel Pentium 4
 address prediction, 192, 206
 multipath execution, 154
 resource flow microarchitectures, 176, 394
 Intel Pentium Pro processor, 122, 203
 Intermediate representation, 119
 Inter-thread dependencies, 347–349
 Intuiting, 34
 Invalidation, 232–233
 IPC, *see* Instructions Per Cycle (IPC)
 IPF, *see* Intel Itanium processor family (IPF)
 Irregular accesses, 176
 Irregular array reference, 167
 Irregular memory access patterns, 163
 Issue resources, 231
 Issues, 154–156, 290–291
 Itanium, *see* Intel Itanium processor (IA-64)
 Iterated Dominance Factors, 317
 Iteration-based probability, 309–310
 IU, *see* Instruction Unit (IU)
 IXD, *see* Instruction cross data (IXD)

J

Jacobi kernel, 164, 166
 Java runtime parallelizing machine, 287–288
 JIT compiler system, 286
 JRPM system, 274
 Jump pointers, 168, 170–171, 171

K

Kin machine/processor, 138–139, 147, 151–152
 Knuth studies and algorithm, 273, 275

L

Last actions, 53
 Last actions caveat, 51, 53
 Last branch actions, 53, 66
 Last Fetched Store Table (LFST), 229

Last outcomes, 53
 Last-value predictor, 221
 Latency reduction/tolerance, 172
 Latest load/store, 306
 Lazy evaluation, 154
 LDS, *see* Linked data structure (LDS)
 Length, 39
 Levo microarchitecture, 138, 146–148, 150–151
 LFST, *see* Last Fetched Store Table (LFST)
 Likeliness, 311
 Likely taken/likely non-taken, 48
 Limitations, branch prediction, 70
 Limited support systems, 122–123
 Linear function test replacement, 315
 Line size, 11
 Linked data structure (LDS), 168–171
 Live paths, 145
 Load context, 60
 Load-fetch-to-schedule, 194
 Load-fetch-to-use delay, 211, *see also* Fetch-load-to-use delay
 Load operations, 30, 188, 191
 Load promotion, 190, 204–207, 206, 209–211
 LoadStore cache, 208
 Load/store parallelism, *see also* Memory dependence prediction
 address-based scheduling, 381–384, 382–383
 exploiting, 357–361
 memory dependence prediction, 356
 performance potential, 378–379, 379
 Load-store-queue (LSQ) component, 408–409
 Locality analysis, 164
 Local variables, 304
 Location in memory, 31
 Lock-free instruction cache, 18
 LOCs, *see* Abstract memory locations (LOCs)
 Logic language machines, 154
 Loop caches, 103
 Loop peeling, 166
 Loop unrolling, 166
 Loops, 308
 LSQ, *see* Load-store-queue (LSQ) component
 L1 vs. L2 instruction cache, 103, 406

M

M88ksim benchmark, *see* Instruction precomputation
 Machine scalability, 414–415
 Magid machine, 150
 Mainline path, 137, 410
 Maintenance during optimizations, 291
 Malloc function, 304–305
 MAP, *see* Memory Address Prediction (MAP)

- Masking logic, 94
Matching, partial, 98–99
May modify operator, 311
May reference operator, 311
MCD, *see* Minimal Control Dependencies (MCD)
MDPT, *see* Memory dependence prediction table (MDPT)
MDST, *see* Memory dependence synchronization table (MDST)
MediaBench benchmark suite, 264
Memory, resource flow microarchitectures, 396, 399
Memory Address Prediction (MAP), 228
Memory based dependency and operations, 188–189
Memory bypassing, 207–209
Memory Conflict Buffer (MCB), 359
Memory consistency, 36
Memory dependence prediction
 address-based scheduling, 381–384, 382–383
 basics, 355–357, 387–389
 dynamic methods, 359–361
 experimental results, 374–386
 hybrid static/dynamic methods, 359, 360
 implementation framework, 369–373
 load/store parallelism, 357–361, 378–379, 379, 381–384, 382–383
 memory dependence speculation, 361–365, 363–364
 methodology, 375–378, 376–377
 mimicking, 360, 365–368, 367
 multiple dependences, 373, 373
 naive memory dependence speculation, 379–380, 380
 performance potential, 378–379, 379
 related work, 374
 results, 374–386
 speculation, 382, 384–386, 386–387
 static methods, 358, 359
 synchronization, 382, 384–386, 386–387
 working example, 370–371, 372
Memory dependence prediction table (MDPT), 369–371, 385
Memory dependence speculation, 356, 361–365, 363–364, 387
Memory dependence speculation and synchronization, 388
Memory dependence synchronization table (MDST), 369–371, 385
Memory Disambiguation Table (MDT), 348
Memory filter unit (MFU), 406, 414–415
Memory level parallelism register tracking, 192
Memory load reference, 313
Memory operand storage, 402–403, 403
Memory operand transfer buses, 415
Memory operations, 202–204, 204
Memory Order Buffer (MOB), 189, 201, 209
Memory profile, 271
Memory-side hardware prefetching, 173
Memory store reference, 313
Memory wall, 3
Mesa benchmark, *see* Instruction precomputation
MFU, *see* Memory filter unit (MFU)
Microarchitectural support, 341, 345–349
Microarchitecture examples, 148–154
Microarchitectures speculation, 6
Mimicking, 360, 365–368, 367
Minimal Control Dependencies (MCD), 153
MinneSPEC, 253
Minsky’s Conjecture, 36, 70
MIPS/MIPS R10000, 187, 253, 375
Mispredictions, 58, 136
Miss information status history register (MSHR), 18–19, 21, 23
Mississippi Delta prefetching (MS Delta), 286, 287
MOB, *see* Memory Order Buffer (MOB)
Models, 112–115, 113, 276–278
Modified locations, 313
Monolithic prediction, 68–69
Most likely correct, 301
Most recently used (MRU) order, 177–178
Motivation, multipath execution, 136–137, 138
Mowry’s algorithm, 164–167, 165
MS Delta, *see* Mississippi Delta prefetching (MS Delta)
Mul instruction, 226
MultiCluster machine, 395
Multiflow Trace 300 series, 123
Multi-level speculation, 321–323, 323–325, 325
Multipath execution
 Amdahl’s Law, ILP version, 155
 bandwidth, 156
 basics, 3, 135–136, 157
 branch path/instruction ID, 144–145
 branch tree geometry, 138, 140–144
 characterization, 137–148
 classically based hardware, 148–150
 compiler-assisted, 148
 confidence estimation, 155
 data speculation, 148
 essentials, 136–137, 138
 functional language machines, 154
 future directions, 157
 granularity, 146–147
 hardware, 148–152
 issues, 154–156
 logic language machines, 154
 microarchitecture examples, 148–154
 motivation, 136–137, 138

multiprocessors, 152–154
 non-classically based hardware, 150–152
 operation phases, 145–146
 pipeline depth, 155
 predication, 147, 154
 resource flow microarchitectures, 416
 status, 157
 taxonomy, 137–148, 138–139
Multi-phase trace construction, 100–101
Multiple-branch predictor, 91
Multiple-definition problem, 127
Multiple dependences, 373, 373
Multiple discriminators, 52
Multiple workload stability, 290
Multiprocessors, 152–154
Multiscalar processors and architectures, 226, 349
Multithreading and multithreaded environments,
see also Threads
 basics, 40
 branch prediction, 69–70
 processors, 333
Multithreading and speculation, *see also Threads*
 architectures, 336–339, 337–338, 349
 basics, 4–5, 333–335, 349–350
 committing process, 346–347
 helper threads, 339–341
 microarchitectural support, 341, 345–349
 spawning, 341–345, 343
 speculative architectural threads, 341–349
 speculative state storage, 347

N

Naive dynamic memory disambiguation
 technique, 205–206, 206
Naive memory dependence speculation, 363,
 379–380, 380
Names, 399
Natural pointer techniques, 168–170, 169
NET, *see* Next Executing Tail (NET)
Next Executing Tail (NET), 277
 Next line prefetching, 14–15, 15
 Nonblocking caching, 18, 18–19, 20
 Nonclassically based hardware, 150–152
 Nonconflicting load promotion, 204
 Nonspeculative address calculation techniques,
 191–192
 Nonspeculative code, 322
 Nonspeculative updates, 101
 Non-strided accesses, 176
 Not anticipated expression, 317
Not Taken, *see* Taken/Not Taken path ID

O

Observable order, 36
OoO, *see* Out-of-Order (OoO)
Opcode field, 33
Operand-address space, 65
Operand block, 400
Operand space, 65
Operands
 basics, 32
 fetch, 33
 filtering, 404–406
 first class entity, 396–397
 forwarding, 403–404, 405
 memory, storage, 402–403, 403
 resource flow microarchitectures, 402
 snooping, 403–404, 405
Operating environment, 62–63
Operating System, 35
Operation, trace caches, 92, 94–95, 97, 97
Operation code, 33
Operation phases, 145–146
Optimizations, maintenance during, 291
Optimizations, speculative
 ΦInsertion step, 317, 317–318
 basics, 314–316, 316
 CodeMotion step, 320–321, 321
 conservative speculation, 319, 320
 Downsafety step, 319–320
 profile-driven speculation, 320
 Rename Step, 318, 318–319
Original load instruction, 322
Otherthan call statements, 314
Out-of-Order (OoO)
 address prediction, 187–189, 193, 200, 203
 fetch, 18, 18–19, 20
 memory dependence prediction, 356, 370
Output dependence, 305–306

P

Parallel access cache, 13
Parallel Execution Window processor, 395
Parallel instruction cache access, 102
Parallelism motivation, 395–396
Parser benchmark, *see* Instruction precomputation
Partial dead-code elimination (PDE), 280
Partial-disambiguation, 225
Partial matching, 98–99
Partial redundancy, 317
Partial redundancy elimination (PRE)
 basics, 5
 compilation and speculation, 310, 314–315,
 328–329, 329

- profile-based speculation, 280
Partial reverse if-conversion, 116
Partially anticipated expression, 317
Partially anticipated speculatively expression, 317, 320
Partially ready (P-ready) code motion algorithm, 327
Path associativity, 97
Path-based approach, 52
Path IDs, 144–145, 398, 402
Paths, 135
PDE, *see* Partial dead-code elimination (PDE)
Pentium 4, *see* Intel Pentium 4
Pentium Pro, *see* Intel Pentium Pro processor
Performance
 branch prediction, 36–42
 instruction precomputation, 253, 255–260, 260
 memory dependence prediction, 378–379, 379
Performance monitoring unit (PMU), 273–274, 286
Persistent architected register state, 415
Perturbation, 291
PEs, *see* Processing elements (PEs)
Pipeline depth, 39, 155
Pipeline stage(s), 39, 139, 146
Pipelining, 37–39, 38, 43–45, 44
Plackett and Burman design, 260–262
PMU, *see* Performance monitoring unit (PMU)
Pointer-chasing codes, 168
Pointer-chasing problem, 168
Pointer prefetching, 168–171, 169, 171
Polypath machine, 149
Power 4, *see* IBM Power 4
Power 620 processor, 374
PRE, *see* Partial redundancy elimination (PRE)
Precomputation Table (PT), 247, *see also*
 Instruction precomputation
Predicate, 396, 399
Predicate clearing and setting, 115
Predicate comparison instructions, 113
Predicate Decision Logic Optimization (PDLO), 118
Predicate information, 400
Predicate promotion, 118
Predicate saving and restoring, 115
Predication, *see also* Branch predication
 basics, 110
 multipath execution, 147, 154
 profile-based speculation, 280
Predictability, 195–199, 195–201
Predicted/not predicted path, 138, 145
Predictions, 71
Predictors, data dependence, 226, 228–229
Predict Taken, 49
Prefetch Address, 40
Prefetch arrays, 170, 171
Prefetch buffers, 163
Prefetch distance, 166
Prefetch instruction, 163
Prefetch pointer generation loop, 171
Prefetch scheduling, 166
Prefetching, 40, 192–193, 199–200, 202, *see also*
 specific type
Previous value, 403
Princeton multiPath (PrincePath) machine, 150, 154
Probability, 301
Probable Calling Branch, 61
Procedure calls, 306–308, 307
Process, 145, 147
Processing elements (PEs), 410
Processor-side hardware prefetching, 173
Profile A, Run A, 253, 255, 255
Profile AB, Run A, 257, 258
Profile B, Run A, 255–257, 256
Profile-based speculation
 basics, 260–270, 292
 code layout, 280–281
 collection, 272–275
 commonly used profiles, 270–271
 compile-time profiling, 276
 computation reuse, 289–290
 continuous profiling, 277–278
 control flow profile, 270–271
 data layout, 281
 data prefetching, 282–286
 hardware performance monitoring, 273–274
 hot-cold optimizations, 279–280
 hot data stream prefetching, 284–286
 instruction level parallelism, 278–281
 instrumentation, 273
 issues, 290–291
 Java runtime parallelizing machine, 287–288
 maintenance during optimizations, 291
 memory profile, 271
 Mississippi Delta prefetching, 286, 287
 multiple workload stability, 290
 perturbation, 291
 program change, 291
 runtime profiling, 276–277
 software-based speculative precomputation, 290
 software-hardware collaborative profiling, 275
 special hardware, 274–275
 speculative parallel threading, 288–289
 static analysis, 272
 stride prefetching, 282–283, 282–284, 285
 thread level speculations, 287–290

- trace scheduling, 278–279
 - updating, 291
 - usage models, 276–278
 - value profile, 271
 - Profile-driven speculation, 320
 - Profiling branch actions, 50
 - Program, 29
 - Program changes, 291
 - Programming model, 32
 - Prolog benchmarks, 154
 - Prologue loop, 167, 171
 - Pruning, 145–146
 - Pseudo associative cache, 13
 - Pseudo-random trees, 138, 142, 144
 - Pure eager execution, 135

 - R**
 - Read-after-read operations, 201, 209
 - Read-after-write operations, 201
 - Real variables, 311
 - Recovery, 229, 232–233
 - Recovery block, 321
 - Recovery Code, 321–329
 - Recurrent loads, 179
 - Reduced Instruction Set Computer (RISC), 43, 68, 187
 - Redundant computations, 247–248, 248–249
 - Reference-based probability, 309
 - Reference ID, 305
 - Reference locations, 313
 - Register, 396, 399, 402–403, 403
 - Register allocation, 280
 - Register based dependency and operations, 188–189
 - Register filter unit (RFU), 406, 414–415
 - Register promotion, 315
 - Register tracking, 191
 - Regular memory access patterns, 163
 - Rename step, 315, 318, 318–319
 - Renaming, 399
 - Replicated correlation, 178
 - Resolution buses, branch, 149
 - Resolved branches, 135
 - Resource Flow Computing, 394
 - Resource flow microarchitectures
 - active station idea, 399–402, 401
 - basics, 393–395, 416
 - distributed scalable resource-flow
 - microarchitecture, 410–416, 411
 - dynamic dependency ordering, 397–398
 - filtering, operands, 404–406
 - first class entity, 396–397
 - forwarding, operands, 403–404, 405
 - handling multipath execution, 398–399
 - memory operand storage, 402–403, 403
 - names, 399
 - operands, 396–397, 402–406, 403, 405
 - parallelism motivation, 395–396
 - register, 402–403, 403
 - renaming, 399
 - resource flow, 396–406
 - result forwarding buses, 404–406
 - small resource-flow microarchitecture, 408, 408–410
 - snooping, operands, 403–404, 405
- Result forwarding buses, 404–406
- Retirement based verification, 231
- Retirement resources, 231
- Return instruction, 59
- Returns, 58–60, 59
- Reverse if-conversion, 119
- RFU, *see* Register filter unit (RFU)
- Right predictions, 49, 67
- RISC, *see* Reduced Instruction Set Computer (RISC)
- Runahead threads, 70
- Runtime data access reordering, 172
- Runtime disambiguation, 358
- Runtime profiling
 - alias profiling, 304–305
 - basics, 303–304, 304
 - data dependence profiling, 305–310
 - dependence probability, 309, 309–310
 - loops, 308
 - procedure calls, 306–308, 307
 - profile-based speculation, 276–277
 - shadow variables, 305–306, 306
-
- S**
- Same memory location, 305
- SAT, *see* Speculative Architectural Threads (SAT)
- Scaling trends, 9–10
- Scheduling, 280
- Selection policy, 99–100
- Selective Eager Execution (SEE), 149
- Selective memory dependence speculation, 364
- Semantics, 34
- Sequential basic blocks, 94
- Sequential prefetching, 173, *see also* Stride and sequential prefetching
- Serial access cache, 13
- Serial instruction cache access, 102
- Set associative cache, 11, 13
- SF, *see* Synonym File (SF)
- SG, *see* Sharing group (SG)
- Shadow memory, 305

- Shadow variables, 305–306, 306
Sharing group (SG), 411
Short branches, 152
Sideline path, 137
Signal operation, 367
Silent forwards, 405
Simplicity, 70
Simulation methodology, 253, 254
Single path tree, *see* Unipath tree
Sink-based dependence, 309–310
Size, 403
Size economies, 64
Skewed multipath trees, 140–144
Skip Lists, 170
Small resource-flow microarchitecture, 408, 408–410
Snarfing, 404
Snooping, operands, 403–404, 405
Software-based speculative precomputation, 290
Software-hardware collaborative profiling, 275
Software pipelining, 166
Software prefetching
 architectural support, 162–163
 array prefetching, 163–167
 basics, 161–162
 indexed arrays, 167
 Mowry’s algorithm, 164–167, 165
Source-based dependence, 309–310
Space, 305
SPARC V9, 122
Spawning, 137, 342–346, 343
SPEC89 benchmark suite, 278
SPEC95 benchmark suite
 data speculation, 229
 instruction precomputation, 263
 memory dependence prediction, 375, 376
SPEC2000 benchmark suite, 314, 323
SPEC-CPU benchmark suite, 118
SPECfp95 benchmark suite, 228, 376
SPECint95 benchmark suite, 275
SPECINT2000 benchmark suite
 data speculation, 222, 236
 instruction precomputation, 247, 253
 profile-based speculation, 277–278, 282, 284, 286, 289
SPECint/SPECfp benchmark suite, 152–154, 197–198
SPEC JBB2000 benchmark suite, 286
Speculation
 instruction precomputation, 262–263
 memory dependence prediction, 382, 384–386, 386–387
Speculation, data dependence, 224–229, 227, 235
Speculation flag, 319–320
Speculation/synchronization, 388
Speculative address calculation techniques, 192–194
Speculative analysis framework, 310, 310–314
Speculative Architectural Threads (SAT), 335–338, 341–349
Speculative check statements, 320
Speculative code, 322
Speculative disambiguation, 359
Speculative full redundancy, 317
Speculative load instruction, 322
Speculatively redundant, 316
Speculative Memory Bypassing, 228–229
Speculative Memory Cloaking, 228
Speculative memory disambiguation
 basics, 200–201
 characterization, 201–204
 compiler based speculative load promotion, 309–211
 load promotion, 204–207
 memory bypassing, 207–209
Speculative multithreaded architecture, 335
Speculative parallel threading (SPT), 288–289
Speculative state storage, 347
Speculative thread-level parallelism (TLP), 335
Speculative thread loops (STLs), 275, 287
Speculative updates, 101
Speculative Versioning Cache, 348
Speculative weak updates, 312, 317–321
Split branch, 137, 145
SPMD machine, 154
SPSM architecture, 349
SPT, *see* Speculative parallel threading (SPT)
Squash invalidation, 362
Squashing, 3, 217
SSA, *see* Static Single Assignment (SSA)
SSID, *see* Store set identifier (SSID)
SSIT, *see* Store Set Identifier Table (SSIT)
Stacks, 59, 59–62
STAMPede project, 349
StarJIT system, 276
State, 35–36
Static analysis, 272
Static branch prediction, 48
Static DEE tree
 basics, 138, 143
 hardware, 150–151
 instruction fetch, 156
 theory-based trees, 141–142, 143
Static disambiguation, 358
Static guesses, 48
Static instruction pair, 369
Static methods, 358, 359
Static program order, 88
Static sequence of instructions, 29

- Static Single Assignment (SSA), 310, 313–320,
see also Speculative analysis framework
- Steady state loop, 167, 171
- STL, *see* Speculative thread loops (STLs)
- Storage for speculative state, 347
- Store barrier approach, 374
- Store barrier cache, 374
- Store operations, 30, 188
- Store parallelism, *see* Memory dependence prediction
- Store set identifier (SSID), 229
- Store Set Identifier Table (SSIT), 229
- Store set method, 207
- Stored program computer, 32
- Stream buffers, 15–18, 17
- Stream tags, 69
- Streams, 26
- Strength reduction, 315
- Stride and sequential prefetching, 174–176, 175
- Stride prefetching, 173, 282–283, 282–284, 285
- Stride-value predictor, 219, 221
- Stuff_bar operation, 121–122
- Stuff operation, 121–122
- Subroutine calls, 58–60, 59
- Subroutine Entry Point, 62
- Subroutine Return, 61–62
- Superscalar processing, 68–69
- Superspeculative architecture, 395
- Superthreaded architecture, 349, 395
- Synchronization, 382, 384–386, 386–387, 388
- Synonym File (SF), 209
- Synonyms, 373
- T**
- Table updates, 66–67
- Taken branches, 88
- Taken/Not Taken path ID
- basics, 145
 - branch prediction, 49
 - data speculation, 217
 - resource flow microarchitectures, 398
- Taken Twice, 49
- Tandem branch prediction, 66
- Target addresses
- basics, 54
 - Branch History Table, 55–57, 56
 - fetch width, 58
 - instruction prefetching, 54
 - mispredictions, 58
 - trace caches, 95, 97
- Target prefetching, 15, 15
- Taxonomy, 137–148, 138–139, 189–194
- Temporal average behavior, 48
- Texas Instruments C6X, 122
- The Halting Problem, 31
- Theory-based skewed trees, 141
- Thread-Level Data Speculation, 348
- Thread-level parallelism (TLP), 5, 333
- Thread start point, 342
- Threads, *see also* Multithreading and multithreaded environments; Multithreading and speculation
- branch prediction, 39–40, 69–70
 - committing process, 346–347
 - control independence, 344
 - data independence, 344
 - granularity, 146
 - helper threads, 192, 339–341
 - inter-thread dependencies, 347–349
 - microarchitectural support, 341, 345–349
 - multithreading speculation, 4–5
 - path ID, 145
 - profile-based speculation, 287–290
 - runahead threads, 70
 - size, 344
 - speculative, 341–349
 - speculative architectural threads, 341–349
 - speculative multithreaded, 336–339, 337–338, 349
 - speculative parallel threading, 288–289
 - speculative state storage, 347
 - storage for speculative state, 347
 - thread level speculations, 287–290
 - workload balance, 344
- Tiling, 172
- Time, 305
- Timeliness, 161
- Time ordering tag, 402
- Time-tag, 404
- Time-tag value, 400
- Timing, 43–45, 44
- TLB, *see* Translation Look-Aside Buffer (TLB)
- Total-disambiguation, 225
- Trace Cache, 2
- Trace caches
- basics, 87–88, 87–89, 90, 104
 - branch prediction, 99
 - core fetch unit, 91–92, 92–93, 94, 102
 - design space, 97–103
 - indexing strategy, 98
 - instruction cache, 101–103
 - instruction fetch unit, 89–97
 - loop caches, 103
 - L1 vs. L2 instruction cache, 103
 - multi-phase construction, 100–101
 - nonspeculative updates, 101
 - operation, 92, 94–95, 97, 97

- parallel instruction cache access, 102
partial matching, 98–99
path associativity, 97
selection policy, 99–100
series instruction cache access, 102
speculative updates, 101
traces, 89, 91
Trace identifier, 91
Trace predictor, 98
Trace scheduling, 278–279
Trace selection, 91
Traces, 89, 91
Transactions, 404
Translation Look-Aside Buffer (TLB), 145
Transmeta system, 276, 359
Traversal loads, 179
True data, 4
True dependence, 387
True values, 34
Turing studies, 30
Twolf benchmark, *see* Instruction precomputation
Value predictability, 197–199, 199–201
Value predictors, 218–222, 221
Value profile, 271
Value reuse, 246–247, 252, 259–260, 260
Value Reuse Table (VRT), 246–247, 252
VAX, *see* Digital Equipment VAX
Verification, 230–232
Very high ILP environments, 68–70
VHPT, *see* Virtual Hashed Page Table (VHPT)
Virtual Hashed Page Table (VHPT), 274
Virtuality, 33–34
Virtual variables, 311, 313
VLIW machines, *see* DAISY systems
von Neumann programming, 29–30
Vortex benchmark, *see* Instruction precomputation
Vpr-Place benchmark, *see* Instruction precomputation
Vpr-Route benchmark, *see* Instruction precomputation
VRT, *see* Value Reuse Table (VRT)

U

- Unambiguous memory dependences, 360
Unconditional branch, 34
Unipath execution, 136
Unipath tree, 140
Unnecessary prefetches, 164
Updates, 291
Upper-bound performance, 253, 255, 255
Usage models, 276–278
Use-def factored chain, 311

V

- Validity, operands, 231
Value, 403–404

W

- Wait operation, 367
Warp Engine, 397
Way prediction, 192
Way prediction cache, 13–14
When of branch prediction, *see* Timing
WillBeAvailable, 315
Working sets, 62–63
Workload balance, 344
Wrong-path prefetching, 24, 26
Wrong predictions, 49, 68

Y

- Y-pipe machine, 148–149