UNIVERSITY OF CALIFORNIA, SAN DIEGO

Predictor-Directed Data Prefetching for Pointer-based Applications

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

by

Suleyman Sair

Committee in charge:

Professor Brad Calder, Chairperson
Professor Sujit Dey
Professor Rajesh Gupta
Professor Alex Orailoğlu
Professor Dean Tullsen

2003

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

Chair

University of California, San Diego

2003

Dedications

This dissertation is the culmination of not only years of effort, but also of the training, teaching, and support of many people. I would first like to acknowledge gratefully my advisor, Dr. Brad Calder, to whom I owe much of my academic and professional development. Thanks for many years of putting up with me and always giving me not what I wanted but what I needed. Thanks to him, the words *status* and *both* are irreversibly engraved in my brain.

Dr. David Kaeli also has a lot to do with my professional development. After all, he's the one who introduced me to computer architecture. I would also like to thank the other members of my thesis committee. Dr. Sujit Dey, Dr. Rajesh Gupta, Dr Alex Orailoğlu, and Dr. Dean Tullsen. I am grateful to them not only for reading this dissertation and approving it, but for their expert advice, encouragement and valuable feedback.

Equally important in my development (and enjoyment of the whole Ph.D. process) were my fellow architecture students at UCSD. They created an atmosphere that will be difficult to replicate ever again. To Glenn, who has been my great friend since I joined the Ph.D. program four years ago. He introduced me to innertube water polo, extreme hiking in the Topanga Canyon, rollerblading and a whole lot of other things which have made my stay in San Diego extremely enjoyable. Most importantly I have to thank him (and partly my parents I guess) for finding an inordinate number of anagrams of my name. To Tim, my co-conspirator and partner in crime. He has been a co-author in all but one my papers, so I owe him big time. Thanks for introducing me to hockey – and for countless hours spent in Mission Bay trying to sail during windless days (not including the day where you caused me to fly off my laser). To Jamison for setting up PBS and then having to answer a billion emails on our problems with it. To Barbara for her contagious laughter. To Hillary for making me forget that

I was actually at work. To Beth, master coordinator and organizer. To John, for hours of hard work administering machines. To Lori for her expert advice and making sense of things when no one else was able to. To Chandra for being my virtual roommate. To Eric for his unique perspective on things. To Erez for his serenades and sweet tomatoes and hot peppers. To Floria for her incredible will to get things done no matter that the obstacle. To Wei for playing hockey with us and showing us his mad skills. To Jeff for managing to organize weekly volleyball games despite everyone's impossible schedule. To Satish for being there the first and only time I got pulled over by the Border Patrol - wait that was because of you! To Rakesh for taking over backup duties from me. To Jeremy for introducing me to Flamebroiler. As a matter of fact, I owe all occupants of the architecture lab (past and current) a debt of thanks, for both their intellectual support, collaborations, and friendship.

Thanks to the staff at UCSD for countless hours of help and patience. To Julie Conner, who single-handedly coordinates the graduate department in a truly admirable manner. To the incredibly competent CSE support staff for handling tons of questions.

And, of course, my thanks to my best and closest friend Şule. Throughout my career, she has continuously supported me, repeatedly comforted me and put some sense into me. I can imagine neither the last four years nor the next forty without her.

Last and most important, I thank my family: my parents, Zehra and Erdoğan Şair, and my brother Ömer, who gave me the upbringing and education that allowed me to reach this point. I never could have done it without them.

Since a large number of people provided me their support, advice and knowledge, I chose to write the thesis in the first person plural. This is my modest way of acknowledging their contributions to this work.

Ílim ilim bilmektir
Ílim kendin bilmektir
Sen kendini bilmezsin
Ya nice okumaktir.

Knowledge should mean a full grasp of knowledge:
Knowledge means to know yourself, heart and soul.
If you have failed to understand yourself,
Then all of your reading has missed its call.

- Yunus Emre – a thirteenth century Turkish dervish.

TABLE OF CONTENTS

Signature Page . . . iii

Dedication Page . . . iv

Epigraph . . . vi

Table of Contents . . . vii

List of Figures . . . x

List of Tables . . . xix

Acknowledgments . . . xx

Vita and Publications . . . xxii

Abstract . . . xxiv

I Introduction . . . 1

II Motivation . . . 7
A. Miss Stream Classification Based on Data Access Types . . . 9
1. Next-Line Misses . . . 9
2. Stride Misses . . . 10
3. Same Object Misses . . . 10
4. Pointer Misses . . . 12
B. Prefetching Focused at Different Load Stream Classifications . . . 17
1. Next-Line . . . 17
2. Stride-based Prefetching . . . 18
3. Same Object Prefetching . . . 18
4. Pointer-Based Prefetching . . . 19
C. Methodology . . . 21
1. Baseline Architecture . . . 21
D. Miss Classification Results . . . 23
1. Unclassified Loads . . . 27
2. Hardware Classification . . . 31
3. Performance Results . . . 37
E. Quantifying Object and Pointer Behavior . . . 39
1. Object Fan-Out . . . 39
2. Variability . . . 42

F. Summary 44

III Prior Work 47
A. Address Prediction 48
1. Stride 48
2. Context/Markov Predictor 49
B. Hardware Prefetching Models 53
1. Fetch Stream Prefetching 53
2. Demand-Based Prefetching 54
3. Decoupled/Stream Prefetching 56
C. Software Prefetching Models 62
D. Summary 64

IV Predictor-Directed Stream Buffers 66
A. Predictor-Directed Stream Buffers 67
1. Predictor-Directed Stream Buffer Implementation 69
2. Stride-Filtered Markov Predictor 74
3. Allocation Filtering 76
4. Prediction Confidence 78
5. Stream Buffer Priority 79
6. TLB Translation and Prefetching 80
B. Methodology 80
1. Baseline Architecture 82
C. Prefetching Performance 84
1. Results 87
2. Sensitivity of Results 98
D. Summary 100

V Pointer Cache 104
A. Pointer Cache 105
1. Identifying Pointer Loads 106
2. Pointer Cache Architecture 107
3. An Example Pointer Cache Use 109
B. Value Prediction 109
C. Using Pointer Cache for Value Prediction and Main Thread Prefetching 111
1. Using the Pointer Cache to Predict Values 111
2. Using the Pointer Cache to Aid Prefetching 112
D. Methodology 112
E. Pointer Cache Performance 115
1. Previous Prefetching Mechanisms 116

2. Main Thread Pointer Cache Prefetching 118
F. Summary 126

VI Pointer Cache Assisted Data Prefetching 129
A. Using Pointer Cache with Speculative Precomputation 132
1. Control Oriented Speculative Precomputation 132
2. Using the Pointer Cache with Speculative Precomputation . . . 134
3. Making Speculative Threads 134
4. Spawning Speculative Threads 136
5. Maintaining Control of the Speculative Threads 137
B. Prefetching Performance 138
1. Pointer Cache Assisted Speculative Precomputation 139
2. Summary of Results 141
C. Summary 147

VII Summary 149

VIII Future Work 153
A. Future Work on PSB 153
B. Future Work on Pointer Cache Assisted Prefetching 154
1. Pointer Cache Chaining 154
2. Memory-backed Prediction Structures 155
3. Early Access to Predictors 156
4. Multi-pronged Prefetchers 157

Bibliography 158

LIST OF FIGURES

I.1 A typical 3 level memory hierarchy design. As the caches get bigger, they also get slower due to wire scaling. 2

I.2 Recurrent load instructions facilitating object transitions create a serial dependence chain amongst themselves. Since pointer based applications have big pointer working sets, pointer loads frequently miss all the way to the main memory, and the program can only execute at the speed of main memory accesses. 3

II.1 Code example from SPEC'2000 floating point benchmark equake. This code exemplifies array accesses typically causing Next-line and Stride misses. 11

II.2 Potential miss stream depicting Next-line and Stride misses corresponding to sequential and striding array accesses respectively. 11

II.3 Code example from SPEC'2000 integer benchmark `mcf`. This code exemplifies linked data structure accesses typically causing Same Object and Pointer misses. 13

II.4 Code representing the objects used in Figure II.3. This code exemplifies the types of linked data structures used in pointer-based applications. Each `node` object occupies 120 bytes on a 64-bit architecture. 13

II.5 Potential miss stream depicting Same Object and Pointer misses corresponding to linked data structure accesses. Object boundaries are depicted with vertical bars and transitions are marked by arrows. The first miss following a transition is classified as a Pointer miss while misses to different fields of the same object are classified as Same-Object misses. 14

II.6 The traversal path execution follows when the code in Figure II.3 is executed on the data structure shown on the left. As shown on the right, if we start at the light gray node, the traversal ends at the dark gray node. 15

II.7 Classification of SPEC 2000 program load misses into the four prefetching models of next-line, stride, same-object, and pointer. 24

II.8 Classification of Olden program load misses into the four prefetching models of next-line, stride, same-object, and pointer. 28

II.9 Classification of Pointer program load misses into the four prefetching models of next-line, stride, same-object, and pointer. 28

II.10 Load Miss Classification Hardware. 32

II.11 Classification prediction accuracy for the suite of pointer intensive applications. 34

II.12 Classification prediction accuracy for the Olden benchmark suite. 35
II.13 SPEC'CINT00 integer benchmark suite performance results when assigning perfect load latency for loads that match the different classifications. 36
II.14 SPEC'CFP00 floating point benchmark suite performance results when assigning perfect load latency for loads that match the different classifications. 36
II.15 Olden benchmark suite performance results when assigning perfect load latency for loads that match the different classifications. 38
II.16 Pointer benchmark suite performance results when assigning perfect load latency for loads that match the different classifications. 38
II.17 Fan Out Example 40
II.18 Object fan-out of the pointer-based programs. A histogram of object fan-out is shown for L1 cache misses classified as pointer transitions in Section II.A. 41
II.19 Object fan-out of the Olden benchmark suite. A histogram of object fan-out is shown for L1 cache misses classified as pointer transitions in Section II.A. 42
II.20 Pointer variability for the pointer-based applications. A histogram of pointer variability is shown for L1 cache misses classified as pointer transitions in Section II.A. 43
II.21 Pointer variability for the Olden benchmark suite. A histogram of pointer variability is shown for L1 cache misses classified as pointer transitions in Section II.A. 44

III.1 The operation of the two-delta stride predictor. (a) The predictor is indexed with the Program Counter(PC) of the load instruction and has fields for holding the last miss address, the transient stride, and the stable stride. (b) Each time a load misses we index into the predictor with the load PC and insert the load's information in the table. In the example we update the last address field with the miss address 100. (c) Upon observing the second miss, we update the transient stride (20) as the difference between the current miss address (120) and the previous miss address stored in the table (100). (d) On a miss to address 140 by the same load we observe that the calculated stride ($140 - 120 = 20$) matches the transient stride in the table and update the stable stride field with 20. 50

III.2 Potential traversal pattern through a sorted binary tree searching for a key. 51
III.3 The operation of the Markov predictor. (a) The predictor is indexed with the miss address of the load instruction and has fields for holding the last miss address, and the predicted address. (b) Each time a load misses we index into the predictor with the previous miss address and insert a transition from the previous miss to the current miss. In the example mark the entry indexed with the miss address 100 as the active entry(as indicated with the bold box). (c) Upon observing a miss to address 120, we insert a transition from 100 to 120 and mark the entry indexed with 120 as the active entry. (d) On a miss to address 135, we insert a transition from 120 to 135 and mark the entry indexed with 135 as the active entry and so on. . . . 52
III.4 A dynamic data structure that has a backbone layer connecting groups of rib nodes. 58
III.5 The interaction of stream buffers with the rest of the processor. A stream buffer prefetches consecutive cache blocks, starting with the one that missed in the L1 cache. On subsequent misses, the head of the stream buffer is probed. If the reference hits, that block is transferred to the L1 cache. 59
III.6 A stream buffer architecture guided by a PC-based stride prediction table. 60

IV.1 Stride-based Stream Buffer Architecture. Eight stream buffers are shown (overlapping each other). Each stream buffer can hold N cache blocks. When a stream buffer is allocated, it is assigned a predicted stride to use to generate all of its prefetch addresses. 69
IV.2 A Predictor-Directed Stream Buffer. We modify the stream buffer so it accesses a separate address prediction table to get its next prefetch address. When a stream buffer is allocated necessary prediction information is copied into the stream buffer to enable its access to the address predictor. 70
IV.3 Stride-Filtered Markov Predictor-Directed Stream Buffer Architecture. When a stream buffer is allocated, it is assigned a fixed stride from the stride table. To generate the next prefetch address the last address is (1) looked up in the Markov table and (2) used to calculate a next stride address. If the Markov table hits, then the Markov address is used, otherwise the next stride address is used for the prefetch. 71

IV.4 The number of bits to accurately predict cache misses using the Markov Difference Predictor. The y-axis shows the percent of L1 cache misses that could be correctly predicted given the number of bits used for each entry of the Markov table shown on the x-axis. The cache miss address is predicted by adding together the address used to index the Markov table with the value stored in the Markov table. 76
IV.5 IPC performance results of base case, prior PC-Stride prefetching, and our Predictor-Directed Stream Buffers. 88
IV.6 Predictor accuracy. This is the number of potential correct address predictions for all of the data cache misses for the PC-Stride predictor and the Stride-Filtered Markov predictor. . . 90
IV.7 Prefetch coverage breakdown. This is the number of stream buffer hits divided by the number of base case L1 misses. It represents the number of cache misses we are able to cover/remove with stream buffers compared to an architecture with no prefetching. The PC-Stride and Confidence Priority results are for the baseline prefetching architecture with 8 stream buffers and only one prefetch per cycle. For these configurations, we show the breakdown of the stream buffer hits that completely hide the memory latency and those that only partially hide the cache miss latency. The "X" results show the prefetch coverage for the Confidence Priority scheme with 128 stream buffers (each with 4 entries), where up to 4 predictions and prefetches can be initiated per cycle. 90
IV.8 Load Stall Rate. This is the percent of all dynamic loads that stall for one or more cycles when considering the different stream buffer architectures. These are references that are not full cache hits and are not full stream buffer hits. The percent of load stalls are broken down into those that are complete misses and those that are partial hits. 93
IV.9 Average latency of a load in cycles for the different architectures broken down by where the time is spent. The L1 access time is 3 cycles. Partial hits add to the access time of the level that they hit in. 93
IV.10 Average partial stream buffer hit latency. This is average latency of stream buffer tag hits that still have the prefetch outstanding. These hits therefore do not completely hide the memory latency. 94
IV.11 The percent of cycles the L1-L2 bus and the L2-Mem bus were busy. 96

IV.12 Wasted prefetches. This is the number of prefetches not used by the processor divided by the number of prefetches made. . . 96
IV.13 Average number of in-flight memory requests per cycle. 97
IV.14 Performance results showing different stream buffer configurations using our confidence allocation and priority scheduling technique. The first bar in each group is the baseline prefetching architecture with 8 stream buffers each having 4 entries, and it can issue up to 1 prediction and 1 prefetch per cycle. For each bar titled sbMxN[-P], M indicates the number of stream buffers and N denotes the number of entries in each stream buffer. P implies the maximum number of predictions and prefetches allowed per cycle. If not shown, P is assumed to be 1. 99
IV.15 Performance results for increased cache size. 101
IV.16 Performance results for default Confidence Priority configuration using a 32K 4-way cache in comparison to the baseline (no prefetching) architecture with a 32,128,256, and 512 4-way caches. The results show the trade off between utilizing area for stream buffers versus increasing the size of the L1 data cache. . 101

V.1 Because objects are linked together via pointers, we cannot access an arbitrary object among a group of objects. We have to traverse each object in succession to get to the desired node. Since load instructions executed for facilitating object transitions depend on the data loaded by another, a cache miss by the first load forces the second load to stall until the first load completes. When executing a long sequence of such dependent pointer chasing loads, instructions can only be executed at the speed of the serial accesses to memory. Since pointer based applications have big pointer working sets, pointer loads frequently miss all the way to the main memory, and the program can only execute at the speed of main memory accesses. 106

V.2 Pipeline organization of a processor with a pointer cache. The pointer cache is queried in parallel with the L1 cache and returns the address of the object pointed to. This value is consumed by instructions dependent on the load, breaking the serial nature of the memory access. The pointer cache is updated by pointer based accesses (loads or stores) when they commit. When a pointer based load (a heap load which loads a pointer to another heap location) commits, it updates the pointer cache with the load's (address, value) pair. Store Teaching queries the pointer cache on all stores, and, on a hit, updates the relevant pointer cache entry with the store value. 107
V.3 This figure illustrates the different operations performed on the pointer cache: training (1), prediction (2), and store teaching (3). 109
V.4 Performance improvement from previous prefetching schemes. 117
V.5 Performance impact when the main thread is permitted to access a 256K-entry, 4-way pointer cache with different access latencies. All configurations also use the stride prefetcher. VPonly shows results using the pointer cache only for value prediction. All of the rest of the PC results use the pointer cache for both value prediction and prefetching. 119
V.6 Performance impact when the main thread is permitted to access a 256K-entry, 4-way pointer cache with different access latencies. All configurations also use the SFM prefetcher. VPonly shows results using the pointer cache only for value prediction. All of the rest of the PC results use the pointer cache for both value prediction and prefetching. 120
V.7 Performance impact of varying the number of entries in the pointer cache. All pointer cache configurations utilize a 4-way pointer cache with a 5 cycle access latency. All the configurations also make use of stride prefetching. 122
V.8 Performance impact of varying the number of entries in the pointer cache. All pointer cache configurations utilize a 4-way pointer cache with a 20 cycle access latency. All the configurations also make use of stride prefetching. 123
V.9 Performance impact of varying the number of entries in the pointer cache. All pointer cache configurations utilize a 4-way pointer cache with a 5 cycle access latency. All the configurations also make use of SFM prefetching. 124

V.10 Performance impact of varying the number of entries in the pointer cache. All pointer cache configurations utilize a 4-way pointer cache with a 20 cycle access latency. All the configurations also make use of SFM prefetching. 125

VI.1 Code example from SPEC'2000 integer benchmark mcf. As seen in this code, there are multiple potential transitions to follow(illustrated in bold). Moreover, the decision to choose which transition to follow is guarded by branch instructions(e.g. `if( tmp )`). 131

VI.2 The traversal path execution follows when the code in Figure VI.1 is executed on the data structure shown on the left. As shown on the right, if we start at the light gray node, the traversal ends at the dark gray node. This irregular traversal pattern is a typical behavior in pointer-based applications. . . 132

VI.3 Performance impact from combining SP and stride prefetching when using a 256K-entry, 4-way pointer cache with varying access latency for assisting speculative precomputation threads only. The main thread does not use the pointer cache. 139

VI.4 Performance impact from combining SP and SFM prefetching when using a 256K-entry, 4-way pointer cache with varying access latency for assisting speculative precomputation threads only. The main thread does not use the pointer cache. 140

VI.5 Performance results comparing a processor with different combined L3 and pointer cache sizes and latencies. This chart presents results for the SPEC'2000 benchmarks and `vis`. The lines annotated with `L3` are utilizing the whole area for an L3 cache. The lines annotated with `Ptr$` on the other hand indicate using 2 MB for an L3 and the remaining area for a pointer cache. For the pointer cache configurations, both main thread and speculative precomputation threads are allowed to access the pointer cache. 143

VI.6 Performance results comparing a processor with different combined L3 and pointer cache sizes and latencies. This chart presents results for `dot`, `em3d`, `gawk` and `sis`. The lines annotated with `L3` are utilizing the whole area for an L3 cache. The lines annotated with `Ptr$` on the other hand indicate using 2 MB for an L3 and the remaining area for a pointer cache. For the pointer cache configurations, both main thread and speculative precomputation threads are allowed to access the pointer cache. 144

VI.7 Performance results comparing a processor with a 2M 8-way L3 cache and 256k entry 4-way, 20 cycle pointer cache against a baseline stride prefetching architecture with a 3M, 12-way L3 cache. The pointer cache is used for value prediction and prefetching for all main thread (MT) results, except the third bar, where the pointer cache is only used by the main thread for value prediction. All configurations access their L3 cache in 30 cycles and utilize a stride prefetcher. 145
VI.8 Performance results comparing a processor with a 2M 8-way L3 cache and 256k entry 4-way, 20 cycle pointer cache against a baseline stride prefetching architecture with a 3M, 12-way L3 cache. The pointer cache is used for value prediction and prefetching for all main thread (MT) results, except the third bar, where the pointer cache is only used by the main thread for value prediction. All configurations access their L3 cache in 30 cycles and utilize a SFM prefetcher. 146

LIST OF TABLES

II.1 Description of pointer-based benchmarks used. 22
II.2 SPEC 2000 cache miss behavior. The L1 data cache is a 32K 2-way associative cache with 32 byte lines. The L2 is a unified 1 Meg 4-way associative cache with 64 byte lines. 25
II.3 Pointer-based programs cache miss behavior. The L1 data cache is a 32K 2-way associative cache with 32 byte lines. The L2 is a unified 1 Meg 4-way associative cache with 64 byte lines. . . . 26
II.4 Olden cache miss behavior. The L1 data cache is a 32K 2-way associative cache with 32 byte lines. The L2 is a unified 1 Meg 4-way associative cache with 64 byte lines. 26
II.5 Detailed classification of unclassified L1 misses in SPEC 2000 benchmarks. 30
II.6 Detailed classification of unclassified L1 misses in Pointer and Olden benchmarks. 31

IV.1 Description of benchmarks used. 81
IV.2 Baseline results showing the number of instructions simulated, L1 data cache miss rate, percent of executed instructions that were loads and stores, the IPC for each program, and the percent of cycles the bus was busy from the L1 to L2, and the bus from L2 to main memory was busy. 83
IV.3 Summary of simulation parameters. The top parameters summarize the baseline architecture. The next group lists the additional parameters for the PC-Stride stream buffer architecture. The final group describes the additional parameters on top of that needed for the Predictor-Directed Stream Buffer prefetching architecture. 86

V.1 Assumed baseline architecture simulation parameters. 113
V.2 Details of simulated benchmarks. Data cache miss rates are shown for a processor which performs no hardware prefetching. 115

VI.1 Combined L3 and pointer cache sizes examined along with their assumed access latencies. The latencies are those reported by Cacti 3.0 [67] with a 0.18μm feature size assuming a 1 GHz clock frequency. 141

Acknowledgments

The text of Chapter II is in part a reprint of the material as it appears in the Proceedings of the 8th International Symposium on High-Performance Computer Architecture. The dissertation author was a co-primary researcher and author (with Tim Sherwood) and the co-author listed on this publication ([57]) directed and supervised the research which forms the basis for Chapter II.

The text of Chapters III, and IV are in part reprints of material as it appears in the IEEE Transactions on Computers. The dissertation author was a co-primary researcher and author (with Tim Sherwood) and the co-author listed on this publication ([58]) directed and supervised the research which forms the basis for Chapters III, and IV.

The text of Chapters V and VI are in part a reprint of the material as it appears in the proceedings of the 35th International Symposium on Microarchitecture. The dissertation author was a co-primary researcher and author (with Jamison Collins) and the co-authors listed on this publication ([21]) directed and supervised the research which forms the basis for Chapters V and VI.

VITA

August 2, 1975	Born, Bandirma, Turkey
1997	B.S. in Control and Computer Engineering Istanbul Technical University
1997–1999	Research Assistant, Northeastern University
1998	Internship, Analog Devices, Massachusetts
1999	M.S. in Electrical and Computer Engineering Northeastern University
1999–2003	Research Assistant, University of California, San Diego
2000	Internship, IBM T.J. Watson Research Labs, New York
2003	Doctor of Philosophy University of California, San Diego

PUBLICATIONS

Timothy Sherwood, Suleyman Sair and Brad Calder. "Phase Tracking and Prediction." To appear in the 30th Annual International Symposium on Computer Architecture (ISCA-2003), June 2003, San Diego, CA.

Suleyman Sair, Timothy Sherwood and Brad Calder. "A Decoupled Predictor-Directed Stream Prefetching Architecture." In the IEEE Transactions on Computers, Vol 52, No 5, March 2003.

Satish Narayanasamy, Timothy Sherwood, Suleyman Sair, Brad Calder and George Varghese. "Catching Accurate Profiles in Hardware." In the 9th International Symposium on High-Performance Computer Architecture (HPCA-9), February 2003, Anaheim, CA.

Jamison Collins, Suleyman Sair, Brad Calder and Dean Tullsen. "Pointer-Cache Assisted Speculative Precomputation." In the 35th International Symposium on Microarchitecture (MICRO-35), November 2002, Istanbul, Turkey.

Suleyman Sair, Timothy Sherwood, and Brad Calder. "Quantifying Load Stream Behavior." In the Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA-8), February 2002, Cambridge, MA.

Timothy Sherwood, Suleyman Sair and Brad Calder. "Predictor-Directed Stream Buffers." In the Proceedings of the 33rd International Symposium on Microarchitecture (MICRO-33), December 2000 , Monterey, CA.

Suleyman Sair and Mark Charney, "Memory Behavior of the SPEC'2000 Benchmark Suite." IBM Thomas J. Watson Research Center Technical Report RC 21852 , October 2000.

Suleyman Sair, David R. Kaeli and Jose Fridman. "A Study of Dynamic Branch Prediction for SHARC DSPs." In the Proceedings of the 2nd International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'99), Washington, D.C.

Suleyman Sair, Guiseppe Olivadoti, David Kaeli and Jose Fridman. "DSPTune : A Performance Evaluation Toolset for the SHARC Signal Processor." In the Proceedings of 33rd Simulation Symposium , April 2000, Washington, D.C.

Suleyman Sair, David R. Kaeli and Waleed Meleis. "A Study of Loop Unrolling for VLIW-based DSPs." In the Proceedings of the 1998 IEEE Workshop on Signal Processing Systems (SiPS '98), Boston, MA.

ABSTRACT OF THE DISSERTATION

Predictor-Directed Data Prefetching
for Pointer-based Applications by

Suleyman Sair
Doctor of Philosophy in Computer Science and Engineering
University of California, San Diego, 2003
Professor Brad Calder, Chair

CPU speeds double approximately every eighteen months, while main memory speeds double only about every ten years. These diverging rates imply an impending "Memory Wall", in which memory accesses dominate program performance. An effective way to reduce the observed latency of memory accesses is data prefetching. Data prefetching involves predicting which data the processor will use in the near future and then bring that data into storage locations closer to the execution engine (e.g., caches, dedicated prefetch buffers etc.).

Meanwhile, applications are constantly becoming more complex and demanding. This is reflected in programs through the use of various data structures and constructs provided by the programming language. While caches are extremely successful in handling accesses with good locality, the remaining accesses result in costly cache misses. For these reasons, we performed a study that maps the miss stream to several fundamental access types inherent to programming constructs. The results of this study show that pointer-based applications with accesses to linked data structures present the biggest challenge to the memory subsystem. Consequently, they have the biggest potential for speedup when their adverse effects on program performance are removed by data prefetching. For

these reasons, in this thesis, we focus on data prefetching techniques targeting pointer-based applications which have irregular access patterns.

With a better understanding of the kinds of accesses that cause cache misses, we have developed three prefetching techniques that can handle all these cases. The first technique enhances traditional stream buffer designs by following an address prediction stream instead of a fixed stride as originally proposed. This improves the stream buffers ability to target pointer-based programs.

We also explore utilizing a hardware pointer cache to predict the values of pointer loads in order to break down long dependence chains caused by recurrent pointer accesses. This allows overlapping the execution of dependent instructions with the actual memory access, providing significant benefits.

Another technique, Speculative Precomputation, attempts to prefetch data ahead of its use, by running a shortened version of the program on a spare multi-threaded hardware context. The final scheme we propose builds upon speculative precomputation and uses the pointer cache for the pointer loads in speculative threads. Furthermore, we add control flow instructions to the speculative threads to guide them down the correct traversal path when multiple next transitions are possible. Our results show considerable benefits when these techniques are utilized together, complementing each other.

Chapter I

Introduction

Over the past 30 years memory density has gone up consistently due to ongoing developments in VLSI technology. Because of poor wire scaling however, memory access times have not kept pace with processor cycle times [76]. For example the Intel Pentium 4 processor running at 2 GHz accesses main memory in approximately 500 clock cycles. In the ideal case the processor can execute 3000 instructions within that period. To make matters worse, if the processor does not have other work independent of the required data, it will sit idle for that whole duration. Moreover, within 5 years memory accesses are expected to take as much as several thousand cycles. For these reasons, the memory subsystem has become the most dominant factor in observed program performance.

The increasing performance gap between processors and memory will force future architectures to devote significant resources towards removing and hiding memory latency. A great deal of effort has been invested in reducing the impact of cache misses on program performance. As with any other latency, cache miss latency can be tolerated using compile-time techniques such as instruction scheduling, or run-time techniques including out-of-order issue, decoupled execution, or non-blocking loads. It is also possible to reduce the latency of cache misses using multi-level caches, and prefetching.

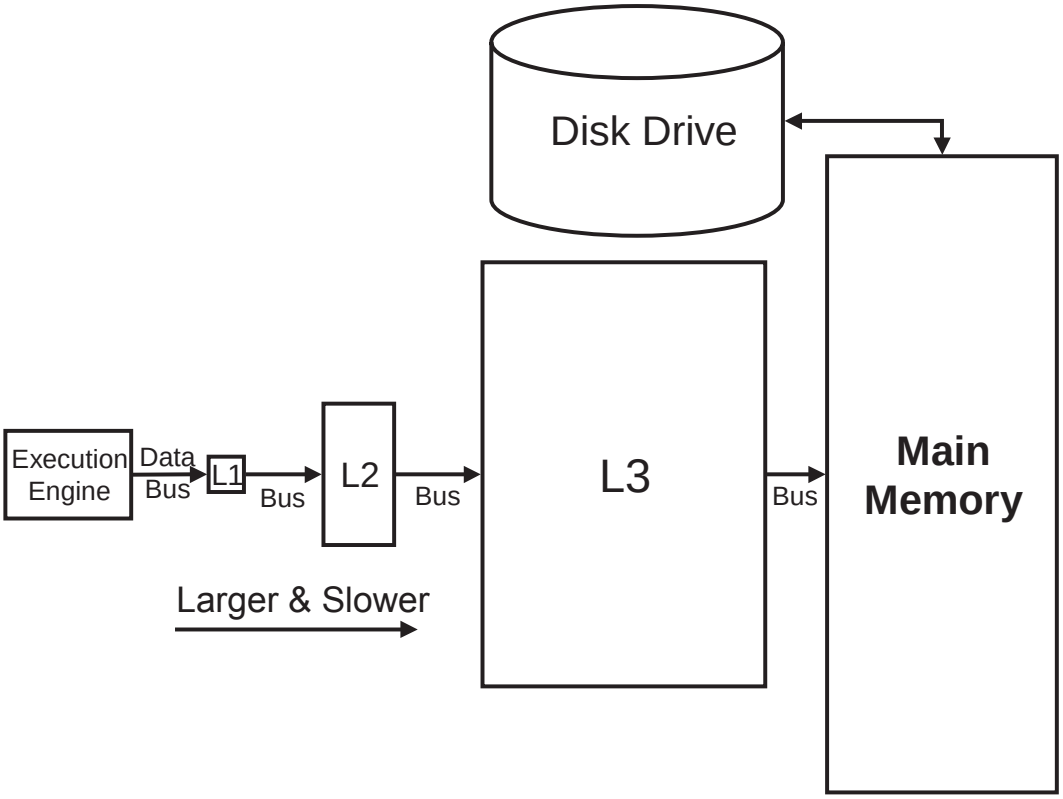

Figure I.1: A typical 3 level memory hierarchy design. As the caches get bigger, they also get slower due to wire scaling.

Processors compensate for the slow accesses to memory by utilizing multiple levels of caches as shown in Figure I.1. Caches are small, fast storage devices used to improve average access time to slow memory. They exploit spatial and temporal locality that exist in data streams. If we assume that location n was recently accessed, spatial locality suggests that locations near n will be accessed in the near future. Similarly, temporal locality implies that location n will be accessed again in the near future. Cache access latency is a function of its size, the larger the cache, the slower it is.

Meanwhile, applications are constantly becoming more complex and demanding. Thus despite the growing size of caches, the working set size is outpacing the growth in cache size. The latency of main memory access is mostly

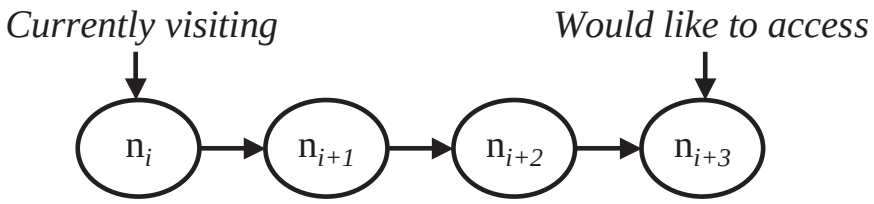

Figure I.2: Recurrent load instructions facilitating object transitions create a serial dependence chain amongst themselves. Since pointer based applications have big pointer working sets, pointer loads frequently miss all the way to the main memory, and the program can only execute at the speed of main memory accesses.

hidden if the data is found in the first level cache. We can not arbitrarily increase L1 cache size to accommodate program working sets however. This is because the continuing downward trend in processor cycle times impose a limit on L1 cache access times and this in turn dictates how big L1 caches can get.

In parallel with the increasing complexity of applications, linked data structures are gaining wide use in programs due to their flexibility. Programs using these structures do not perform well on current processors however. Because objects are linked together via pointers, we cannot access an arbitrary object among a group of objects. We have to traverse each object in succession to get to the desired node as seen in Figure I.2. But because these programs have large working sets, we cannot hold the complete working set in the processor caches, resulting in many cache misses. Because we cannot execute dependent instructions until the cache miss is serviced, the rate at which the program makes progress depends on the level of the memory hierarchy we locate the required data. In other words, if the data request misses all the way to main memory when executing a long sequence of such dependent pointer-chasing loads, instructions can only be executed at the speed of serial accesses to slow main memory.

In order to find a solution that alleviates their negative effects, we need

to first understand the behavior of cache misses. To determine the sources of cache misses we performed a study of data miss stream classification and we present its findings in this thesis. All memory operations and therefore all cache misses, are caused by accesses to data compiled into various data structures. Consequently, we classify misses into one of four categories corresponding to the type of the data access pattern. These are next-line, stride, same-object (additional misses that occur to a recently accessed object), or pointer-based transitions. Next-line and stride misses have been studied extensively in the literature. Our results show that pointer related misses are more difficult in terms of hiding their latency. But at the same time they have the highest potential for performance gains when eliminated from the program. As a result of their performance potential, and the fact that next-line and stride misses have been extensively studied in the literature, we are focusing our research on pointer misses.

Data prefetching can exploit the regularity in programs by predicting future memory accesses and starting them earlier to hide their latency. This principle forms the basis of Predictor-based Data Prefetching. It is an effective method for reducing the effect of load latency in modern processors. One form of hardware-based data prefetching, stream buffers, has been shown to be particularly effective due to its ability to detect data streams and run ahead of them, prefetching as it goes. Unfortunately, in the past, the applicability of streaming was limited to stride intensive code.

In this thesis we propose *Predictor-Directed Stream Buffers* (*PSB*), which allows the stream buffer to follow a general address prediction stream instead of a fixed stride. A general address prediction stream complicates the allocation of both stream buffer and memory resources, because the predictions generated will not be as reliable as prior sequential next-line and stride-based stream buffer implementations. To address this, we examine using confidence-based techniques

to guide the allocation and prioritization of stream buffers and their prefetch requests. Our results show, when using PSB on a benchmark suite heavy in pointer-based applications, that PSB provides a 23% speedup on average over the best previous stream buffer implementation, and an improvement of 75% over using no prefetching at all.

Stream buffers are a very complexity effective approach in dealing with memory latency. Their simplicity is also their Achilles' heel however. One shortcoming of the stream buffer design is the small amount of program state it can hold. With program footprints regularly exceeding multiple GBs, it is very hard to capture the pointer working set of an application in a small predictor.

To that effect, this thesis proposes the use of a pointer cache, which tracks pointer transitions, to aid prefetching. The pointer cache provides, for a given pointer's effective address, the base address of the object pointed to by the pointer. We examine using the pointer cache in a wide issue superscalar processor as a value predictor and to aid prefetching when a chain of pointers is being traversed. When a load misses in the L1 cache, but hits in the pointer cache, the first two cache blocks of the pointed to object are prefetched. In addition, the load's dependencies are broken by using the pointer cache hit as a value prediction.

Another shortcoming of the simple stream buffer design is its inability to adapt to variations in program control flow resulting in a different traversal path when compared to the previous time. These multiple traversal paths are common in pointer-based applications with utilizing objects with multiple outgoing pointer links. Since the execution of control flow instructions determines the traversal path, a thread based prefetching technique, such as speculative precomputation [23], which executes instructions for their prefetching effect can easily be extended to include these control flow instructions as well.

Accordingly, we also examine using the pointer cache to allow speculative precomputation to run farther ahead of the main thread of execution than in prior studies. Previously proposed thread-based prefetchers are limited in how far they can run ahead of the main thread when traversing a chain of recurrent dependent loads. When combined with the pointer cache, a speculative thread can make better progress ahead of the main thread, rapidly traversing data structures in the face of cache misses caused by pointer transitions.

The remainder of the thesis is organized as follows. Chapter II looks at the causes of cache misses and some trends in their behavior that motivate this work. Chapter III reviews some of the relevant prior work in this area. Chapter IV describes our Predictor-Directed Stream Buffer architecture and provides results quantifying its efficiency. Chapter V explores using a Pointer Cache to capture the large pointer working set of programs in order to speed up their execution. In Chapter VI we investigate the use of a Pointer Cache in conjunction with Speculative Precomputation to form a powerful prefetching tool that can adapt to control flow changes in the execution of a program. Chapter VII provides a summary of our conclusions and we wrap up with future research directions in Chapter VIII.

Chapter II

Motivation

As we mentioned in Chapter I, one of the most important impediments to current and future processor performance is the memory bottleneck. Processor clock speeds are increasing at an exponential rate, much faster than DRAM access time improvements [76] and cache misses are becoming more and more dominant in program performance.

In order to design techniques to reduce the adverse affects of data cache misses, we first need to understand what causes them. Since all cache misses are a result of data accesses, we must examine the data structures these load instructions are accessing. This chapter provides an insight into the data structures that programs most commonly use and classify the misses based on the data structure they are accessing with the goal that this information can be used to devise efficient prefetching solutions for each type of data access.

Accordingly, we first show how to classify load miss streams into different classes based on their miss access patterns. We show for a large number of programs what types of accesses are causing misses so that they may be targeted for future research. We further show how this classification can be done efficiently in hardware with a high degree of accuracy, so that architectural structures such as caches or prefetching engines can be made access pattern aware. We classify

these loads into four types of access patterns or streams – (1) next-line loads corresponding to sequential array accesses, (2) striding loads corresponding to hopping or multi-dimensional array accesses, (3) same-object loads causing additional misses to a recently referenced heap object, or (4) pointer-based loads corresponding to misses that occur when we access the object for the first time following a pointer transition.

Out of these four types of cache miss streams, pointer-based streams can be the most difficult to eliminate using existing hardware and software prefetching algorithms. To better understand the behavior of these loads and their applications we examine two new metrics. The first metric, *Object Fan Out*, is used to quantify the number of pointers in an object that are transitioned and frequently miss in the cache. The second metric, *Pointer Variability*, quantifies how many pointer transitions are stable versus how many are frequently changing. A pointer transition is a load that loads a pointer. Pointer variability shows how many times a pointer transition for a given *address* loads a pointer different from the last pointer that was loaded from that address. Programs with low object fan out and pointer variability will be much easier to prefetch, in comparison to programs that have high object fan out and variability, and we show that a large percentage of misses in real programs fall into this second category.

The rest of this chapter is organized as follows. We start by defining the different load miss models and show how the misses for many different types of programs are classified into these models in Section II.A. In Section II.B we describe how these miss types behave in relationship to prior prefetching techniques. After detailing our evaluation methodology in Section II.C, Section II.D shows classification results for an extensive set of programs that are commonly used in computer architecture research. We then present a hardware technique for quickly and efficiently classifying cache misses for the purpose of guiding dynamic

prefetching in Section II.D.2. Section II.D.3 provides an insight into the performance potential of prefetching and removing these different miss streams from the program. Section II.E defines two metrics that correspond to pointer-based miss streams quantifying the difficulty they present to prefetching techniques. Finally we summarize our classification technique and results in Section II.F.

II.A Miss Stream Classification Based on Data Access Types

Even if only a few of the data accesses miss in the on-chip data caches, they can cause severe execution overheads. For example, one cache miss for every 100 accesses can very well double the execution time of a program.

To get a better understanding of the sources of cache misses, we present a classification framework in this chapter. In this framework, the miss stream is separated into four different classes based on the type of data structure being accessed and the nature of the access. Listed in order of increasing complexity they are: next-line, stride, access within an object, and dereferencing of pointers.

II.A.1 Next-Line Misses

Next-line misses are inherent to sequential array accesses or accesses to linked data structures that are allocated sequentially in memory. Figure II.1 shows a piece of code depicting typical access types causing next-line misses. `Equake` is a floating point benchmark taken from the SPEC'2000 benchmark suite. It simulates the propagation of seismic waves in the event of an earthquake. Accesses to array `cor` in Figure II.1 are sequential accesses potentially causing next-line misses. A possible next-line miss stream is shown in Figure II.2 on the top. As seen in the figure, the cache block addresses observed by the processor are adjacent to one another causing next-line misses.

These accesses are the simplest to capture with hardware, a simple stream buffer is very efficient at capturing this type of behavior. The stream buffer can identify accesses to sequential cache blocks and use this information to fetch sequentially down the stream. While this type of access is very simple, it is also very common in a multitude of applications. We classify a cache miss as being a next-line access if it is an access to a cache block that is adjacent to a cache block that was recently fetched.

II.A.2 Stride Misses

Stride misses can be caused by skipping over elements in an array. Additionally, when we access a multidimensional array stored in row major order in column-wise fashion (or vice versa) stride misses occur. When we look at Figure II.1 again, references to array `v` constitute such skipping accesses potentially causing stride misses. A possible striding miss stream is shown in Figure II.2 on the bottom. As seen in the figure, the cache block addresses observed by the processor are separated by a fixed difference called the *stride*.

Stride accesses are the next easiest to capture in hardware and many different prefetching architectures exist to capture this type of behavior. Farkas et al. [26] showed that the most efficient way to capture this sort of behavior is by examining access patterns on a per static load basis. We use this observation to help us define stride access behavior. We define a cache miss to be stride miss if the same stride has been seen twice in a row for the static load that performed the access.

II.A.3 Same Object Misses

So far we have concerned ourselves with regular access patterns, the type that may be commonly found in programs dominated by large multidimensional

```
...
for ( ... ){
   ...
   for (j = 0; j < 4; j++) {
    if (cor[j] == Src.sourcenode)
    {
          v[3 * j] = uf[0];
          v[3 * j + 1] = uf[1];
          ...
    }
    ...
   }
}
```

Figure II.1: Code example from SPEC'2000 floating point benchmark equake. This code exemplifies array accesses typically causing Next-line and Stride misses.

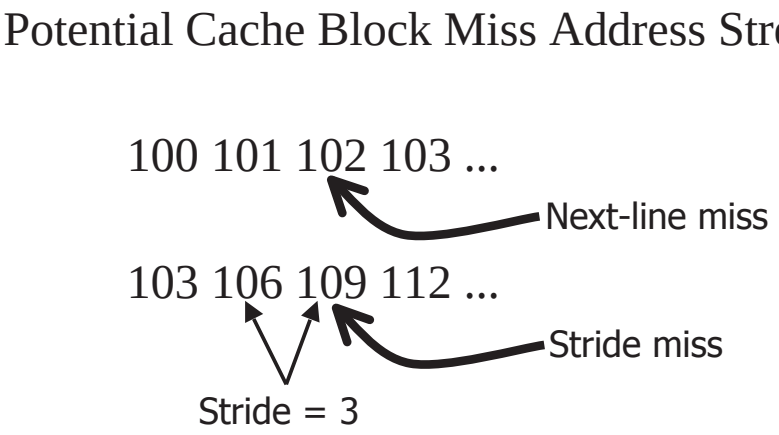

Figure II.2: Potential miss stream depicting Next-line and Stride misses corresponding to sequential and striding array accesses respectively.

arrays. The next two classes may not fall into this category. The class of misses, which we call Same-Object, are non-sequential, non-striding accesses going within a single object. We define a *Same-Object* cache miss as a miss to an object that has already had a miss recently. These misses may possibly be prevented if whenever we access an object we fetch the whole object, or at least those cache blocks of the object that will soon be referenced. These misses can also be targeted by field reordering [19].

Figure II.3 presents a code segment from the program `mcf`. `Mcf` is an integer benchmark taken from the SPEC'2000 benchmark suite. It implements a combinatorial optimization algorithm trying to come up with a feasible route schedule for a public transportation system. Figure II.4 depicts the specific linked data structure used in Figure II.3. Each `node` object occupies 120 bytes on a 64-bit architecture. When we analyze the code, we see that after we touch a new object's `pred` field and incur a cache miss while executing the first line, the access to the `sibling` field of the same object misses in the data cache as well. This happens when our data cache blocks are 32 bytes (or smaller), since the `pred` field is located at offset $0x16$ and the `sibling` field is located at offset $0x32$ in a given object. In this case, the miss to `sibling` would be classified as a Same-Object miss. One such possible miss stream is shown in Figure II.5. In this example we are showing the cache block addresses that result in data cache misses and object boundaries are marked by vertical bars. Since we incur additional misses while accessing the fields of the same object, accesses to address 104, 134, and 214 are classified as Same-Object misses.

II.A.4 Pointer Misses

The final hardware classification that we make is the Pointer class. The *Pointer* class represents misses to objects that are accessed via the dereference

```
...
while( node->pred )
{
   tmp = node->sibling;
    if( tmp )
    {
        node = tmp;
        ...
     }
     else
          node = node->pred;
}
```

Figure II.3: Code example from SPEC'2000 integer benchmark `mcf`. This code exemplifies linked data structure accesses typically causing Same Object and Pointer misses.

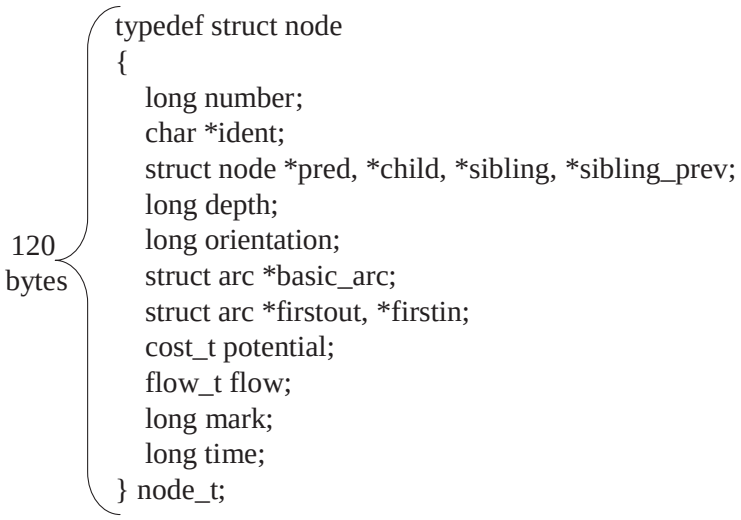

Figure II.4: Code representing the objects used in Figure II.3. This code exemplifies the types of linked data structures used in pointer-based applications. Each `node` object occupies 120 bytes on a 64-bit architecture.

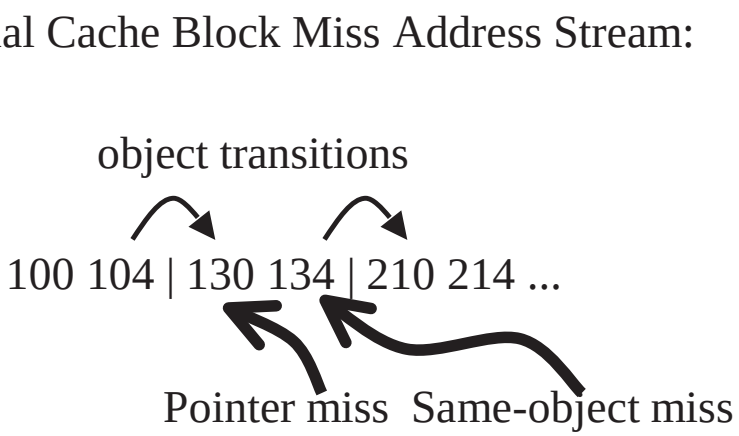

Figure II.5: Potential miss stream depicting Same Object and Pointer misses corresponding to linked data structure accesses. Object boundaries are depicted with vertical bars and transitions are marked by arrows. The first miss following a transition is classified as a Pointer miss while misses to different fields of the same object are classified as Same-Object misses.

of a pointer. If we look at Figure II.3 again, after we transition to a new object following either the *pred* or the *sibling* link, the first access to the new object's *pred* field on the first line will cause a Pointer miss. This is exemplified in Figure II.5 where object transitions are marked with arrows. As shown, the first miss following a transition is marked as a Pointer miss.

Pointer misses can constitute a large portion of total cache misses due to the accesses having very little conventional spatial locality. This is depicted in Figure II.6. The accesses to nodes usually follows an irregular traversal path resulting in a cache miss at each node. This has led to many recent hardware and software schemes that attempt to capture their behavior for the purpose of prefetching. These misses can be targeted by the software techniques of object reordering [44], smart object placement [10], software prefetching [42], or hardware schemes such as jump pointer prefetching [53, 54] and predictor-directed stream buffers [58, 66].

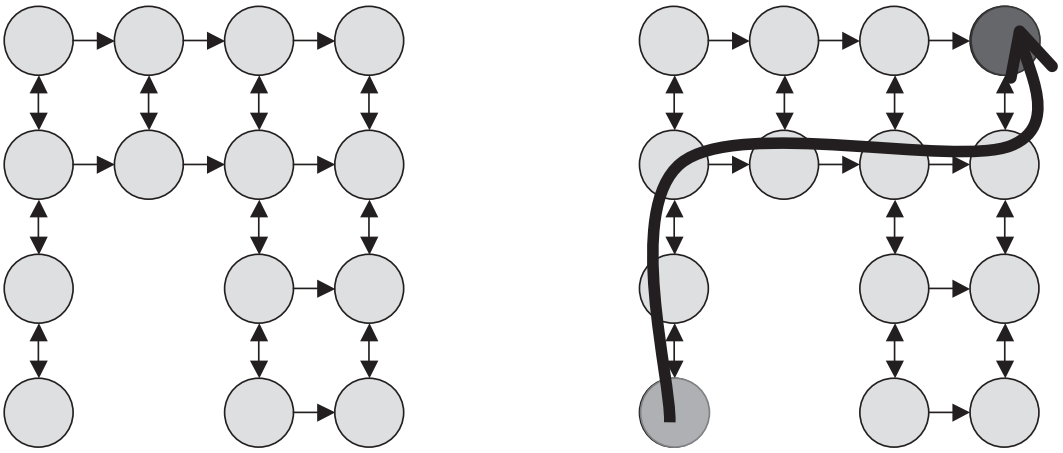

Figure II.6: The traversal path execution follows when the code in Figure II.3 is executed on the data structure shown on the left. As shown on the right, if we start at the light gray node, the traversal ends at the dark gray node.

We classify all cache misses into one and only one of these categories. If there is a cache miss that can fit into more than one category, such as loads with a stride of 1, we classify them into the simplest category possible, which in this case would be next-line.

In the description of the classifications please note the use of the term "recently". In order to capture the recent behavior of the loads we use a profiling technique called *Windowing*. We build a window of the last cache misses and loads, and use only this information when classifying subsequent loads. More specifically we have four windows and every time there is a cache miss, a window is checked for hits for the four different models. This hit information is used to calculate the classification of the cache miss. The window to find next-line misses holds a list of the past addresses that have missed with one block size added to them. Whenever a cache miss occurs, we simply check for a match in the window. A hit in any of the elements of the window indicates that the load

was of the next-line type. After this, we update the window with the most recent miss information.

To find stride misses, we use a window storing PC values and the two most recent addresses of each load instruction. If there is a hit on a load PC in the window, we compute the difference between the current address and previous address and compare this to the difference between the two previous addresses stored in the window. A match indicates a stride miss classification.

Misses to the same heap object can be easily detected using a similar technique used for next-line detection, but this time we store the base address rather than the possible next-line address. The base address from a cache miss is looked up in the window, and a hit indicates that this miss is classified as an access to the same object. On update, each cache miss stores its base address into this window.

To detect pointer based loads we use another window which is updated with the value of every load instruction. When there is a cache miss we check the window to see if we can find the base address of the missing load. If there is a hit then we know that a prior load recently loaded the base pointer for the object that just missed and hence it is a pointer miss.

Using windowing prevents all loads being classified as "next-line", since only a fixed amount of history is kept track of. The window models the recent working set of load misses that could potentially trigger a prefetch of the load that missed. The window size is limited, because the prefetched block could only reside in a prefetch buffer or cache without being used for a window of time before being evicted. For the results presented in this section we capture the last 200 cache misses and the last 500 loads for pointer tracking.

II.B Prefetching Focused at Different Load Stream Classifications

In Section II.A we described the types of misses we wish to classify and how they relate to different data access patterns. In this section we categorize and describe prior software and hardware prefetching research into the classes of misses they target, i.e. next-line, stride, same-object, and pointer traversals. This classification corresponds to an increasing implementation complexity of hardware prefetching techniques.

II.B.1 Next-Line

The simplest form of prefetching is to prefetch the next cache block that occurs after a given load. This form of prefetch is very accurate, since programs have a lot of spatial locality.

Next-Line Prefetching (NLP) was proposed by Smith [68], where each cache block is tagged with a bit indicating when the next block should be prefetched. When a block is prefetched, its tag bit is set to zero. When the block is accessed during a fetch and the bit is zero, a prefetch of the next sequential block is triggered and the bit is set to one.

Jouppi introduced *stream buffers*, as a high latency hiding form of a next-line prefetching architecture [34]. The stream buffers follow multiple streams prefetching them in parallel and these streams can run ahead independent of the instruction stream of the processor. They are designed as FIFO buffers that prefetch consecutive cache blocks, starting with the one that missed in the L1 cache. On subsequent misses, the head of the stream buffer is probed. If the reference hits, that block is transferred to the L1 cache.

II.B.2 Stride-based Prefetching

A logical extension of next-line prefetching is *stride-based prefetching*. This scheme allows the prefetcher to eliminate miss patterns that follow a regular pattern but access non-sequential cache blocks. This type of access frequently occurs in scientific programs using multidimensional arrays.

Palacharla and Kessler [49] suggested a *non-unit stride* detection mechanism to enhance the effectiveness of stream buffers. This technique uses a *minimum delta* non-unit detection scheme. With this scheme, the dynamic stride is determined by the minimum signed difference between the past N miss addresses. If this minimum delta is smaller then the L1 block size, then the stride is set to the cache block size with the sign of the minimum delta. Otherwise, the stride is set to the minimum delta.

Farkas et al. [26] made an important contribution by extending this model to use a *PC-based* stride predictor to provide the stride on stream buffer allocation. The PC-stride predictor determines the stride for a load instruction by using the PC to index into a stride address prediction table. This differs from the minimum-delta scheme, since the minimum-delta uses the global history to calculate the stride for a given load. A PC-stride predictor uses an associative buffer to record the last miss address for N load instructions, along with their program counter values. Thus, the stride prediction for a stream buffer is based only on the past memory behavior of the load for which the stream buffer was allocated.

II.B.3 Same Object Prefetching

Programs make use of different types of data structures to accomplish their final goal. Often times, logically related data are grouped together into an *object* to enhance semantics. The amount of data located inside an object does

not always fit into a single cache block, and accesses to various parts of the same object can cause multiple cache misses. To eliminate these incidental misses, one could trigger the prefetch of the whole object once a miss occurs to data within the object. This could require the prefetching algorithm to know/predict the size of an object.

Zhang and Torrellas [78] recognized the benefit of grouping together fields or objects that are used together, and prefetching these all together as a prefetch group of blocks. They examined using user added grouping instructions that allowed the user to group together fields/objects that should be prefetched together. These groupings are then stored in a hardware buffer, and as soon as one of them is referenced and misses, all the cache blocks in the group are prefetched.

II.B.4 Pointer-Based Prefetching

As logically related data is collected into an object, objects that are related are also connected to each other via pointers. *Pointer-based prefetching*, either predicts or accesses these pointer values to prefetch the next object that is likely to be visited after the current one.

The inherent dependency between neighbor objects limits the amount of latency that can be hidden by the prefetching algorithm. This is known as the *pointer-chasing problem* [42]. The imposed serialization of object accesses constrain the prefetcher from running enough ahead of the execution stream to hide the full memory latency.

Luk and Mowry [42] examined prefetching for Linked Data Structures (LDS). They examined the phenomenon of using pointer chaining for prefetching to hide latency. Greedy Prefetching was used to prefetch down all the pointers in a given heap object. They also examine adding *jump-pointers* to hook up a

heap object X to another heap object Y that occurs earlier in the pointer chain, by adding an explicit jump-pointer from $YtoX$. This approach can hide more latency than their demand based greedy algorithm, but comes at a cost of adding jump-pointers into their structures. In addition, this could potentially perform badly if the structure of the LDS changes radically between traversals over the structure.

Roth et al. [53] propose analyzing the producer-consumer relationship among loads to alleviate the effects of the pointer-chasing problem. In this scheme, load instructions that produce object addresses are linked together to facilitate a prediction chain. Prefetches read address values from memory and initiate another prefetch using the value just prefetched as an address. They examine prefetching one iteration ahead of the current execution to reduce the number of useless prefetches. Furthermore, Roth and Sohi [54] extend the jump-pointer prefetching technique by providing hardware, software and cooperative schemes to facilitate linking objects together. These different techniques provide a variety of trade-off points between prefetch accuracy and prefetching overhead.

Markov prefetching has been proposed as an effective technique for correctly predicting pointer-based loads [33]. When a cache miss occurs, the miss address would index into a Markov prediction table that provides the next set of possible cache addresses that have followed that miss address in the past. After these addresses are prefetched, the prefetcher stays idle until the next cache miss.

The decoupled architecture for prefetching pointer-based miss streams we propose in this thesis (see Chapter IV) extends the stream buffer architecture proposed by Farkas et al. [26] to follow prediction streams instead of a fixed stride. Different predictors can be used to direct this architecture making it quite adept at finding both complex array access and pointer chasing behavior over a variety of applications.

II.C Methodology

We make use of both profiling and detailed cycle accurate simulation in this study. When performing profiling we use Compaq's ATOM [71] tool, to gather miss rates and perform base line classifications.

The simulator used in this study was derived from the SimpleScalar/Alpha 3.0 tool set [9], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user-level instructions, performing a detailed timing simulation of an aggressive 8-way dynamically scheduled microprocessor with two levels of instruction and data cache memory. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch mis-prediction.

To perform our evaluation we collected results from the complete SPEC 2000 integer benchmark suite, selected SPEC 2000 floating point benchmarks, the popular programs from the Olden benchmark suite, and a set of other pointer intensive programs. The pointer intensive programs we will examine in detail are described in Table II.1. All programs were compiled on a DEC Alpha AXP-21264 processor using the DEC FORTRAN, C or C++ compilers under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`).

II.C.1 Baseline Architecture

Our baseline simulation configuration models a next generation out-of-order processor microarchitecture. We have selected the parameters to capture underlying trends in microarchitecture design. The processor has a large window of execution; it can fetch up to 8 instructions per cycle. It has a 128 entry re-order buffer with a 64 entry load/store buffer. To compensate for the added complexity of disambiguating loads and stores in a large execution window, we increased the store forward latency to 2 cycles.

Program	Description
burg	A program that generates a fast tree parser using BURS technology. It is commonly used to construct optimal instruction selectors for use in compiler code generation. The input used was a grammar that scribes the VAX instruction set architecture.
deltablue	A constraint solution system implemented in C++. It has an abundance of short lived heap objects.
dot	Dot is taken from the AT&T's GraphViz suite. It is a tool for automatically making hierarchical layouts of directed graphs. Automatic generation of graph drawings has important applications in key technologies such as database design, software engineering, VLSI and network design and visual interfaces in other domains.
equake	Equake is from the SPEC 2000 benchmark suite. The program simulates the propagation of elastic waves in large, highly heterogeneous valleys, such as California's San Fernando Valley, or the Greater Los Angeles Basin. The goal is to recover the time history of the ground motion everywhere within the valley due to a specific seismic event. Computations are performed on an unstructured mesh that locally resolves wavelengths, using a finite element method.
mcf	Mcf is from the SPEC 2000 benchmark suite. It is a combinatorial optimization algorithm solving a minimum cost network flow problem.
sis	Synthesis of synchronous and asynchronous circuits. It includes a number of capabilities such as state minimization and optimization. The program has approximately 172,000 lines of source code and performs a lot of pointer arithmetic.
vis	VIS (Verification Interacting with Synthesis) is a tool that integrates the verification, simulation, and synthesis of finite-state hardware systems. It uses a Verilog front end and supports fair CTL model checking, language emptiness checking, combinational and sequential equivalence checking, cycle-based simulation, and hierarchical synthesis.

Table II.1: Description of pointer-based benchmarks used.

To make sure that the load classification speedups we report are from eliminating those load memory latencies and not from compensating for a conservative memory disambiguation policy, we implemented perfect store sets [20]. Perfect store sets cause loads to only be dependent on stores that write to the same memory, i.e. when they are actually dependent instructions. In this way loads will not be held up by false dependencies.

In the baseline architecture, there is an 8 cycle minimum branch misprediction penalty. The processor has 8 integer ALU units, 4-load/store units, 2-FP adders, 2-integer MULT, and 2-FP MULT/DIV. The latencies are: ALU 1 cycle, MULT 3 cycles , FP Adder 2 cycles, FP Mult 4 cycles, and FP DIV 12 cycles. All functional units, except the divide units, are fully pipelined allowing a new instruction to initiate execution each cycle. We use a McFarling gshare predictor [45] to drive our fetch unit. Two predictions can be made per cycle with up to 8 instructions fetched.

We rewrote the memory hierarchy in SimpleScalar to better model bus occupancy, bandwidth, and pipelining of the second level cache and main memory. The L1 instruction cache is a 32K 2-way associative cache with 32-byte lines. The baseline results are run with a 32K 2-way associative data cache with 32-byte lines. A 1 Megabyte unified 4-way L2 cache is simulated with 64-byte lines. The L2 cache has a latency of 12 cycles. The main memory has an access time of 120 cycles. The L1 to L2 bus can support up to 8 bytes per processor cycle whereas the L2 to memory bus can support 4 bytes per cycle.

II.D Miss Classification Results

The first thing to look at before we begin discussing the classification of loads is the cache miss rates for the various programs. Tables II.2, II.3, and II.4 show the data input used to run each program, the L1 and L2 cache miss behavior,

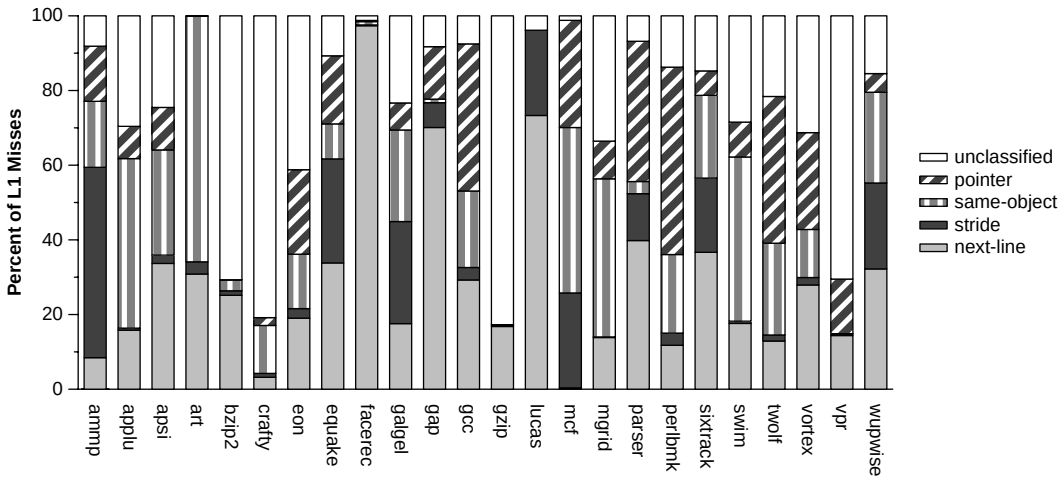

Figure II.7: Classification of SPEC 2000 program load misses into the four prefetching models of next-line, stride, same-object, and pointer.

and the percent of loads executed by each program. The L1 and L2 cache misses are both in terms of the average number of cache misses that occurred per 1K of executed instructions.

The programs consist of the full set of SPEC 2000 integer programs, a subset of the SPEC 2000 floating point programs that have been used in recent prefetching and pre-computation papers, the Olden benchmark suite, and a set of pointer intensive programs. We chose the Olden benchmarks that have shown performance improvements from prior prefetching papers.

We now show the results of applying the classification technique described above over several suites of programs. Figure II.7 shows the classification of the loads that miss in the L1 cache for the SPEC 2000 benchmarks. The four programs of interest from the SPEC 2000 suite that have a significant amount of cache misses include `art`, `ammp`, `equake` and `mcf`, and all of these have 80% or more of their misses classified, with `mcf` having the largest pointer behavior of these applications.

The classification of the programs from the Olden suite can be seen in

benchmark	input	L1 MissPer1kI	L2 MissPer1kI	% Loads
ammp	ref	14.14	8.42	8.29%
applu	ref	28.42	13.57	25.89%
apsi	ref	13.70	3.34	21.34%
art	110	15.04	0.01	4.84%
bzip2	graphic	4.49	0.90	17.48%
crafty	ref	6.08	0.03	22.32%
eon	cook	0.09	0.00	10.06%
equake	ref	44.76	16.29	31.91%
galgel	ref	24.14	2.39	21.64%
gap	ref	1.90	0.88	14.70%
gcc	200	10.30	0.81	21.65%
gzip	graphic	7.47	0.11	17.47%
lucas	ref	0.92	0.47	4.25%
mcf	ref	130.16	91.54	28.34%
mgrid	ref	21.81	6.13	29.34%
parser	ref	13.00	1.92	19.66%
perlbmk	diffmail	4.62	0.08	23.00%
swim	ref	45.31	17.23	19.73%
twolf	ref	19.81	2.45	20.26%
vortex	two	2.38	0.29	21.00%
vpr	place	12.77	1.21	20.81%
wupwise	ref	6.34	2.94	16.56%

Table II.2: SPEC 2000 cache miss behavior. The L1 data cache is a 32K 2-way associative cache with 32 byte lines. The L2 is a unified 1 Meg 4-way associative cache with 64 byte lines.

benchmark	input	L1 MissPer1kI	L2 MissPer1kI	% Loads
burg	rrh-vax	15.82	0.89	17.63%
deltablue	long	56.12	1.11	27.93%
sis	markex	4.58	0.04	36.32%
vis	clma	7.67	2.17	19.91%
dot	small	90.03	68.85	32.91%

Table II.3: Pointer-based programs cache miss behavior. The L1 data cache is a 32K 2-way associative cache with 32 byte lines. The L2 is a unified 1 Meg 4-way associative cache with 64 byte lines.

benchmark	input	L1 MissPer1kI	L2 MissPer1kI	% Loads
health	5 500 1 1	122.93	0.36	34.86%
mst	1024 1	9.63	5.53	18.19%
perimeter	12 1	13.58	9.42	17.60%
treeadd	20 1	9.20	4.56	21.30%
tsp	100000 1	1.53	0.65	6.94%

Table II.4: Olden cache miss behavior. The L1 data cache is a 32K 2-way associative cache with 32 byte lines. The L2 is a unified 1 Meg 4-way associative cache with 64 byte lines.

Figure II.8. `Health` has by far the largest miss rate of all the programs, and is also the most dominated by the pointer behavior. The other programs have less significant miss rates and are more balanced in the types of misses that they exhibit. The one counter example to this is `treeadd` which is dominated by next-line prefetchable structures. All `treeadd` does is allocate a tree in depth-first order, and then traverse the tree in the same order. This results in the tree being created where every left child is allocated right after its parent node in memory. Furthermore, each right child allocated right after the last prior depth first search leaf. This match in the allocation and traversal orders causes almost all of the cache misses to potentially be covered by next-line prefetching. For that reason, `treeadd` has almost no pointer misses.

Figure II.9 shows the same classification technique applied to our own set of pointer intensive benchmarks. As the name suggests, the suite of applications that we have assembled are dominated by pointer behavior. However this pointer behavior is not the only form of misses. There is a mix of access patterns seen, from next-line to pointer behavior, stride and same-object. The benchmark with the highest miss rate, `dot`, is also the program most dominated by pointer behavior. The programs `sis` and `vis` also show very strong pointer behavior.

The ability to do this classification in software is useful to quantify applications allowing researchers to find applications exhibiting certain behavior or to guide profile-directed optimizations. However, there are still some questions to be answered about those loads that are left unclassified.

II.D.1 Unclassified Loads

As can be seen in Figure II.7, some programs have a fair number of cache misses that do not fit into the categories of next-line, stride, same-object or pointer transition. These cache misses have arithmetic operations (not captured

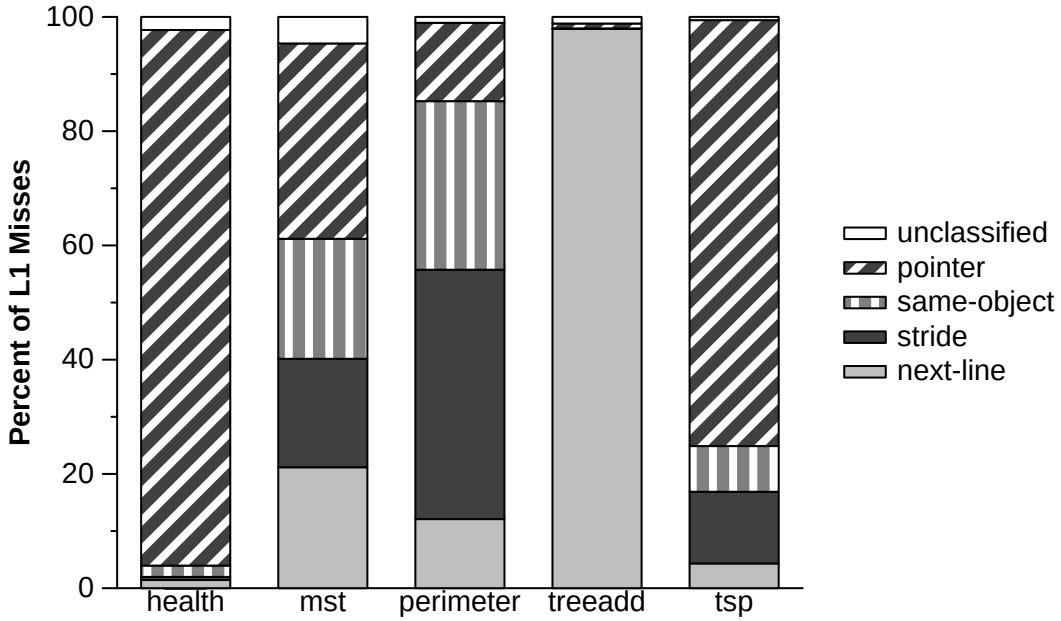

Figure II.8: Classification of Olden program load misses into the four prefetching models of next-line, stride, same-object, and pointer.

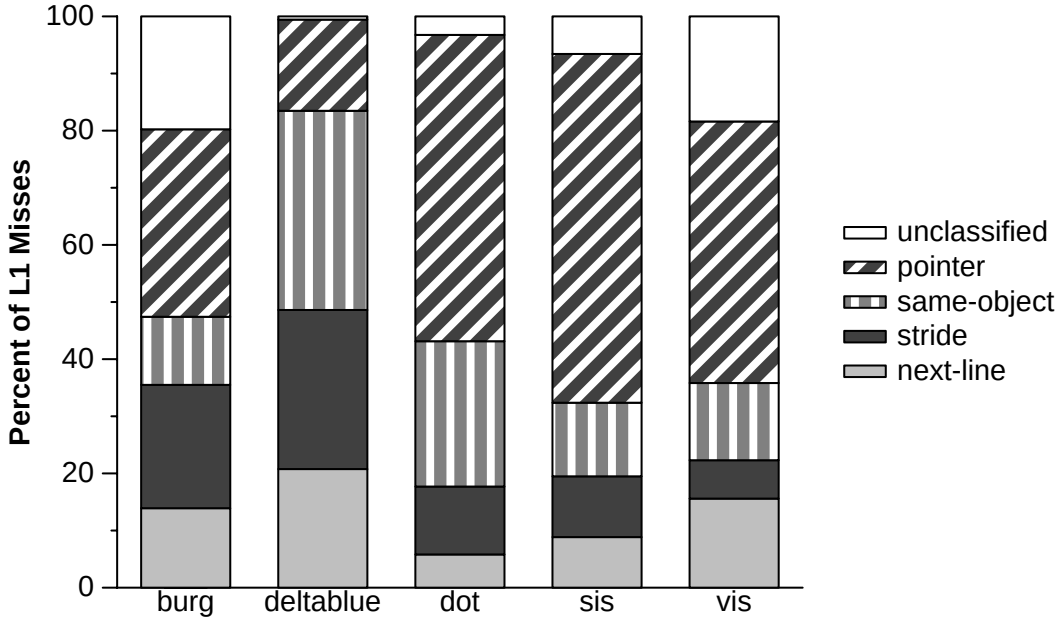

Figure II.9: Classification of Pointer program load misses into the four prefetching models of next-line, stride, same-object, and pointer.

by stride) used to calculate their effective addresses. These classifications are not easily captured by existing hardware prefetchers. For these cache misses, we use three additional types of cache miss classifications. To determine the classification we search back over the dependency chain used to calculate the effective address.

We classify cache misses as *Recurrent* if there is an instruction that is inside of a loop that has the same logical register definition as one of its operands, and the operand register being defined has an address stored in it. A load whose effective address is calculated in this manner is producing its effective address each iteration of the loop off of the prior loop's address calculation. Cache misses that are classified as recurrent perform arithmetic operations to produce the effective address not captured by stride prefetching.

A cache miss is labeled as *Base Address* if there is an instruction in the load's dependency chain that uses the same address over and over again in a calculation to produce the load's effective address. This occurs when the calculation for the effective address is performed off of the same base address every loop iteration.

Finally, a load is labeled as *Complex* if it is not recurrent and it is not base-address, and the load's effective address is calculated from a prior load in the dependency chain, and that prior load's value was an address. The address from that prior pointer load is used in an equation to produce the effective address that missed in the cache.

Tables II.5 and II.6 present a detailed look into load misses that go unclassified as described above. Most of the unclassified misses are recurrent pointer misses, potentially indicating that the loop induction variable is updated in a non-linear fashion. This makes these misses unclassifiable to next-line or stride predictors.

benchmark	% Unclassified Recurrent	% Unclassified Base Address	% Unclassified Complex
ammp	6.24%	0.94%	0.95%
applu	14.05%	1.00%	14.54%
apsi	7.91%	3.06%	13.56%
art	0.00%	0.06%	0.02%
bzip2	70.65%	0.03%	0.00%
crafty	79.45%	1.37%	0.01%
eon	23.75%	5.61%	11.85%
equake	10.60%	0.08%	0.02%
fma3d	0.19%	0.03%	1.01%
galgel	9.26%	0.19%	13.89%
gap	8.05%	0.21%	0.00%
gcc	6.52%	0.93%	0.07%
gzip	82.58%	0.13%	0.00%
lucas	1.88%	0.90%	1.08%
mcf	1.19%	0.01%	0.01%
mgrid	17.59%	0.20%	15.76%
parser	6.53%	0.21%	0.05%
perlbmk	12.24%	1.41%	0.09%
swim	13.85%	0.79%	0.11%
twolf	26.76%	0.67%	1.04%
vortex	20.00%	0.78%	0.82%
vpr	29.61%	1.63%	0.03%
vpr	57.53%	5.41%	7.56%
wupwise	1.51%	0.18%	13.77%

Table II.5: Detailed classification of unclassified L1 misses in SPEC 2000 benchmarks.

benchmark	% Unclassified Recurrent	% Unclassified Base Address	% Unclassified Complex
burg	19.45%	0.31%	0.00%
deltablue	0.49%	0.07%	0.00%
dot	3.23%	0.02%	0.00%
sis	5.69%	0.85%	0.03%
vis	17.16%	0.38%	0.86%
health	1.58%	0.64%	0.07%
mst	4.55%	0.08%	0.02%
perimeter	1.03%	0.00%	0.00%
treeadd	1.14%	0.01%	0.00%
tsp	0.31%	0.00%	0.25%

Table II.6: Detailed classification of unclassified L1 misses in Pointer and Olden benchmarks.

II.D.2 Hardware Classification

In order to take advantage of classification we need to provide a way for it to be done efficiently in hardware at run time. To accomplish this we make use of the windowing technique described in Section II.A, along with a very small fully associative buffer. The classification hardware keeps information in the buffer for the last N cache misses and then performs a lookup on its tables during a cache miss. Different types of matches mean different classifications for that load.

Figure II.10 shows the proposed classification architecture. The basic structures in the architecture are small CAMs each with an update pointer. Every time there is a cache miss, the CAM is checked for hits for the four different models. This hit information is used to calculate the classification of the cache miss. The structure is then updated at the update pointer and the update pointer is incremented to the next entry. The structure is therefore accessed in two ways,

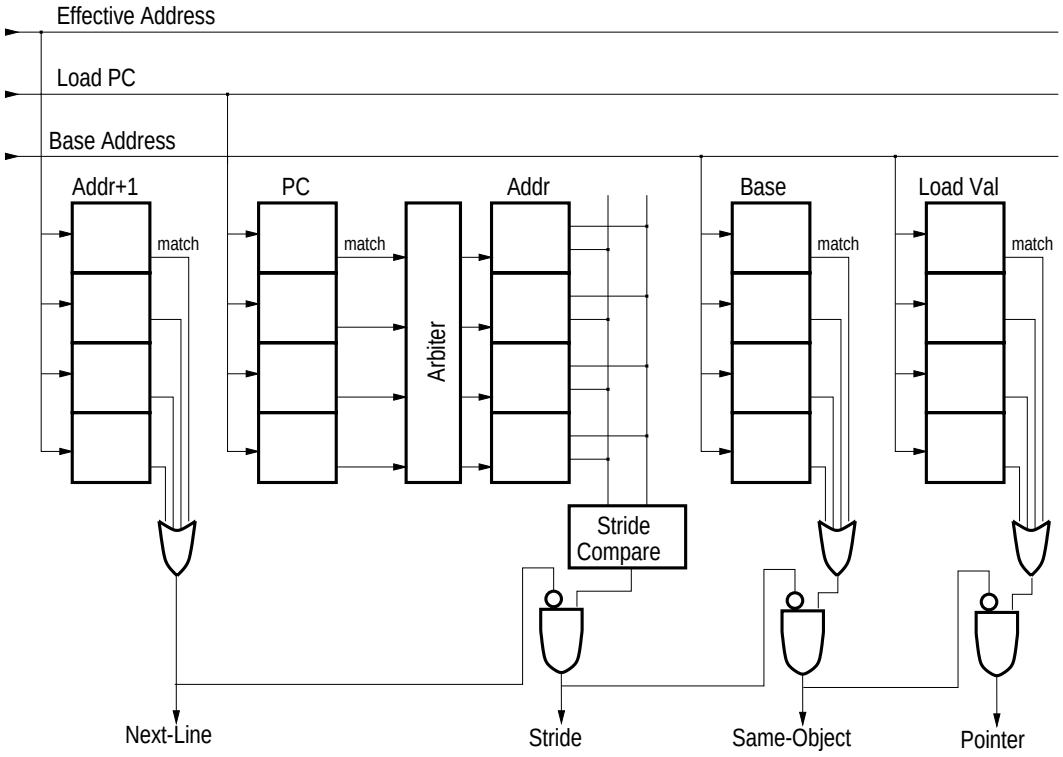

Figure II.10: Load Miss Classification Hardware.

a parallel lookup and a rotating register style update using the update pointer.

The structure to find next-line misses is a small CAM with a list of the past addresses that have missed with one block size added to them. Figure II.10 is drawn showing a CAM with 4 entries, while each actual CAM has 32 entries in the architecture we modeled. Whenever a cache miss occurs, we simply check for a match in the CAM. A hit in any of the elements of the CAM indicates that the load was of the next-line type. Then after this information is computed we update the CAM with the most recent address information.

To find stride misses we add a structure that performs a parallel lookup as in the CAM for next-line misses, but this time we attempt to match the PC of the load instruction rather than the address that is being loaded as shown in Figure II.10. If there is a hit in the CAM, the two most recent addresses

for that load are output and subtracted. The output of this subtraction is then added to the previous load address and we compare this result with the current load address with a match indicating a successful classification. This circuit is very similar in behavior to the arbiter used in the issue stage of an out of order processor, but much smaller. It is further simplified by the fact that the operation can be multi-cycle and pipelined.

Misses to the same heap object can be easily detected using the same type of structure used for next-line detection, but this time we store the base address rather than the possible next-line address. The base address from a cache miss is looked up in the CAM, and a hit indicates that this miss is classified as an access to the same object. On update, each cache miss stores its base address into this CAM.

To detect pointer based loads we add one last small structure, which is another CAM. This CAM is updated with the result of every load using the update pointer associated with that CAM. Because the result of all recent loads are stored in the CAM, loads to pointers are captured. When there is a cache miss we check the CAM to see if we can find the base address of the missing load. If there is a hit then we know that a prior load recently loaded the base pointer for the object that just missed and hence it is a pointer miss.

In using this classification scheme, the first miss to an object for a pointer-based application most likely will be classified as a pointer miss, and all subsequent misses to that same object would be classified as same object misses.

To test this hardware scheme we compare it against the real classifications that were presented in Section II.D. We use a very small hardware window size, of N=32. This means that we only need 896 Bytes of storage to implement this architecture, and this could be further reduced to around 256 Bytes if partial tags are used with a small hash function.

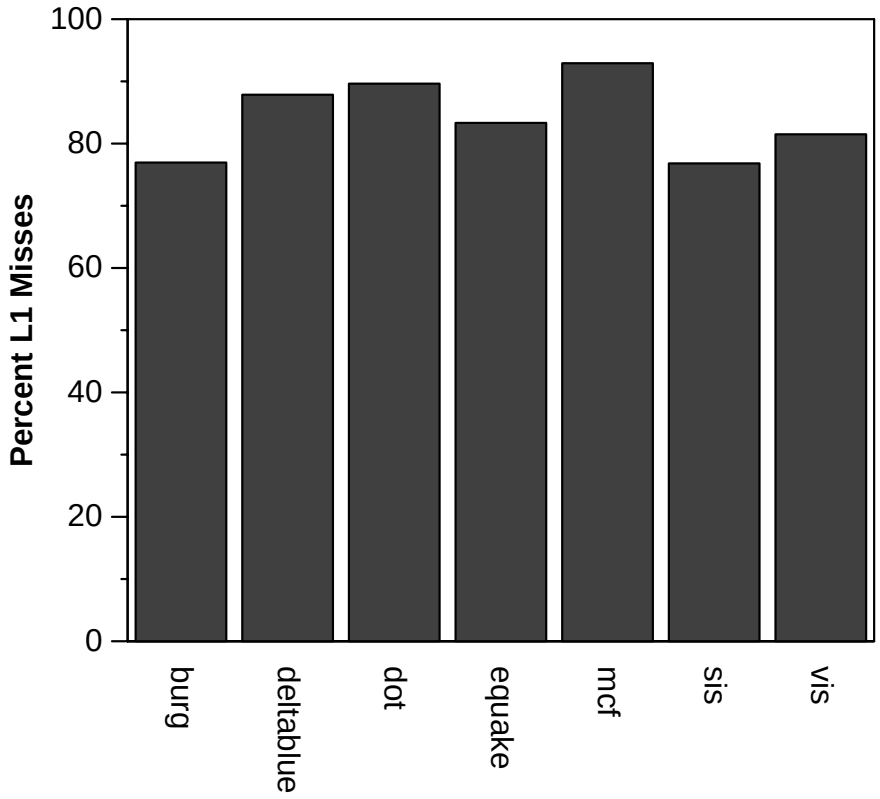

Figure II.11: Classification prediction accuracy for the suite of pointer intensive applications.

To evaluate the classification hardware, we would like to predict what the *next* classification will be for a given load. To accomplish this we keep a small direct mapped table, which is indexed by the address of the load. In this table we store the last known classification of the load, as generated by the classification hardware. We then compare this value stored in this table to the true classification of the load. Figures II.11 and II.12 show the accuracy of this classification prediction mechanism over the pointer and Olden benchmark suites if a prediction table of size 128 is used.

The prediction accuracies show that by using the presented architecture

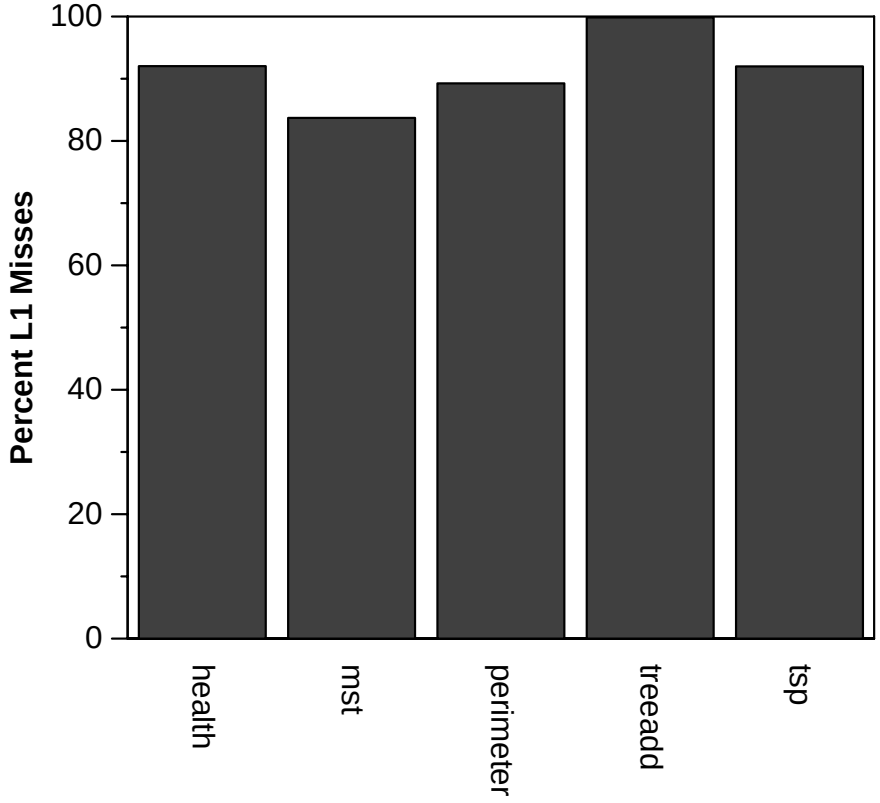

Figure II.12: Classification prediction accuracy for the Olden benchmark suite.

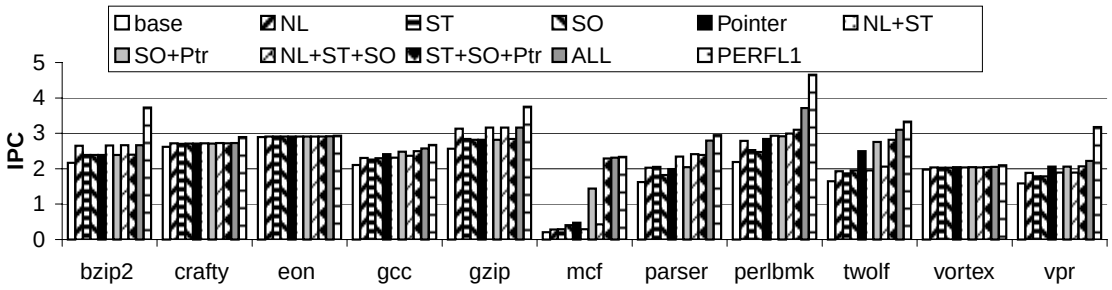

Figure II.13: SPEC'CINT00 integer benchmark suite performance results when assigning perfect load latency for loads that match the different classifications.

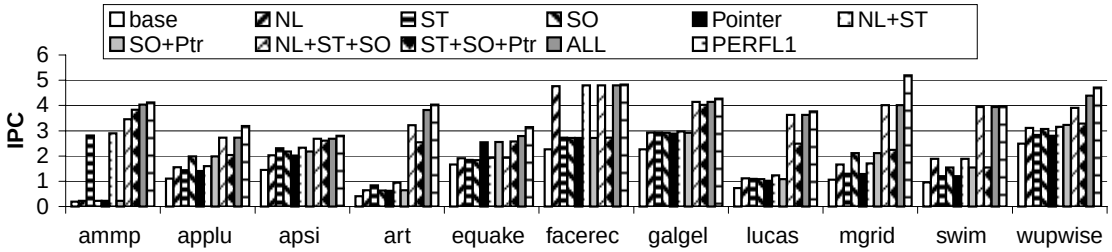

Figure II.14: SPEC'CFP00 floating point benchmark suite performance results when assigning perfect load latency for loads that match the different classifications.

we can correctly classify the majority of cache misses for the applications examined. The programs that the predictive classification had the most trouble with were `burg` and `sis`. For `burg` the predictor is caught jumping between stride and pointer classifications, this is because the application happened to allocate some of its objects at a fixed stride from each other. This causes stride to be predicted when a stride is observed by the classification hardware, but pointer is the still the true access type. This is not a problem because either answer is really valid, but it could be fixed with the addition of a small amount of hysteresis. The classification for `Sis` is around 80% because the program performs pointer arithmetic to load some of its data, and these are not accurately classified.

II.D.3 Performance Results

In order to measure the potential benefit of applying our classification scheme, we ran detailed performance simulations using the SimpleScalar model described in section II.C. The goal of these simulation results are to show the potential IPC performance if all of the cache misses are eliminated related to one or more of the prior four types of load miss classifications using the classification hardware presented in the prior section. Figures II.13, II.14, II.15, and II.16 show IPC results when we assume perfect load latency for these loads. The first bar shows IPC results for the baseline architecture. The next bar (NL) shows results when loads classified as next-line using our hardware classification architecture are given perfect L1 latency (they do not miss in the cache). The remaining bars show the same optimization applied to loads classified as stride (ST), same-object access (SO), pointer accesses (Pointer), and combinations of different classes. The bar (All) shows the IPC where all loads classified (i.e. next-line, stride, same-object and pointer loads) are assumed to hit in the cache. The last bar (PerfL1) shows the IPC when there are no memory stalls. During simulation, each L1 cache miss is passed to the classification hardware. If the cache miss is classified as one of the types of misses we are eliminating, then the cache access is performed with no latency.

For all of the Olden benchmarks, a single load classification dominates the misses for `health` and `treeadd`. Applying a single optimization targeting specific loads achieves very good results. All of `Health's` important load misses are pointer misses. All of the important misses in `treeadd` are captured by the next-line classification.

Figure II.16 presents the results for a suite of pointer-intensive applications. No one particular load class dominates the accesses. In this regard, a prefetching algorithm needs to be able to handle all four of these different classes

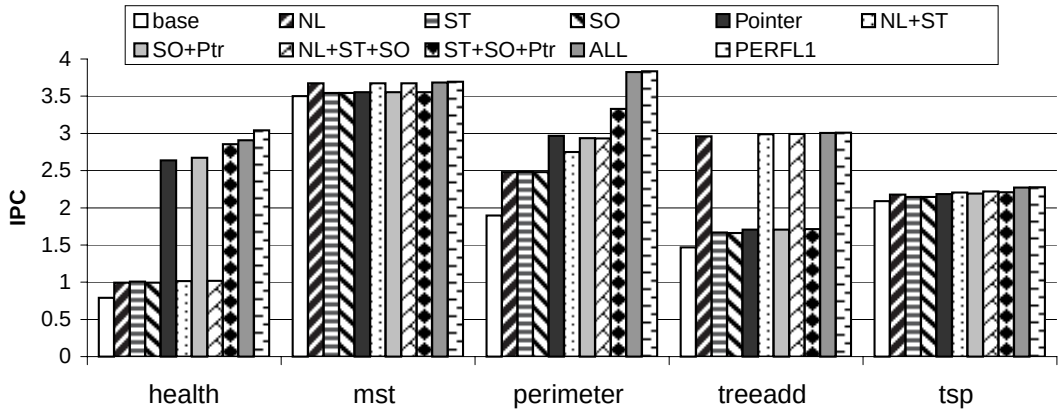

Figure II.15: Olden benchmark suite performance results when assigning perfect load latency for loads that match the different classifications.

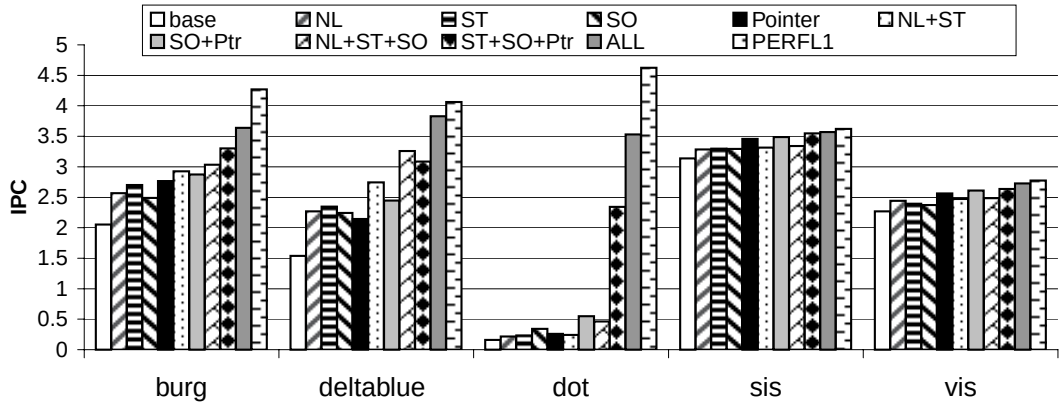

Figure II.16: Pointer benchmark suite performance results when assigning perfect load latency for loads that match the different classifications.

of loads at the same time in order to provide significant speedups. This is shown in `burg`, `deltablue`, `dot`, `mcf`, and `vis`. This is in stark contrast to most of the Olden benchmarks, in which handling just one of the classifications provides complete speedups.

II.E Quantifying Object and Pointer Behavior

One of the most challenging types of access patterns to capture are pointer transition patterns. These are often also the most critical to capture because of the high degree of dependence typically seen between pointer loads, and the poor spatial locality exhibited by this type of access. As seen in Section II.D.3 pointer misses provide the largest speedup when eliminated from the program. In order not to miss on this performance potential, we need a deeper understanding of pointer behavior.

There are two major factors that make capturing pointer behavior difficult. Pointer structures often have a high degree of fan out making the path to be traversed more difficult to choose. In addition, pointer transitions can be dynamic by their very nature and can change dramatically through the execution of the program via insertions or deletions to the data structure. Applications that have a high degree of fan-out and pointer transition variability will potentially be harder to accurately prefetch. In this section we provide an analysis of these two factors over our set of pointer based programs and the Olden benchmark suite.

II.E.1 Object Fan-Out

A given heap object that contains a set of n pointers to other objects, is said to have a *fan-out* of n. For example, a binary tree with a right and left child is said to have a fan-out of 2, while a tree with three child pointers is said to have a fan-out of 3. When calculating fan-out for an object, we only count a

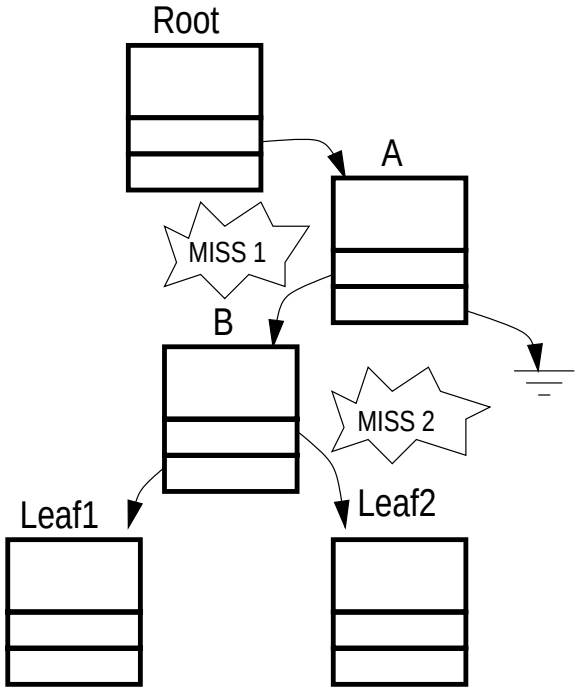

Figure II.17: Fan Out Example

pointer to another object as part of the fan-out if it is actually traversed at least once during execution.

Since we are concerned with memory performance, we are interested in pointer transitions from object A to object B that result in a cache miss. In this example, we are concerned with the object fan-out of A, because the fan-out (number of pointer transitions) out of A will influence the ability of the hardware to prefetch the cache miss transition to B. Figure II.17 shows an example of this. Suppose that we have the small tree, where A has one `NULL` child and one child transition to B, and node B has two real children. Now suppose that there is a cache miss when the program attempts to transition to node B, noted as $Miss1$ in the diagram. This cache miss will be classified as having a fan-out of 1, because it was the dereference of a pointer from an object (A) with a fan-out of 1. Cache $Miss2$ on the other hand will be noted as having a fan-out of 2,

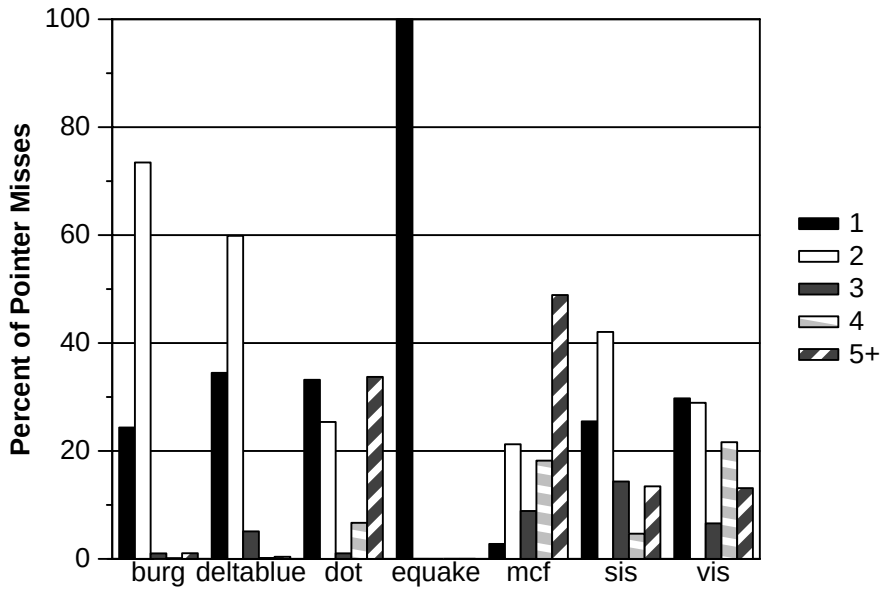

Figure II.18: Object fan-out of the pointer-based programs. A histogram of object fan-out is shown for L1 cache misses classified as pointer transitions in Section II.A.

because it comes from the dereference of node B, which has a fan-out of 2.

Now that we have this measure of fan-out, we wish to see how the misses are distributed across objects with different fan-outs. Figures II.18 and II.19 show the histogram of fan-out misses for both the pointer-based programs we have chosen and the Olden benchmarks. The fan-out results are shown for the L1 cache misses that are classified as pointer transitions in figures II.7, II.8 and II.9. Looking at the graphs in figure II.18, the fan-out for *equake* stands out. `Equake` has all of its misses coming from objects with a fan-out of 1, such as a simple linked list. The programs `deltablue` and `burg` are split between objects that have a fan-out 1 or 2 that transition to a miss, while `dot`, `mcf`, `sis`, and `vis` have misses from objects with many transitions to chose from. The dominate fan-out for `dot` and `mcf` is at 5 or greater.

This is in stark contrast to the behavior of the Olden benchmarks seen

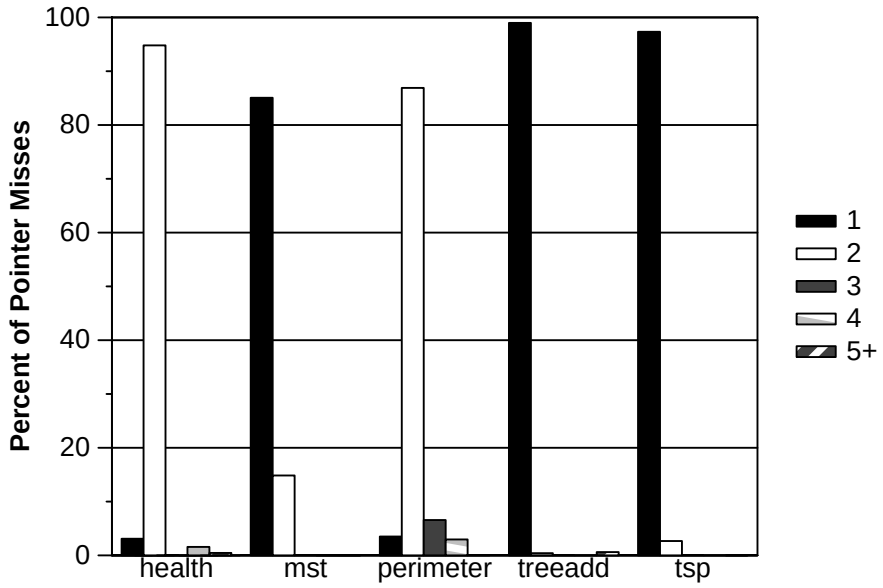

Figure II.19: Object fan-out of the Olden benchmark suite. A histogram of object fan-out is shown for L1 cache misses classified as pointer transitions in Section II.A.

in Figure II.19, where all of the programs are dominated by a single fan-out of either 1 or 2. This shows that the behavior of the Olden benchmarks is dominated by a single simple homogeneous data structure, which is not representative of the complexity inherent in the other pointer based applications.

II.E.2 Variability

Another factor that makes prefetching of pointer structures difficult is the fact that the pointer transitions to other objects changes over the lifetime of the application. In order to understand how the pointer structures change over time we add a new metric called variability.

The *variability* of a pointer in a program is the number of different values (addresses) it has over the life time of the program. In order for a data structure to change, the pointers within the structure must point to different objects. Every

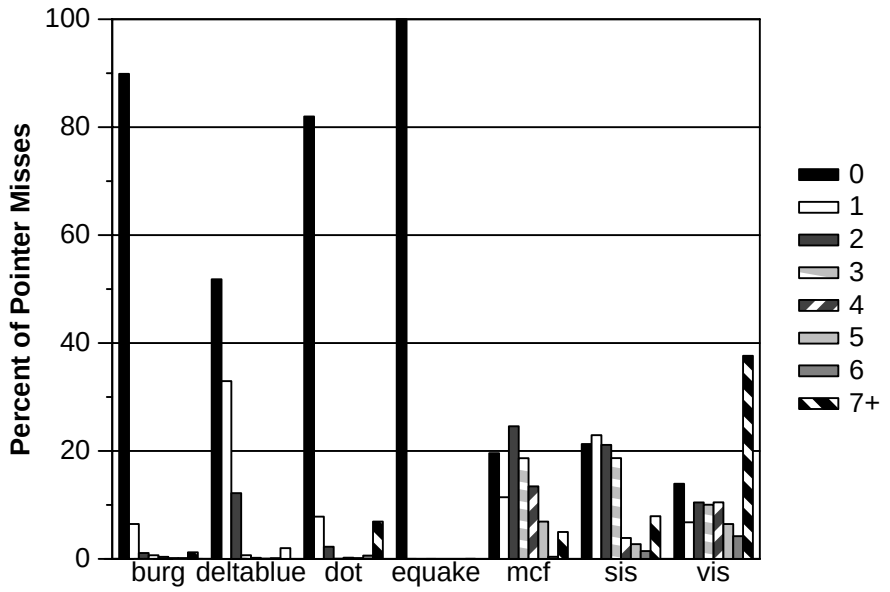

Figure II.20: Pointer variability for the pointer-based applications. A histogram of pointer variability is shown for L1 cache misses classified as pointer transitions in Section II.A.

time we see one of these changes, we record that it changed. After the program has completed running, we put all of the cache misses associated with a pointer address into a bucket based upon the number of different addresses stored in that pointer (the variability) during execution. From this we make the histograms seen in Figures II.20 and II.21. This shows the percent of pointer classification L1 misses that had the different degrees of variability.

In analyzing Figure II.20, `equake` again shows that the pointer transitions do not change during the execution after the initial data structure has been set up. This correlates to the fan-out results in Figure II.18, which showed each object has only one outgoing edge creating misses and that transition retains its value throughout the program. `Mcf`, `sis` and `vis` also are interesting to look at as most of the misses are caused by objects with high variability. These programs are difficult to accurately prefetch as the data stream is constantly changing and

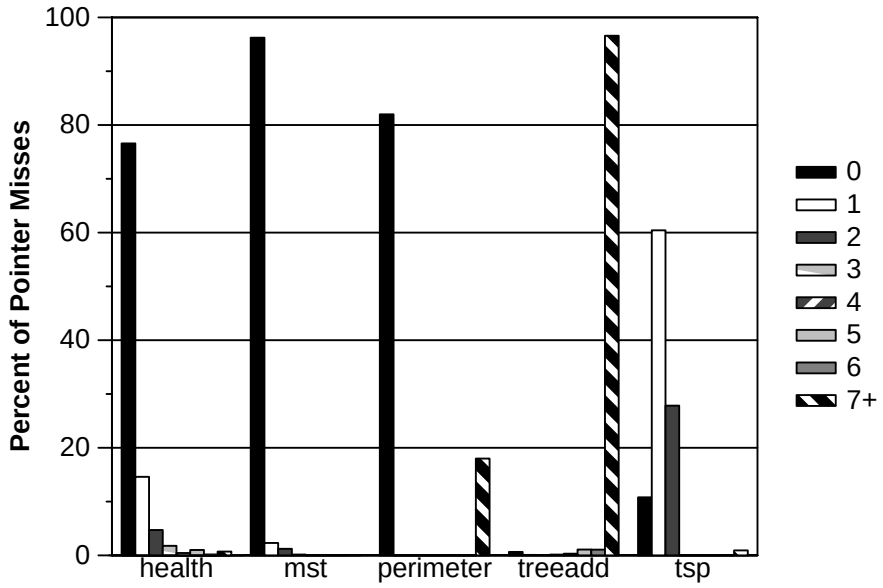

Figure II.21: Pointer variability for the Olden benchmark suite. A histogram of pointer variability is shown for L1 cache misses classified as pointer transitions in Section II.A.

hence, is highly unpredictable.

Most of the Olden benchmarks data structures remain fairly static with little variability, making them suitable for software based prefetching techniques [42]. The one program that is, at least at the surface, counter to this characterization is `treeadd`. The reason why `treeadd` is shown to have a high degree of variability is that only 3% of all of its cache misses were classified as pointer misses as shown in Figure II.8. These 3% of the misses came from a load in malloc which temporarily stores the address of newly allocated memory, and the pointer is overwritten repeatedly.

II.F Summary

The gap between processor performance and memory latency continues to grow at an astonishing rate, and because of this the memory hierarchy con-

tinues to be the target of a great deal of architectural research. In this paper, we present both an analysis of cache miss behavior, to help guide researchers in future cache and prefetching research, and a dynamic hardware technique to perform classification on the fly, which will enable architectural structures to be access pattern aware.

We classify load access patterns into one of four types, next-line, stride, same-object (additional misses that occur to a recently accessed object), and pointer-based transitions. These four access patterns account for more than 90% of all cache misses in the programs we examined. We then show a hardware technique that can detect this behavior using very little on-chip area. The dynamic classification technique presented can accurately predict more the 77% of cache misses as being of the correct type for all programs. On average across all programs, the technique correctly classifies 85% of all misses.

We also evaluate the potential benefit of correctly identifying load classes and the effect of removing their memory latency. This in effect simulates a perfect prefetcher for each class of loads. Our results show that a multi-pronged attack is needed to hide the majority of the memory latency. Prefetching only a single class of load does not provide noticeable benefits for the pointer-based collection of applications we examined. In contrast, removing the latency for only a single load stream classification achieved perfect results for a few of the Olden benchmarks.

In addition to the hardware classification technique presented, we further study those misses identified as pointer-based. Pointer-based misses have become the subject of a great deal of research in recent years and for future research it is important to understand their behavior.

To quantify the behavior of pointer loads, we examined two metrics each weighted by the number of cache misses for pointer-based loads. We use the fan-out metric of objects to quantify the branching factor that data structures have.

For a set of programs that are actually used to solve real problems, the fan-out tends to be both large and non-uniform. In contrast, the Olden benchmark suite shows both a very regular and a very small fan-out.

In addition to fan-out, we also examine how often a pointer transition changes over the life time of the application. To track this we keep a list of every pointer in the program and note how many times the pointer's value changes over the execution. We found that while about half the programs simply build and then destroy a large data structure, the other half change their data structures around quite often. This can make prefetching techniques that are based on learning the access pattern much more difficult to implement. The Olden benchmarks showed very little variability in their pointer transitions.

This study shows that a prefetching technique needs to be multifaceted and target multiple types of misses to be successful in achieving observable speedups. Furthermore, object fan-out and variability metrics indicate that of the four different miss types, pointer misses are the most difficult to handle. However, as shown in Section II.D.3, pointer misses also have the highest speedup potential when removed from the application. To take advantage of this performance potential, we describe the three prediction-based data prefetching techniques we propose in the remainder of this thesis.

Chapter III

Prior Work

Much work has been put into data prefetching in order to reduce the observed memory latency of data accesses. These include software schemes, hardware prefetching architectures, or a combination of the two. At the heart of data prefetching lies address prediction. Whether we have a software or a hardware scheme, we need to be able to predict future data addresses to issue prefetches for them. Compiler-based prefetching annotates load instructions or inserts explicit prefetch instructions to bring data into the cache before it is needed to hide the load latency. These techniques use locality analysis to determine where to insert prefetch instructions and show significant improvements [47]. Hardware-based prefetching can dynamically predict prefetch address streams and predict prefetch addresses that may be hard to find using compiler analysis. They can either prefetch into the data cache or into dedicated prefetch buffers to reduce pollution in the data cache. Depending on the complexity of the address predictor, these techniques can incur significant hardware costs. Compiler and hardware-based prefetching can be used together, since the compiler can be used to prefetch load instructions for which it can accurately determine locality information, and the hardware prefetcher can be used for those load address patterns not captured.

We will now examine some of the more relevant prior work in detail in

this chapter. We first provide details on prior work in address prediction and then relevant hardware and software prefetching techniques.

III.A Address Prediction

To guide hardware-based prefetching, accurate address prediction is needed. In performing this research, we examined using stride-based address prediction, Markov/context address prediction, and correlated address prediction.

III.A.1 Stride

A *stride* predictor [16, 25] keeps track of not only the last address referenced by a load, but also the difference between the last address of the load and the address before that. This difference is called the stride. The predictor speculates that the new address seen by the load will be the sum of the last address value and the stride. We chose to use the two-delta stride predictor [25, 60], which only replaces the predicted stride with a new stride if that new stride has been seen twice in a row.

The operation of the two-delta stride predictor is shown in Figure III.1. The predictor is indexed with the Program Counter (PC) of the load instruction and has fields for holding the last miss address, the transient stride, and the stable stride. Each time a load misses we index into the predictor with the load PC and insert the load's information in the table. When we see a load for the first time we update the last address field with the miss address. Upon observing each miss from this point on, we calculate the stride as the difference between the current miss address and the previous miss address stored in the table. We then store this in the predictor table as the transient stride. On subsequent misses we check if the calculated stride matches the transient stride in the table and if so

update the stable stride field with the calculated stride value.

III.A.2 Context/Markov Predictor

Context [60, 61, 75] and *Markov* [13, 14, 33] predictors are fundamentally similar, in that each predictor bases its prediction on a set of past values seen. An order k context/Markov predictor uses the k past values to predict the next one. It can only provide a prediction, if the given pattern has been seen and the transition is recorded into a prediction table.

Context predictors can capture miss streams that do not exhibit arithmetic patterns but repetition patterns. These misses are usually created by accesses to Linked Data Structures such as trees. The misses that occur when we traverse a sorted binary tree in search of a specific key is a good example III.2.

A typical Markov predictor counts the occurrences of a particular value immediately following a certain context. Thus, in general, there are as many counters as contexts (value patterns). The predicted address is the one that has the maximum counter value. In an actual implementation, the number of contexts that are being followed by the predictor are limited, bounded by the number of counters in the hardware table.

A Markov predictor assumes that the address stream seen in a program can be efficiently modeled by a Markov model. A Markov model is a set of states and transition frequencies where each state has a probability of transition to another. Each transition from address A to B is assigned a weight representing the fraction of As that are followed by a B. The Markov predictor described in [33] is a first order context predictor as it uses only the last address to predict the next one as shown in Figure III.3. The predictor has fields for holding the last miss address, and the predicted address. Each time a load misses we index into the predictor with the previous miss address and insert a transition from the

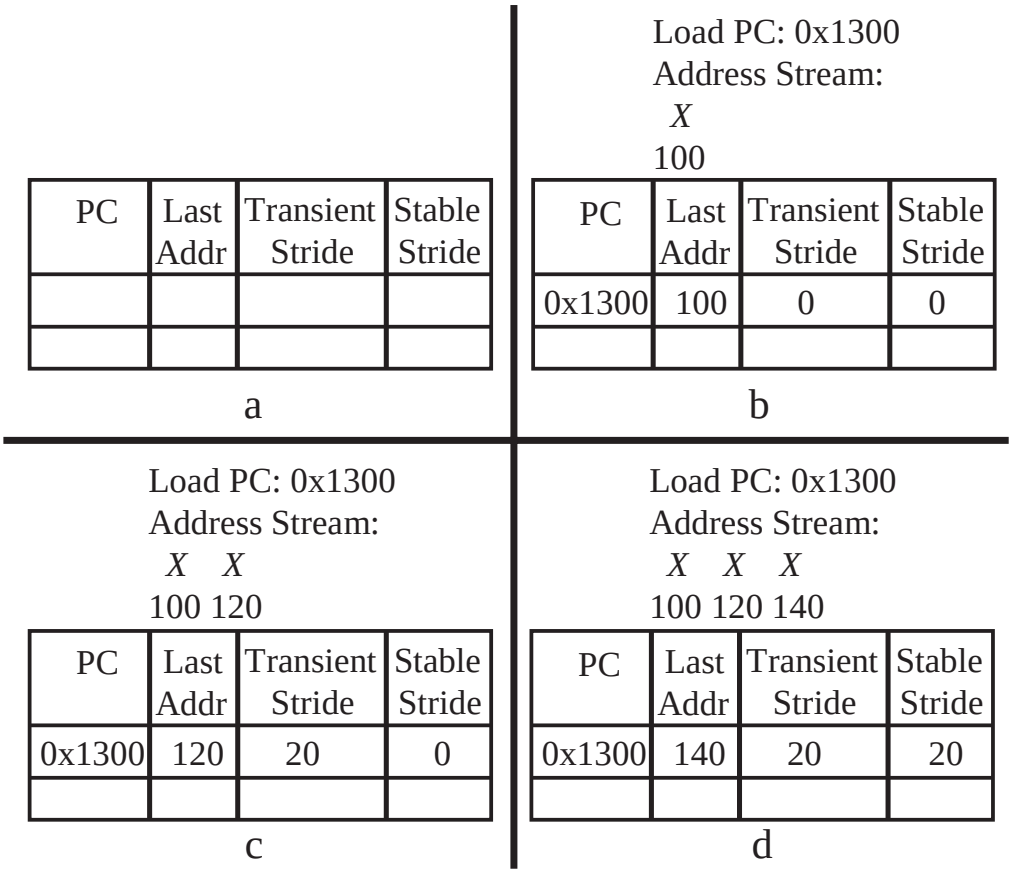

Figure III.1: The operation of the two-delta stride predictor. (a) The predictor is indexed with the Program Counter(PC) of the load instruction and has fields for holding the last miss address, the transient stride, and the stable stride. (b) Each time a load misses we index into the predictor with the load PC and insert the load's information in the table. In the example we update the last address field with the miss address 100. (c) Upon observing the second miss, we update the transient stride (20) as the difference between the current miss address (120) and the previous miss address stored in the table (100). (d) On a miss to address 140 by the same load we observe that the calculated stride ($140 - 120 = 20$) matches the transient stride in the table and update the stable stride field with 20.

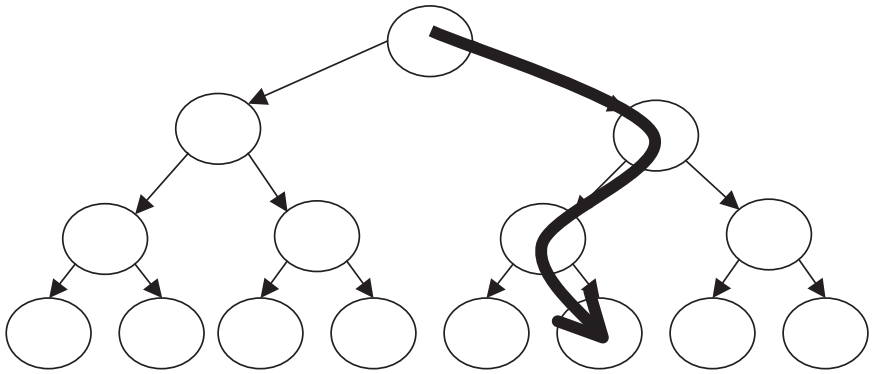

Figure III.2: Potential traversal pattern through a sorted binary tree searching for a key.

previous miss to the current miss. In the example mark the entry indexed with the miss address 100 as the active entry(as indicated with the bold box). Upon observing a miss to address 120, we insert a transition from 100 to 120 and mark the entry indexed with 120 as the active entry.

Bekerman et al. [4] propose yet another context-based predictor. For every load, they combine a series of past base addresses (they state that 4 is enough for reasonable accuracy - essentially amounting to a fourth order Markov predictor), to generate a history and store it into a first-level table. They use that history as an index into a second level table that stores a predicted *base* address. They then add the load's static offset (which could be stored in the first-level table) with the predicted base address. By using base addresses, a high-level of global correlation is achieved for multiple load instructions accessing different fields in the same object.

In this thesis, we only provide results for stride and first order Markov-based prediction. We simulated higher order Markov predictors and the correlation predictor [4], but saw little to no improvement in prediction accuracy and coverage over the first order Markov predictor for the programs we examined. This is partially due to the fact that correlated loads lie within the same

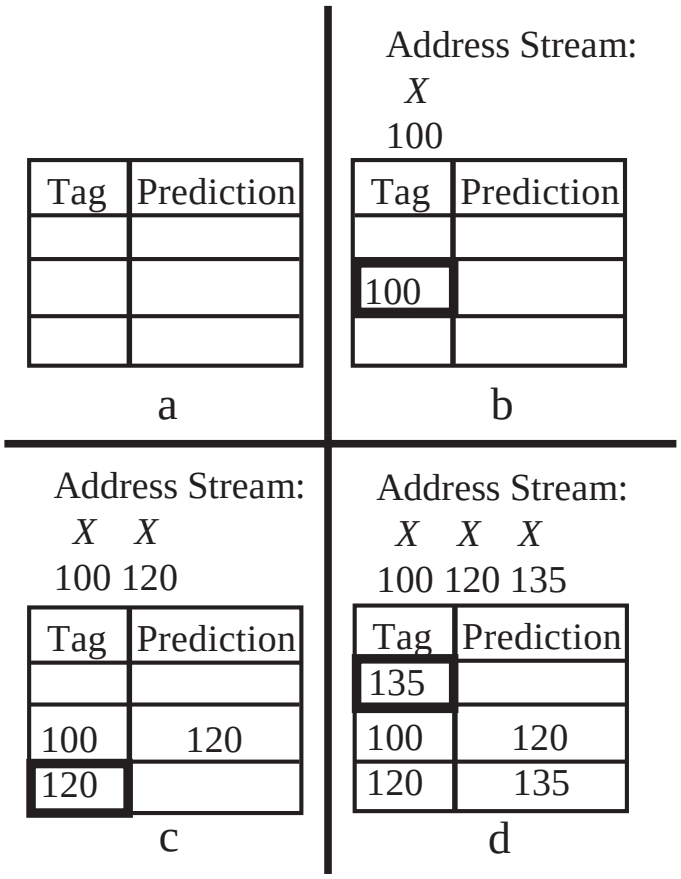

Figure III.3: The operation of the Markov predictor. (a) The predictor is indexed with the miss address of the load instruction and has fields for holding the last miss address, and the predicted address. (b) Each time a load misses we index into the predictor with the previous miss address and insert a transition from the previous miss to the current miss. In the example mark the entry indexed with the miss address 100 as the active entry(as indicated with the bold box). (c) Upon observing a miss to address 120, we insert a transition from 100 to 120 and mark the entry indexed with 120 as the active entry. (d) On a miss to address 135, we insert a transition from 120 to 135 and mark the entry indexed with 135 as the active entry and so on.

cache block for the programs we examined. Therefore, correctly predicting the correlated load provides less gains in terms of prefetching, since we perform our predictions and prefetches at the cache block granularity.

III.B Hardware Prefetching Models

We classify the prior hardware prefetching research into three models – Fetch Stream Prefetching, Demand-Based Prefetching, and Decoupled Prefetching.

III.B.1 Fetch Stream Prefetching

The first model follows the branch prediction or fetch stream, predicting and prefetching addresses [6, 15, 17, 30].

Chen and Baer [15] proposed an approach to provide the load prediction early by using a Look-Ahead PC(LA-PC), which can run ahead of the normal instruction fetch engine. The LA-PC is guided by a branch prediction architecture that runs ahead of the fetch engine, and is used to index into an address prediction table to predict data addresses for cache prefetching. Since the LA-PC provided the instruction address stream ahead of the normal fetch engine, they were able to initiate data cache prefetches farther in advance than if they had used the normal PC, which in turn allowed more of the data cache miss penalty to be masked. The amount of load latency that can be hidden is dependent upon how far the look-ahead PC can get in front of the execution stream.

Reinman et al. [51] extended the approach of Chen and Baer [15] to instruction prefetching. In their approach, they only have one branch predictor instead of two as in Chen and Baer. This is accomplished by decoupling the branch predictor from the instruction cache with a fetch target queue between them. The queue is used to store fetch block predictions, which are then fed into

the instruction cache in a later cycle. The fetch addresses in the queue are used to perform instruction cache prefetching. These same fetch addresses can also be used to guide data prefetching, similar to what was proposed in [15].

They recently extended this approach to perform power-efficient instruction prefetching by decoupling the tag component of the instruction cache access from the data component of the cache access [52]. The tag component verifies if an address is in the cache in a separate cycle before the data component access for the instruction lookup. If the fetch address is not found, it is prefetched, while the fetch address is queued up to be consumed by the data component. In this new design, the data component access consumes significantly less power, since only one way of the data component is driven, and the way was determined during the tag access in a prior cycle. They are currently extending this design to fetch stream data cache prefetching.

III.B.2 Demand-Based Prefetching

The second model can be classified as demand-based prefetching. In this approach an action such as a cache miss or the use of a cache block has to occur for each prefetch generated.

An early example of a demand-based prefetching architecture is *Next Line Prefetching* (NLP) by Smith [68], where each cache block was tagged with a bit indicating when the next block should be prefetched. When a block is prefetched its tag bit is set to zero. When the block is accessed during a fetch and the bit is zero, a prefetch of the next sequential block is triggered and the bit is set to one.

Another demand-based prefetching architecture is Shadow Directory Prefetching by Charney and Puzak [13]. In this approach, each L2 cache block has a shadow address associated with it. The shadow address points to the

cache block accessed right after the corresponding cache block, providing a simple Markov transition. A hit in the L2 cache with a valid shadow entry triggers a prefetch of the shadow address. Alexander and Kedem [1] examined using a similar Markov table, but distributed over the DRAM modules, which are used to prefetch cache blocks from DRAM array into an SRAM buffer.

A recently proposed demand-based prefetcher identifies/predicts when an L1 data cache block becomes "dead" (i.e. evictable). It then uses an address predictor to prefetch a cache block to be potentially used in the future in the place of that "dead" block [36]. The result of the address predictor is similar to shadow prefetching [13], predicting a cache block that was previously mapped to the same block frame. It will potentially record the address that occurred right after the dead cache block was evicted, and then predict this same address when that cache block is predicted to be "dead" in the future.

The last example we will discuss is the Markov prefetcher used by Joseph and Grunwald [33]. When a cache miss occurred, the miss address would index into their Markov prediction table to provide the next set of possible cache addresses that have followed this miss address before. After these addresses are prefetched, the prefetcher stays idle until the next cache miss. They do not use the predicted addresses to re-index into the table to generate more predictions for prefetching.

They also examined storing bits in their Markov predictor table to indicate whether the prefetch address was actually used after being prefetched. If not, it was not used for a given number of times, then the prefetch would not be performed. A similar filter was examined by Luk and Mowry [43] where a pollution counter is kept with each cache block in the L2 cache. This counter keeps track of the number of times a cache block was prefetched from the L2, but not used. When the counter was above a filter threshold the prefetch request

would be cancelled. They found this to be a very beneficial for a prefetching architecture that stored prefetched blocks directly into the instruction cache. We did not examine this pollution filter because we store our prefetched blocks into a prefetch buffer before moving those that hit into the instruction cache.

III.B.3 Decoupled/Stream Prefetching

In this model the prefetcher is loosely decoupled from the instruction fetch stream and can potentially prefetch down multiple streams independent of what the instruction fetch stream is doing.

General Decoupled Models

An access decoupled architecture partitions programs into a prefetching instruction stream and an execution instruction stream [5, 28, 32]. As long as the prefetch stream can run ahead of the execution stream, the memory latency can be masked. Roth et al. [53, 54] have examined both a software and a hardware approach for prefetching linked data structures using a decoupled model.

Yang and Lebeck [77] examined an architecture which uses the compiler to create small prefetch kernels of instructions, which are executed in parallel with the original application on a separate prefetch engine. Yang and Lebeck [77] examine a decoupled processing model where linked data structure kernels are executed ahead of the actual computation, pushing the data to the processor instead of pulling it as in normal prefetching. The examine *traversal kernels*, which is a compact representation of a given linked data structure recurrence.

A similar technique was recently proposed by Solihin et al. [69]. They present a scheme where context-based prefetching is performed by a User-Level Memory Thread running on a simple general-purpose processor in memory. A software handler records L2 miss addresses in a correlation table which resides

in memory. When a stripped down version of the target application executes on the processor in memory, the software handler accesses the correlation table to prefetch several blocks into the L2 cache. To provide high coverage and timely prefetches, their correlation table stores several levels of successor misses per entry.

Another prefetching technique targeting pointer-based applications is Dependence-Based Prefetching (DBP) [53]. DBP identifies recurrent pointer-chasing loads in hardware, and then prefetches them sequentially in a prefetch engine. DBP can exploit memory parallelism for simple backbone and rib traversals shown in Figure III.4. In this figure there is a list of `rib` nodes, which are connected to each other. Separate lists of rib nodes are linked together via a chain of `backbone` nodes. DBP cannot create overlap among separate rib lists for more complex traversals because it pursues only one backbone node ahead of the CPU to minimize useless prefetches. A follow-up paper to DBP proposed Cooperative Chain Jumping [54], a technique that combines DBP with jump pointers for backbone and rib structures. Jump pointers are artificial pointers added to the data structure to improve the timeliness of prefetches. They create memory parallelism amongst the nodes on the backbone, and then DBP hardware sequentially prefetches each rib. Cooperative Chain Jumping can create memory parallelism across nodes from different rib-lists. However, once again it does so only for backbone and rib traversals, and it requires artificial jump pointers. More complex structures where there are multiple potential traversal paths cannot be easily prefetched without heavy jump pointer assistance.

In contrast, the techniques that we explore in this study are independent of the data structure being traversed. Especially the techniques presented in Chapters V and VI are powerful prefetching schemes that can accurately prefetch down arbitrary data structures due to its control flow sensitive nature.

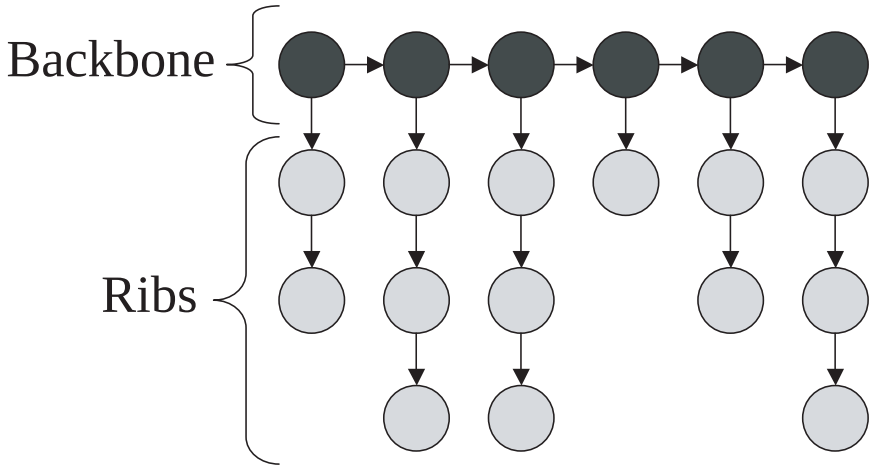

Figure III.4: A dynamic data structure that has a backbone layer connecting groups of rib nodes.

Stream Buffers

Jouppi introduced *stream buffers* to improve direct mapped cache performance [34]. The stream buffers follow multiple streams prefetching them in parallel and these streams can be decoupled from the instruction stream of the processor. They are designed as FIFO buffers that prefetch consecutive cache blocks, starting with the one that missed in the L1 cache. On subsequent misses, the head of the stream buffer is probed. If the reference hits, that block is transferred to the L1 cache.

Figure III.5 shows how the stream buffers interact with the rest of the system. On an L1 miss, we allocate a stream buffer. Following allocation, the stream buffers starts prefetching sequential cache blocks until all stream buffer entries are filled. At this point the stream buffer sits idle until a hit or a deallocation occurs. Meanwhile, when a load instruction enters the pipeline, we query the stream buffers in parallel with the L1 data cache. On an L1 miss and a hit in the stream buffers, we transfer the cache block from the stream buffer to the L1 cache. We also forward the required data to the register file. Subsequently,

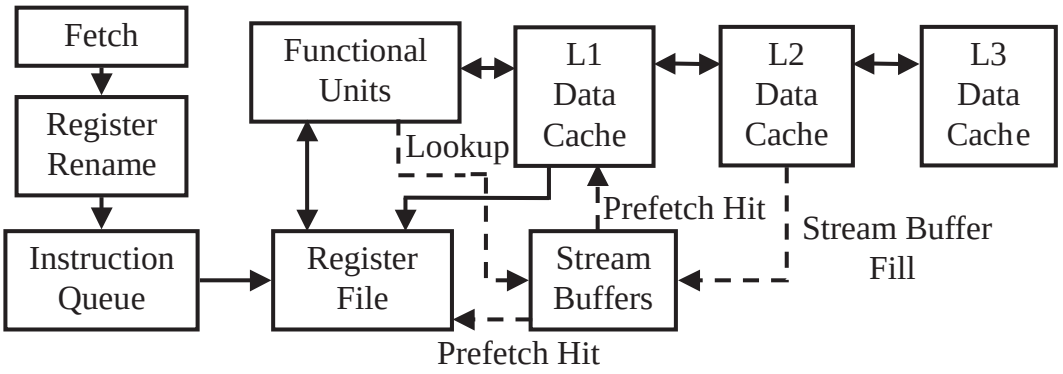

Figure III.5: The interaction of stream buffers with the rest of the processor. A stream buffer prefetches consecutive cache blocks, starting with the one that missed in the L1 cache. On subsequent misses, the head of the stream buffer is probed. If the reference hits, that block is transferred to the L1 cache.

the stream buffer becomes available for further prefetches.

Palacharla and Kessler [49] suggested two techniques to enhance the effectiveness of stream buffers: *allocation filters* and a *minimum delta non-unit stride* detection mechanism. The filter prevents a stream buffer from being allocated until two consecutive misses occur for the same stream. With the non-unit stride scheme, the dynamic stride is determined by the minimum signed difference between the miss address and the past N miss addresses. If this minimum delta is smaller then the L1 block size, then the stride is set to the cache block size with the sign of the minimum delta. Otherwise, the stride is set to the minimum delta. To implement the non-unit stride detection an address indexed stride table is used. To find the striding behavior the memory is divided up into chunks, and associated with each chunk is a stride.

Farkas et al. [26] made an important contribution by extending Palacharla and Kessler's model to use a *PC-based* stride predictor to provide the stride on stream buffer allocation. The PC-stride predictor determines the stride for a load instruction by using the PC to index into a stride address prediction table. This

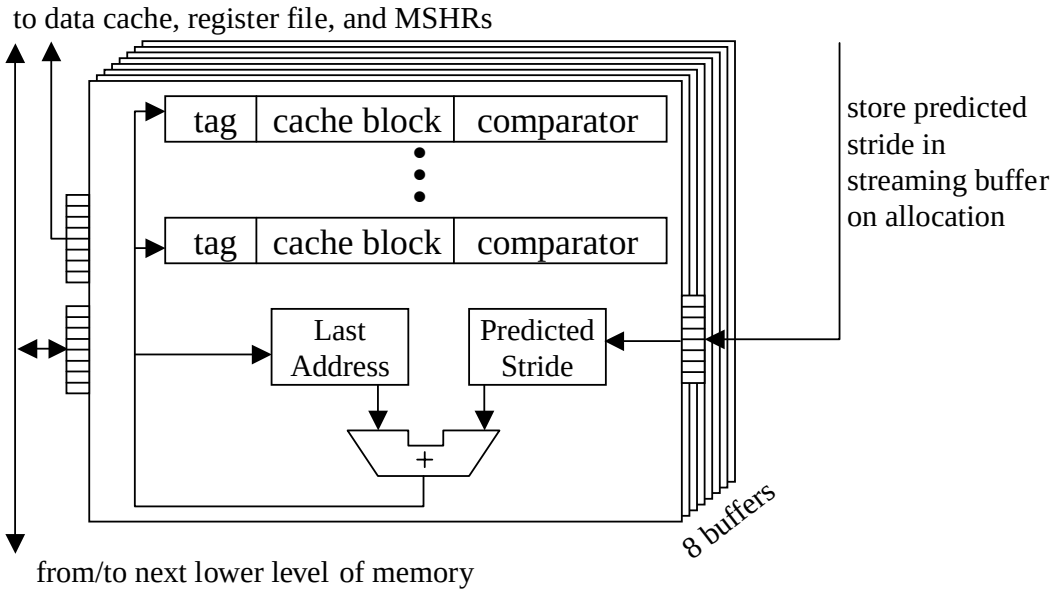

Figure III.6: A stream buffer architecture guided by a PC-based stride prediction table.

differs from the minimum-delta scheme, since the minimum-delta uses the global history to calculate the stride for a given load. PC-stride predictor uses an associative buffer to record the last miss address for N load instructions, along with their program counter values. Thus, the stride prediction for a stream buffer is based only on the past memory behavior of the load for which the stream buffer was allocated. We compared the global minimum-delta approach to the PC-stride predictor and found that it was uniformly outperformed by the per-load stride detector. Therefore, we only present comparison results of our approach with PC-based stride prediction stream buffers. Figure III.6 shows a set of eight stream buffers with this configuration.

Farkas and Jouppi [27] further enhanced the stream buffer design of Palacharla and Kessler by enforcing the streams being followed by multiple stream buffers to be non-overlapping. This prevented duplication and saved bus bandwidth. Furthermore, instead of the FIFO structure which had been originally

proposed by Jouppi in [34], they used a fully-associative stream buffer lookup, which we model.

Speculative Pre-Execution

Several thread-based prefetching paradigms have been proposed, including Collins et al.'s Speculative Precomputation (SP) [23], Zilles and Sohi's Speculative Slices [80], Roth and Sohi's Data Driven Multithreading [55], Luk's Software Controlled Pre- Execution [41], and Annavaram's Data Graph Precomputation [2].

The goal of this area of research is to (1) identify loads that are problematic, (2) construct a minimal sequence of instructions (kernel) needed to produce the address stream for these loads, and then (3) execute this minimal kernel speculatively on an idle thread in a Simultaneous Multithreading processor [74]. These architectures attempt to prefetch the address stream ahead of the execution stream, via the minimal kernel running on a spare multi-threaded hardware context.

Speculative precomputation [23] works by identifying the small number of static loads, known as delinquent loads, that are responsible for the vast majority of memory stall cycles. Precomputation slices (p-slices), sequences of dependent instructions which, when executed, produce the address of a future delinquent load, are extracted from the program being accelerated. When an instruction in the non-speculative thread that has been identified as a trigger instruction reaches some point in the pipeline (typically commit or rename), the corresponding p-slice is spawned into an available SMT thread context.

Speculative slices [80] focus largely on the use of precomputation to predict future branch outcomes and to correlate predictions to future branch instances in the non-speculative thread, but they also support load prefetching.

Software controlled pre-execution [41] focuses on the use of specialized, compiler inserted code that is executed in available hardware thread contexts to provide prefetches for a non-speculative thread.

Data graph precomputation [2] explores the runtime construction of instruction dependence graphs (similar to the p-slices of SP) through analysis of instructions currently within the instruction queue. The constructed graphs are speculatively executed on a specialized secondary execution pipeline.

In general, the advantage of these decoupled approaches is that they follow an actual instruction stream allowing them to more effectively determine the pointer traversals to follow, since they can potentially pre-execute branch instructions. A potential disadvantage of speculative pre-execution is that the minimal kernels can sometimes have a hard time running far enough ahead of the miss stream to hide the miss latency due to dependencies in the pointer chain.

In comparison, predictor-based approaches can run ahead more easily when traversing the pointer chain, since they are based on predicting the address. This of course assumes that a compressed version of the pointer traversal over the miss stream can be captured in our predictor. The disadvantage is that these techniques can have lower prefetch accuracy when it comes to objects with many next pointer traversals, unless the prefetcher incorporates branch prediction information to help guide the address predictor down its prediction stream.

III.C Software Prefetching Models

There is a significant amount of research on software-based prefetching where the compiler is used to analyze loops and insert prefetch instructions [12, 47]. For pointer-based applications, Luk and Mowry examined using the compiler to insert prefetches for Linked Data Structures [42]. They examined adding *jump pointers* to hook up a heap object X to another heap object Y a number of

objects earlier in the structure, by adding an explicit pointer from Y to X. On future traversals of the data structure, the targets of these extra pointers are prefetched. This technique has the ability to hide more latency than demand pointer chasing as in the greedy algorithm, but comes at a cost of adding in jump pointer structures. In addition, this could potentially perform badly if the structure of the linked data structure changes radically between traversals over the structure.

Natural jump pointers on the other hand are existing pointers in the data structure used for prefetching. It is assumed that when an object is visited one of its neighbors will also be accessed in the near future and prefetches are issued for all the pointer fields in the object. This is a greedy technique that is often referred to as *Greedy Prefetching*. These techniques were introduced by Luk and Mowry [42] and refined in [35] and [54]. In [54], Roth and Sohi proposed adding explicit prefetch fields to data structures, and these fields are prefetched each time a structure is processed. These fields could be either explicitly inserted into a data structure, or added as padding when allocating the data structure. They examine using a compiler and hardware approach to guide the prefetches for these additional fields.

As discussed in Section III.B.3, Roth et al. [53] also proposed using a Correlation Table and Potential Producer Window to find the load instructions that produce addresses, and the corresponding loads that use those addresses. Their Dependence-Based prefetching architecture then finds the instructions that load pointer addresses during the fetch stage, which in turn can initiate a stream of prefetches, basically pre-executing the linked data structure's kernel.

Luk and Mowry also examined using correlation profiles to determine where to insert prefetch instructions for linked structures [48]. Karlsson et al. [35] extend Luk and Mowry's research by building prefetch arrays, and then prefetch-

ing these nodes several links ahead to try and mask pointer latency. In [38], Lipasti et al. describe a compiler approach for inserting prefetches for pointer-based applications. They insert prefetches for pointer arguments that are passed to a procedure, before the call site in a program.

Zhang and Torrellas [78] recognized the benefit of grouping together fields or objects that are used together, and prefetching these all together as a prefetch group of blocks. They examined adding specialized grouping instructions that allowed the user to group which fields/objects should be prefetched together. These groupings are then stored in a hardware buffer, and as soon as one of them is referenced and misses, all the cache blocks in the group are prefetched.

Recently Chilimbi and Hirzel [18] proposed an automated software approach based on correlation. Their scheme first gathers a data reference profile via sampling. Next, they process the trace to extract data reference sequences that frequently repeat in the same order. At this point, the system inserts prefetch instructions to detect and prefetch these frequent data references. The sampling and optimization are done dynamically at runtime with very low overhead.

III.D Summary

As described in this chapter, there is a plethora of research on data prefetching. This large body of work focuses on overcoming the challenges brought about by the constantly diverging CPU-Memory access speeds.

If we categorize the prior art on data prefetching, we can see that most of the work has focused on highly predictable address streams such as strided accesses in single or multidimensional arrays. As a matter of fact, the Intel Pentium 4 processor has a built in hardware prefetcher that implements a prefetching technique similar to stream buffers. Even though a few software techniques focus on the irregular accesses seen in pointer-based applications, there has been only

a few efforts in tackling this problem in hardware, mainly due to the difficulty of implementing address predictors that are large enough to capture the program's pointer working set. This thesis proposes a cost-effective predictor to overcome this challenge in Chapter V.

Chapter IV

Predictor-Directed Stream Buffers

As elaborated in the previous chapters, there has been a great deal of effort in reducing the impact of cache misses on program performance. However, as we stated in Chapter III a significant portion of that research is on regular miss streams such as Next-line or Stride misses. Since we are focusing on hardware prefetching techniques for pointer-based applications, in this chapter we propose an efficient prefetching architecture building upon the simple *Stream Buffer* framework.

As described in Chapter III stream buffers were originally proposed by Jouppi [34] to prefetch a stream of sequential cache blocks. When a cache miss occurs, the next sequential cache block is allocated into a stream buffer. The stream buffer then prefetches sequential cache blocks from that address, as bandwidth permits, until the buffer is full. As prefetches are used, new data is brought in, keeping the buffer far enough in advance of the data's use so that it can potentially hide the entire latency. Along with introducing allocation filters, Palacharla and Kessler [49] extended stream buffers by associating a stride with

each stream buffer. They examined providing a stride from a table which was indexed by the area of memory being accessed. Farkas et al. [26] further extended this research by using a PC indexed stride table, which allows for detection of many strides over the same region of memory.

In this chapter we propose a new form of stream buffer called the *Predictor-Directed Stream Buffer* (PSB). Instead of associating a fixed stride with each buffer, we use a *predictor* to generate the next address to prefetch. Then that prefetch address can be used to generate the next predicted address to prefetch. After allocation, a stream buffer is decoupled from the execution stream. In other words, it can independently prefetch down a predicted address stream. We simulate the use of a hybrid *Stride Filtered Markov* (SFM) predictor to direct stream buffer prefetching and find it is quite adept at finding both complex array access and pointer chasing behavior over a set of pointer intensive benchmarks.

Farkas et al. [26] show the importance of using allocation filters to prevent the stream buffers from being allocated and deallocated too often and for too many streams, an effect we call *stream thrashing*. We propose a technique based on confidence for eliminating stream thrashing as well as making more effective use of available processor and predictor resources. This is done by using confidence to guide stream buffer allocation and prefetch prioritization.

The rest of the chapter is organized as follows. Section IV.A describes our PSB architecture. Simulation methodology and benchmark descriptions can be found in Section IV.B. Section IV.C presents results for our architecture, and our conclusions are summarized in Section IV.D.

IV.A Predictor-Directed Stream Buffers

We will now describe our Predictor-directed Stream Buffer (PSB) architecture. The PSB architecture resides on chip and prefetches data from the

L2 cache and main memory into the stream buffers. If a prefetch request is not found in the L2, it will service the request from main memory. We concentrate on stream buffers instead of the other architectures described in the previous section because of their simple yet effective design, their ability to follow a prefetch stream independent of the fetch stream, and the design fits nicely with an on-chip prefetcher to try to hide L2 and main memory latency.

We present an approach that extends the PC indexed stream buffer design of Farkas et al. [26]. As described in Chapter III, the PC index scheme uses a stream buffer which is guided by a static stride, provided at allocation time by a per-PC stride table as shown in Figure IV.1. This approach can work well for stride-based applications, but the stream buffers do not follow the correct stream for non-stride based load patterns, such as during the traversal of a recursive data structure.

To address this problem, we propose Predictor-Directed Stream Buffers (PSB). The general idea of a PSB is to use a predictor to generate an address stream for prefetching. The predictor takes as input some prediction information, such as the last address accessed and history information, and then generates a prediction for a given stream buffer. This prediction is then stored back into the stream buffer, and the prediction information in the stream buffer is updated. In this way we can generate prediction n from prediction $n-1$. The base of the recursion is a cache miss which causes a stream buffer allocation. This is illustrated in Figure IV.2 and a PSB implementation is shown in Figure IV.3.

There are two major parts to PSB, a per-stream history which is stored with each stream buffer, and an address predictor which is shared between stream buffers. The per-stream history is used to keep data about a particular stream buffer including confidence information, and local stride. This data may be used for a variety of purposes, such as indexing into the address predictor. The primary

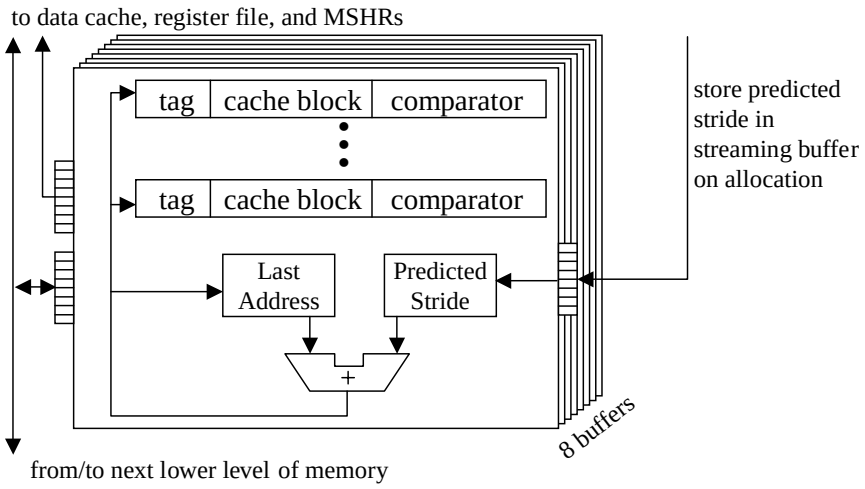

Figure IV.1: Stride-based Stream Buffer Architecture. Eight stream buffers are shown (overlapping each other). Each stream buffer can hold N cache blocks. When a stream buffer is allocated, it is assigned a predicted stride to use to generate all of its prefetch addresses.

service of the per-stream history is to store the current speculative state which can be fed to the predictor to generate the next address prediction for that stream buffer. The prediction from the address prediction table is then used to update the state information in the stream buffer so that a new speculative prediction can be made. It is a key point that the address prediction table is *not* updated when the stream buffer makes a prediction, this step is done separately in the write-back stage when a load has a data cache miss. This model allows the stream buffer to follow the address prediction stream of any type of address predictor.

IV.A.1 Predictor-Directed Stream Buffer Implementation

Figure IV.3 shows a specific implementation of our predictor-directed stream buffer architecture. Each stream buffer holds (1) the PC of the load that caused the stream buffer to be allocated, (2) the last predicted address for

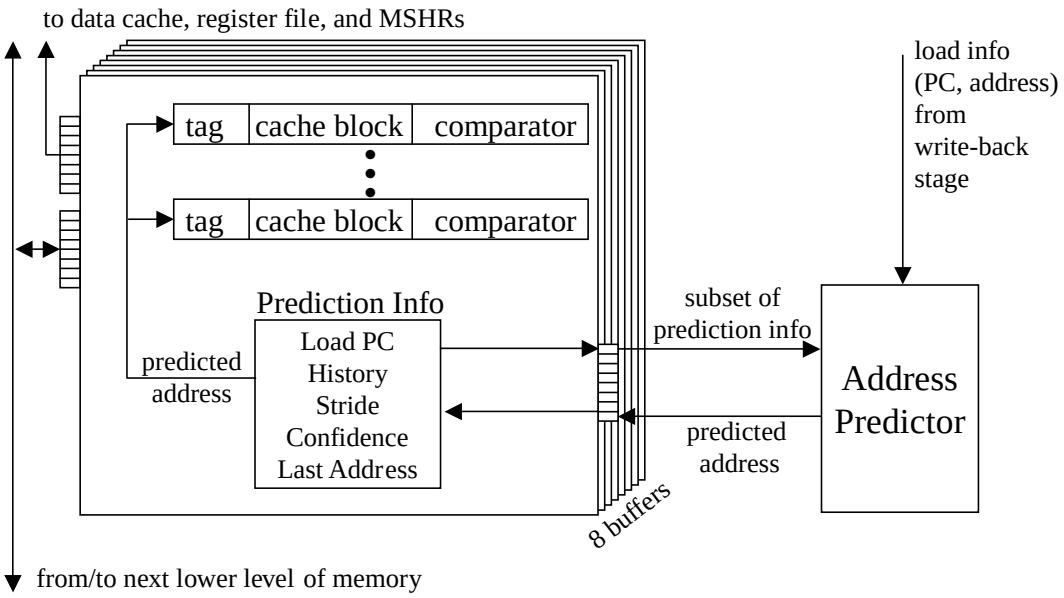

Figure IV.2: A Predictor-Directed Stream Buffer. We modify the stream buffer so it accesses a separate address prediction table to get its next prefetch address. When a stream buffer is allocated necessary prediction information is copied into the stream buffer to enable its access to the address predictor.

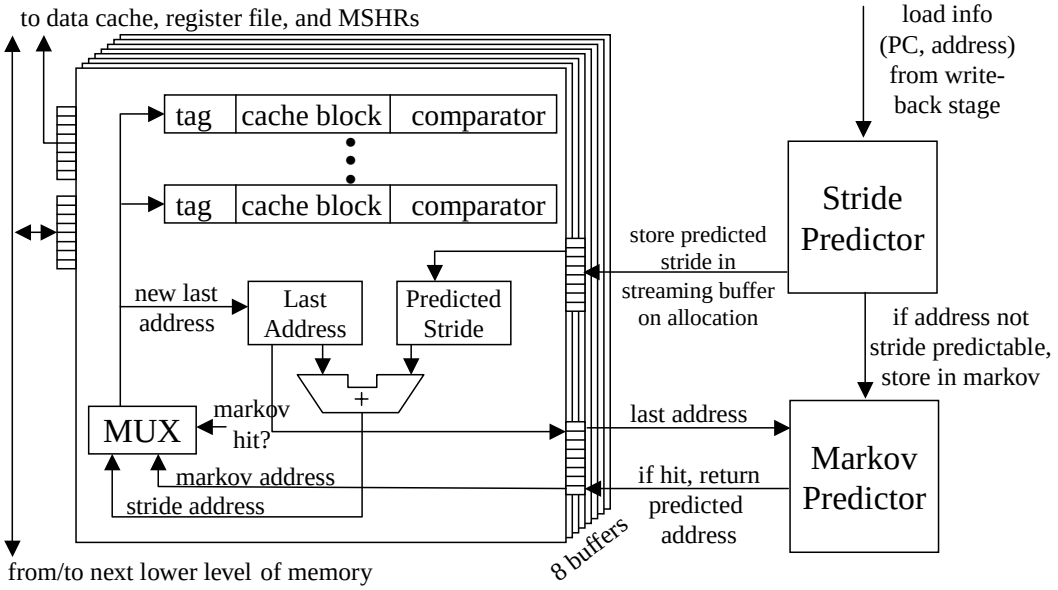

Figure IV.3: Stride-Filtered Markov Predictor-Directed Stream Buffer Architecture. When a stream buffer is allocated, it is assigned a fixed stride from the stride table. To generate the next prefetch address the last address is (1) looked up in the Markov table and (2) used to calculate a next stride address. If the Markov table hits, then the Markov address is used, otherwise the next stride address is used for the prefetch.

the load, and (3) any additional prediction information (e.g., history state or confidence) needed to perform the next address prediction. The stream buffer is on-chip next to the address predictor, which in our case is a stride-filtered Markov predictor.

There are several stages of execution a stream buffer will go through over the course of a program, starting with the allocation of a stream and ending with it's reallocation. We now describe the initialization and steady state operation of a stream buffer.

Allocation A stream buffer is allocated, subject to allocation filters (see Section IV.A.3), when a load executes and it misses both in the data cache and the

stream buffer. If a load miss is assigned to a stream buffer, the load PC, current address, and any additional prediction information are copied to the stream buffer from the address predictor. This initialization stage is only done once per allocation, and is directed only from predictor to stream buffer. The state of the address predictor is not modified. This copied state will later be used for indexing into the prediction table.

Prediction Each cycle, one stream buffer is chosen to make a prediction using the address predictor, according to priority heuristics described in Section IV.A.5. The information stored in the stream buffer is used to index into the address predictor, returning the next predicted address, and potentially updating the stream buffer's history information. We properly model allowing only a single prediction per cycle to be generated from the predictor. Due to the fact that only one request (miss or prefetch) can be processed by the bus from the L1 to the L2 cache at a time, the predictor was not a bottleneck even with the one prediction per cycle limitation.

Once a stream buffer has been allocated, the stream buffer's history information is updated after each prediction. The address prediction table, as was mentioned earlier, remains unchanged while generating a prediction for a stream buffer. For example, a design such as a context predictor which uses a history of the last N addresses to index into the address predictor would store the history of its last N predictions in the stream buffer. This would then be used as an index into the address predictor each cycle. The history of the last N addresses stored in the stream buffer is updated after a prediction, *not* the state in the address prediction table. Therefore, the stream buffer maintains its own prediction history information.

Before inserting the prediction into the stream buffer, the stream buffers are searched in parallel for the cache block of the predicted address. This was used

by Farkas et al., [26] to prevent stream buffers from prefetching down overlapping paths. To alleviate the pressure on the stream buffer ports and tags, we replicated the tags and added a dedicated port for this check. Note that this does not introduce a significant overhead as there are only 8 stream buffers and they each have only 4 entries. If the prediction was found to be already resident in a buffer entry, then it is ignored, no useful prediction is made that cycle, and the stream buffer prediction history information is updated. If prediction is not found in a stream buffer, the prediction is stored in the stream buffer's least recently used entry, and that entry is marked as ready for prefetching. Once all entries have been predicted for a stream buffer, no further entries will be predicted until (1) an entry is cleared during a lookup (it is a hit), or (2) the stream buffer is reallocated.

Prefetching Once an entry has a valid prediction associated with it, it is ready to be prefetched. We only allow prefetching to occur if the L1-L2 bus is free at the start of any given cycle. When the bus is free, a stream buffer with an entry containing a valid un-prefetched prediction is chosen using the priority scheduling algorithms described in Section IV.A.5. The prefetch is then sent to the lower levels of memory and the entry is marked as prefetched and waiting.

Lookup When a load performs a lookup in the L1 data cache, it searches all of the stream buffer entries in parallel for a hit. For our results, we assume the data cache lookup latency is 3 cycles whereas the stream buffer lookup latency is a single cycle. If there is a hit in the stream buffer, and the data is not in the data cache, the cache block stored in the stream buffer is moved into the data cache. If there is a tag hit in the stream buffer, but the block is not ready yet, the tag is moved into a data cache MSHR, and the data cache handles the block when it comes back from memory. For a stream buffer hit, the corresponding stream buffer entry is freed for a new prediction and prefetch.

We will now describe our design using a Stride-Filtered Markov (SFM)

address predictor, although any address predictor [4, 33, 60, 61, 75] can be used to guide the predictor-directed stream buffer. We examined several types of predictors (including stride with correlated [4]), but only provide results for a SFM table, as it performed uniformly better.

IV.A.2 Stride-Filtered Markov Predictor

Charney and Reeves [14] and also Joseph and Grunwald [33] introduced *Markov* prefetching, and provided results for a "stride and Markov in series" predictor. We use this predictor to guide our predictor-directed stream buffer, and make a few minor improvements which are described below.

To provide address prediction for the stream buffers we use a *Stride-Filtered Markov* (SFM) predictor. The predictor has a two-delta stride table in front of a Markov prediction table, as shown in Figure IV.3. In the write-back stage, the load instruction is checked to see if it hit or missed in the L1 data cache. The prediction table is only updated on a miss (i.e., we are predicting the miss stream). In addition, our implementation does not update the predictor with loads that receive their value forwarded from stores, since we found little benefit from prefetching these loads.

In the write-back stage, the load-PC (for a missed load) is used to index into the stride table. The stride table stores (1) the last address for the load, (2) the last stride for the load, (3) the 2-delta stride, and (4) some confidence information. To facilitate confidence-guided allocation filtering (see Section IV.A.3), we use a saturating counter which reflects the predictability of the miss stream generated by that load instruction. Furthermore, we have a separate confidence counter for both the stride predictor and the Markov predictor. These counters are used in determining which predictor to use when making a prediction. If the stride calculated by (current miss address - last address) does not match the last

stride or 2-delta stride, then the Markov table is updated noting the transition from last address to current address. The last address is stored as the tag, and the current address as the data entry. Accordingly, when that same last address is seen again, it will get a hit in the Markov table, predicting the next miss address not captured by the stride predictor.

When a stream buffer is allocated, the 2-delta stride table is accessed, and a stride is assigned to the stream buffer. If there is no 2-delta stride in the table, then a next line stride is assigned by default. We select between a stride prediction and a Markov prediction based on the values of the corresponding confidence counters stored in the stride predictor.

The stride assigned to the stream buffer is used to generate the predicted address if the Markov table misses during the prediction generation. Otherwise if there is a Markov table hit, then the Markov address is used for the prediction.

For the SFM predictor examined in this chapter, we only use the current address to index into the Markov part of the table, in other words we present results from a first order Markov predictor. We examined using higher order Markov predictors as in [33], but found that it provided little improvement, confirming their results. The only additional information we copy into the stream buffer from the predictor is some confidence information, to guide priority scheduling and prediction as described below.

Differential Markov Predictor

In order to reduce the size of the Markov predictor table we store into the table *only the difference (at the cache block granularity) between consecutive cache miss addresses*, rather than the full address as is done in prior work. To calculate the address to prefetch, a stream buffer adds its last missing address to the signed offset contained in the table. The table is still indexed by the last

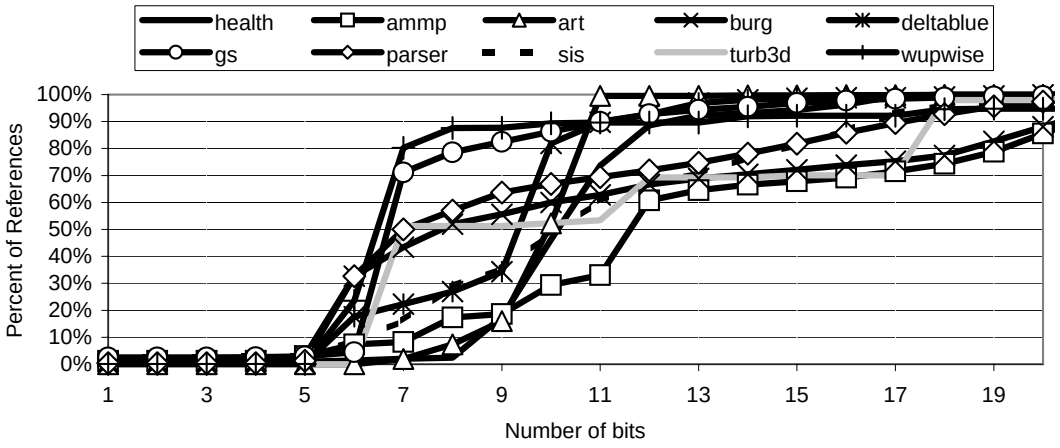

Figure IV.4: The number of bits to accurately predict cache misses using the Markov Difference Predictor. The y-axis shows the percent of L1 cache misses that could be correctly predicted given the number of bits used for each entry of the Markov table shown on the x-axis. The cache miss address is predicted by adding together the address used to index the Markov table with the value stored in the Markov table.

miss as in the standard Markov table. Figure IV.4 shows how many bits are needed to represent the address difference for all of the miss transitions found in the Markov table. The results show that having 20 bits captures almost all of the transitions. This number could perhaps be further reduced by smart heap memory allocation which could place objects with high temporal locality close to one another. In addition, the tag size can also be reduced by storing only partial address tags. In this chapter we use a Markov table with 2K entries, which uses a total of 5 KBytes for the data storage.

IV.A.3 Allocation Filtering

Stream buffer allocation is one of the most important parts of a stream buffer architecture. Since there are only a small number of stream buffers, there is high contention, as every data cache miss could potentially allocate a stream

buffer.

Farkas et al. [26] showed that using *two miss stride filtering* provides good results for a PC-based stream buffer. Two miss filtering only allocates a stream buffer for a load when it misses 2 times in a row, and the last two strides are identical. For our predictor-directed stream buffers we examine two methods for filtering allocation – a general form of two miss filtering, and using our new prediction confidence to guide allocation.

When updating the SFM predictor for a load that misses in the cache, both the PC-based stride table and the address based Markov table are indexed, and potentially updated. Our two-miss allocation filter allows a load to allocate a stream buffer when the load has two cache misses in a row, and both times the load would have been correctly predicted using either the stride predictor or the Markov predictor. If this occurs, then it allocates a stream buffer. This modified scheme is our two-miss allocation filter.

The second heuristic we examine uses address prediction confidence to guide stream buffer allocation. Each entry in the PC-based table stores an *accuracy counter*, which is incremented every time the load's update address matches the prediction of the stride or Markov table, and decremented when it does not match. The saturating counter reflects the ability of the predictor being able to predict the load's misses. By separating the confidence counters from the stream buffer we can gauge how well a particular load's miss stream can be predicted before we allocate a stream buffer to it. This is used to avoid stream thrashing. When a stream buffer is allocated, it copies the accuracy confidence counter into a *priority counter* in the stream buffer. Maintaining the priority counter is described in more detail in the next section.

On a cache miss, the accuracy confidence counter in the prediction table guides stream buffer allocation. If the address prediction confidence level (accu-

racy counter) of the load is above an allocation threshold, it is allowed to contend for a stream buffer. We performed several experiments with different threshold values and our results suggest that a threshold value of 1 is appropriate for our benchmark suite. In addition, a load is only allocated a stream buffer if there is at least one stream buffer whose *priority* confidence counter is less or equal to the *accuracy* confidence counter of the load. If the load's accuracy confidence is lower than all of the stream buffers priority confidence, then a stream buffer will not be allocated for it.

IV.A.4 Prediction Confidence

When an address is touched by both a stride-predictable load instruction and one that is not, we need a mechanism to choose the correct predictor to predict the next address. To facilitate this, each load entry in the stride predictor has a saturating confidence counter for each of the predictors. When we update the predictor in write-back stage, we generate a stride prediction by adding the stride and the last address information in the stride predictor. We also generate a Markov prediction by indexing into the Markov table with the last address. We then compare the current miss address to the stride and Markov predictions. The counter is decremented if its corresponding prediction is wrong. Otherwise, it is incremented.

When a stream buffer is allocated, this confidence information is copied into the stream buffer. The stream buffer uses these counter values to choose between a stride and a Markov prediction. When the stride prediction confidence counter is higher, a stride prediction is generated. In contrast, if the Markov prediction counter is higher, we use the Markov prediction if there is a Markov table hit, otherwise we make a stride prediction. In case of a tie, the Markov table is given priority. We found this choosing mechanism to provide better

results than giving static priority to either predictor.

IV.A.5 Stream Buffer Priority

The predictor and bus create a resource constraint, since there are potentially several stream buffers which have empty entries, or have predicted addresses waiting to be prefetched. We examine two approaches for determining which stream buffer should get access to the predictor and L1-L2 bus each cycle.

The first heuristic is *Round-Robin*, giving each buffer an equal chance at performing a prediction or prefetch. A pointer is kept to the last stream buffer to perform a prediction and another pointer for the last entry to issue a prefetch. The stream buffers are then sequentially examined in round-robin order, looking for a buffer with an entry in need of prediction or a predicted entry ready to be prefetched.

The second heuristic uses *Priority Counters* to guide which stream buffer gets to perform the next prediction or prefetch. Every time there is a lookup and the stream buffer gets a hit, the priority counter is incremented by a constant value (2 in our implementation). To enable the reuse of stream buffers that had high confidence but outlived their usefulness, after several allocation requests (i.e. data cache misses that also miss in stream buffers) we decrement each stream buffer's priority counter by a value of 1. We found using 10 L1 data cache misses as our aging period provided decent results. Intuitively, this policy tries to deallocate stream buffers that do not incur frequent hits. If a stream buffer can eliminate 5% or more of the misses then the priority counter value will not be affected by aging because on average it will get a hit during a 20 miss period. When determining which stream buffer gets to use the predictor or perform a prefetch, the stream buffers are examined in the order from highest priority to lowest based on their priority counter. If there are several stream buffers that

are at the same confidence level, we use an LRU policy to choose the winner.

As described earlier in Section IV.A.3, the priority counter is also used to guide stream buffer allocation along with the accuracy counters. A stream buffer will only be re-allocated for a data cache miss if the load's prediction accuracy confidence is greater than or equal to a stream buffer's priority counter. Therefore, stream buffers that are performing useful prefetches will stay allocated and have a longer lifetime. When a stream buffer is allocated, the accuracy confidence is copied into the stream buffer's priority counter.

IV.A.6 TLB Translation and Prefetching

As we store the virtual effective address of a load in our predictor, we need to translate this to a physical address before we access memory. For this study, we assumed a multi-ported TLB that can support two demand accesses and one prefetch access per cycle. On a prefetch, we access the data TLB for the translation and perform a replacement if necessary. In essence, this amounts to TLB prefetching [59]. However, we did not observe any benefits or performance losses caused by this approach, as the benchmarks we have used had only a small number of TLB misses (except for `ammp`). As an optimization, the TLB translations could potentially be stored with each stream buffer when the stream buffer is allocated. Then a TLB lookup would only need to be performed when the next virtual prefetch address goes outside the current page boundary.

IV.B Methodology

The simulator used in this study was derived from the SimpleScalar/Alpha 3.0 tool set [9], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user-level instructions, performing a detailed timing simulation of an aggressive 8-way dynamically scheduled micro-

Program	Description
health	A hierarchical health-care system simulator taken from the Olden Benchmark suite (input: 3 500).
ammp	This benchmark solves the ODE defined by Newton's equations for the motions of the atoms in the system on a protein-inhibitor complex which is embedded in water (input: ref).
art	The Adaptive Resonance Theory 2 neural network is used to recognize objects in a thermal image. The objects are a helicopter and an airplane. The neural network is first trained on the objects. After training is complete, the learned images are found in the scanfield image (input: ref).
burg	A program that generates a fast tree parser using BURS technology. It is commonly used to construct optimal instruction selectors for use in compiler code generation. The input used was a grammar that scribes the VAX instruction architecture.
deltablue	A constraint solution system which is implemented in C++, with an abundance of short lived heap objects (input: long).
gs	Ghostscript is an implementation of Adobe Systems' PostScript (tm) language. The input run converts a PostScript file into a jpeg.
parser	The Link Grammar Parser is a syntactic parser of English, based on link grammar, an original theory of English syntax. Given a sentence, the system assigns to it a syntactic structure, which consists of set of labeled links connecting pairs of words (input : ref).
sis	Synthesis of synchronous and asynchronous circuits (input: simplify). It includes a number of capabilities such as state minimization and optimization. The program has approximately 172,000 lines of source code and a good deal of pointer arithmetic (input: markex).
turb3d	Simulates isotropic, homogeneous turbulence in a cube with periodic boundary conditions in x,y,z coordinate directions (input: ref).
wupwise	Wupwise is an acronym for Wuppertal Wilson Fermion Solver, a program in the area of lattice gauge theory involving quantum chromodynamics (input: ref).

Table IV.1: Description of benchmarks used.

processor with two levels of instruction and data cache. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch mis-prediction.

To perform our evaluation, we collected results for the programs shown in Table IV.1. The programs were compiled on a DEC Alpha AXP-21164 processor using the DEC FORTRAN, C and C++ compilers under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`). Table IV.2 shows the number of instructions simulated, L1 data cache miss rate, percent of executed instructions that were loads and stores, the IPC for each program, and the percent of cycles the bus from the L1 to L2, and the bus from the L2 to main memory were busy (occupied). `Turb3d` was fast forwarded 1.3 billion instructions [63] before gathering statistics. `Ammp, art, parser` and `wupwise` were also fast forwarded by 2, 2.9, 10.7 and 2.5 billion instructions respectively. These fast forward numbers were derived using the SimPoint tool [64]. Results for `health` are provided for comparison purposes as this is a popular benchmark in prefetching studies. It is not intended to reflect the true merits of our prefetching architecture. More detailed analysis of the pointer behavior (pointer variability and fan-out) for these programs can be found in [57].

IV.B.1 Baseline Architecture

Our baseline simulation configuration models a next generation out-of-order processor microarchitecture. We've selected the parameters to capture underlying trends in microarchitecture design. The processor has a large window of execution; it can fetch up to 8 instructions per cycle. It has a 128 entry re-order buffer with a 64 entry load/store buffer. To compensate for the added complexity of disambiguating loads and stores in a large execution window, we increased the store forward latency to 2 cycles.

program	#inst (Million)	%L1 Miss Rate	%loads	%stores	IPC	L1-L2 %bus utilization	L2-Mem %bus utilization
health	11	50.72	36	14.2	0.76	37.4	0.5
ammp	300	26.69	27.6	4.5	0.19	3.6	11.4
art	300	57.27	30.2	7.6	0.41	23.9	60.3
burg	300	18.60	19.1	18.7	1.97	17.4	4.9
deltablue	96	35.10	28.9	9.9	1.47	38.3	4.1
gs	300	3.42	19.2	6.8	3.43	4.2	0.9
parser	300	5.63	27.7	8.8	1.48	7	4.1
sis	300	3.99	28.7	36.8	10.5	4.8	0.6
turb3d	300	9.23	23.3	16.2	2.58	24.6	13.9
wupwise	300	4.26	22.2	7.5	2.42	5.7	9.9

Table IV.2: Baseline results showing the number of instructions simulated, L1 data cache miss rate, percent of executed instructions that were loads and stores, the IPC for each program, and the percent of cycles the bus was busy from the L1 to L2, and the bus from L2 to main memory was busy.

To make sure that the prefetching speedups we report are from actual prefetching benefit and not from compensating for a conservative memory disambiguation policy, we implemented perfect store sets [20]. Perfect store sets cause loads to only be dependent on stores that they are actually dependent upon. In this way loads will not be held up by false dependencies making the prefetcher look better. All the architectures simulated in this study utilize perfect store sets.

In the baseline architecture, there is an 8 cycle minimum branch misprediction penalty. The processor has 8 integer ALU units, 4-load/store units, 2-FP adders, 2-integer MULT/DIV, and 2-FP MULT/DIV. The latencies are: ALU 1 cycle, MULT 3 cycles, Integer DIV 12 cycles, FP Adder 2 cycles, FP Mult 4 cycles, and FP DIV 12 cycles. All functional units, except the divide units, are fully pipelined. We use a McFarling gshare predictor [45] to drive our fetch unit.

Two predictions can be made per cycle with up to 8 instructions fetched.

We rewrote the memory hierarchy in SimpleScalar to better model bus occupancy, bandwidth, and pipelining of the second level cache and main memory. For the majority of our results, the L1 instruction cache is a 32K 2-way associative cache with 32-byte lines. The baseline results are run with a 32k 4-way associative data cache with 32-byte lines. A 1 Megabyte unified L2 cache is simulated with 4-way associativity and 64-byte lines. The L1 cache has a 3 cycle hit latency while stream buffer hits incur a 1 cycle latency. The L2 cache has a latency of 12 cycles, and is pipelined three accesses deep. The main memory has an access time of 120 cycles. The L1 to L2 bus can support up to 8 bytes per processor cycle whereas the L2 to memory bus can support 4 bytes per cycle.

IV.C Prefetching Performance

This section compares predictor-directed stream buffers to the best performing prior stream buffer approach. This is the PC-based stride stream buffers of Farkas et al. [26], which was described in Chapter III. We call their approach *PC-Stride*. Loads that miss in the data cache are kept track of in a 256 entry 4-way associative stride address prediction table. On a miss, the predicted stride is copied into the stream buffer to guide the predictions. We examined using PC-stride tables larger than 256 entries, but they provided little to no improvement.

For our PSB architecture, we also use a 256 entry 4-way PC-stride address prediction table to filter stride predictions out of a 2K entry Markov table. We use a differential Markov table as described in Section IV.A.2, where each entry in the Markov table is only 20-bits (total table size of 5 Kbytes). The advantage of PSB over PC-Stride is that we can accurately follow non-stride based miss patterns. For the counter stored in our stride table, which we use to estimate accuracy in the allocation policy, we used a saturating value of 7.

For the priority counters in the stream buffers, we used a saturating value of 12. Table IV.3 summarizes the different simulation parameters and their respective values.

For both the PC-Stride and the PSB architectures we used 8 stream buffers, each with 4 entries unless otherwise noted. All stream buffers are checked in parallel on a lookup. In addition, when a stream buffer generates a prediction, all stream buffers are checked to guarantee that the stream buffers do not follow overlapping streams.

There are two possible address types we could choose to predict, virtual or physical. If virtual addresses are used, these will have to pass through the TLB before being prefetched from memory. On the other hand, if physical addresses are used, the effectiveness of the predictors will be greatly diminished when address traces cross page boundaries. We chose to predict virtual addresses and translate the prefetches with the TLB. In this study we assumed an additional port on the TLB to handle the prefetch requests, and there can be at most one prefetch per cycle, unless otherwise noted. If the predictions are accurate, these prefetch requests will actually generate the equivalent of a TLB prefetch [59]. Therefore, using effective cache prefetching performs the function of a dedicated TLB prefetcher. However, the benchmarks we examine do not have high enough TLB miss rates to see much benefit from the TLB prefetching.

To reduce the size of the tables used in the prediction phase of prefetch, we employ three small optimizations. First, in order to reduce the size of the stride PC-table, we only perform inserts when there has been a cache miss. In this way we can reduce the PC-table size down to 256 entries while still capturing all of the critical loads that we may come across. In addition to that, to reduce the size of the stride and the Markov tables, the tables *only store cache block address*, instead of the full address. Just using the cache block aligned address instead of

parameter	value
L1 data cache	32K, 4-way, 32B blocks
L1 instruction cache	32K, 2-way, 32B blocks
Unified L2 cache	1M, 4-way, 64B blocks
L1-L2 bus bandwidth	8 bytes/cycle
L2-Memory bus bandwidth	4 bytes/cycle
L1 hit latency	3 cycles
Load-store forward latency	2 cycles
L2 hit latency	12 cycles
Memory access latency	120 cycles
TLB miss latency	30 cycles
Number of stream buffers	8
Number of stream buffer entries	4
Stream buffer hit latency	1 cycle
Stride predictor size	256 entries, 4-way set associative
Markov predictor size	2048 entries
Markov predictor entry size	20 bits
Predictor access latency	1 cycle
Accuracy counter saturation	7
Priority counter saturation	12
Allocation threshold	1
Accuracy counter increment	1
Accuracy counter decrement	1
Priority counter increment	2
Priority counter decrement	1
Priority counter aging period	10 L1 misses

Table IV.3: Summary of simulation parameters. The top parameters summarize the baseline architecture. The next group lists the additional parameters for the PC-Stride stream buffer architecture. The final group describes the additional parameters on top of that needed for the Predictor-Directed Stream Buffer prefetching architecture.

the full byte address, reduces each entry by 5 bits while still being sufficient to perform any prefetch. Finally, to reduce the size of the Markov predictor table we only store *the difference* (at the cache block granularity) between consecutive cache miss addresses, rather than the full address. For the programs we looked at, our results indicate 20-bits are sufficient to cover almost all of the miss transitions in the Markov predictor.

IV.C.1 Results

To analyze the benefits of our approach, we now show the overall effect that the stream buffers have on the performance of the system. Figure IV.5 shows performance results for the baseline architecture, the best prior approach, PC-Stride, and our new Predictor-Directed Stream Buffers with and without confidence based allocation and priority. PSB results are shown for all four combinations of the allocation filter and priority scheduler. These are (1) two miss allocation filter with round-robing scheduling (2Miss-RR), (2) two miss allocation filter with priority confidence scheduling (2Miss-Pri), (3) confidence allocation with round-robin scheduling (Conf-RR), and (4) confidence allocation with priority scheduling (Conf-Pri). The results were generated for several pointer-based applications, and two stride-based FORTRAN programs. We ran several FORTRAN programs, and they all had similar performance to the results shown for `turb3d` and `wupwise`.

The most notable performance improvements can be seen on `health`, `ammp`, `burg`, and `deltablue`. All of these programs benefit significantly from the addition of a Markov predictor. The speedups over the best previously known stream buffer technique, PC-Stride, range from under 1% for `gs` which has very little pointer behavior, to over 100% for `ammp` which has many misses, most of which go to main memory. `Wupwise` exhibits one shortcoming of the confidence-

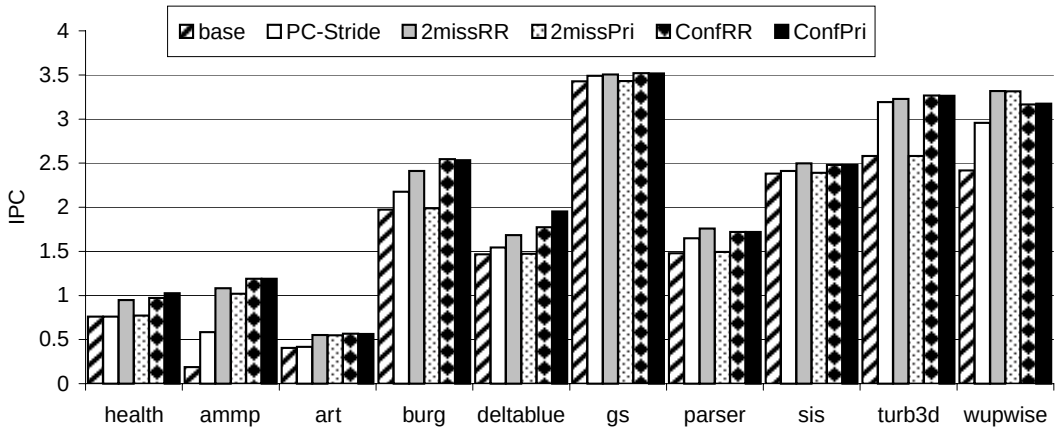

Figure IV.5: IPC performance results of base case, prior PC-Stride prefetching, and our Predictor-Directed Stream Buffers.

based allocation scheme. For this program, the confidence-based configurations allocate stream buffers to references that are L1 misses but are often found in the L2 cache. The configurations based on two miss filters on the other hand allocate buffers for references that are L2 misses as well. Since the two miss filter approach is able to hide longer latencies, even though it hides fewer misses in the big picture, it achieves higher speedup than confidence-based allocation. One remedy for this problem can be to assign higher priority/confidence to stream buffers that prefetch streams that service the slower levels of the memory hierarchy. For most of the programs, confidence based allocation works better. Using confidence based allocation makes sure that stream buffers are only deallocated when they are no longer providing useful prefetches.

Accuracy and Coverage

In understanding how the performance of the system will be affected by stream buffers it is important to verify that the optimizations do not detrimentally affect prediction accuracy and prefetch coverage.

To quantify the benefit of using the Markov predictor with our PSB architecture, we measured the accuracy of the address predictors at predicting cache misses. To measure accuracy, we count the number of times the predictor (either PC-Stride or Stride-Filtered Markov) was able to correctly predict the miss address, for all cache misses. When a new miss occurs for the SFM predictor, we generate a prediction from the stride PC and Markov tables, and see if the miss would have been correctly predicted by the Markov prediction if there was a Markov hit, otherwise by the stride predictor. Note, this prediction accuracy is not the accuracy of the stream buffers. It is the accuracy of the address predictors for predicting every cache miss on the non-speculative path of execution. The accuracy of the two-delta stride predictor used for PC-Stride prefetching, and the Stride-Filtered Markov predictor used in our PSB configurations are shown in Figure IV.6. The results show that all but two programs experience a significant increase in miss stream prediction when using the SFM predictor instead of only a PC-Stride predictor.

We now consider the coverage of the prefetches, and how it relates to predictor accuracy. Coverage is defined as the percentage of base case misses that have their latency *reduced* by either completely hiding the memory latency or hiding some part of it via stream buffer prefetching. The bars in Figure IV.7 show coverage results for PC-Stride and the Confidence Priority PSB when using 8 stream buffers, where at most one prefetch can be initiated per cycle. The figure also presents the breakdown of the stream buffer hits that completely hide the memory latency and those that only partially hide the cache miss latency. The "X" results show the prefetch coverage for the Confidence Priority scheme with 128 stream buffers (each with 4 entries), where up to 4 predictions and prefetches can be initiated per cycle. The results indicate that the Predictor-Directed Stream Buffers increase the useful prefetches by augmenting the PC-

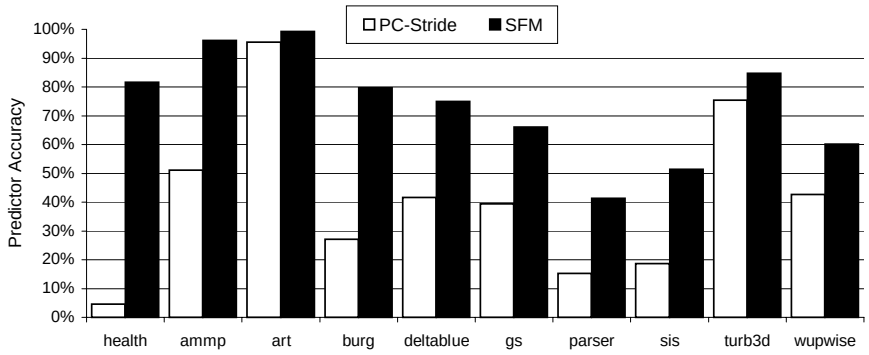

Figure IV.6: Predictor accuracy. This is the number of potential correct address predictions for all of the data cache misses for the PC-Stride predictor and the Stride-Filtered Markov predictor.

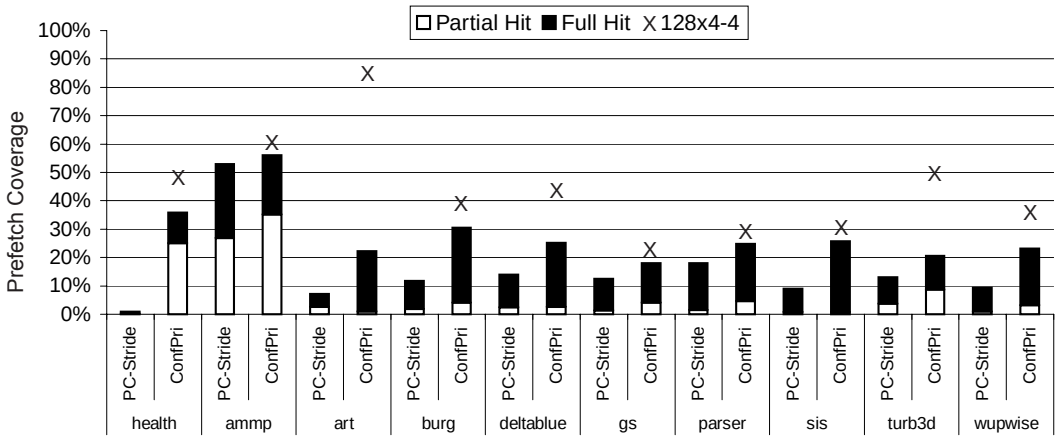

Figure IV.7: Prefetch coverage breakdown. This is the number of stream buffer hits divided by the number of base case L1 misses. It represents the number of cache misses we are able to cover/remove with stream buffers compared to an architecture with no prefetching. The PC-Stride and Confidence Priority results are for the baseline prefetching architecture with 8 stream buffers and only one prefetch per cycle. For these configurations, we show the breakdown of the stream buffer hits that completely hide the memory latency and those that only partially hide the cache miss latency. The "X" results show the prefetch coverage for the Confidence Priority scheme with 128 stream buffers (each with 4 entries), where up to 4 predictions and prefetches can be initiated per cycle.

Stride scheme with prefetches it could not have captured before.

When comparing predictor accuracy in Figure IV.6 with prefetching coverage Figure IV.7, we can see that the coverage is not as high as it could be. Note that Figure IV.6 is an upper bound for the coverage PSB could achieve, since the accuracy results are for updating the predictor after each prediction on the non-speculative path. There are two main reasons for the low coverage. The first reason is due to stream buffer contention amongst these high confidence miss streams preventing the coverage of some of the predictable streams. In Figure IV.7, we show an "X" for a special configuration of Confidence Priority PSB with 128 stream buffers, each with 4 entries, where we can perform up to 4 predictions and 4 prefetches per cycle. The IPC performance results for this configuration are discussed in Section IV.C.2. When removing the stream buffer contention and increasing prefetch bandwidth, coverage increases significantly for programs like `health`, `art`, `deltablue` and `turb3d`. In fact, the coverage for `art` increases to 85% of all cache misses. The second reason for the lower than expected coverage is that a stream buffer follows only an address prediction stream, and it is decoupled from the execution stream. Once a stream buffer is allocated, it builds each prediction upon a prior prediction, so the probability of each subsequent prediction being correct will decrease. In addition, we've seen that pointer transitions guarded by conditional branches end up choosing the incorrect Markov predictor entry for some predictions, because no branch information is available to guide the speculative prediction stream. Therefore, incorporating branch history information into the predictor may be able to improve the stream buffer prefetch coverage.

Timeliness of Prefetches

The third criteria for a prefetch architecture is that it provides its results in a timely manner. Figure IV.7, along with Figures IV.8, IV.9, and IV.10 quantify the timeliness of the prefetches.

Because loads are scheduled optimistically assuming an L1 hit latency, it is important to measure the number of loads that now have 1 or more stall cycles (beyond a full L1 hit or full stream buffer hit), which is shown in Figure IV.8. This figure shows the percent of loads that stall due to either a complete miss in the cache and stream buffer or stall due to a partial hit. The percentage of loads that encounter a stall is significantly reduced with PSB for all programs except `ammp`. For `ammp` the total number of stalling loads is increased by around 30% over PC-Stride. Even so, the performance for `ammp` is better for PSB than PC-Stride.

`Ammp` has two important loads dependent upon each other in a loop, which our Confidence Priority scheduling scheme accurately focuses on. The round-robin scheduling used by PC-Stride causes delay in initiating the prefetches for these important loads. For one of these loads, the prefetch always occurs late resulting in the execution stream waiting on a high latency stall (comparable to memory access latency) for that partial hit. This allows the second load to be prefetched with its prefetch latency being completely hidden, since it is overlapped with the prior prefetch. This pattern repeats during this loop's execution resulting in PC-Stride having fewer partial hits, but its partial hits have a long latency almost as high as a main memory access. The Confidence Priority architecture on the other hand, focuses the prefetch resources on these two loads. This results in shorter stalls for both loads. This allows the execution stream to keep up with the prefetch stream of every load, making it hard to completely hide the prefetch latency for these two loads. This results in more partial hits

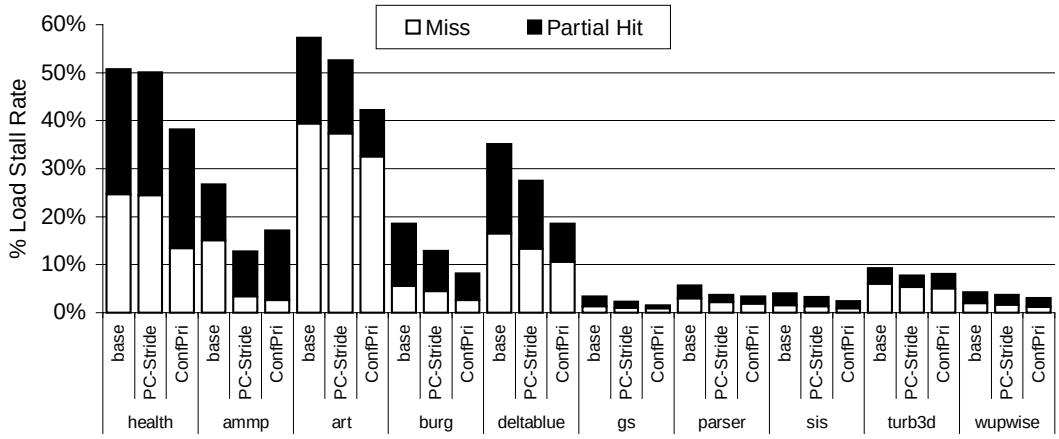

Figure IV.8: Load Stall Rate. This is the percent of all dynamic loads that stall for one or more cycles when considering the different stream buffer architectures. These are references that are not full cache hits and are not full stream buffer hits. The percent of load stalls are broken down into those that are complete misses and those that are partial hits.

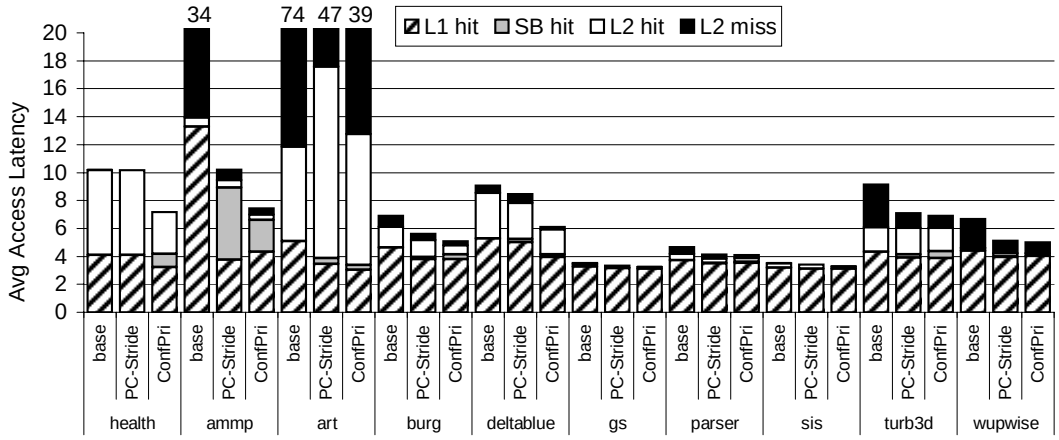

Figure IV.9: Average latency of a load in cycles for the different architectures broken down by where the time is spent. The L1 access time is 3 cycles. Partial hits add to the access time of the level that they hit in.

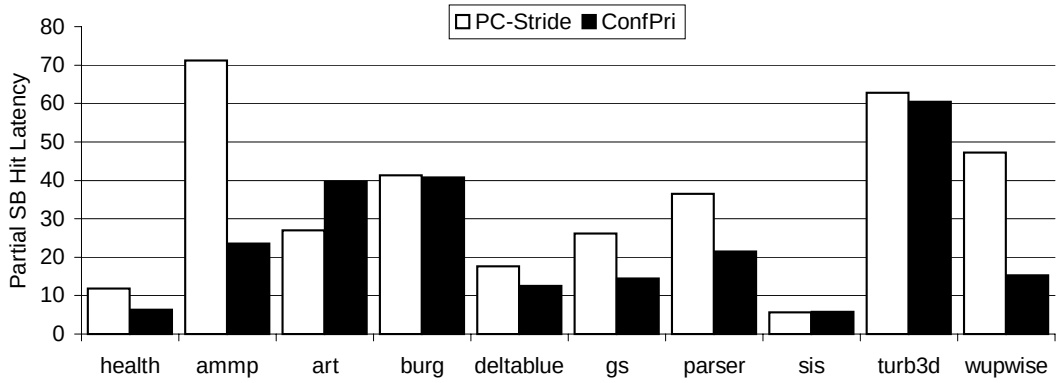

Figure IV.10: Average partial stream buffer hit latency. This is average latency of stream buffer tag hits that still have the prefetch outstanding. These hits therefore do not completely hide the memory latency.

for Confidence Priority as Figure IV.8 shows, but the latency of each partial hit is significantly smaller than PC-Stride as seen in Figure IV.10. This results in overall better performance for Confidence Priority over PC-Stride.

Figure IV.9 shows the average load latency for the different benchmarks and techniques, showing where the time is being spent for the percent of loads that stall from Figure IV.8. The L1 cache hit latency is 3, anything above this is a cycle spent waiting by a load for its result. Since some L1 hits are only partial hits (i.e. the tag hits but data is not ready), average L1 hit latency can actually be higher than 3 cycles. We break down the latency by where it is waiting for memory from L1 (L1-hit), L2 (L2-hit), Memory (L2-Miss), or a Stream Buffer (SB hit). Cycles for partial hits are charged to the structure that the reference is due to arrive at. If the prefetches we made were not timely, the cycles spent waiting for each load would show up as waiting for stream buffers. The results show that after applying the optimizations there is only one program, `ammp`, that spends a significant number of cycles waiting on the stream buffer partial-hits. `Ammp` has a significant number of prefetch hits that need to go all the way out to

main memory to retrieve their blocks, and as described above the prefetches are not occurring early enough to cover the full latency.

Provided earlier in Figure IV.7 is the percent of all stream buffer prefetches that successfully cover all of the latency associated with that load (full hits), versus the fraction that only cover some of the latency (partial hits). According to Figure IV.7, `health` and `ammp` incurred a significant number of partial stream buffer hits. To discover the effect this has on performance, we analyzed average partial stream buffer hit latencies in Figure IV.10. We found that with `health` there is very little residual latency after prefetching. Even though the prefetcher cannot cover the full memory latency, it is able to cover most of it. Hence the partial hit latency has very little impact on the performance. As mentioned above, the only benchmark with significant partial hit latencies is `ammp`. Still, in comparison to PC-Stride, the prefetches are significantly more timely for `ammp` due to the priority scheduling.

Memory Controller and Bus Utilization

Figure IV.11 shows the percent of bus utilization for both the bus from the L1 to the L2, and the bus from the L2 to main memory. The total amount of L1 bus utilization is only increased significantly for two programs with our technique, `health` and `ammp`. However, we saw in Figure IV.5, these bus utilization increases are more than justified for the amount of speedup achieved. In fact, for three of the programs, `art`, `turb3d`, and `wupwise`, our combination of techniques simultaneously achieves *both* speedup and reduces bus utilization over PC-Stride techniques.

Figure IV.12 shows the percent of useless prefetches for the different configurations examined. These are the prefetches that never end up being used by the processor. This graph shows that in some cases the accuracy of the new

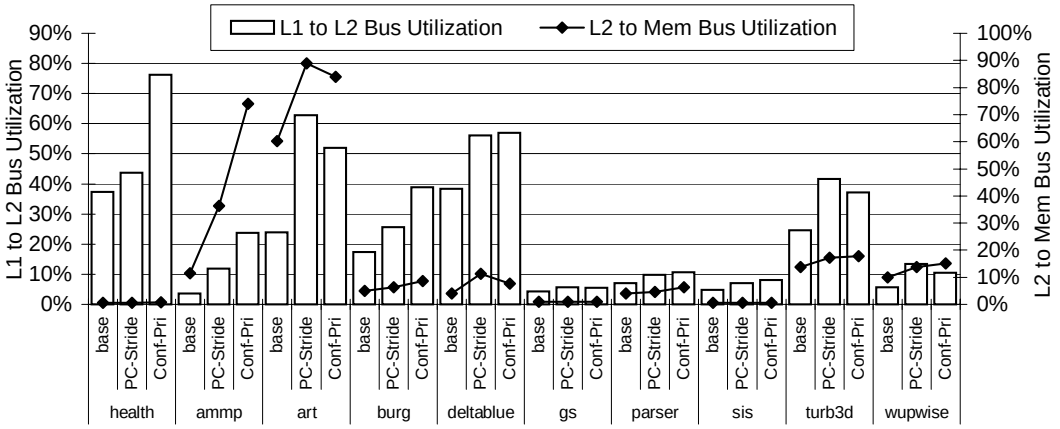

Figure IV.11: The percent of cycles the L1-L2 bus and the L2-Mem bus were busy.

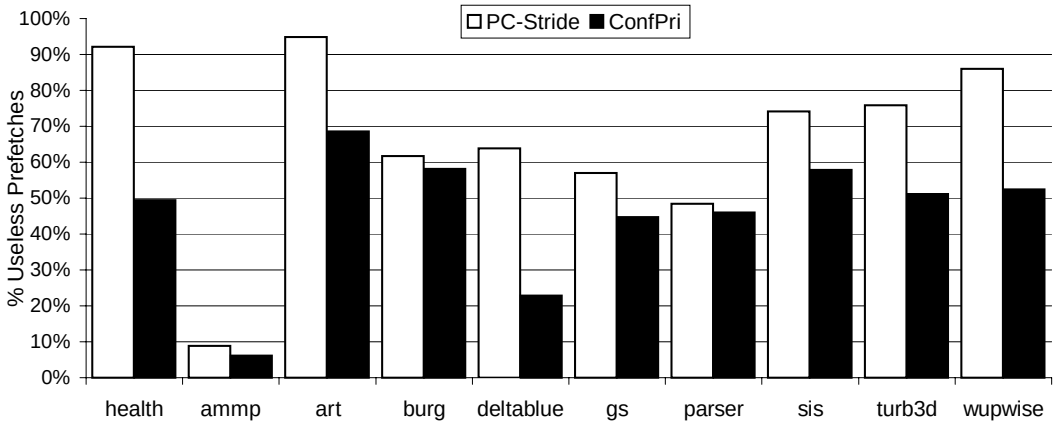

Figure IV.12: Wasted prefetches. This is the number of prefetches not used by the processor divided by the number of prefetches made.

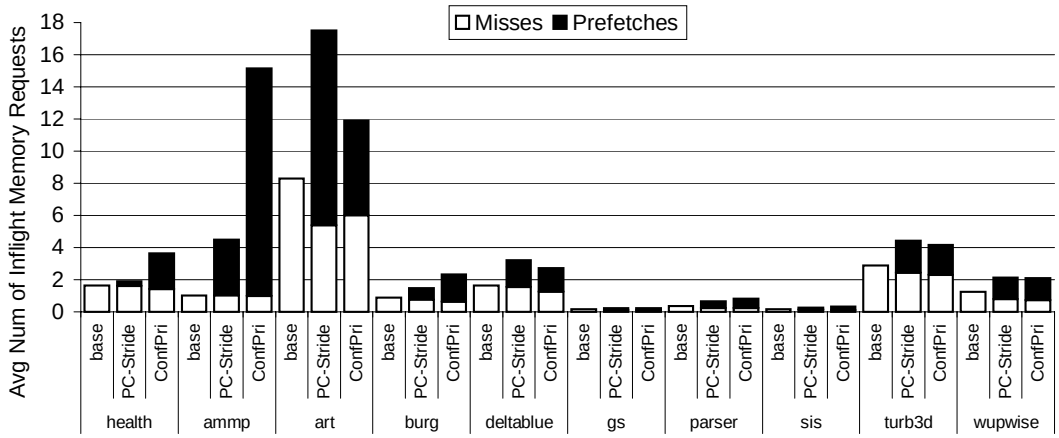

Figure IV.13: Average number of in-flight memory requests per cycle.

prefetcher more than doubles that of PC-Stride. This can be attributed to using the confidence counters to guide allocation. This allows stream buffer allocation to concentrate on highly predictable loads, and avoids replacing stream buffers that are receiving a lot of hits. Stream thrashing is a serious problem for programs with large amounts of missing loads as is the case in both large programs and tight inner loops, which are highly software pipelined. We found performing loop unrolling and software pipelining increases the number of load instructions in the program, which can degrade the performance of stream buffers. If an architecture has stream buffers, a loop with a hardware predictable reference stream may achieve better performance with limited loop unrolling, and instead use the stream buffers to hide the load latency. As expected, in cases where the Confidence Priority scheme is significantly better than the PC-Stride scheme in terms of wasted prefetches, such as `art`, `deltablue`, `turb3d` and `wupwise`, the bus is utilized at the same or a higher level of efficiency.

In order to gain more insight into the pressure placed on the memory subsystem (i.e. bus, caches, MSHRs etc.), we looked at the number of in-flight memory requests at each clock cycle. Figure IV.13 shows the average number of

memory requests divided into demand cache misses and prefetches. The maximum number of in-flight L1 misses range from 6 for `sis` to 24 for `art`. The maximum number of prefetches for PC-Stride vary between 13 for `sis` and 32 for `art`. For our Confidence Priority configuration, these numbers fall between 14 and 32 for the same programs. For PC-Stride, at any given cycle the maximum combined number of outstanding prefetch and demand misses deviates between 16 for `sis` and 60 for `art`. For the Confidence Priority, the maximum number of combined in-flight memory requests is 14 for `sis` at one end of the spectrum and at 68 for `turb3d` at the other end. `Art` exhibits an interesting behavior where the average number of L1 misses increases with the Confidence Priority scheme as opposed to the PC-Stride technique. This can be attributed to the significant speedup (40%) achieved over the PC-Stride technique. The shorter execution time causes the misses to cluster together and increases the average number of L1 misses. On average, except for `art`, `deltablue` and `turb3d`, programs have more in-flight prefetches with Confidence Priority over PC-Stride due to the clustering effect of shorter execution time and higher prefetch accuracy. As the stream buffer entries are used, they become available for more prefetches. Consequently, higher stream buffer hit rates result in higher in-flight prefetch requests.

IV.C.2 Sensitivity of Results

We evaluated the performance of our best performing technique across a range of stream buffer configurations. These results are summarized in Figure IV.14. The first bar in each group corresponds to the baseline Confidence Priority prefetching architecture with 8 stream buffers each having 4 entries (sb8x4). This architecture can issue up to 1 prediction and 1 prefetch per cycle. Results are shown varying the number of stream buffers (8, 32 and 128). The last two bars

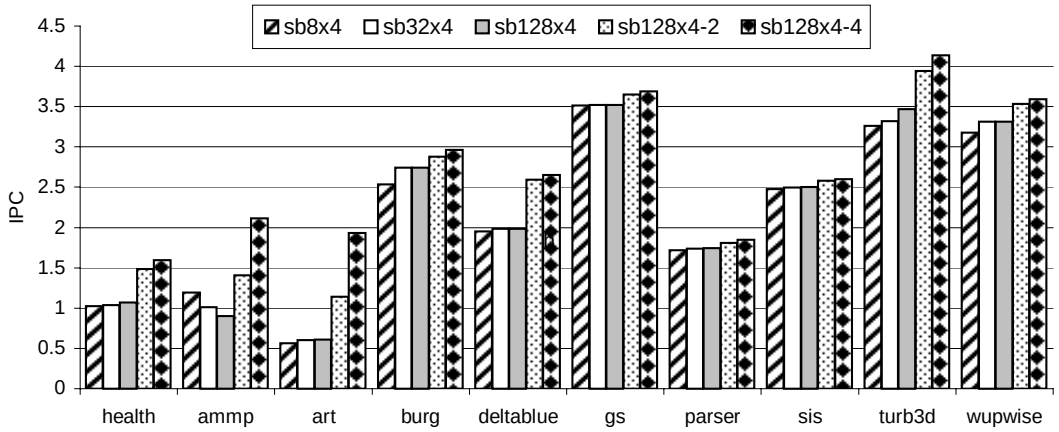

Figure IV.14: Performance results showing different stream buffer configurations using our confidence allocation and priority scheduling technique. The first bar in each group is the baseline prefetching architecture with 8 stream buffers each having 4 entries, and it can issue up to 1 prediction and 1 prefetch per cycle. For each bar titled sbMxN[-P], M indicates the number of stream buffers and N denotes the number of entries in each stream buffer. P implies the maximum number of predictions and prefetches allowed per cycle. If not shown, P is assumed to be 1.

show increasing the maximum number of potential predictions and prefetches allowed per cycle to 2 and then 4 for the 128 stream buffer configuration. For these last two bars, the L1-L2 and L2-Memory bus bandwidths are adjusted by 2 or 4 times with the prefetch capabilities to expose more available performance. Half of the programs show considerable speedups with increased prefetch bandwidth. As mentioned in Section IV.C.1, programs with highly predictable streams gain the most benefit from increased prefetch bandwidth.

We also examined varying the number of stream buffer entries from 2 to 16 for the same configurations shown in Figure IV.14, but do not graph their results. We found that only two programs (`turb3d` and `wupwise`) significantly

benefit from the lengthening of stream buffers over the default length of 4. These two programs have access patterns that are easy to predict far in advance (Figure IV.12), combined with still uncovered latency (Figure IV.9), and available L1 bandwidth (Figure IV.11).

The speedup that we are achieving is due to the hiding of latency associated with capacity problems in the L1 cache. This is shown by Figure IV.15, where we look at the performance when increasing the cache size to 64K 4-way and 128K 4-way cache. It can be seen that the speedup obtained over PC-Stride is still seen over a reasonable set of cache configurations for the programs we examined. We also investigated the performance obtained by simply using a larger L1 data cache for the baseline architecture without prefetching to determine if the larger cache can better capture the working set, or if it is better to use the hardware resources for prefetching. Results in Figure IV.16 indicate that in most cases the performance of our PSB architecture with a 32K 4-way data cache is comparable to 512K 4-way cache results. For some of the programs, the working set does not even fit into a 512K cache.

Finally, we performed simulations to measure the sensitivity of our architecture to various predictor access latencies ranging from 1 cycle to 3 cycles. All the results shown in the figures were simulated with a single cycle predictor access. Although not shown here, results for varying predictor access latency indicate that the predictor can take 3 cycles to access without affecting performance, assuming the predictor is pipelined.

IV.D Summary

We chose to focus on stream buffers because of their ability to follow address streams independent of what the fetch stream is doing. Previous stream buffer architectures were limited to streaming only address patterns with a fixed

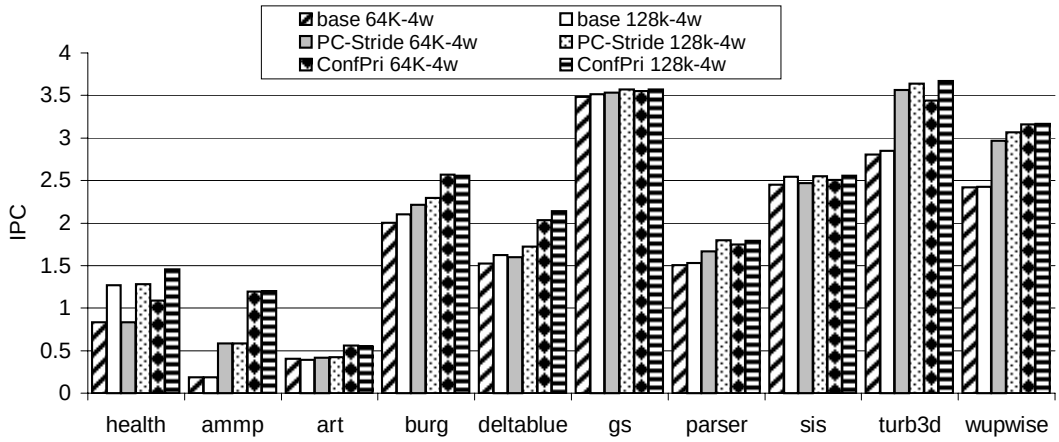

Figure IV.15: Performance results for increased cache size.

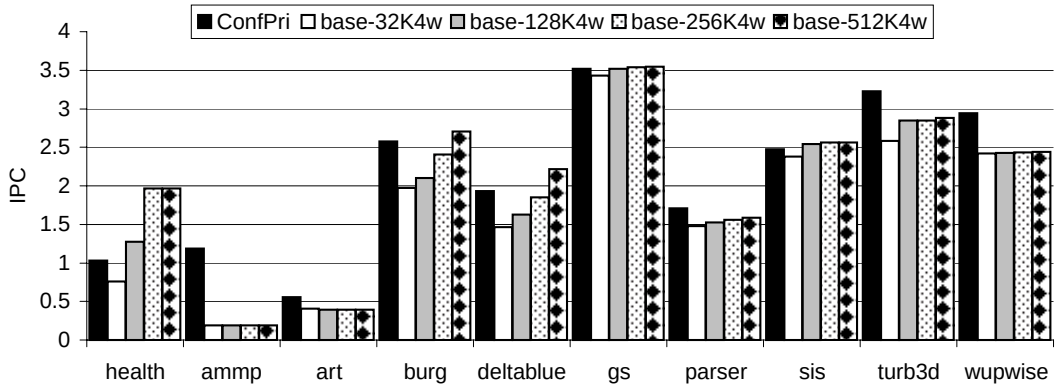

Figure IV.16: Performance results for default Confidence Priority configuration using a 32K 4-way cache in comparison to the baseline (no prefetching) architecture with a 32,128,256, and 512 4-way caches. The results show the trade off between utilizing area for stream buffers versus increasing the size of the L1 data cache.

stride [26], which limits their benefit for commercial pointer-based applications. To go beyond this limit we presented a new stream buffer architecture, Predictor-Directed Stream Buffers, that allow stream prefetching of any sort of predictable address patterns.

While Predictor-Directed Stream Buffers can be used in conjunction with any address prediction scheme, we examine them using a Stride-Filtered Markov predictor. The SFM predictor captures complex sequential, stride, and pointer behaviors as well as tightly integrated combinations of those. We proposed a modified version of the Markov part of this predictor, which we call a Differential Markov predictor, whose table data size was only 5 Kbytes for the results we presented.

We show two problems that arise when allocating resources for Predictor-Directed Stream Buffers, allocation of stream buffers and sharing of memory bandwidth. We present two confidence-based techniques, allocation filtering and priority scheduling, that overcome these problems and allow the stream buffers to perform efficiently. For the applications we examined, Predictor-Directed Stream Buffers provided a 75% speedup on average over no prefetching, and 23% average speedup over the best performing prior stream buffer architecture.

The stride-filtered Markov predictor we use has the potential to achieve a 74% prediction accuracy when predicting the L1 misses. But, the baseline prefetching architecture (8 stream buffers) only covers (prefetches) 28% of the L1 base case misses over the set of benchmarks studied. We showed that if we can increase both the numbers of streams (128 stream buffers) that can be prefetched and the bandwidth (up to 4 prefetches per cycle) then the coverage can be increased to 45% of the cache misses. This aggressive design achieved a 49% speedup over our 8 stream buffer PSB architecture, and resulted in a 178% speedup over the baseline no prefetching architecture. The other reason for the

lower coverage is that the stream buffers are using the predictor speculatively building prediction upon prediction, which will result in lower accuracy. Related to this, we've seen that pointer transitions guarded by conditional branches can end up choosing the incorrect Markov predictor entry for some predictions, because there is no branch information guiding the speculative prediction stream. Therefore, we present a control flow sensitive prefetching technique in Chapter VI.

Considering the fact that 4GHz processors are currently being manufactured and tested, memory latencies are expected to be in the several hundred to a thousand cycles range. Our PSB architecture can successfully hide the memory latency for data that is stored in on-chip caches (L2 and L3), where the miss stream can compactly be represented in a prediction buffer. This fits in well with recently proposed architectures such as Solihin et al.'s [69], where PSB would focus on eliminating latency due to data that exhibits temporal locality within the on-chip caches, and their processor in memory approach would target misses that go off-chip.

Chapter V

Pointer Cache

The difference between the speed of computation and the speed of memory access (the CPU-memory gap) continues to grow. Meanwhile, the working set and complexity of the typical application is also growing rapidly. Thus, despite the growing size of on-chip caches, the performance of many applications is increasingly dependent on the observed latency of the memory subsystem.

As we pointed out in Chapter II pointer chasing loads frequently miss in the on-chip data caches due to the large working set of applications. This causes the program to execute at the speed of accesses to main memory because all the other instructions processing the data in the object depend on the pointer chasing load instruction to make the transition into the new object. To address this, in this chapter we introduce a specialized cache used to aid the handling of pointer loads. The *Pointer Cache* is specifically targeted at speeding up recurrent pointer accesses. The pointer cache only stores pointer transitions (from one heap object to another), effectively providing a compressed representation of the important pointer transitions for the program. We examine using the pointer cache as a prediction table of object pointer values when a load executes. We also use a hit in the pointer cache to initiate a prefetch of the object to provide pointer-based prefetching for the main thread of execution. To keep the pointer cache up to

date in the face of changing data, our design uses *Store Teaching*, to update the pointer transitions when they change.

The pointer cache organizations we examine are not necessarily small (or fast) structures, but are an alternative to overly large on-chip cache structures that provide little marginal performance on many applications. We show that a pointer cache in combination with a smaller traditional cache can be a more effective use of processor transistors.

The rest of the chapter is organized as follows. Section V.A describes the Pointer Cache. Section V.B explores the related work on value prediction while Section V.C focuses on the pointer cache as a value predictor used in accelerating single-thread performance. Simulation methodology and benchmark descriptions can be found in Section V.D. Section V.E presents performance results, and Section V.F concludes.

V.A Pointer Cache

The Pointer Cache holds mappings between heap pointers and the address of the heap object they point to. This structure is organized like a cache, but with only word-length lines since we want to store only the important pointer transitions to maximize the utility of the structure. Since we are only interested in capturing pointer transitions, the pointer cache stores load values *only if the address of the pointer and the address of the object it points to (i.e. the value of the load) fall within the range of the heap.*

The primary function of the pointer cache is to break the serial dependence chains in pointer chasing code V.1. When one load depends on the data loaded by another, a cache miss by the first load forces the second load to stall until the first load completes. When executing a long sequence of such dependent pointer-chasing loads, instructions can only be executed at the speed of the serial

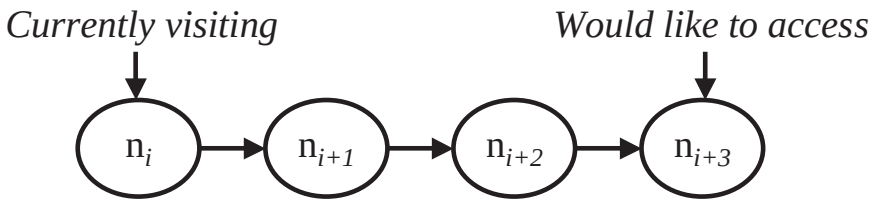

Figure V.1: Because objects are linked together via pointers, we cannot access an arbitrary object among a group of objects. We have to traverse each object in succession to get to the desired node. Since load instructions executed for facilitating object transitions depend on the data loaded by another, a cache miss by the first load forces the second load to stall until the first load completes. When executing a long sequence of such dependent pointer chasing loads, instructions can only be executed at the speed of the serial accesses to memory. Since pointer based applications have big pointer working sets, pointer loads frequently miss all the way to the main memory, and the program can only execute at the speed of main memory accesses.

accesses to memory.

V.A.1 Identifying Pointer Loads

Only pointer loads are candidates to be inserted into the pointer cache. We assume simple hardware support for identifying pointer loads, which are loads that access a heap address, loading a value which is also in the range of the heap. A load is identified as a pointer load if the upper N bits of its *effective address* match the upper N bits of the *value being loaded*. In this study we assume N to be 6 bits. We found that not inserting loads into the pointer cache whose effective addresses point to the stack provided higher performance than including them. Therefore, all load instructions with the stack pointer as a source register are classified as not being pointer loads in our study. This approach for dynamically identifying heap loads was proposed by Cooksey et al. [24]. They found that

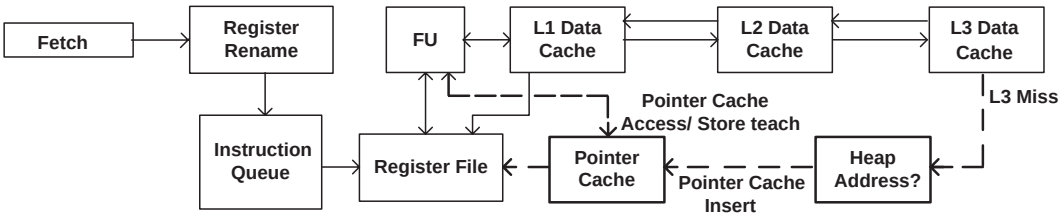

Figure V.2: Pipeline organization of a processor with a pointer cache. The pointer cache is queried in parallel with the L1 cache and returns the address of the object pointed to. This value is consumed by instructions dependent on the load, breaking the serial nature of the memory access. The pointer cache is updated by pointer based accesses (loads or stores) when they commit. When a pointer based load (a heap load which loads a pointer to another heap location) commits, it updates the pointer cache with the load's (address, value) pair. Store Teaching queries the pointer cache on all stores, and, on a hit, updates the relevant pointer cache entry with the store value.

comparing the upper 8 bits was sufficient for their benchmark suite.

V.A.2 Pointer Cache Architecture

Figure V.2 shows a processor organization for including a pointer cache. The pointer cache is queried in parallel with the L1 cache and returns the address of the object pointed to. This value is consumed by instructions dependent on the load, breaking the serial nature of the memory access. Pointer cache entries are optionally tagged with the source memory address they correspond to in order to avoid false hits. In this study, we assume partial tags which essentially amounts to using the pointer cache for value prediction.

Because the pointer cache predicts a full memory address, the returned value can be directly used by instructions dependent on the load. Subsequent loads may result in further accesses to the cache and/or pointer cache, allowing

parallel access to these serial data structures.

The pointer cache is updated by pointer based accesses (loads or stores) when they commit. When a pointer based load (a heap load that loads a pointer to another heap location) commits, it queries the pointer cache with the address it had accessed. If no entry is found for this address, and the load had missed in L3 cache, a new pointer cache entry is allocated for the load. We only install a new pointer entry on L3 misses. This reduces contention for pointer cache entries and ensures maximum benefit is derived from each pointer cache entry. More aggressive or targeted filtering techniques are also possible. If a pointer load hits in any level of data cache, but the loaded value does not match the value stored in the pointer cache entry, then that pointer cache entry is updated.

Existing pointer cache entries must be updated when the program modifies pointer values. Otherwise, the pointer cache will lose effectiveness by returning incorrect information when queried. Such pointer cache misspeculations (in which a pointer cache hit occurs, but an incorrect memory value is returned), are avoided through the use of *Store Teaching* to compensate for the dynamic nature of data structures. This technique queries the pointer cache on all stores, and, on a hit, updates the relevant pointer cache entry with the store value. Store teaching updates occur in the commit stage of the pipeline. If a pointer cache with full tags employs store teaching, and is exposed to the system's cache coherence mechanisms, then values loaded from the pointer cache can safely be used by the processor without special care for misspeculation. However, in this research we assume only partial tags, so the value returned from the pointer cache could be incorrect. Even though we found this situation to be rare, the address returned by the pointer cache must be treated as speculative. Therefore, the load cannot commit until it verifies the predicted value against the actual loaded value. Section V.C describes this situation in more detail.

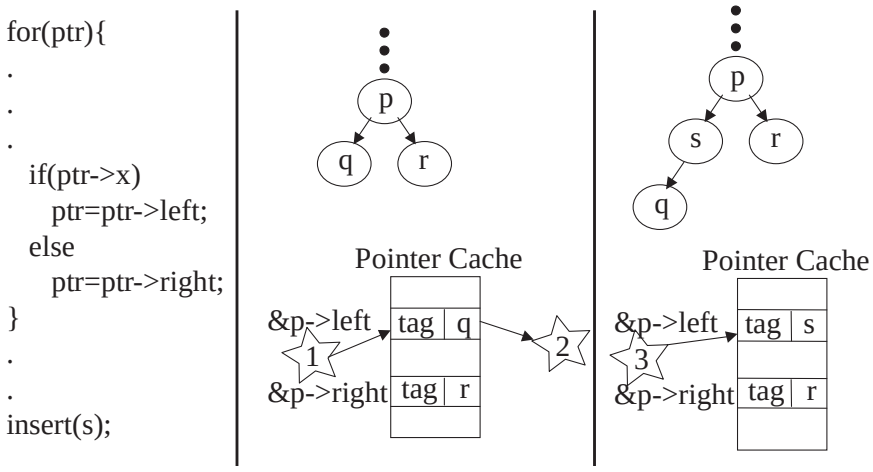

Figure V.3: This figure illustrates the different operations performed on the pointer cache: training (1), prediction (2), and store teaching (3).

V.A.3 An Example Pointer Cache Use

Figure V.3 illustrates the possible operations on a pointer cache. In this example `q` and `r` are the left and right children of `p` respectively. On the first traversal of this data structure, the pointer cache is updated with `q` at `&p->left` when we execute the `ptr=ptr->left` transition (1). Similarly, when we follow the other link we store `r` at `&p->right`. On a subsequent traversal of the data structure, a dereference of `ptr->left` which misses in cache results in a pointer cache hit, returning the address of `q` (2). Finally, when a later modification of the data structure takes place, (in this case, causing `p->left` to point at `s`), the pointer cache detects the update via store teaching, and updates the corresponding pointer cache entry (3).

V.B Value Prediction

The pointer cache is much more space efficient when used as a value predictor (i.e. tagged with partial tags instead of full tags). In this section,

we briefly introduce relevant value prediction work before describing the pointer cache as a value predictor in more detail in the next section.

Value prediction predicts the actual data value that is to be brought in from memory, allowing instructions dependent on the load to speculatively execute with the predicted value. If the prediction is correct, this breaks true data dependencies since the value is produced without having to wait on the load instruction.

When a load is value predicted, the value is used to update the current value and status of the load's destination register. This value will then be seen and used by subsequent instructions. The load still takes its normal path of execution for a non-speculative load — it calculates its effective address, accesses the store buffer and memory. When the load's value becomes available it is checked against the predicted value of the load instruction for miss-speculation. This is called a *check-load*, since the load is used to check the predicted value.

Several architectures have been proposed for value prediction including last value prediction [39, 40], stride prediction [29, 31], context predictors [60], and hybrid approaches [75]. Calder et al. examined the importance of confidence prediction to improve value predictor accuracy [11].

Lee et al. [37] proposed a decoupled value prediction architecture that worked in conjunction with a trace cache. Value prediction occurred in the write-back stage, and predictions were stored in a Value Prediction Buffer accessed in parallel with the trace cache. This allows more flexibility in the design of the value predictor, as predictor access time is no longer on the critical path.

Zhou et al. [79] proposed studying computational locality in the global value history. They describe a prediction scheme, the gDiff predictor, to exploit stride-based global locality. Their experiments show that there exists very strong stride-based locality in global value histories; and, many instructions that are

hard-to-predict using local history-based predictors become highly predictable using global history-based predictors. They also demonstrate the use of the gDiff predictor in exploring the global stride locality in the load address stream. Their results show that the gDiff predictor achieves higher coverage and accuracy in predicting load addresses as well as predicting addresses of missing loads only when compared with local predictors or a Markov predictor with larger prediction tables.

V.C Using Pointer Cache for Value Prediction and Main Thread Prefetching

We first examine using the pointer cache for a single-threaded wide-issue superscalar processor. This section describes using the pointer cache for two distinct purposes. The first purpose is to use the values out of the pointer cache to break true data dependencies exposing additional ILP. The second purpose is to initiate prefetching when a load hits in the pointer cache.

V.C.1 Using the Pointer Cache to Predict Values

We access the pointer cache with a load's effective address in parallel with the data cache. A pointer cache hit provides a predicted value for the load. This provides the base address for an object to potentially be used by subsequent loads to access that object's fields. For programs where the critical loop path consists of instructions performing serialized pointer chasing, value predicting the base addresses effectively allows multiple loop iterations to execute in parallel.

If the predicted address differs from the value which was loaded, the misspeculation must be repaired, either by re-executing instructions dependent on the load, or by flushing the pipeline. Note that very few misspeculations (less

than 0.2% of all pointer cache accesses) occurred in our results because of the use of store teaching and 10-bit partial tags.

Different pointer cache configurations are also possible. One such design is the gDiff predictor proposed by Zhou et al. [79]. Using the gDiff predictor to implement the pointer cache is part of our future research.

V.C.2 Using the Pointer Cache to Aid Prefetching

When using a pointer cache to provide predicted values, the consumers of this value are typically loads to various fields of the object. The predicted object's cache blocks will not be loaded into the data cache (assuming they are not already there) until the first consumers of each block of the object are executed. It can take several cycles from the time the pointer cache prediction was made until these first consumers are executed. This can occur because of resource contention, or because the first consumer of a block may be statically compiled a reasonable distance from the base pointer load. Consequently, programs can benefit from initiating the prefetch of the object when we get a hit in the pointer cache.

To use the pointer cache for main thread prefetching, when a load hits in the pointer cache, we initiate a prefetch of the pointed-to address and the sequential next line after that into the L2 cache. We blindly prefetch two blocks for the object, since many objects do not fit into one cache block. More accurately predicting the object size for prefetching is a goal of planned future work.

V.D Methodology

Benchmarks are simulated using SMTSIM [72], a cycle accurate, execution driven simulator that simulates an out-of-order, simultaneous multithreading processor. SMTSIM executes unmodified, statically linked Alpha binaries. Ta-

Pipeline Structure	8 stage pipeline, 1 cycle misfetch penalty, 6 cycle minimum mispredict penalty
Fetch	8 instructions total from up to two threads
Branch Predictor	88kbit 2Bc-gskew branch predictor modeled after the EV8 [62] branch predictor but with instantaneous global history update (which also accounts for its smaller size)
	256 entry 4-way associative BTB
Execution Resources	6 total int units, 4 can perform mem ops, 3 fp. All units pipelined, 256 int and 256 fp renaming regs 128 entry int and fp instruction queues. Each thread has a 384 entry commit buffer
Memory Hierarchy	64KB, 2-way instruction cache, 64KB, 2-way data cache 256KB, 4-way shared L2 cache (10 cycles round trip time) 2048KB, 8-way shared L3 cache (30 cycle round trip time) Memory has a 230 cycle round trip time, 128 entry instruction and data TLB TLB misses handled by pipelined, on chip TLB miss handler, 60 cycle latency
Stream Buffers	8 stream buffers of 4 entries with 1 cycle access time
Address Predictor	256-entry 4-way Stride prediction table, or 256-entry 4-way Stride prediction table to filter stride predictions from a 2,048-entry Markov predictor
Multithreading	8 total hardware thread contexts

Table V.1: Assumed baseline architecture simulation parameters.

ble V.1 shows the configuration of the processor modeled in this research. Programs are simulated for 300 million committed instructions or until completion, starting with a cold cache.

We study eight memory limited benchmarks in the rest of this thesis. `mcf`, `parser`, and `vpr` are from the SPECINT2k suite and `em3d` is from the Olden suite. `Dot` is taken from the AT&T's GraphViz suite. It is a tool for automatically making hierarchical layouts of directed graphs. Automatic generation of graph drawings has important applications in key technologies such as database design, software engineering, VLSI and network design and visual interfaces in other domains. We also used `gawk`, which is the GNU Project's implementation of the

AWK programming language. Finally our benchmark suite includes `sis` which is an interactive program for the synthesis of both synchronous and asynchronous sequential circuits, and `vis` which integrates the verification, simulation, and synthesis of finite-state hardware systems.

Benchmarks from the SPEC suite are compiled for a base SPEC build, and other benchmarks are compiled with gcc -O4. All simulations except `em3d` and `gawk` run for 300 million instructions after first skipping program initialization code as determined by the SimPoint tool [64, 65]. `Em3d` and `gawk` are run from the beginning until completion. Table V.2 presents additional details on the simulated benchmarks, including the fraction of simulated instructions which are loads, and data cache miss rates for each level of cache when executing each program with no prefetching.

We examine two architectures that use stream buffers. The first is the baseline program counter based stride prefetcher proposed by Farkas et al. [26] (stride prefetching). The second is the *Stride-filtered Markov* (SFM) predictor proposed in Chapter IV. For all stream buffer configurations, we use the confidence-based allocation and priority-based scheduling described in Chapter IV. The stride configuration uses a 256-entry, 4-way associative stride address prediction table. For the stride-filtered Markov scheme, we utilize a similar 256-entry 4-way stride address prediction table to filter stride predictions out of a 2K-entry Markov table (approximately 4KB). For both the stride and the stride filtered Markov architectures we use 8 stream buffers, each with 4 entries. All stream buffers are checked in parallel on a lookup. In addition, when a stream buffer generates a prediction, all stream buffers are checked to guarantee that the stream buffers do not follow overlapping streams. Unless otherwise noted, all the techniques utilize 8 stream buffers with 4 entries each guided by either the PC-Stride or the SFM predictor.

Bench	FFwd	% lds	L1 MR	L2 MR	L3 MR
`dot`	5.1B	40.0%	26.7%	88.7%	97.1%
`em3d`	0	30.9%	7.6%	89.0%	74.5%
`GNU awk`	0	28.3%	1.2%	25.5%	67.8%
`mcf`	50.8B	30.8%	10.6%	82.2%	83.2%
`parser`	228B	23.4%	2.3%	59.9%	37.8%
`SIS`	5B	27.6%	2.3%	58.4%	23.0%
`VIS`	5B	24.5%	1.9%	84.4%	86.4%
`vpr`	31.8B	32.6%	2.8%	67.9%	50.1%

Table V.2: Details of simulated benchmarks. Data cache miss rates are shown for a processor which performs no hardware prefetching.

Each pointer cache entry is comprised of a 10-bit partial tag and a 26-bit differential address field. The address field stores the difference between the indexing address and the value stored in the entry to conserve area. Using 26 bits of difference was enough to cover the whole memory space for the programs we examined. The overall size of the 256K entry pointer cache is 1.1MB, which is about the size of a 1MB cache along with its tags. As for the partial tags, tag-width sensitivity analysis showed that tags wider than 8 bits result in very few tag mismatches — less than 1%.

V.E Pointer Cache Performance

This section compares the pointer-cache assisted uniprocessor architecture to prior prefetching techniques which have been shown to perform well on pointer intensive programs. The prior techniques include speculative precomputation [23] (SP) alone as described in Chapter III and Predictor-directed Stream Buffers [58, 66] (PSB) as described in Chapter IV.

V.E.1 Previous Prefetching Mechanisms

We first explore the performance impact of prior prefetching schemes. Figure V.4 shows performance results with no prefetching, prefetching using a program counter stride predicted stride stream buffer (stride prefetching), Stride-Filtered Markov (SFM), Speculative Precomputation (SP), and two new combinations not examined in prior literature which combine speculative precomputation with stride and SFM guided stream buffers respectively.

When combining speculative precomputation with stream buffer prefetching, we allow the speculative threads to access the stream buffers and the address predictor along with the main thread. To make the stride predictor thread-aware, we extended the load PCs used to update the predictor with thread IDs. This increases the accuracy of the predictor on a per-thread basis, maximizing the stream buffer efficiency. Speculative threads are treated as equals with the main thread with respective to all stream buffer functions (e.g., spawning stream buffers on a data cache miss).

The results show that stride-based prefetching provides large speedups for `dot`, `em3d`, and `parser`. Speculative precomputation by itself performs well on `sis` and `vpr`, but misses a lot of potential benefit the simple stride prefetcher achieves. Therefore, we combined the stride prefetcher with speculative precomputation, and found that this provided better overall results when compared to either technique alone. The results show that combining stride with speculative precomputation provides on average 29% speedup over only using stride, and 25% speedup over only using speculative precomputation.

When we look at the SP and SFM configuration, we observe a 40% speedup over only using speculative precomputation on average. We do not see any noticeable speedups over SFM prefetching however. This is mainly because the benefit SFM provides to the speculative threads is offset by stream thrashing

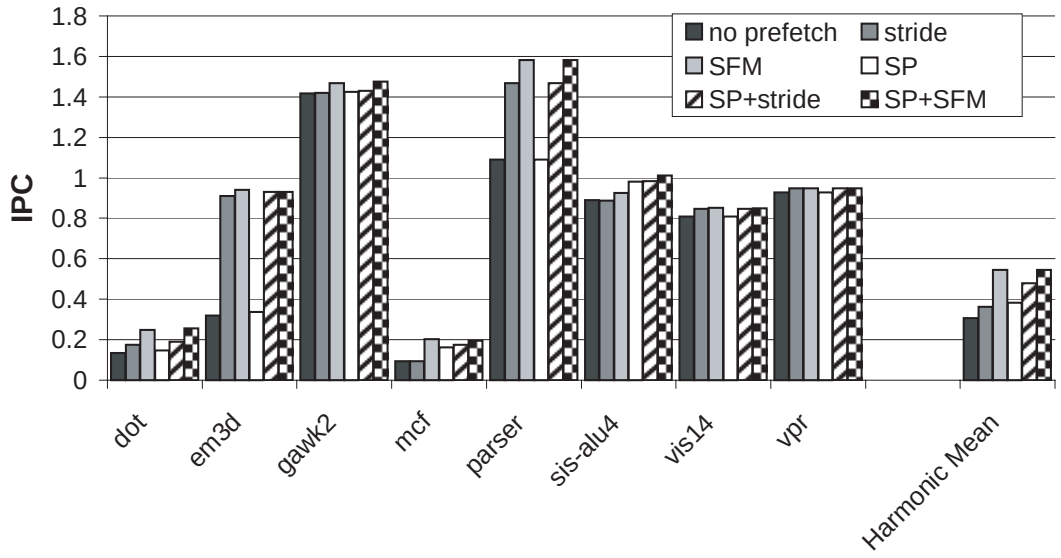

Figure V.4: Performance improvement from previous prefetching schemes.

caused by the SP threads competing for stream buffers alongside the main thread. We can increase the number of stream buffers and partition them between the main thread and the speculative threads to prevent this. Since we wanted to keep the hardware budget devoted to stream buffers, we did not consider this solution however. Still, speculative precomputation and SFM prefetching combination still outperforms the SP and stride prefetching architecture by 10%.

Unlike the other benchmarks, `em3d` shows a very dramatic speedup (almost 200%) from using only stride prefetching over the next best approach. The traversal of `em3d`'s data structures occur in the same order in which they memory allocated, and the data structure pointers do not change during execution. Because of this and the fact that the memory allocator allocates the objects to sequential addresses in a linear order, `em3d`'s object traversal is highly stride prefetchable as seen in Chapter II.

Two benchmarks, `dot` and `mcf`, involve significant numbers of dependent pointer dereferences. The SFM scheme provides the best performance for these

two programs because the stream buffers are able to use address prediction to run ahead of the main thread. SP in contrast, is hampered by the serial nature of these memory accesses, and has difficulty prefetching ahead of the main thread. Section VI.B.1 illustrates how the pointer cache enables SP to overcome this limitation.

V.E.2 Main Thread Pointer Cache Prefetching

In this section we evaluate the benefits of allowing the main thread to use the pointer cache (PC) to hasten the issue of instructions dependent on a pointer load. We compare several pointer cache configurations with varying capacity and access latency to a baseline architecture of PC-Stride prefetching (labeled stride in the graphs) or SFM prefetching (labeled SFM in the graphs).

Figures V.5 and V.6 display the effects of permitting the main thread access to a pointer cache with varying access times (between 5 and 20 cycles). All the pointer cache configurations are 4-way set-associative and have 256K entries, except the last one with 16 Meg entries. The second and the sixth bars show the effect of only using the pointer cache for value prediction. The remaining bars use the pointer cache for value prediction and prefetching into the L2 on a pointer cache hit. All results in Figure V.5 have a baseline stride prefetching stream buffer and all results in Figure V.6 have a baseline stream buffer architecture guided by the SFM predictor.

The results show that access latency is not a significant factor in performance. This can be attributed to the pointer cache allowing the main thread to hide long latencies associated with off-chip cache misses. Prefetching the pointer cache target into L2 cache works particularly well for `mcf` and `parser` over only using the pointer cache for value prediction. Since an increase in pointer cache hit latency results in a delay in initiating prefetches for the target object, these

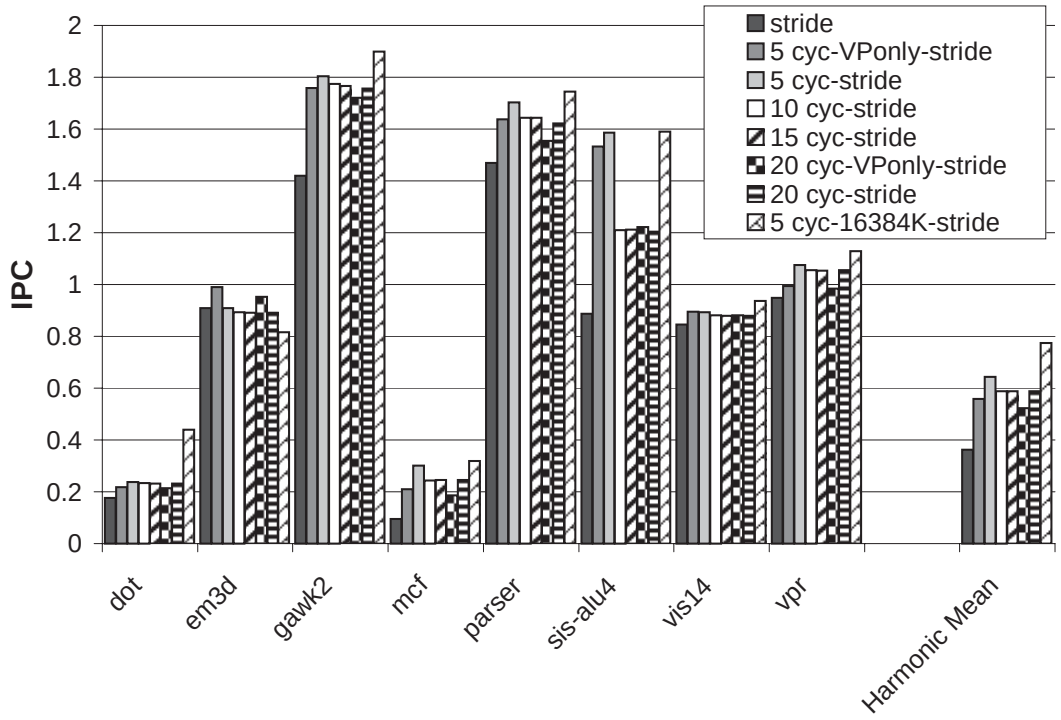

Figure V.5: Performance impact when the main thread is permitted to access a 256K-entry, 4-way pointer cache with different access latencies. All configurations also use the stride prefetcher. VPonly shows results using the pointer cache only for value prediction. All of the rest of the PC results use the pointer cache for both value prediction and prefetching.

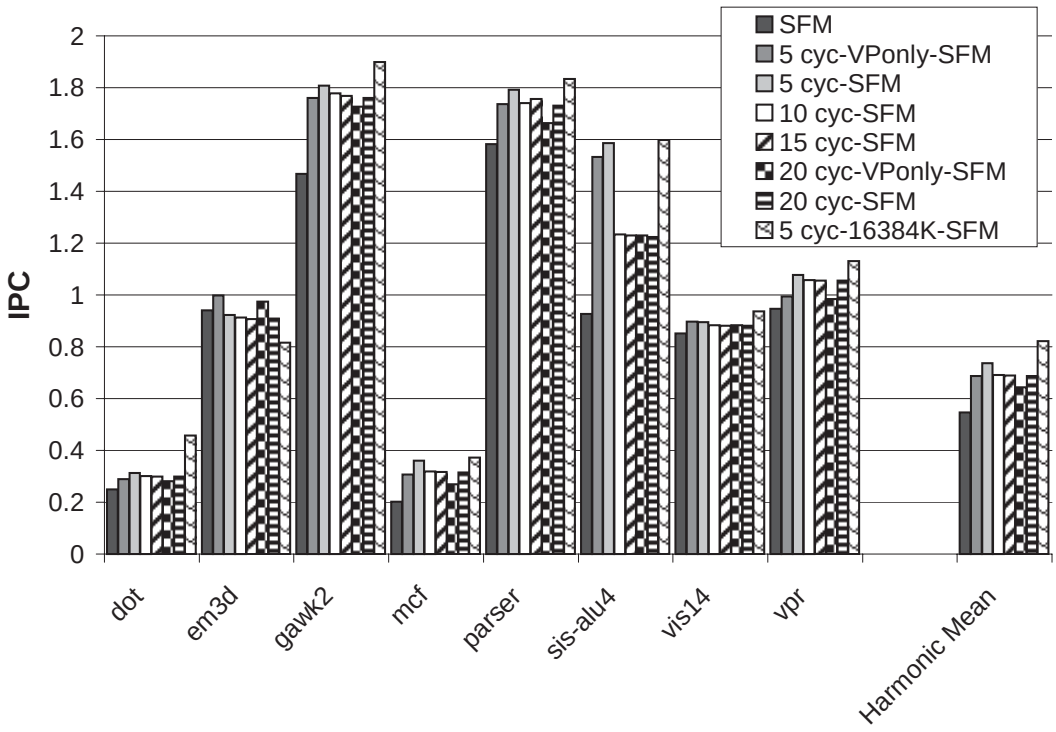

Figure V.6: Performance impact when the main thread is permitted to access a 256K-entry, 4-way pointer cache with different access latencies. All configurations also use the SFM prefetcher. VPonly shows results using the pointer cache only for value prediction. All of the rest of the PC results use the pointer cache for both value prediction and prefetching.

benchmarks suffer some performance degradation with a 20 cycle pointer cache. In comparison, `dot` also enjoys a significant benefit from additional prefetching, but it is not affected by the increase in access time since it is more capacity-limited than latency-limited as described below.

For `em3d`, value prediction only performs better than prefetching. This is because data structures in both this program are small, and prefetching two cache lines pollutes the cache by bringing in irrelevant data. This is especially true for `em3d`, where stride prefetching is already effective at prefetching the same loads which are targeted by the pointer cache.

Even though the relative speedups achieved when using the SFM prefetcher in the baseline is lower that those enjoyed by the stride prefetcher, another trend evident in Figures V.5 and V.6 is that using the SFM prefetcher in the baseline outperforms the baseline with the stride prefetcher.

Figures V.7 and V.8 show results for using the pointer cache for value prediction and prefetching varying the number of pointer cache entries. Figure V.7 provides results for a pointer cache that is accessed in 5 cycles while Figure V.8 is for a pointer cache that is accessed in 20 cycles. All configurations utilize stride prefetching in these graphs. These results show that performance is independent of the access latency until we utilize larger pointer caches with 512K entries or more. At this point, the capacity problems are solved and latency becomes the major factor in determining performance.

Figures V.9 and V.10 show the same results for using a pointer cache backed up by a SFM prefetcher. Figure V.9 provides results for a pointer cache that is accessed in 5 cycles while Figure V.10 is for a pointer cache that is accessed in 20 cycles. Trends observed in Figures V.7 and V.8 are also evident in Figures V.9 and V.10 with the exception that the architectures using the SFM prefetcher exhibit higher performance across all configurations. Moreover, the

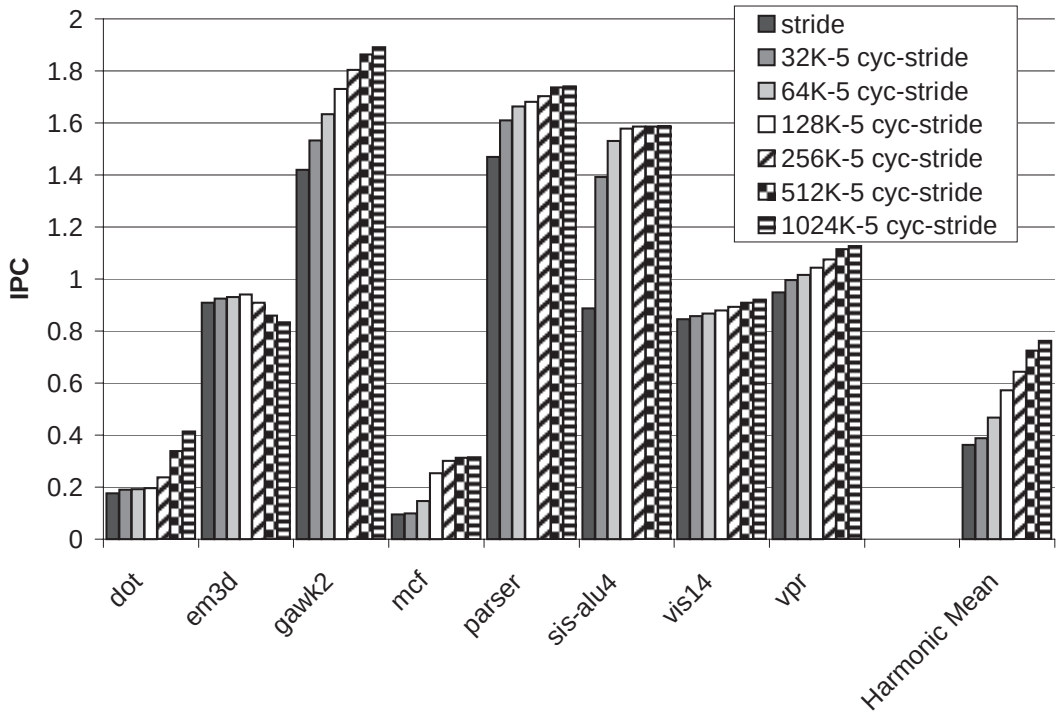

Figure V.7: Performance impact of varying the number of entries in the pointer cache. All pointer cache configurations utilize a 4-way pointer cache with a 5 cycle access latency. All the configurations also make use of stride prefetching.

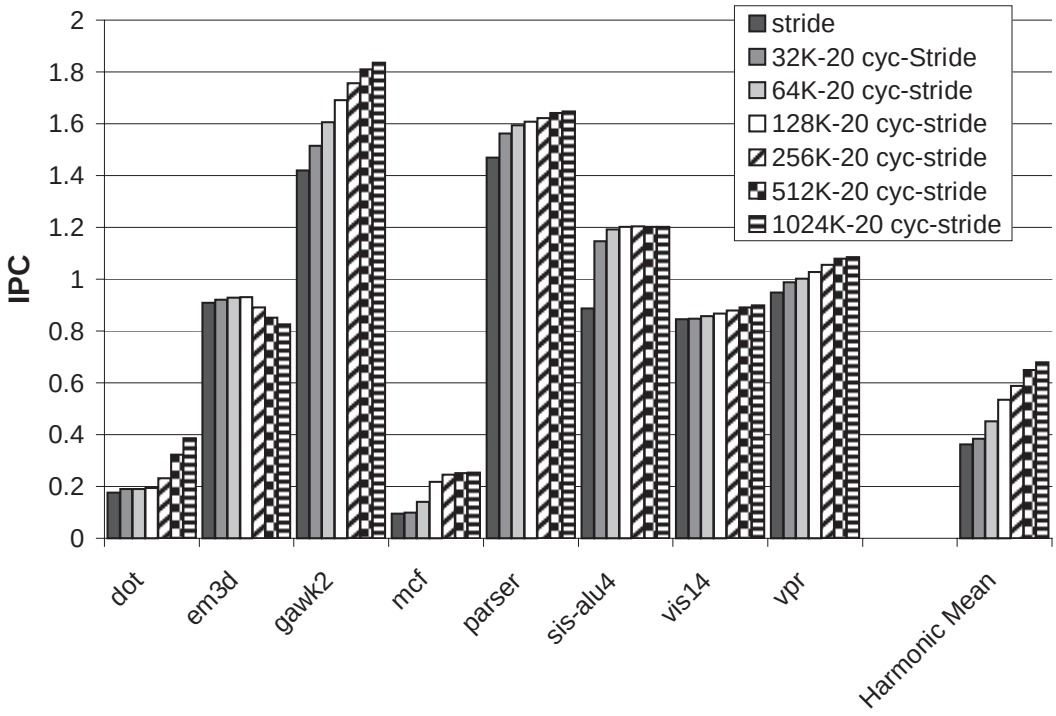

Figure V.8: Performance impact of varying the number of entries in the pointer cache. All pointer cache configurations utilize a 4-way pointer cache with a 20 cycle access latency. All the configurations also make use of stride prefetching.

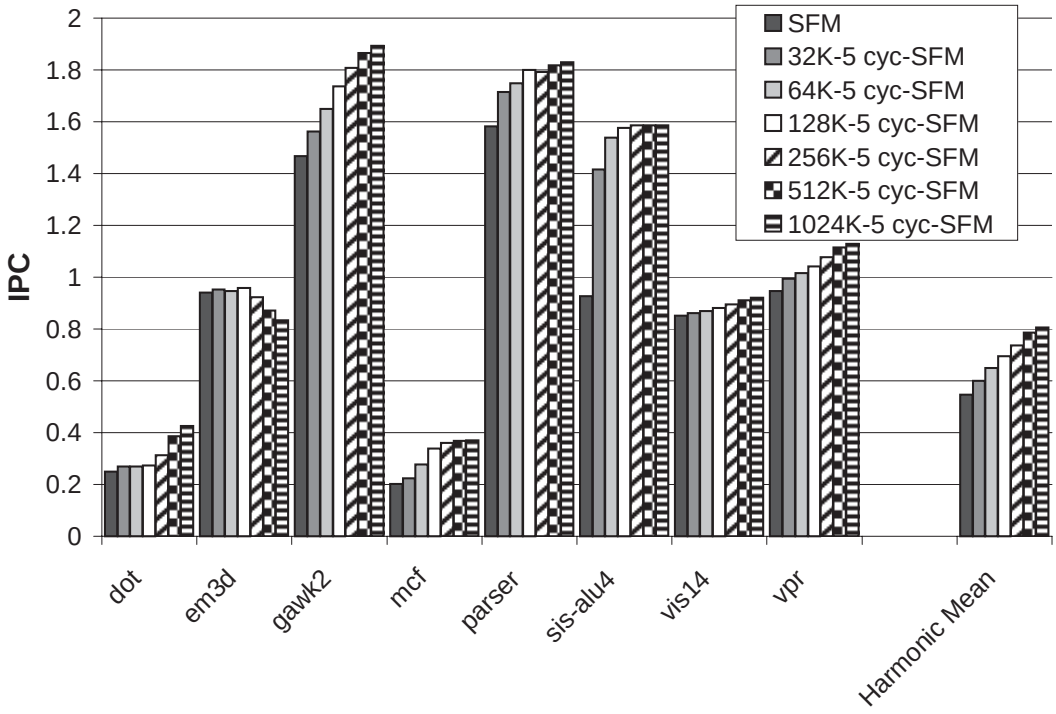

Figure V.9: Performance impact of varying the number of entries in the pointer cache. All pointer cache configurations utilize a 4-way pointer cache with a 5 cycle access latency. All the configurations also make use of SFM prefetching.

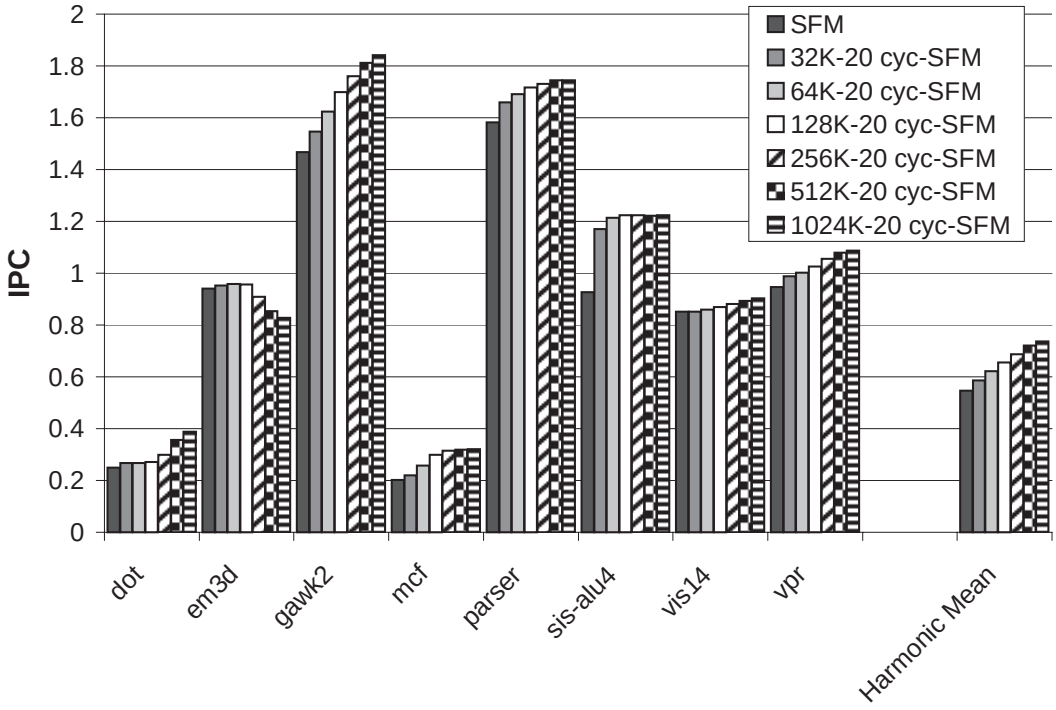

Figure V.10: Performance impact of varying the number of entries in the pointer cache. All pointer cache configurations utilize a 4-way pointer cache with a 20 cycle access latency. All the configurations also make use of SFM prefetching.

speedup curves are less steep when compared to those in Figures V.7 and V.8 because the SFM predictor is doing a good job of targeting applications with simple control flow.

Overall, these results show that performance is more sensitive to the capacity of the pointer cache than to the latency to access it. With the exception of `dot` and `gawk`, the pointer data set for each program fits into a 256K entry, 4-way pointer cache. `Dot` has a large working set — it has a 3% hit rate on a 2M, 8-way L3 cache — and incurs pointer cache capacity misses for pointer caches with less than one million entries. `Gawk` also traverses long recursive data structures, creating the need for a large pointer cache. It is interesting to note that `gawk`, `parser` and `sis` attain noticeable speedups even with a relatively small 32K entry pointer cache.

V.F Summary

A wide range of applications — from games to database management systems — are based on dynamic data structures linked together via pointers. However, such accesses are often not governed by the localities exploited by traditional cache organizations. Furthermore, misses to such pointer-based loads, especially recurrent load accesses, significantly restrict parallelism and expose the full latency to memory.

In this chapter we proposed using a Pointer Cache to accelerate processing of pointer-based loads. The pointer cache provides a prediction of the object address pointed to by a particular pointer. If the load misses in cache, consumers of this load can issue using this predicted address from the pointer cache, even though the load itself is still outstanding. Using a 5-cycle pointer cache for just value prediction provided a 25% speedup over Stride-filtered Markov prefetching for a single-threaded processor.

We also examine using the pointer cache to initiate prefetches for the main thread of execution. On a pointer cache hit, prefetches for the first two cache blocks of the object are initiated. This provides an additional 12% speedup on average over using the pointer cache for just value prediction because the object's cache blocks are accessed at the time of prediction instead of waiting until their first use.

Another contribution is the examination of adding stream buffer prefetching to speculative precomputation. We found that a speculative precomputation architecture that incorporated a stride prefetching architecture provided 25% speedup over using only SP. Even with this improvement, we found that applying speculative precomputation with stride to a suite of pointer intensive applications was ineffective for half of the programs we examined, often because of recurrent loads. To address this, in the next chapter we propose using the pointer cache to increase the effectiveness of speculative precomputation by supplying speculative threads with pointer load values when they otherwise would have been forced to stall due to cache misses.

On the other hand, using the SFM predictor guided stream buffers in conjunction with SP was more efficient providing 40% speedup over SP. The SFM predictor is able to provide the speculative threads with recurrent load values improving their efficiency. However, they also increase the pressure on the allocation process contending for stream buffers alongside the main thread. Therefore using the pointer cache to assist speculative threads with recurrent pointer loads is more beneficial as discussed in the next chapter.

Furthermore, since the pointer cache configurations we examine are neither small, nor fast, we could improve the effectiveness of the pointer cache significantly if we can use them in a setting where the pointer cache is queried much earlier than the pointer load instruction. Speculative precomputation allows that

by executing these problematic loads in speculative thread contexts. Moreover they could guide the pointer cache predictions down the right traversal path if we incorporate the necessary branch instructions in the speculative precomputation slices. These optimizations are discussed in more detail in the next chapter.

Chapter VI

Pointer Cache Assisted Data Prefetching

Data prefetching is one technique that reduces the observed latency of memory accesses by bringing data into the cache or dedicated prefetch buffers before it is accessed by the CPU. One can classify data prefetchers into three general categories. Hardware data prefetchers [16, 33, 34, 66] observe the data stream and use past access patterns and/or miss patterns to predict future misses. Software prefetchers [47] insert prefetch directives into the code with enough lead time to allow the cache to acquire the data before the actual access is executed. Recently, the expected emergence of multi-threaded processors [74] has led to thread-based prefetchers [2, 22, 23, 41, 46, 55, 56, 70, 80], which execute code in another thread context, attempting to bring data into the shared cache before the primary thread accesses it.

However, traditional prefetching techniques have difficulty with sequences of irregular accesses. A common example of this type of access is pointer chains, where the code follows a serial chain of loads (each dependent on the previous one) to find the data it is looking for.

Some hardware approaches [66] can run ahead of the main program when traversing a pointer-chain, since they can predict the next address. This assumes that a history of the pointer traversal over the miss stream can be captured in the predictor. The accuracy of these techniques drops when running too far ahead of the main thread, since the techniques are based entirely on prediction. Accuracy can degrade because of (1) the address predictor being used, and (2) having pointer traversals in a program guarded by branches (e.g., tree traversal) and these techniques do not incorporate control flow into their prediction.

Thread based techniques, such as speculative precomputation [22, 23] (SP), have the advantage of using code from the actual instruction stream, allowing them to accurately precompute load addresses. A potential shortcoming of speculative precomputation is that cache misses can prevent the speculative thread from making progress faster than the main thread when traversing pointer chains with little other intervening computation,

In this chapter we show that using a pointer cache to provide pointer base addresses overcomes serial accesses to recurrent loads for an SMT processor with support for prefetching via speculative precomputation. The pointer cache is particularly effective in combination with speculative precomputation for two reasons. It prevents the speculative thread from being hampered by long, serial data accesses. Data speculation occurs in the speculative thread rather than the main thread, allowing greater distance between the speculative computation and the committed computation. Furthermore, the control instructions in the speculative threads allow them to follow the correct object traversal path resulting in very accurate preloading of cache lines to be required by the main thread.

Figure VI.2 illustrates why it is imperative to have a control-flow sensitive prefetching mechanism. Here we present the `mcf` example one more time to emphasize the irregular nature of pointer based applications. When we execute

```
...
while( node->pred )
{
   tmp = node->sibling;
    if( tmp )
    {
        node = tmp;
        ...
     }
     else
          node = node->pred;
}
```

Figure VI.1: Code example from SPEC'2000 integer benchmark mcf. As seen in this code, there are multiple potential transitions to follow(illustrated in bold). Moreover, the decision to choose which transition to follow is guarded by branch instructions(e.g. `if( tmp )`).

the piece of code in Figure VI.1 on the data structure shown on Figure VI.2.A, we follow the path drawn on Figure VI.2.B assuming that the traversal starts with the light gray node. As shown in Figure VI.2.B, at each step of the traversal, there are multiple paths that execution can follow. Since the decision to choose which transition to follow is made by control flow instructions, including control flow information in a prefetching algorithm would increase its accuracy considerably in the face of complex structures with multiple transitions.

The rest of the chapter is organized as follows. Section VI.A describes utilizing the pointer cache in accelerating speculative precomputation. Section VI.B presents performance results, and Section VI.C concludes.

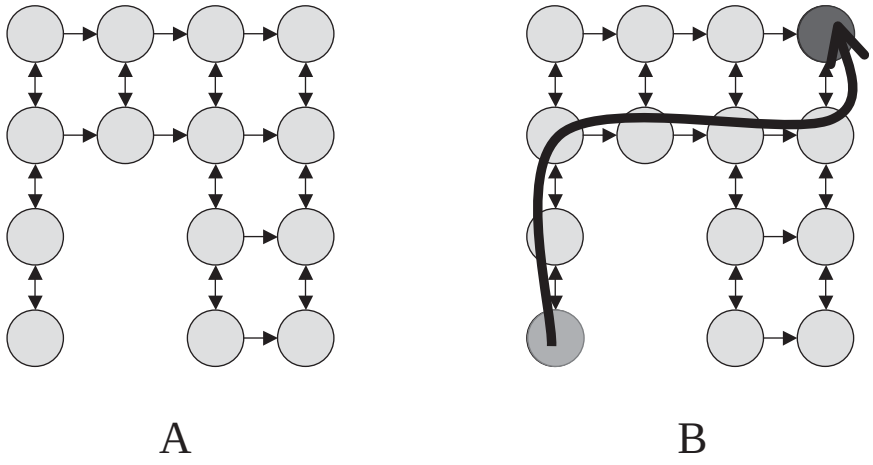

Figure VI.2: The traversal path execution follows when the code in Figure VI.1 is executed on the data structure shown on the left. As shown on the right, if we start at the light gray node, the traversal ends at the dark gray node. This irregular traversal pattern is a typical behavior in pointer-based applications.

VI.A Using Pointer Cache with Speculative Precomputation

This section describes the use of the pointer cache to aid speculative precomputation, enabling it to get farther ahead of the main thread of execution. Unlike the main thread, speculative threads are not bound by the correctness requirement, and can freely integrate pointer cache address predictions into their thread context without verification.

VI.A.1 Control Oriented Speculative Precomputation

This work assumes a Speculative Precomputation architecture similar to that described in [23], on a simultaneous multithreading [73, 74] (SMT) processor. For this work, speculative threads are generated off-line and trigger instructions are identified manually. Speculative threads fetch instructions out of a dedicated

hardware Slice Cache, avoiding fetch related conflicts with the main thread.

Unlike the prior speculative precomputation work in [23], speculative threads constructed for this research can contain important conditional branches. One of the primary benefits of SP in this architecture is the ability to correctly follow control flow. Allowing SP to resolve branch instructions and recover from branch mispredictions provides the ability to accurately traverse control flow to find out what pointer transition to follow next. This is particularly important when there are several possible next pointer transitions for a given loop construct.

Control speculation in the speculative threads is handled just like a non-speculative thread — fetch is guided by branch prediction, and on a branch misprediction all instructions following the mispredicted branch are squashed and the speculative thread is redirected down the correct control path.

Because slices contain branches in this research, poor branch prediction resulting from a cold branch predictor could hamper the speculative threads' ability to get ahead of the main thread. Therefore, we assume an architecture that allows the speculative thread to benefit from the branch predictor training of the main thread. For conditional branch prediction, we would ideally like to have the same hash bits for the speculative thread branch PCs as the original main thread code. Thus, we assume that the speculative thread code is laid out so that the PC bits used for the branch prediction hash are the same between the branch in the main thread and its duplicate in the speculative thread. Therefore, the speculative thread will use the same 2-bit counters for prediction as the main thread, but in our implementation the speculative threads do not update the conditional branch 2-bit counters when the branch completes. For branch target address prediction, the speculative branch stream needs to be executed once and inserted into the branch target buffer before taken branches on the SP thread can enjoy no fetch stalls.

VI.A.2 Using the Pointer Cache with Speculative Precomputation

The pointer cache is used by speculative threads only as a value prediction. When a speculative thread executes a load, the pointer cache is accessed in parallel. On a pointer cache hit, the load is marked as ready to commit (in addition to waking up its dependent instructions) *even though its memory request is still outstanding*. Since the speculative thread doesn't have to be correct, we do not wait until the load comes back from memory before committing a load that hits in the pointer cache. In comparison, the main thread, when using the pointer cache speculatively, must validate the value prediction before the load and its dependencies can complete execution, which may cause the main thread's fetch to stall due to a full instruction window. Thus, SP makes it possible for speculative computation to advance well ahead of the main thread when the speculative thread incurs a sequence of pointer cache hits.

VI.A.3 Making Speculative Threads

Speculative threads are constructed manually, using an approach similar to the automatic construction described in [22]. We start with loads that account for the majority of misses for the program, which we call *delinquent loads*. The program's static instructions are analyzed in the reverse order of execution from a delinquent load, building up a slice of instructions the load is directly and indirectly dependent upon. As instructions are traversed, the set of registers (the register *live-in* set) necessary to compute the address accessed by the delinquent load is maintained — when an instruction is encountered which produces a register in this set, the instruction is included in the slice, and the register set is updated by clearing the dependence on the destination register, and adding dependences for the instruction's source registers. If, during analysis, another load which has been identified as delinquent is analyzed, it is automatically included

in the slice.

If only a single dominant control path leads to the delinquent load, only this control path is traversed (in reverse order) when constructing the slice, and the conditional branches necessary to compute this control path are not included in the slice. Slice construction terminates when analyzing an instruction far enough from the delinquent load that a spawned thread can provide a timely prefetch, or when further analysis will add additional instructions to the slice without providing further performance benefits. In this form, a slice consists of a sequence of instructions in the order they were analyzed.

The single path slices constructed in this work are triggered typically between 40 and 100 instructions prior to the delinquent load, and contain between 10 and 15 instructions. In some cases, moving the trigger instruction further back would make the slice less profitable because doing so requires a significant increase in the number of executed speculative instructions. For example, when the targeted delinquent loads are preceded by computation of a hash function, including this hash function in the slice may degrade performance.

If multiple control paths lead to the delinquent load, the constructed slice must contain the dependence chains from each path, and must also include the relevant branches to determine which path to take when executing the slice. In addition, if there are multiple delinquent loads on different paths that define the same register, then those loads and their corresponding branches are included into the same slice. For example, the slice formed for Figure V.3 includes the loop branch, the `if (ptr->x)` branch and its register operand definitions, and the two loads `ptr=ptr->left` and `ptr=ptr->right`.

Slices which contain multiple paths are typically targeting delinquent loads within a loop. The slice formation is terminated when analysis reaches an instruction sufficiently far preceding the loop (in which case the last instruction

added is marked as a trigger instruction), or when the live-in set converges for all of the control flow paths and the instructions in the slice. Branches that are not critical to the slice are marked as *predict-only*. This means that those branches will only generate a prediction, and the speculative thread will not verify the correctness of the prediction. In addition, these branches cannot cause a speculative thread to be terminated, which we describe in more detail below. These predict-only branches are inserted into speculative threads in order to keep the global branch history information accurate for the main thread, and to keep the speculative branch outcome FIFO synchronized with the branches seen by the main thread. Keeping global branch history accurate in the speculative thread is important since the main thread and speculative thread share the 2-bit predictor entries.

The multiple-path slices with control flow consist of between 8 and 40 static instructions for the programs we examined, with some instructions unique to a particular control path. In comparison, the original targeted loops contain between 27 and 171 instructions. Some slices consist of a significant number of the instructions making up a loop (>50% of the loop instructions) when address computation is complex, or when a significant number of branch outcomes must be computed.

VI.A.4 Spawning Speculative Threads

When an instruction in the main thread that has been identified as a trigger instruction reaches the register rename stage, a speculative thread is spawned into a hardware thread context if one is available and another instance of the same speculative thread is not already executing. The speculative thread's context is initialized by copying the necessary register live in values from its parent thread, and by copying the global history register value (at the time the

trigger instruction was fetched) from its parent thread, ensuring the accuracy of the branch predictions performed by the speculative thread.

Speculative threads spawn further child threads via explicit child spawn instructions (the *chaining triggers* of [23]) in the speculative thread. Unlike when spawning threads off of the main thread, a child thread is spawned even if there is already another instance of that particular thread executing. A speculative thread that attempts to fetch a spawn instruction when all thread contexts are occupied is prevented from doing so, and stops fetching until a thread context becomes available.

VI.A.5 Maintaining Control of the Speculative Threads

The branch predictor for our SMT architecture is shared among all of the threads, but each thread maintains its own global history of executed branches. We call this the branch outcome FIFO queue. When a speculative thread is forked, the parent thread's speculative global history register, when the trigger point was fetched, is copied to the speculative thread. From that point on, the speculative thread performs its own branch predictions and keeps track of its predicted global branch history in the FIFO global history queue.

The speculative thread keeps track of the outcomes of all executed branches that have not committed in the main thread, starting from the trigger point. This enables two important mechanisms — killing of speculative threads which have deviated from the main thread's control flow, and a feedback mechanism to control how far ahead a speculative thread is allowed to go.

The first mechanism is achieved by comparing the head of this FIFO with each main thread branch that commits, and killing the speculative thread when they differ. If the branch outcomes agree, the head entry is removed. A speculative thread can be on the wrong path because the formation of the

speculative thread has left out an instruction (e.g., a store) that would at a later point affect the outcome of a branch on the speculative thread. Note, that if the FIFO entry was produced by a predict-only branch, this check is not performed. Predict-only branches do not cause a speculative thread to be terminated.

The second mechanism is achieved by preventing a speculative thread from fetching when this FIFO exceeds a limit. Each slice can have its own limit. The multi-path slices we created had this limit set between 8 and 256 branches, to target traversing several loop iterations ahead.

The branch outcome FIFOs can also provide a simple mechanism for guiding main thread branch prediction with outcomes computed in a speculative thread. However, because this study focused only on the use of speculative threads for prefetching, computed branch outcomes are not used for this purpose. This is a topic of future work.

VI.B Prefetching Performance

This section presents results for the pointer-cache assisted speculative precomputation architecture and compares it to several prior prefetching techniques. We also show how using the pointer cache for both the main thread and speculative precomputation threads stacks up against using a larger cache in the baseline architecture.

The simulation methodology for the results presented here is the same as the previous chapter's except for the addition of speculative precomputation. The speculative precomputation technique makes use of an SMT processor with 8 thread contexts. As described in the previous section, the precomputation slices are created by hand.

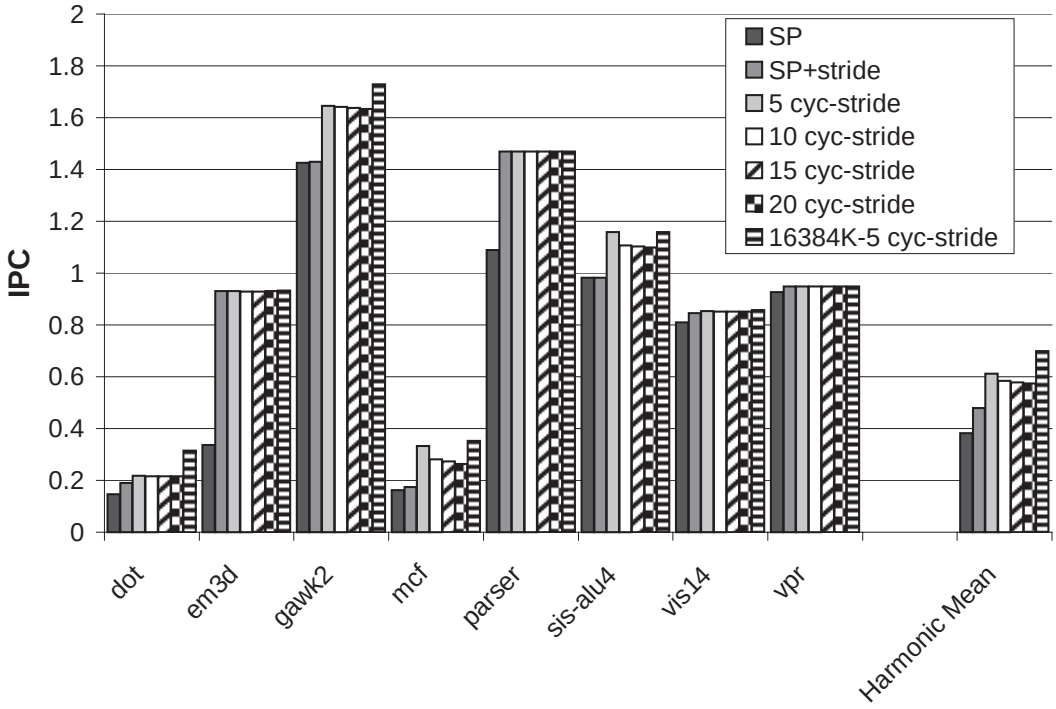

Figure VI.3: Performance impact from combining SP and stride prefetching when using a 256K-entry, 4-way pointer cache with varying access latency for assisting speculative precomputation threads only. The main thread does not use the pointer cache.

VI.B.1 Pointer Cache Assisted Speculative Precomputation

Figure VI.3 shows the performance impact when combining SP and stride prefetching while permitting speculative threads to access a pointer cache for value prediction only. For these results the pointer cache is *not* used by the main thread of execution. The performance benefits from permitting speculative threads access to the pointer cache are often significant, varying between 2% and 236% over a processor with no pointer cache.

Figure VI.4 presents the same results when the SFM prefetcher is used instead of the stride prefetcher. As seen in the previous chapter, architectures

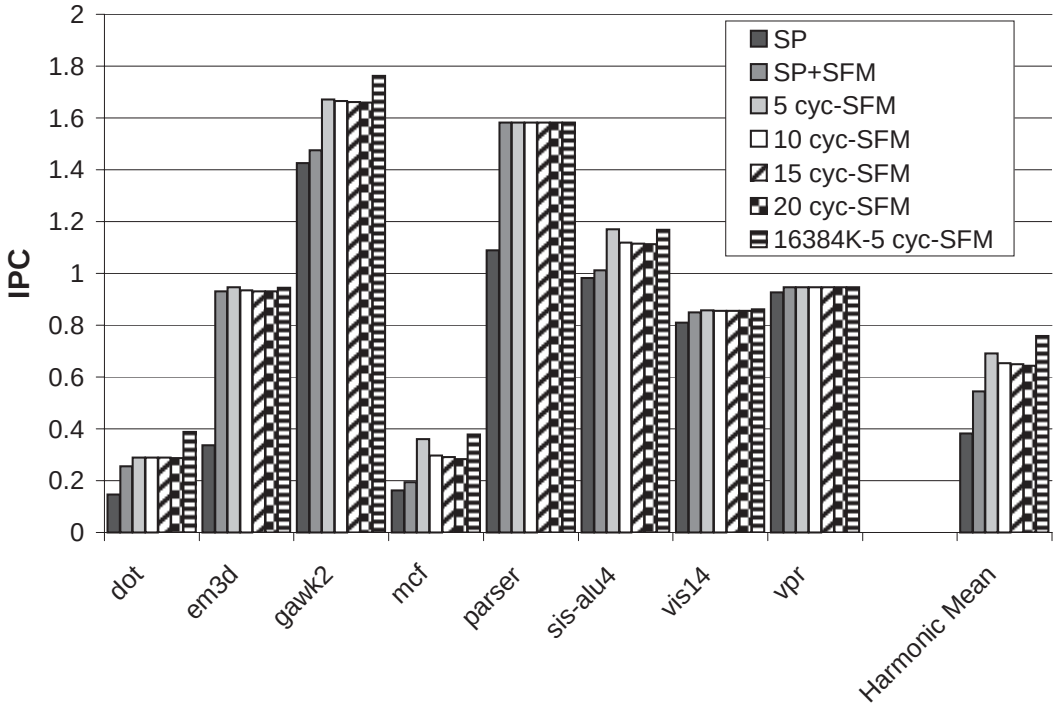

Figure VI.4: Performance impact from combining SP and SFM prefetching when using a 256K-entry, 4-way pointer cache with varying access latency for assisting speculative precomputation threads only. The main thread does not use the pointer cache.

Combined L3 Cache and Pointer Cache Size	Access Latency
3 MB	17 cycles
4 MB	21 cycles
5 MB	26 cycles
6 MB	28 cycles
7 MB	34 cycles
8 MB	35 cycles

Table VI.1: Combined L3 and pointer cache sizes examined along with their assumed access latencies. The latencies are those reported by Cacti 3.0 [67] with a 0.18μm feature size assuming a 1 GHz clock frequency.

using the SFM predictor attain higher performance.

In comparing Figures V.5 and VI.3 (or Figures V.6 and VI.4), some benchmarks achieve a smaller speedup when combining SP with the pointer cache than when only the main thread using the pointer cache. It is not always possible (or profitable) to construct slices targeting some loads in a program, but these loads may achieve hits in the pointer cache when accessing it directly by the main thread. Maximal benefit is achieved by permitting both the main thread and speculative threads access to the pointer cache, which we present next.

VI.B.2 Summary of Results

It is not surprising to see that adding a pointer cache improves performance. The real question is whether the transistors used for a pointer cache are better utilized when used for other structures such as larger caches. We now look at the performance tradeoff involved with using different combined L3 cache and pointer cache areas. We used Cacti 3.0 [67] to compute the access latencies for the cache configurations examined. These results are presented in Table VI.1. All the configurations with a pointer cache use the remaining area after taking

out 2 MB out of our combined area budget for an L3 cache. For example for a combined 5 MB size, we use a 2 MB L3 and a 3 MB pointer cache. The latencies of the structures are based on the combined size. So in the previous example, both the L3 cache and the pointer cache are accessed in 26 cycles.

Figures VI.5 and VI.6 show the performance tradeoffs involved for the benchmarks we examined when both the main thread and the speculative precomputation threads are allowed to access the pointer cache. The results are divided into two graphs for easier studying. For most of the applications, having a pointer cache in conjunction with a 2 MB L3 cache is better than having a larger L3. This result supports the intuition that for pointer-based applications, having a dedicated storage for pointer loads is more valuable than having general-purpose storage. The pointer cache remains beneficial until we reach an L3 size of 8 MB for most benchmarks we examined. `Em3d` is an exception where it does consistently better with a larger L3. As we have seen in Figure V.6 in the previous chapter, this is because `em3d` incurs significant cache pollution due to the prefetching of the next cache line when we incur a pointer cache hit. The bigger the pointer cache gets, the more hits we incur, and thus the worse `em3d` performs. As shown with the dotted line Figure VI.6, if we use pointer cache only for value prediction, the performance of `em3d` compares relatively similar to that of using a large L3 cache. Meanwhile `dot` and `sis` perform better with a pointer cache for all the sizes examined. What is more interesting is that the performance curves appear to remain flat, indicating having a pointer cache will still be better than a large L3 for a few more generations of L3 sizes.

We now compare the performance of the pointer cache architecture to a baseline architecture that uses the transistors we devote to the pointer cache to instead increase the size of the L3 cache. Figure VI.7 compares the performance of speculative precomputation with stride prefetching using a 3 MB L3 cache

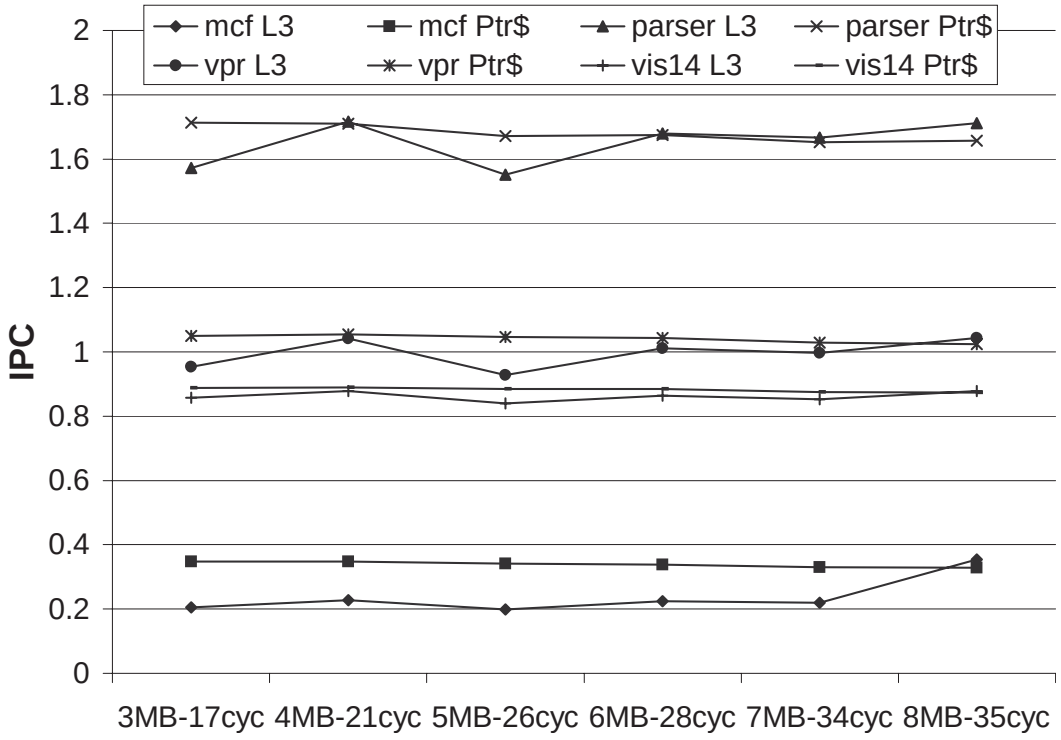

Figure VI.5: Performance results comparing a processor with different combined L3 and pointer cache sizes and latencies. This chart presents results for the SPEC'2000 benchmarks and `vis`. The lines annotated with `L3` are utilizing the whole area for an L3 cache. The lines annotated with `Ptr$` on the other hand indicate using 2 MB for an L3 and the remaining area for a pointer cache. For the pointer cache configurations, both main thread and speculative precomputation threads are allowed to access the pointer cache.

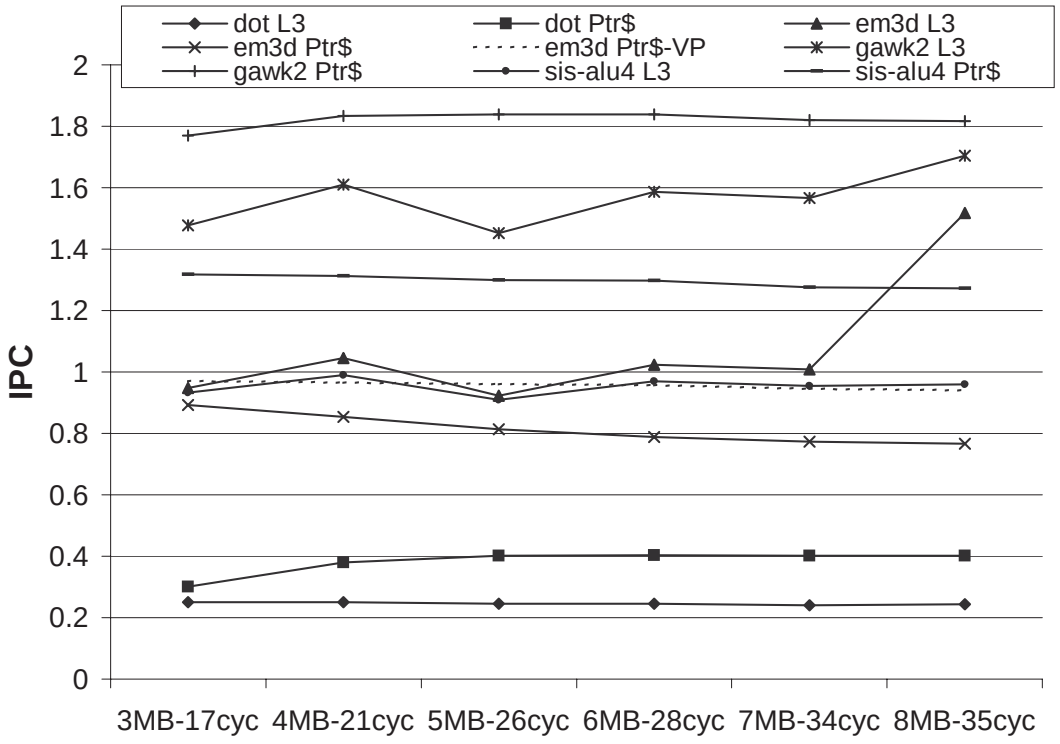

Figure VI.6: Performance results comparing a processor with different combined L3 and pointer cache sizes and latencies. This chart presents results for `dot`, `em3d`, `gawk` and `sis`. The lines annotated with `L3` are utilizing the whole area for an L3 cache. The lines annotated with `Ptr$` on the other hand indicate using 2 MB for an L3 and the remaining area for a pointer cache. For the pointer cache configurations, both main thread and speculative precomputation threads are allowed to access the pointer cache.

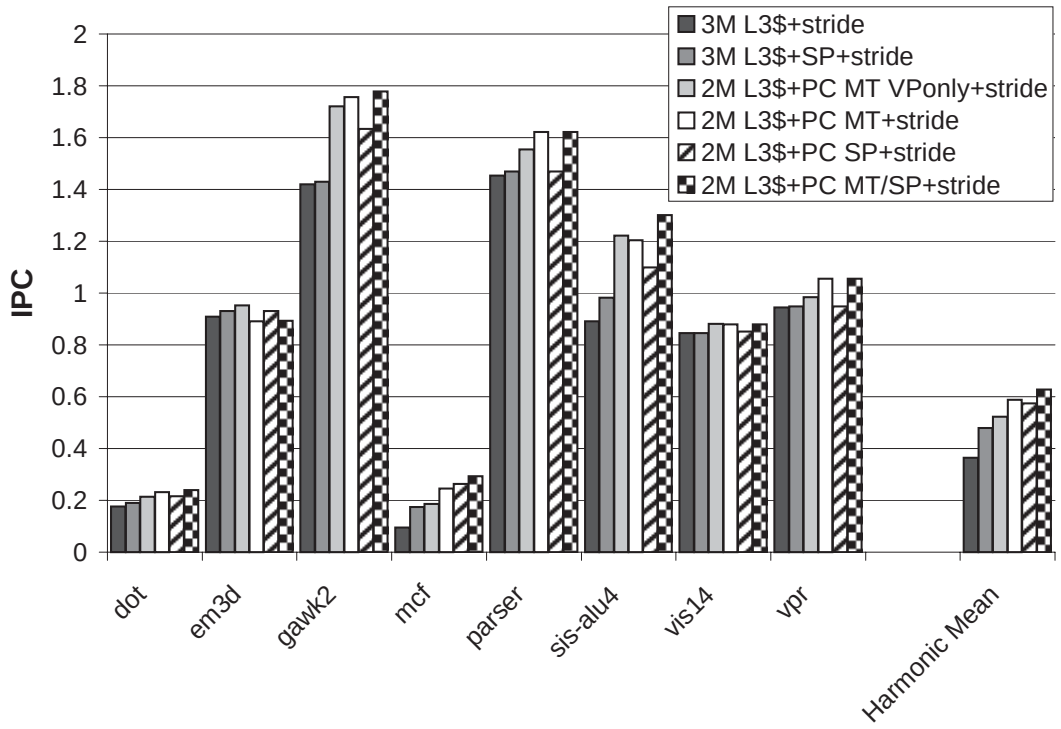

Figure VI.7: Performance results comparing a processor with a 2M 8-way L3 cache and 256k entry 4-way, 20 cycle pointer cache against a baseline stride prefetching architecture with a 3M, 12-way L3 cache. The pointer cache is used for value prediction and prefetching for all main thread (MT) results, except the third bar, where the pointer cache is only used by the main thread for value prediction. All configurations access their L3 cache in 30 cycles and utilize a stride prefetcher.

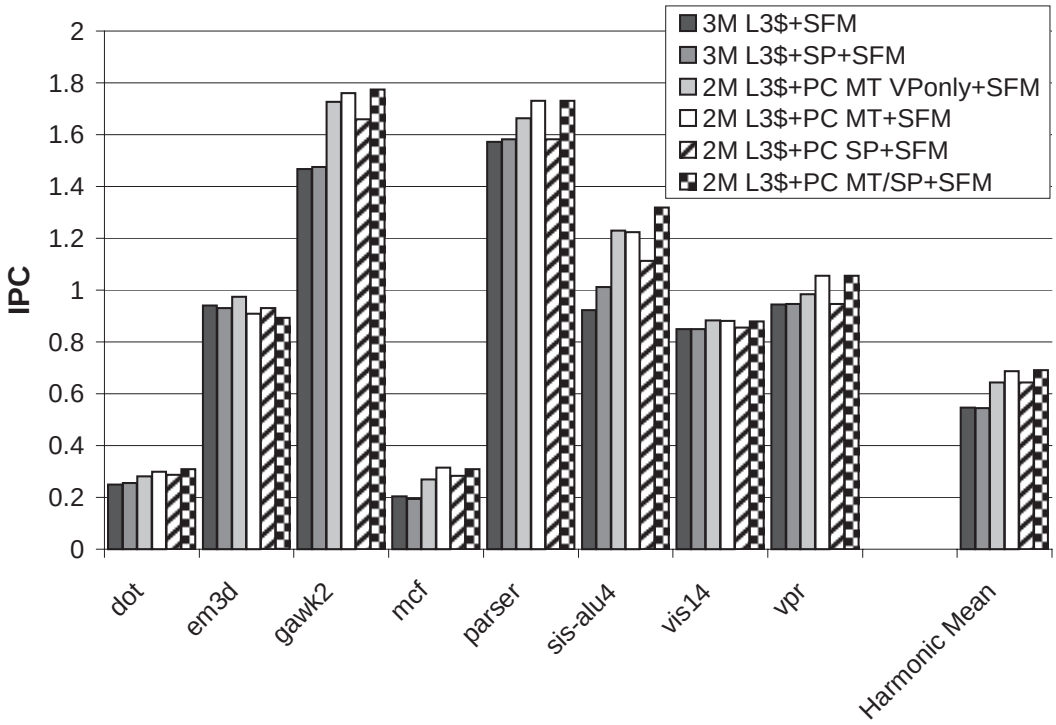

Figure VI.8: Performance results comparing a processor with a 2M 8-way L3 cache and 256k entry 4-way, 20 cycle pointer cache against a baseline stride prefetching architecture with a 3M, 12-way L3 cache. The pointer cache is used for value prediction and prefetching for all main thread (MT) results, except the third bar, where the pointer cache is only used by the main thread for value prediction. All configurations access their L3 cache in 30 cycles and utilize a SFM prefetcher.

to the performance a 1 MB pointer cache and a 2 MB L3 cache. The 1 MB pointer cache is 4-way associative and has 256K entries, each requiring 36 bits (as described in Section V.D). The 1 MB pointer cache is accessed in 20 cycles to present a fair comparison to the 3MB L3 cache with 30 cycles access time (which is the L2 access time plus 20 cycles). We present results for four pointer cache configurations: (1) allowing only the main thread to access the pointer cache to perform value prediction, (2) allowing only the main thread to access the pointer cache for value prediction and prefetching target objects, (3) allowing only the speculative threads to access the pointer cache for value prediction, and finally (4) allowing both the main thread and the speculative threads to access the pointer cache for value prediction while using it also for prefetching for the main thread.

The same results for using a SFM prefetcher in the baseline is provided in Figure VI.8.

For programs which experience a large number of serialized pointer chasing related misses, the pointer cache provides very significant performance benefits over increasing the L3 cache size; `mcf` achieves a speedup of more than 220% from using a pointer cache compared to increasing the size of the L3 cache when using the stride prefetcher in the baseline. Only `em3d`, which does not significantly benefit from the pointer cache, performs better with the larger L3 cache.

VI.C Summary

In this chapter we presented using a Pointer Cache to accelerate processing of pointer-based loads in speculative precomputation threads. This enables the speculative thread to prefetch far ahead of the main thread. When executing recurrent loads, the pointer cache is invaluable, as otherwise a cache miss by any load prevents the speculative thread from making any further progress ahead of

the main thread until the data returns. A challenge for speculatively prefetching object transitions, is dealing with objects that have several possible next transition fields. To address this, we present a new form of speculative thread that includes additional control flow in order to accurately guide which next object transition to take. This allows the speculative threads to preload the cache lines that will soon be demanded by the main thread of execution. Our results show that stride prefetching with speculative precomputation using a 2 MB L3 cache and a 1 MB pointer cache is able to achieve 53% speedup on average over stride-based prefetching with a 3 MB L3 cache. When both the main thread and speculative precomputation threads using the pointer cache for prefetching, the speedup increases to 63% on average. This shows that a pointer cache can be an attractive alternative to increasing the L3 on-chip cache to help reduce the memory bottleneck.

Chapter VII

Summary

In this thesis, we have investigated several data prefetching techniques targeting pointer-based applications. These applications are not handled well by traditional memory hierarchies and we need something new. We presented Predictor-directed Stream Buffers (PSB) and the Pointer Cache as parts of a solution. The PSB provides a simple prefetcher to target short-latency misses (i.e. misses served by on chip caches) and the Pointer Cache complements it by handling recurrent pointer loads which frequently miss all the way to main memory and slow the progress rate of the program down to the main memory access times.

Throughout the thesis we have postulated that prefetching for pointer-based applications if far from trivial. We performed a classification study of load misses to verify this presumption. We separated the miss stream into four different classes based on the data structure being accessed. These are (1) next-line loads corresponding to sequential array accesses, (2) striding loads corresponding to skipping or multi-dimensional array accesses, (3) same-object loads causing additional misses to a recently referenced heap object, or (4) pointer-based loads corresponding to misses that occur when we access the object for the first time following a pointer transition. Our results showed that even though pointer misses

constitute 50% of all the misses, when eliminated from the program, they have the potential to improve execution time by almost 90%. Furthermore, our object fan-out and pointer variability metrics also quantify the challenges met by a prefetching technique targeting these applications. In light of this study, we concluded that a prefetching solution needed special attention to address several issues when dealing with pointer-based applications:

- Breaking down the serial dependence chains imposed by pointer chasing load instructions,
- Incorporating control flow decisions into the prefetching algorithm,
- Generating future addresses fast through prediction/caching.

Our solution to these issues comprises of a multi-pronged attack which uses a small but efficient stream buffer architecture to help eliminate on-chip misses while utilizing speculative precomputation along with a specialized pointer cache to target long latency misses.

As mentioned in Chapter III, one form of hardware-based data prefetching, stream buffers, has been shown to be particularly effective due to its ability to detect data streams and run ahead of them, prefetching as it goes. Unfortunately the applicability of streaming was limited to stride intensive code.

We proposed *Predictor-Directed Stream Buffers* (*PSB*), which allows the stream buffer to follow a general address prediction stream instead of a fixed stride. Since we are particularly interested in eliminating pointer misses, we investigated using a *Stride-filtered Markov* (*SFM*) predictor to guide the stream buffers. This predictor has a two-delta stride table in front of a Markov prediction table, making it quite adept at finding both complex array access and pointer chasing behavior. A general address prediction stream complicates the allocation of both stream buffer and memory resources, because the predictions generated

will not be as reliable as prior sequential next-line and stride-based stream buffer implementations. To address this, we examine using confidence-based techniques to guide the allocation and prioritization of stream buffers and their prefetch requests. Moreover, we analyze a confidence-based choosing mechanism in order to prioritize select predictions made by the stride and the Markov predictors. Our results show, when using PSB on a benchmark suite heavy in pointer-based applications, that PSB provides a 28% speedup on average over the best previous stream buffer implementation, and an improvement of 93% over using no prefetching at all.

When we analyzed the reasons why PSB did not attain the full performance potential of eliminating pointer misses, we found that the stream buffers were not able correctly identify the next object when multiple transitions at any given object were present. Furthermore, the 4KB Markov table was not big enough to hold the complete pointer working set of most applications. Since we designed the PSB to be a light-weight prefetcher, we could not have increased the Markov predictor's size without incurring serious costs in the access time and area constraints of the stream buffer design.

Instead we proposed the *Pointer Cache*, a specialized cache that can be placed off the critical path. The pointer cache tracks pointer transitions to aid prefetching. It provides, for a given pointer's effective address, the base address of the object pointed to by the pointer. We examine using the pointer cache in a wide issue superscalar processor as a value predictor and to aid prefetching when a chain of pointers is being traversed. When a load misses in the L1 cache, but hits in the pointer cache, the first two cache blocks of the pointed to object are prefetched. In addition, the load's dependencies are broken by using the pointer cache hit as a value prediction. We found this to be an effective technique in speeding up the execution of pointer chasing code. Our results show that 50%

speedup for value prediction and 71% for value prediction and prefetching.

To add in the control flow information into the prefetching algorithm, we also explored using the pointer cache to allow speculative precomputation to run farther ahead of the main thread of execution than in prior studies. Since speculative precomputation readily executes instructions for their prefetching side effect, we added control flow instructions and instructions that compute the branch outcome to the speculative threads. We also allowed the speculative threads to recover from branch mispredictions to increase their lifetime and consequently their usefulness. We utilized a synchronization mechanism through a FIFO branch outcome queue to enforce the speculative thread down the path that the main thread will soon follow. Furthermore, previously proposed thread-based prefetchers were limited in how far they can run ahead of the main thread when traversing a chain of recurrent dependent loads. When combined with the pointer cache however, a speculative thread makes better progress ahead of the main thread, rapidly traversing data structures in the face of cache misses caused by pointer transitions. Overall we found using the pointer cache to aid speculative precomputation provides a 53% speedup. Naturally, the best combination is allowing both the main thread and the speculative threads access to the pointer cache, which provides a 63% speedup when compared to the 16% speedup we get when we utilize the pointer cache area as part of a regular L3 cache.

Another interesting result of this dissertation is quantifying the benefits of using speculative precomputation in conjunction with stream buffers. We investigated two different configurations, one using a stride predictor, and the other using the SFM predictor we proposed. We found that over a baseline speculative precomputation prefetching architecture, adding a stride prefetcher provides 25% speedup compared to the 40% improvement in execution time when we add an SFM prefetcher.

Chapter VIII

Future Work

By no measure have we completely eliminated the ill effects of cache misses on program performance. This chapter provides an overview of our vision for future research avenues related to data prefetching.

VIII.A Future Work on PSB

Even though the PSB architecture is fine-tuned to achieve the best possible performance for the set of applications we examined, there is still room for improvement in:

- A better confidence update policy. We can update confidence counters in stream buffers based on the amount of latency they hide instead of a binary value depending on a hit or a miss.
- A better address predictor. As we mentioned in Chapter IV, the major shortcoming of the PSB architecture is its control-flow insensitive nature. To address this, we can include branch history information into the indexing function of a context predictor to guide prefetching down the traversal path more accurately.

- A smarter prefetcher. Most linked data structures utilize objects that span multiple cache lines. We can annotate the address predictor with the maximum dynamic offset observed by each base address to prefetch the complete object at once. To reduce the impact on the data buses, we can further optimize this method by only prefetching blocks that are accessed frequently (i.e. hot blocks of the object).

Investigating the use of our PSB architecture for instruction prefetching is also an interesting research topic. Annavaram et al. [3] build call graphs to implement effective instruction prefetching for database applications. One could achieve similar results using a context predictor to predict when calls occur and the next-line or stride predictor to prefetch consecutive instruction blocks that are consumed the majority of the time.

VIII.B Future Work on Pointer Cache Assisted Prefetching

There are a number of hardware challenges that future microarchitects will need to face to design the next generation of microprocessors. One of the primary means of increasing the speed of microprocessors has been to scale the feature size of the current technology. As feature sizes continue to shrink, it has become evident that wire latencies are not scaling with transistor latencies. This trend has been termed the *interconnect scaling bottleneck* [7, 8].

VIII.B.1 Pointer Cache Chaining

As the clock speed of the processor continues to increase, correspondingly dropping the cycle time of the processor, the addition of large memory structures (e.g. pointer cache) to an architecture may be constrained by how

fully they can be pipelined or how small they can get while still providing benefits.

One aspect of our future work explores reducing the size of the pointer cache while maintaining its latency hiding capabilities. Just as in jump pointer chaining [54], we can link every Nth object in the pointer cache effectively reducing its size to $1/N$th of its original size and still providing the same latency tolerance. In this scheme N would have to be adjusted with respect to the amount of work that exists in the program before moving on to a new object. Once the prefetch stream gets far enough ahead of the execution stream, we can allow sequential prefetching of objects since we can tolerate the miss latency as long as the miss serviced before the execution stream needs the relevant data.

VIII.B.2 Memory-backed Prediction Structures

It is expected that by 2014 the memory density will be roughly 30 times what it is today. The average chip will have a capacity of approximately 25 GB. To maintain the same level of performance, one area we plan to investigate is utilizing some of this vast memory as a secondary storage for predictor data which can then be fetched (or even prefetched) into the smaller and faster first level predictor tables when necessary.

One could implement the pointer cache as a memory-backed predictor. We can divide data memory into chunks and measure the relative access frequencies of these data blocks. When observed over fixed intervals, this vector of access frequencies form a data signature for a given interval. We could use these signatures to build up a history and use context prediction to predict the next phase signature given a set of past phase signatures. This would separate the program into phases based on the data segments being accessed and it could be used to group prediction information (such as pointer cache entries) that are related to

that phase to be stored and retrieved from memory backed predictors when the same phase is predicted to occur in the future.

VIII.B.3 Early Access to Predictors

In [50], Reinman et al. propose a decoupled front-end architecture that uses a fetch target queue (FTQ), which is a small FIFO queue used to store predicted fetch addresses from the branch prediction unit. The FTQ provides a decoupled front-end that allows the branch predictor and instruction cache to run more independently. The FTQ can also be used to index into other PC-based predictors (such as value or address predictors) further ahead in the pipeline.

One way to take advantage of the FTQ is to build the pointer cache as a PC-indexed value predictor. In this design, the pointer cache can still be fairly large. The fetch addresses stored in the FTQ can initiate the predictor access earlier in the pipeline. This can allow for larger and more accurate predictors to be used, and even allow these predictors to be located off-chip.

Another possibility is to implement the pointer cache as a global stride-based value predictor as proposed by Zhou et al. in [79]. Their gDiff predictor exploits stride-based global locality. Their experiments show that there exists very strong stride-based locality in global value histories; and, many instructions that are hard-to-predict using local history-based predictors become highly predictable using global history-based predictors. They also demonstrate the use of the gDiff predictor in exploring the global stride locality in the load address stream. Their results show that the gDiff predictor achieves higher coverage and accuracy in predicting load addresses as well as predicting addresses of missing loads only when compared with local predictors or a Markov predictor with larger prediction tables.

VIII.B.4 Multi-pronged Prefetchers

Within the next five years, memory latencies are expected to be in the several thousand cycle range. Already about 80% of a chip's die area is devoted to memory. This is only expected to increase with each generation of process technology. Even though there will be a significant amount of on-chip storage, the latencies to access various components of this storage will continue to be a dominating factor in performance. To this effect, we need a multi-pronged attack to target both on-chip (those that are serviced by the on-chip L2 or L3 caches) and off-chip misses (those that are serviced by off-chip caches or the main memory). This solution should include a simple and fast component that focuses on eliminating latency due to data that exhibits temporal locality within the on-chip caches, and a long-range approach (such as the push prefetching model where there is a prefetching engine in memory) targeting misses that go off-chip.

Bibliography

[1] T. Alexander and G. Kedem. Distributed prefetch-buffer/cache design for high performance memory systems. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture*, February 1996.

[2] M Annavaram, J. Patel, and E. Davidson. Data prefetching by dependence graph precomputation. In *28th Annual International Symposium on Computer Architecture*, June 2001.

[3] M. Annavaram, J. M. Patel, and E. S. Davidson. Call graph prefetching for database applications. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 281–290, January 2001.

[4] M. Bekerman, S. Jourdan, R. Ronen, G. Kirshenboim, L. Rappoport, A. Yoaz, and U. Weiser. Correlated load-address predictors. In *26th Annual International Symposium on Computer Architecture*, May 1999.

[5] A. Berrached, L. Coraor, and P. Hulina. A decoupled access/execute architecture for efficient accesss of structured data. In *In the Hawaii International Conference on System Services*, January 1993.

[6] B. Black, B. Mueller, S. Postal, R. Rakvic, N. Utamaphethai, and J. P. Shen. Load execution latency reduction. In *12th International Conference on Supercomputing*, June 1998.

[7] M. Bohr. Interconnect scaling - the real limiter to high-performance ulsi. In *Tech. Dig. of the International Electron Devices Meeting*, pages 241–244, December 1995.

[8] M. Bohr. Silicon trends and limits for advanced microprocessors. *Communications of the ACM*, 41(3):80–87, March 1998.

[9] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[10] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, October 1998.

[11] B. Calder, G. Reinman, and D. Tullsen. Selective value prediction. In *26th Annual International Symposium on Computer Architecture*, May 1999.

[12] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, April 1991.

[13] M.J. Charney and T.R. Puzak. Prefetching and memory system behavior of the spec95 benchmark suite. *IBM Journal of Research and Development*, 41(3), May 1997.

[14] M.J. Charney and A.P. Reeves. Generalized correlation based hardware prefetching. Technical Report EE-CEG-95-1, Cornell University, February 1995.

[15] T.F. Chen and J.L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 51–61, October 1992.

[16] T.F. Chen and J.L. Baer. Effective hardware-based data prefetching for high performance processors. *IEEE Transactions on Computers*, 5(44):609–623, May 1995.

[17] C. Chi and C. Cheung. Hardware-driven prefetching for pointer data references. In *International Conference on Supercomputing*, pages 377–384, June 1998.

[18] T. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the SIGPLAN'02 Conference on Programming Language Design and Implementation*, June 2002.

[19] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, May 1999.

[20] G. Chrysos and J. Emer. Memory dependence prediction using store sets. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[21] J. Collins, S. Sair, B. Calder, and D. Tullsen. Pointer cache assisted prefetching. In *35th International Symposium on Microarchitecture*, November 2002.

[22] J. Collins, D. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *34th International Symposium on Microarchitecture*, December 2001.

[23] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *28th Annual International Symposium on Computer Architecture*, June 2001.

[24] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, October 2002.

[25] R. J. Eickemeyer and S. Vassiliadis. A load instruction unit for pipelined processors. *IBM Journal of Research and Development*, 37:547–564, July 1993.

[26] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[27] K. Farkas and N. Jouppi. How useful are non-blocking loads, stream buffers and speculative execution in multiple issue processors? In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 78–89, January 1995.

[28] M. Farrens and A.Pleszkun. Implementation of the pipe processor. *IEEE Computer*, January 1991.

[29] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. EE Department TR 1080, Technion - Israel Institue of Technology, November 1996.

[30] J. Gonzalez and A. Gonzalez. Speculative execution via address prediction and data prefetching. In *11th International Conference on Supercomputing*, pages 196–203, July 1997.

[31] J. Gonzalez and A. Gonzalez. The potential of data value speculation to boost ilp. In *12th International Conference on Supercomputing*, 1998.

[32] G.P. Jones and N.P. Topham. A comparison of data prefetching on an access decoupled and superscalar machine. In *30th International Symposium on Microarchitecture*, December 1997.

[33] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[34] 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.

[35] M. Karlsson, F. Dahgren, and P. Stenstrom. A prefetching technique for irregular accesses to linked data structures. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, January 2000.

[36] A. Lai, C. Fide, and B. Falsafi. Dead-block prediction and dead-block correlating prefetchers. In *28th Annual International Symposium on Computer Architecture*, June 2001.

[37] S. Lee, Y. Wang, and P. Yew. Decoupled value prediction on trace processors. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 231–240, January 2000.

[38] M. Lipasti, W. Schmidt, S. Kunkel, and R. Roediger. Spaid: Software prefetching in pointer and call intensive environments. In *28th International Symposium on Microarchitecture*, November 1995.

[39] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *17th International Conference on Architectural Support for Programming Languages and operating Systems*, pages 138–147, October 1996.

[40] M.H. Lipasti and J.P. Shen. Exceeding the dataflow limit via value prediction. In *29th International Symposium on Microarchitecture*, December 1996.

[41] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *28th Annual International Symposium on Computer Architecture*, June 2001.

[42] C.-K. Luk and T. C. Mowry. Compiler based prefetching for recursive data structures. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[43] C.-K. Luk and T. C. Mowry. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In *31st International Symposium on Microarchitecture*, December 1998.

[44] C.-K. Luk and T. C. Mowry. Memory forwarding: Enabling aggressive layout optimizations by guaranteeing the safety of data relocation. In *ISCA99*, pages 88–99, May 1999.

[45] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Equipment Corporation, Western Research Lab, June 1993.

[46] A. Moshovos, D. Pnevmatikatos, and A. Baniasadi. Slice processors: An implementation of operation-based prediction. In *International Conference on Supercomputing*, June 2001.

[47] T.C. Mowry, M.S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, October 1992.

[48] T.C. Mowry and C.-K. Luk. Predicting data cache misses in non-numeric applications through correlation profiling. In *27th International Symposium on Microarchitecture*, September 1997.

[49] S. Palacharla and R. Kessler. Evaluating stream buffers as secondary cache replacement. In *21st Annual International Symposium on Computer Architecture*, April 1994.

[50] G. Reinman, T. Austin, and Brad Calder. A scalable front-end architecture for fast instruction delivery. In *26th Annual International Symposium on Computer Architecture*, May 1999.

[51] G. Reinman, B. Calder, and T. Austin. Fetch-directed instruction prefetching. In *32nd International Symposium on Microarchitecture*, November 1999.

[52] G. Reinman, B. Calder, and T. Austin. A power efficient speculative fetch architecture. Technical Report UCSD-CS2000-0657, University of Califonia, San Diego, June 2000.

[53] A. Roth, A. Moshovos, and G. Sohi. Dependence based prefetching for linked data structures. In *Eigth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[54] A. Roth and G. Sohi. Effective jump-pointer prefetching for linked data structures. In *26th Annual International Symposium on Computer Architecture*, May 1999.

[55] A. Roth and G. Sohi. Speculative data-driven multithreading. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, January 2001.

[56] A. Roth, C. B. Zilles, and G. S. Sohi. Micro-architectural miss/execute decoupling. In *International Workshop on Memory access Decoupled Architectures and Related Issues*, October 2000.

[57] S. Sair, T. Sherwood, and B. Calder. Quantifying load stream behavior. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, February 2002.

[58] S. Sair, T. Sherwood, and B. Calder. A decoupled predictor-directed stream prefetching architecture. *IEEE Transactions on Computers*, 52(3):260–276, 2003.

[59] A. Saulsbury, F. Dahlgren, and P. Stenstrom. Recency-based TLB preloading. In *27th Annual International Symposium on Computer Architecture*, June 2000.

[60] Y. Sazeides and J. E. Smith. The predictability of data values. In *30th International Symposium on Microarchitecture*, pages 248–258, December 1997.

[61] Y. Sazeides and J. E. Smith. Modeling program predictability. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[62] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. In *29th Annual International Symposium on Computer Architecture*, May 2002.

[63] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, University of Califonia, San Diego, August 1999.

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

[65] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, October 2002. http://www.cs.ucsd.edu/users/calder/simpoint/.

[66] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *33rd International Symposium on Microarchitecture*, December 2000.

[67] P. Shivakumar and N. Jouppi. Cacti version 3.0. http://research.compaq.com/wrl/people/jouppi/CACTI.html, August 2001.

[68] J. E. Smith and W.-C. Hsu. Prefetching in supercomputer instruction caches. In *Proceedings of Supercomputing*, November 1992.

[69] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *29th Annual International Symposium on Computer Architecture*, May 2002.

[70] Y. Song and M. Dubois. Assisted execution. Technical Report CENG 98-25, University of Southern California, October 1988.

[71] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[72] D.M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, December 1996.

[73] D.M. Tullsen, S.J. Eggers, J. Emer, H.M. Levy, J. Lo, and R. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[74] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[75] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th Annual International Symposium on Microarchitecture*, December 1997.

[76] Wm. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. Technical Report CS-94-48, University of Utah, 1, 1994.

[77] C. Yang and A. Lebeck. Push vs. pull: Data movement for linked data structures. In *International Conference on Supercomputing*, June 2000.

[78] Z. Zhang and J. Torrellas. Speeding up irregular applications in shared-memory multiprocessors: Memory binding and group prefetching. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[79] H. Zhou, J. Flanagan, and T.M. Conte. Detecting global stride locality in value streams. In *30th Annual International Symposium on Computer Architecture*, June 2003.

[80] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *28th Annual International Symposium on Computer Architecture*, June 2001.