DTIC FILE COPY

# Efficient Analysis of Caching Systems

By

James Gordon Thompson

B.S.S.E. (University of Florida) 1971
M.S.S.E. (University of Florida) 1972

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

GRADUATE DIVISION

OF THE

UNIVERSITY OF CALIFORNIA, BERKELEY

AD-A196 912

DTIC
S ELECTE D
AUG 0 3 1988
CAD

Approved: ........................................ 9/22/87
Chairman Date

John K. Ousterhout ........................ 9/22/87

Ronald W. Wolff ........................ 9/23/87

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

88 8 02 184

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

## REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFIT/CI/NR 88-93
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): EFFICIENT ANALYSIS OF CACHING SYSTEMS
5. TYPE OF REPORT & PERIOD COVERED: PHD ~~MS~~ THESIS
6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): JAMES GORDON THOMPSON
8. CONTRACT OR GRANT NUMBER(s):
9. PERFORMING ORGANIZATION NAME AND ADDRESS: AFIT STUDENT AT: UNIVERSITY OF CALIFORNIA BERKELEY
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
11. CONTROLLING OFFICE NAME AND ADDRESS:
12. REPORT DATE: 1988
13. NUMBER OF PAGES: 239
14. MONITORING AGENCY NAME & ADDRESS (If different from Controlling Office):

AFIT/NR
Wright-Patterson AFB OH 45433-6583

15. SECURITY CLASS. (of this report): UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report): DISTRIBUTED UNLIMITED: APPROVED FOR PUBLIC RELEASE

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report): SAME AS REPORT

18. SUPPLEMENTARY NOTES:

Approved for Public Release: IAW AFR 190-1
LYNN E. WOLAVER Lynn Wolaver 19 Jul 88
Dean for Research and Professional Development
Air Force Institute of Technology
Wright-Patterson AFB OH 45433-6583

19. KEY WORDS (Continue on reverse side if necessary and identify by block number):

20. ABSTRACT (Continue on reverse side if necessary and identify by block number): ATTACHED

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

# Efficient Analysis of Caching Systems

*James Gordon Thompson*

## ABSTRACT

This dissertation describes innovative techniques for efficiently analyzing a wide variety of cache designs, and uses these techniques to study caching in a network file system. The techniques are significant extensions to the stack analysis technique (Mattson et al., 1970) which computes the read miss ratio for all cache sizes in a single trace-driven simulation. Stack analysis is extended to allow the one-pass analysis of:

1) writes in a write-back cache, including periodic write-back and deletions, important factors in file system cache performance.
2) sub-block or sector caches, including load-forward prefetching.
3) multi-processor caches in a shared-memory system, for an entire class of consistency protocols, including all of the well-known protocols.
4) client caches in a network file system, using a new class of consistency protocols.

The techniques are completely general and apply to all levels of the memory hierarchy, from processor caches to disk and file system caches. The dissertation also discusses the use of hash tables and binary trees within the simulator to further improve performance for some types of traces. Using these techniques, the performance of all cache sizes can be computed in little more than twice the time required to simulate a single cache size, and often in just 10% more time.

In addition to presenting techniques, this dissertation also demonstrates their use by studying client caching in a network file system. It first reports the extent of file sharing in a UNIX environment, showing that a few shared files account for two-thirds of all accesses, and nearly half of these are to files which are both read and written.

It then studies different cache consistency protocols, write policies, and fetch policies, reporting the miss ratio and file server utilization for each. Four cache consistency protocols are considered: a polling protocol that uses the server for all consistency controls; a protocol designed for single-user files; one designed for read-only files; and one using write-broadcast to maintain consistency. It finds that the choice of consistency protocol has a substantial effect on performance; both the read-only and write-broadcast protocols showed half the misses and server load of the polling protocol. The choice of write or fetch policy made a much smaller difference.

Approved: 9/22/87

Prof. Alan Jay Smith Date

# Efficient Analysis of Caching Systems

*James Gordon Thompson*

## ABSTRACT

This dissertation describes innovative techniques for efficiently analyzing a wide variety of cache designs, and uses these techniques to study caching in a network file system. The techniques are significant extensions to the stack analysis technique (Mattson et al., 1970) which computes the read miss ratio for all cache sizes in a single trace-driven simulation. Stack analysis is extended to allow the one-pass analysis of:

1) writes in a write-back cache, including periodic write-back and deletions, important factors in file system cache performance;
2) sub-block or sector caches, including load-forward prefetching;
3) multi-processor caches in a shared-memory system, for an entire class of consistency protocols, including all of the well-known protocols; and
4) client caches in a network file system, using a new class of consistency protocols.

The techniques are completely general and apply to all levels of the memory hierarchy, from processor caches to disk and file system caches. The dissertation also discusses the use of hash tables and binary trees within the simulator to further improve performance for some types of traces. Using these techniques, the performance of all cache sizes can be computed in little more than twice the time required to simulate a single cache size, and often in just 10% more time.

In addition to presenting techniques, this dissertation also demonstrates their use by studying client caching in a network file system. It first reports the extent of file sharing in a UNIX environment, showing that a few shared files account for two-thirds of all accesses, and nearly half of these are to files which are both read and written.

It then studies different cache consistency protocols, write policies, and fetch policies, reporting the miss ratio and file server utilization for each. Four cache consistency protocols are

considered: a polling protocol that uses the server for all consistency controls; a protocol designed for single-user files; one designed for read-only files; and one using write-broadcast to maintain consistency. It finds that the choice of consistency protocol has a substantial effect on performance; both the read-only and write-broadcast protocols showed half the misses and server load of the polling protocol. The choice of write or fetch policy made a much smaller difference.

Approved: ______________________________
Prof. Alan Jay Smith Date

DTIC
COPY
INSPECTED
6

Accesion For
NTIS CRA&I ☑
DTIC TAB ☐
Unannounced ☐
Justification ________
By ________
Distribution /
Availability Codes
Dist | Avail and/or Special
A-1

# Efficient Analysis of Caching Systems

Copyright © 1987

James Gordon Thompson

## Acknowledgements

Many people deserve thanks for their help in making this dissertation possible. I want to particularly recognize the contributions of two special people — my wife, Bev, and my advisor, Alan Smith. Bev held our family together while I ''played student''; caring for our children, running the house, and sacrificing her own ambitions to mine. This is a debt I can never repay.

Alan is a brilliant advisor whose insights kept my research on course. His comments were often slow in coming, sometimes biting, but invariably were on the mark and led to a clearer understanding or articulation of the problems and solutions. If my reasoning abilities were half as good as Alan's intuition this dissertation would have been completed long ago.

I would also like to thank my other two readers for wading through what I know is an imposing document, and for providing useful comments and suggestions. I particularly want to thank John Ousterhout for getting me interested in this topic during my first semester at Berkeley. John also gets my thanks for letting me play with Gremlin.

I would like to thank the United States Air Force for giving me the opportunity to return to school. My thanks also to the unknown members of the Berkeley Computer Science Division admissions committee who decided to take a chance on me, giving me the opportunity to study at a world-class school.

Many others have helped me along the way and deserve thanks: Songnian Zhou and Herve' Dacosta for providing excellent traces; Mark Hill for suggesting the sub-block caching technique and providing comments on early versions of Chapter 3; Frank Olken for his insights into the techniques of Chapter 4 and for his valuable comments on the chapter; and Oivind Kure for many thoughtful discussions whenever I ran into technical roadblocks, and for loaning me his car. Others, too numerous to mention, have helped in many ways, and receive my sincere thanks.

My special thanks go to my mother and father for their inspiration, love, and encouragement in this and all other things I have attempted. Any success I have achieved in life has been inspired by their model.

Finally, I would like to again thank my family, Bev, Jill, and Scott, for their support and understanding during the many times I was ''the man who came to dinner''. Without their patient love I could never have finished — and would not have wanted to try.

The material presented here is based on research supported in part by the National Science Foundation under grants CCR-8202591 and MIP-8713274, by the State of California under the MICRO program and by IBM Corporation, Digital Equipment Corporation, Hewlett Packard Corporation, and Signetics Corporation.

# Table of Contents

Acknowledgements .......... i

Table of Contents .......... iii

List of Figures .......... xi

List of Tables .......... xiv

**1. Introduction** .......... 1

**2. Background** .......... 6

2.1. Cache Memories .......... 6

2.2. Metrics .......... 10

2.3. Trace Driven Simulation .......... 13

2.4. Non-Stack Algorithms .......... 18

2.5. Extensions to Stack Analysis .......... 20

**3. Efficient (Stack) Algorithms for Analysis of Write-Back** .......... 22

3.1. Summary .......... 22

3.2. Introduction .......... 22

3.3. Write-Back Stack Algorithm .......... 24

3.3.1. The Write-Back Problem .......... 24

3.3.2. A Non-stack Algorithm .......... 24

3.3.3. Dirty Set Inclusion Property .......... 26

3.3.4. Writes Avoided .......... 27

3.3.5. Dirty Push Computation .......... 29

3.3.6. Warm Start .......... 30

3.4. Extensions ........ 32

3.4.1. Write-Through ........ 33

3.4.2. Periodic Write-Back ........ 34

3.4.3. Deletions ........ 34

3.4.4. Flush Back ........ 37

3.5. Sector Cache Simulation ........ 37

3.5.1. Background ........ 38

3.5.2. The Stack Simulation Problem ........ 39

3.5.3. Extensions ........ 41

3.5.3.1. Write Back ........ 41

3.5.3.2. Load Forward ........ 42

3.6. Experimental Results ........ 47

3.6.1. The Trace Data ........ 47

3.6.2. Run-Time Comparison ........ 51

3.6.3. Write-Back Probability ........ 52

3.7. Conclusions ........ 57

**4. Improving the Efficiency of Stack Simulation ........ 58**

4.1. Summary ........ 58

4.2. Introduction ........ 58

4.3. Review of Stack Implementations ........ 60

4.3.1. Linked List ........ 61

4.3.2. Hash Table ........ 63

4.3.3. Bennett and Kruskal's Algorithm ........ 64

4.3.3.1. Bounded Space Bennett and Kruskal's Algorithm ........ 65

4.3.4. AVL Tree .......... 69

4.3.5. Deletions .......... 74

4.4. Hybrid Method .......... 76

4.5. Summary of Implementations .......... 80

4.6. Experimental Performance .......... 80

4.6.1. The Simulator .......... 81

4.6.2. The Trace Data .......... 81

4.6.3. Trace File Characteristics .......... 82

4.6.4. Distance Distributions .......... 85

4.6.5. Comparison of Methods .......... 89

4.7. Conclusions .......... 92

**5. One-Pass Analysis Techniques for Multi-Processor Caches** .......... 93

5.1. Summary .......... 93

5.2. Introduction .......... 93

5.3. One-Pass Algorithms .......... 95

5.3.1. Review of Stack Algorithms .......... 95

5.3.2. Efficient One-Pass Algorithms .......... 96

5.4. Metrics for Multi-processor Simulations .......... 98

5.5. Cache Consistency Protocols .......... 99

5.6. One-Pass Algorithms for MOESI Protocols .......... 105

5.6.1. Initial Assumptions .......... 105

5.6.1.1. Synchronous Reference Streams .......... 105

5.6.1.2. Replacement Priority .......... 107

5.6.1.3. Consistent Rule Usage .......... 107

5.6.2. A Simple Example - the Berkeley Protocol .......... 108

5.6.3. A Second Protocol .......... 113

5.6.4. A Complex Example - the Dragon Protocol .......... 116

5.6.5. Related Memory Sizes .......... 123

5.7. The MOESI Protocols .......... 125

5.7.1. State Variables .......... 125

5.7.2. Further Restrictions .......... 126

5.7.2.1. Berkeley Protocol .......... 127

5.7.2.2. Invalidation .......... 130

5.7.2.3. Write-Through Rules .......... 133

5.7.2.4. Pass/Copy-Back .......... 134

5.7.3. Proof of One-Pass Algorithms .......... 135

5.7.3.1. Dirty Inclusion .......... 136

5.7.3.2. Validity Inclusion .......... 137

5.7.3.3. Single Dirty Cache .......... 138

5.7.3.4. Sharing Inclusion .......... 139

5.7.3.5. Single Exclusive Cache .......... 141

5.7.4. Variables (Revisited) .......... 142

5.8. Formal Algorithms .......... 143

5.8.1. Read Algorithm .......... 145

5.8.2. Write Algorithm .......... 147

5.8.2.1. "Preferred" Protocol Algorithm .......... 148

5.8.2.2. Berkeley Protocol Algorithm .......... 148

5.8.2.3. Dragon Protocol Algorithm .......... 149

5.8.2.4. Illinois Protocol ........ 150

5.8.2.5. Write-Through Protocol Algorithm ........ 151

5.8.3. Pass Algorithm ........ 151

5.8.4. Flush Algorithm ........ 152

5.9. Other Protocols ........ 153

5.9.1. Firefly Protocol ........ 153

5.9.2. Write Once ........ 156

5.10. Implementation Issues - Partial Invalidation ........ 159

5.11. Conclusions ........ 161

**6. A Class of File System Consistency Protocols** ........ 162

6.1. Summary ........ 162

6.2. Introduction ........ 162

6.3. Differences Between File System and Processor Caches ........ 163

6.4. Concurrent Access ........ 165

6.5. Existing Schemes ........ 168

6.5.1. Cedar ........ 168

6.5.2. Andrew ........ 168

6.5.3. SUN Network File System (NFS) ........ 169

6.5.4. Sprite ........ 169

6.6. A Class of Consistency Control Protocols ........ 170

6.6.1. Basic Approach ........ 170

6.6.2. State Attributes ........ 171

6.6.3. States ........ 172

6.6.4. Client Actions ........ 174

6.6.5. Server Actions ........ 176
6.6.6. Exclusive Cache Actions ........ 179
6.7. Selected Protocols ........ 181
6.7.1. Sprite Protocol ........ 182
6.7.1.1. Variation: Notification of Write Flush ........ 182
6.7.2. Exclusive Protocol ........ 185
6.7.2.1. Variation: Lazy Invalidation ........ 185
6.7.3. Read-Only Protocol ........ 186
6.7.4. Write Broadcast Protocol ........ 188
6.8. One-Pass Analysis ........ 190
6.8.1. Compatible States ........ 190
6.8.2. Assumptions and Restrictions ........ 192
6.8.2.1. Invalidation ........ 192
6.8.2.2. Write Broadcast ........ 192
6.8.2.3. Related Memory Sizes ........ 193
6.8.3. Proof of One-Pass Algorithm ........ 194
6.9. Conclusions ........ 195
**7. Evaluation of Client Caching in a Network File System** ........ 196
7.1. Summary ........ 196
7.2. Introduction ........ 197
7.2.1. Relation to Other Work ........ 197
7.3. The Traces ........ 198
7.4. File Sharing ........ 203
7.5. The Simulator ........ 204

7.6. Consistency Policies .......... 207

7.6.1. Sprite Protocol .......... 207

7.6.2. Exclusive Protocol .......... 208

7.6.3. Read-Only Protocol .......... 209

7.6.4. Write Broadcast Protocol .......... 212

7.6.5. Optimal Protocol .......... 213

7.7. Write Policy .......... 214

7.8. Fetch Policy .......... 218

7.9. Conclusions .......... 223

**8. Conclusions and Future Directions** .......... 224

8.1. One-Pass Analysis .......... 224

8.2. Simulation Techniques .......... 225

8.3. Results .......... 226

8.3.1. Network File System Consistency Protocols .......... 226

8.3.2. Other File System Simulation Results .......... 226

8.3.3. File Sharing .......... 227

8.3.4. Probability of Write-Back .......... 227

8.4. Future Work .......... 227

8.4.1. Adaptive MOESI Protocols .......... 227

8.4.2. Broadcast File System Protocols .......... 228

8.4.3. Name Caching .......... 228

8.4.4. Server Caching .......... 229

8.4.5. More Data .......... 230

8.5. Summary .......... 231

9. References .......... 232

# List of Figures

Figure 2.1: Examples of stack maintenance using LRU replacement .............................. 14

Figure 2.2: Examples of stack maintenance using a stack replacement algorithm ............ 15

Figure 2.3: Cache contents using Least Frequently Used policy .................................... 18

Figure 2.4: Cache contents using one-block prefetch ................................................... 19

Figure 3.1: The count of dirty pushes may be negative after warm start ......................... 31

Figure 3.2: Revised count of dirty pushes after warm start ........................................... 32

Figure 3.3: A gap "jumps" down the stack .................................................................. 35

Figure 3.4: Alternative layouts for a block/sector in a cache ......................................... 38

Figure 3.5: Valid levels in a sector cache ..................................................................... 41

Figure 3.6: A situation where load-forward can violate inclusion .................................. 44

Figure 3.7: A situation where load-forward preserves inclusion .................................... 44

Figure 3.8: Probability of a dirty push ......................................................................... 53

Figure 4.1: Abstract Stack Analysis Algorithm ............................................................ 61

Figure 4.2: Bennett and Kruskal's (B&K) tree representation ....................................... 66

Figure 4.3: Example of update to B&K tree ................................................................. 67

Figure 4.4: B&K tree of Figure 4.3 after compaction ................................................... 68

Figure 4.5: Binary tree representation of memory stack ............................................... 69

Figure 4.6: Single and 2-rotations in AVL tree ............................................................ 72

Figure 4.7: AVL tree changes due to reference string {ABCDEFADBG} ...................... 73

Figure 4.8: Simulation time for varying values of B .................................................... 79

Figure 4.9: Stack distance distributions for various traces ............................................ 88

Figure 5.1: Relationships between MOESI states ......................................................... 102

Figure 5.2: The order of events can affect bus activity ........ 106

Figure 5.3: Unbounded regions result from inconsistent rule usage ........ 108

Figure 5.4: Examples of actions using the Berkeley Protocol ........ 111

Figure 5.5: Examples of possible pairs of configurations ........ 112

Figure 5.6: Actions using the second protocol ........ 114

Figure 5.7: Actions using the Dragon protocol ........ 118

Figure 5.8: The simple Dragon configurations produce others ........ 121

Figure 5.9: The general shared Dragon configuration produces others ........ 122

Figure 5.10: Possible cache size relationships ........ 124

Figure 5.11: Configurations using equal cache sizes ........ 124

Figure 5.12: Possible configurations assuming inclusion ........ 126

Figure 5.13: The Berkeley protocol violates sharing inclusion ........ 127

Figure 5.14: Whether state E or $E^S$ depends on the variable $E^S$ ........ 130

Figure 5.15: Violation of inclusion using invalidation rules ........ 131

Figure 5.16: Violation of dirty inclusion using write-through ........ 133

Figure 5.17: Pass can violate sharing inclusion ........ 134

Figure 5.18: Effect of actions on SPass region ........ 135

Figure 5.19: General One-Pass Multi-processor Algorithm ........ 144

Figure 5.20: The effect of Reads on the sharing level ........ 146

Figure 5.21: Read Algorithm ........ 147

Figure 5.22: Main Write algorithm ........ 147

Figure 5.23: Write algorithm for Preferred protocol ........ 149

Figure 5.24: Write algorithm for Berkeley protocol ........ 149

Figure 5.25: Write algorithm for Dragon protocol ........ 150

Figure 5.26: Write algorithm for Illinois protocol ........ 151

Figure 5.27: Write algorithm for Write-Through protocol ........ 151

Figure 5.28: Pass algorithm ........ 152

Figure 5.29: Flush algorithm ........ 152

Figure 5.30: Violation of the dirty level caused by Firefly protocol ........ 154

Figure 5.31: There is a single dirty region for the Firefly protocol ........ 155

Figure 5.32: Write algorithm for Firefly protocol ........ 156

Figure 5.33: Write-once violates dirty inclusion ........ 157

Figure 5.34: Write-once violates dirty inclusion ........ 158

Figure 5.35: Write-Once protocol algorithms ........ 159

Figure 5.36: Implementation of partial invalidation ........ 160

Figure 6.1: Relationships between attributes and states ........ 173

Figure 6.2: Dirty inclusion is violated by write broadcast ........ 193

Figure 7.1: Sprite protocol ........ 208

Figure 7.2: Exclusive protocol ........ 209

Figure 7.3: Read-Only protocol ........ 210

Figure 7.4: Write broadcast protocol ........ 213

Figure 7.5: Comparison of consistency protocols ........ 215

Figure 7.6: Write-through transfer ratio ........ 216

Figure 7.7: Comparison of write policies ........ 217

Figure 7.8: One-block prefetch policies ........ 219

Figure 7.9: Full-file prefetch policies ........ 221

Figure 7.10: Effect of varying the definition of a ''small'' file ........ 222

# List of Tables

Table 3.1: General Trace File Characteristics ........ 49

Table 3.2: Comparison of Execution Times ........ 52

Table 3.3: Probability of Dirty Push - Program Address Traces With Flushing ........ 54

Table 3.4: Probability of Dirty Push - Program Address Traces Without Flushing ........ 54

Table 3.5: Probability of Dirty Push - UNIX File System Traces ........ 55

Table 3.6: Percentage of Blocks and Reference By Class ........ 56

Table 4.1: Time complexity of Linked List implementation ........ 61

Table 4.2: Definitions of terms used in Time Analyses ........ 62

Table 4.3: Time complexity of Linked List with Hash Table ........ 63

Table 4.4: Time complexity of bounded Bennett & Kruskal's tree ........ 68

Table 4.5: Effect of AVL Balance Schemes ........ 71

Table 4.6: Time complexity of AVL tree implementation ........ 73

Table 4.7: Time complexity of Hybrid implementation ........ 79

Table 4.8: A comparison of simulation stack implementations ........ 80

Table 4.9: Types of Trace Events ........ 83

Table 4.10: General Trace File Characteristics ........ 84

Table 4.11: Stack Distance Distribution Summary ........ 86

Table 4.12: Comparison of simulation time for all techniques ........ 90

Table 4.13: Fundamental Operations per Event ........ 91

Table 5.1: MOESI protocols - Local Requests ........ 103

Table 5.2: MOESI Protocols - Bus Requests ........ 103

Table 5.3: Berkeley Protocol ........ 109

Table 5.4: Possible Configurations - Berkeley Protocol .......... 110

Table 5.5: Second Protocol .......... 113

Table 5.6: Possible Configurations - Second Example Protocol .......... 115

Table 5.7: Dragon Protocol .......... 117

Table 5.8: Possible Configurations - Dragon Protocol .......... 120

Table 5.9: Revised MOESI protocols .......... 129

Table 5.10: Possible Combinations of Write Rules .......... 132

Table 5.11: Firefly Protocol .......... 153

Table 5.12: Write Once Protocol .......... 157

Table 6.1: Possible file states .......... 173

Table 6.2: Client Actions to Local Requests .......... 175

Table 6.3: Server Actions to Client Requests .......... 178

Table 6.4: Client Actions to Server Requests .......... 180

Table 6.5: Exclusive Client Actions .......... 181

Table 6.6: Client Actions - Sprite Protocol .......... 183

Table 6.7: Server Actions - Sprite Protocol .......... 183

Table 6.8: Client Actions - Exclusive Protocol .......... 184

Table 6.9: Server Actions - Exclusive Protocol .......... 184

Table 6.10: Client Actions - Read-Only Protocol .......... 187

Table 6.11: Server Actions - Read-Only Protocol .......... 187

Table 6.12: Client Actions - Write Broadcast Protocol .......... 189

Table 6.13: Server Actions - Write Broadcast Protocol .......... 189

Table 7.1: Traced Events .......... 199

Table 7.2: Number of Events and Percent of Each Record Type .......... 199

Table 7.3: Summary of Activity Types .................... 201

Table 7.4: Summary by Type of Activity .................... 201

Table 7.5: Major File Types .................... 202

Table 7.6: Summary by Type of File .................... 202

Table 7.7: Concurrent Sharing .................... 203

Table 7.8: Potential Sharing .................... 205

Table 7.9: Weighting Factors for Transfer Ratio .................... 207

Table 7.10: Payoff By Delaying Write .................... 211

Table 7.11: Combination of Write Policies .................... 218

# Chapter 1

# Introduction

The enormous difference between the access times of processor memory and disk, often called the ''access gap'', has spawned interest in caching to reduce the time required to respond to user data requests. By maintaining a file system cache of frequently used blocks, the access time for file system requests can be significantly reduced. In a distributed system, communication delays increase the access time, further increasing the benefits of caching.

The research leading to this dissertation began as a study of caching in a distributed file system using trace-driven simulation. As problems arose in the efficient simulation of certain cache parameters, such as the effect of writes, and as the generality of the solutions to those problems became evident, the thrust of the research changed. This dissertation is therefore primarily a study of innovative ways to efficiently simulate a variety of cache designs, with applicability ranging from on-chip processor caches to file system and disk caches.

Chapter 2 reviews the concepts of caching and discusses some metrics used to analyze cache performance. It then reviews the *stack analysis technique*, which is the basis for the efficient algorithms to follow. Discovered in 1970 by Mattson, et al. [Matt70], stack analysis permits the miss ratios for all cache sizes to be computed in one pass over a trace file, for certain replacement algorithms. This is possible because these algorithms impose an *inclusion property* on the cache contents such that if a block is present in a cache of size $k$ then it is present in all larger caches.

There are a number of situations where stack analysis has not been applied. The first of these is the consideration of writes and the performance of different write policies. If a cache uses a write-through policy then every write results in a memory access. This can constrain the performance of a system which is limited by the bandwidth of the memory path, such as a network file system or a multi-processor bus. This gives incentive to using a write-back policy, which marks the block dirty in the cache but delays the write to memory until later. Write-back has the advantage that several writes to a block can be combined into one write to memory. Furthermore, write-back in a file system cache may allow the file to be deleted before the block is ever written to disk.

Compared to reads, the simulation of write-back for all sizes appears very complex. Simulation of reads is simple because the simulator knows at the time of the reference exactly which sizes require a memory access. On the other hand, a write-back from cache occurs at an indefinite time after the block becomes dirty, and occurs at different times for each cache size (i.e. when the block is replaced). This implies that a lot of work is required to count write-back accesses.

Chapter 3 discusses this problem in more detail, and describes a new technique for analyzing writes in one pass. There are two key observations which make this possible. The first is that the dirty state obeys inclusion, that is, each block has a single *dirty level*, which is the minimum cache size in which the block is dirty. The second discovery is a method of counting writes by observing that each write to a dirty block saves one memory access (compared to write-through) since the dirty block needs to be written to memory only once. Therefore, every write results in a write access for all sizes where the block is clean, but the write-back is avoided for all sizes where it is dirty. Chapter 3 goes on to show that deletions and periodic write-back, particularly important to studies of file system caches, can also be efficiently analyzed.

Chapter 3 also discusses the use of a similar technique to analyze sub-block or *sector* caches. These are caches having a large block size to reduce the number of blocks which need to managed for a given cache size, but which access data in terms of smaller sub-blocks to reduce the bandwidth requirement. Chapter 3 shows that there is a unique *valid level* for each sub-block, which is used to compute the cache sizes where a memory access occurs. The chapter also shows that a useful form of prefetch for such caches, known as load-forward, can be simulated in one pass.

The second problem studied was one previously encountered for database traces [Benn75]; even with stack analysis, simulations of file system caches are slow. In fact, they are so slow that as many as ten simulations of a single size could be done in the same time, negating the advantage of stack analysis. The problem is due to the excessive time required to compute the *stack distance*, or the minimum cache size where the block is valid, when the simplest linked-list implementation of a stack is used against traces with high mean stack distance. Chapter 4 reviews several possible solutions proposed by others, involving data structures ranging from the addition of a hash table to the use of static and balanced tree structures. It also describes a new hybrid combination of a list and tree that is relatively insensitive to the mean stack distance.

The chapter then presents a comparison of the time required for simulations using each of the data structures against a variety of trace data, including several program address traces, disk traces, and UNIX file system traces. It shows that for program address traces, the simple linked-list is adequate. However, for file system and disk traces, a static tree structure, discovered by Bennett and Kruskal [Benn75] and refined by Olken [Olke81], is much more efficient. The hybrid method performs better than the other implementations in several cases, but the improvement is minor.

A third problem arises when stack analysis is extended to multi-processors each with its own cache. Efficient simulation is trivial if the reference streams are independent, but the need to maintain consistency of shared memory introduces dependencies. The question studied is whether multiple caches can be simulated in one pass for all cache sizes: a) if each cache can be of any size, or b) if the cache sizes are constrained (e.g. if all caches are the same size, or if some caches have constant sizes).

Chapter 5 considers these questions for a class of cache consistency protocols for shared-memory multi-processor systems using a backplane bus. It introduces the notion of a *one-pass algorithm* as a generalization of the term *stack algorithm*, for efficient cache simulation algorithms where the state depends on more than the stack contents. A cache management algorithm is called a one-pass algorithm if it can be simulated in time which is $O(N*S)$ and space which is $O(S)$, where $N$ is the number of trace events, and $S$ is the largest cache size of interest. Both the original stack algorithm and the write-back analysis technique meet this definition. The chapter shows that although some consistency protocols are one-pass algorithms for independent cache sizes, most are one-pass only if the memory sizes are related. The remainder of the chapter considers the special case where all caches are the same size, and shows that all protocols of the class are one-pass algorithms.

The key to this result is again inclusion. The cache consistency protocols work by taking actions based on the state of a block in each cache, where the state is a combination of three characteristics: validity, dirtiness or ownership, and sharing. Chapter 5 shows that, given certain reasonable restrictions, each of these characteristics obeys inclusion (e.g. if the block is shared for size *k* then it is shared for all larger sizes). It shows that the state of a block in all caches and sizes can actually be maintained using just a stack per cache plus two variables per block — a dirty level and a sharing level. Chapter 5 goes on to present algorithms for maintaining these variables and computing misses and bus traffic for several common consistency protocols.

Chapter 6 extends this technique to a class of file system cache consistency protocols. It begins by describing the differences between the shared-memory multi-processor of Chapter 5 and network file systems. The primary differences are the existence of higher-level operations and objects in a file system (i.e. open/close of a file), and the lack of an inherent reliable broadcast capability similar to that provided by a backplane bus. The chapter defines an expanded state space to consider these differences, then defines a class of protocols for maintaining consistency between file system caches. The class includes numerous options which can be selected independently for any cache and file. The tradeoffs between the options are discussed in the context of four example protocols from the class. One is a very simple protocol that relies exclusively on the file server to maintain consistency. The second is a protocol which is optimized for files that are only accessed by a single user (private files). The third is optimized for read-only files, whether or not they are shared. The final example is a protocol that uses write broadcast to keep multiple caches consistent even when the file is read-write shared.

Finally, Chapter 7 demonstrates the use of all of the techniques by presenting the results of an analysis of client caching in a network file system. It begins by describing the composition and general characteristics of the file system trace data. It then reports on file sharing, one of the measures which most affects the performance of cache consistency protocols. The study shows that concurrent sharing is very rare (fewer than 3% of all opens are to a file already open by another user), but that the 5% of files that are ever accessed by multiple users (i.e. potentially shared) account for two-thirds of all opens. Over half of the shared accesses are read-only, while 80% of writes are to private files, both of which reduce the demand for consistency controls.

Chapter 7 then presents the actual simulation results using the techniques discussed in the previous chapters to study cache consistency protocols, write policies, and fetch policies. The efficiency of the techniques is shown in the fact that it takes just 10% longer to simulate all sizes compared to simulating a single set of cache sizes. The actual simulation results include the miss ratio and the transfer ratio, which is the ratio of server requests with and without cache, including copy-back and cache consistency overhead.

The study of consistency protocols compares the four example protocols described in Chapter 6. It shows that the choice of consistency protocol has a much greater effect on performance than the choice of write or fetch policy, varying the transfer ratio by a factor of four for large caches. The Read-Only protocol produces less than half the misses and server load of the simple protocol, with similar performance for the Write-Broadcast protocol. Both of these have nearly double the misses and load of an optimal protocol, indicating a potential for improvement through research into refined protocols.

The analysis of write policies shows that delaying writes by 30 seconds avoids 25-30% of all writes, but that nearly the same savings is possible by using write-back just for temporary files. The fetch experiments include one-block and full-file prefetch. Chapter 7 shows that an optimistic one-block look-ahead reduces misses by 30% compared to demand fetch, with little increase in server load. Full-file fetch is shown to be a poor policy unless restricted to small files, because the cost of erroneously fetching a large, randomly-accessed file is so high.

The dissertation concludes with Chapter 8 which summarizes all the results and presents some thoughts on future directions for related research.

# Chapter 2

# Background

## 2.1. Cache Memories

Analysis of memory systems has always been an important part of the design of computer systems. The early concentration on virtual program memory [Bela66] has been recently replaced by emphasis on high-speed processor caches, and by file system buffering or caching. In reality, these can all be viewed as parts of a hierarchy of memory types, where the upper levels of the hierarchy are faster, but generally more costly. Although many of the analysis techniques discussed in this dissertation generalize to multi-level hierarchies [Olke81], we restrict our discussions to a two-level hierarchy, and refer to the top level as a *cache*.

The purpose of caching is to improve the average access time to items in memory by keeping the most frequently used items in a small fast cache memory, leaving the remainder in a larger slower memory. The contents of cache are checked on each reference; if the referenced item is present in cache, then the data is available at the speed of the cache. If not then the data is read into cache from memory, replacing something already cached. The speed of the combined memory system is a function of the two memory speeds and the probability that the referenced item is in cache.

Caches are effective because of the *principle of locality* [Denn72]. This principle says that the items most likely to be referenced next are those "near" the items that have been recently referenced. The two aspects of locality are "temporal" locality and "spatial" locality. *Temporal locality* implies that an item that has been recently referenced has a good chance of been referenced again in a short time. *Spatial locality* implies that items close to a referenced item are also likely to be referenced. This is particularly evident in the sequential reference behavior observed in instructions and within files.

There are a large number of design parameters to any cache, most of which must be considered in any analysis of that design. We briefly present definitions of a number of these. For more detail see [Smit82].

Blocking

The cache may be divided into fixed-size *blocks*, or variable-size *segments*. Blocks are also referred to as *pages* in the context of virtual memory, and *lines* or *sectors* in the context of processor cache. The cache block or line size may be equal to the amount of data retrievable in one memory cycle, or several memory cycles may be required to fetch a block. A larger block size reduces per-block overhead and provides a form of prefetch, discussed below. This dissertation discusses only block caches, although most of the algorithms presented can be generalized to analyze segment caches [Olke81], for example a virtual memory that manages entire programs as a unit, or file system caches that manage whole files.

Replacement Policy

The replacement policy determines which block to remove when the cache is full and a new block must be fetched. Commonly suggested policies include the Least Recently Used (LRU) policy, First-In First-Out (FIFO), Least Frequently Used (LFU), and Random (RAND). An optimal policy, MIN, exists, but is unrealizable in practice because it requires knowledge of the future [Matt70]. The MIN policy does not consider writes or deletes, and is known to be non-optimal if writes are considered [Yu76].

Write Policy

The write policy determines when a modification is presented to the secondary storage. Writes may always go directly to the secondary storage using the *write-through* or *store-through* policy. Alternatively, the write may go to the cache to be written at some later time, usually when the block is about to be replaced, which is called *write-back* or *copy-back*. Write-back is motivated by the expectation that the block will be modified several times before it has to be written. Clearly write-back can never cause more accesses than write-through, and usually far fewer. On the other hand, since it deals in blocks rather than words, write-back may increase the number of bytes written. In addition, dirty blocks may remain in the cache for a long time, leading to reliability issues in large volatile caches such as file system caches in main memory. The decrease in memory traffic from write-back makes it very valuable in systems with limited memory bandwidth, such as shared-bus multi-processor systems. Write-back is also desirable in file system caches because many files are temporary and may *never* have to be written.

### Write-Fetch

If write-back is used in a cache where partial-block modification is allowed, and the block to be written is not in cache (a *write miss*) then it is usually necessary to fetch the block prior to modifying it. This *write-fetch* is needed, for example, if one word of a multi-word block is being written. The alternative is to keep track of the portion(s) of the the cache block that are ''valid'', which becomes difficult when several disjoint portions of the block are written. However, there are situations where write-fetch can be avoided. Two examples are when the entire block is being overwritten, or when the contents of the rest of the block are predictable, such as when the block is a ''new'' block in a file system.

### Prefetch

Because of spatial locality, a reference to a block often implies that the next physical block will soon be referenced. It is possible to take advantage of this anticipated reference and to *prefetch* the next block in advance. This reduces the delay when the next block is actually referenced. Prefetch is advantageous when it can be overlapped with processing of other references, or when two or more blocks can be fetched in much less time than all of them individually, as is the case with disk secondary storage. While it reduces the delay, prefetch will increase memory traffic unless all prefetched blocks are referenced before they are replaced. It may also result in *memory pollution*, where a soon-to-be-referenced block is displaced to make room to prefetch an unnecessary block [Smit78b]. If a prefetch is only permitted in conjunction with a fetch then the policy is a *demand prefetch* policy. Demand prefetch is desirable when the overhead of a fault is large; demand prefetch amortizes this over two (or more) blocks. With modern memory systems and file system caches, it is simple and inexpensive to initiate a prefetch even if the referenced block is present.

### Cache Consistency

Whenever a block may be present in several locations there is a potential for a *cache consistency problem* and the need to ensure that all caches have the same copy of a block. There are two general approaches to consistency control: Write-Update or *Broadcast* and *Invalidation*. The broadcast approach keeps all copies current by sending all changes from the writer to all other cache sites. If the interconnection supports reliable broadcast (as provided by a backplane bus, for example), then broadcast requires a single message. If reliable broadcast is not available then the writer needs to contact the other caches individually, which is usually prohibitively expensive. Broadcast is typically not used in network file systems for this reason. This is discussed further in Chapter 6.

Invalidation schemes maintain consistency by ensuring that all other copies become invalid whenever one cache is written. Invalidation can occur at the time of the write, or at the time of next access from another cache. ''Snooping'' caches on a multiprocessor bus usually invalidate on write, since a single bus action can notify all other caches. An example of invalidation on use is a network file system that ''polls'' the file server on each open to verify the version number of a file. Invalidation is most economical when updates are bursty, that is, when there are several updates from one processor before any reads from others. Broadcast is advantageous when access is more uniform, since it saves the refetch in all other caches, at a cost of one (or a few) write-throughs.

## 2.2. Metrics

The performance of a memory system can be measured in several ways. Perhaps the most widely used is the *miss ratio*, which is the fraction of references that were not satisfied by the cache. Conversely, the hit ratio is the fraction that were satisfied by the cache. The miss ratio is a latency metric since it determines the apparent access time of the memory system. For a multi-level hierarchy, the effective access time is given by $\sum t_i h_i$, where $t_i$ is the access time to the $i$th level, and $h_i$ is the hit ratio to the $i$th level. Sometimes overlooked is the fact that the access time to each level should include any queuing delays. These are usually negligible in a single-processor system, but may become important when several processors compete for access to a single secondary store [Arch86, Lazo86].

The actual computation of miss ratios during simulation varies with the parameters of the cache. Let $N$ be the total number of references, and $m(C)$ be the number of misses in a cache of size $C$. If all references are assumed to be reads, then the miss ratio for a cache of size $C$ is given by

$$MR_R(C) = m(C)/N \tag{2.1}$$

hence the name.

With write-through, where every write is a "miss" (i.e. causes an access to secondary storage), the miss ratio is

$$MR_{WT}(C) = (m_r(C)+W)/N \tag{2.2}$$

where $m_r(C)$ is the number of reads that "miss", and $W$ is the number of write references.

When write-back is used, this becomes

$$MR_{WB}(C) = (m_r(C)+dp(C))/N \tag{2.3}$$

where $dp(C)$ is the number of dirty blocks "pushed" from a cache of size $C$.

If write-fetch is also considered, a write could result in two accesses to secondary storage, one to fetch the block and another later to write it. The miss ratio is now given by

$$MR_{WBWF}(C) = (m(C)+dp(C))/N \tag{2.4}$$

where we have used the fact that a write-fetch is actually just a read reference and occurs if the block reference "misses".

All of the expressions so far assume that the processor must wait for the write to secondary storage to complete before continuing. It is often reasonable to buffer the writes so that the processor can continue almost immediately. In this case delay occurs only if there are enough accesses to create contention. In [Smit79], it is observed that when memory bandwidth is adequate, four store-through buffers are sufficient to largely eliminate queuing for writes. Under this assumption, the write-back miss ratio with write-fetch is again simply

$$MR_{WF}(C) = m(C)/N. \tag{2.5}$$

A related metric is the *traffic ratio,* which is the ratio of traffic between cache and secondary storage compared to the traffic that would be present without a cache [Hill84]. The traffic ratio is increasingly important for analyzing shared-bus systems such as multi-processor architectures or a network file system. Although buffering may eliminate write-back from consideration in the miss ratio, the write traffic is not eliminated, so writes must be considered in the traffic ratio. Also, prefetch may result in increased traffic, since some prefetched blocks may not actually be referenced.

The traffic ratio is dependent on the same factors as the miss ratio and, in addition, depends on the size of data blocks transferred. Suppose that the processor accesses $B_p$ bytes per average memory reference. The traffic without a cache is then $B_p$ times the number of references. Frequently the cache block size, $B_c$ is larger than $B_p$. We assume that each cache miss causes $B_c$ bytes to be transferred. Then a large cache block size may act as a form of prefetch and reduce the miss ratio, but it may also increase the amount of traffic.

The general form of the traffic ratio computation is

$$TR(C,B_c) = [m_r(C) + m_w(C)*Wf + f(C) + dp(C)]*B_c / N*B_p \tag{2.6}$$

where $Wf$ is 1 if write-fetch is used, 0 otherwise; $m_w(C)$ is the number of write misses (i.e. write-fetches); $dp(C)$ is again the number of write-backs, and $f(C)$ is the number of prefetches. Notice that the traffic ratio is identical to the miss ratio when there is no prefetching, no write buffering, and the cache block size is the same as $B_p$.

A third reasonable metric is the *transfer ratio*, which is the ratio of secondary storage accesses with and without cache [Smit78b]. This metric has also been called the *transaction ratio* [Gibs86], the *I/O ratio* [Thom85, Kent86], and the *swapping ratio* [Kubo75]. The transfer ratio is similar to the traffic ratio, but is more appropriate when performance is dominated by the cost of a memory access, relatively independent of the number of bytes transferred. Thus it is appropriate for disk caches, and often for networks using small (1K or less) messages. For example, the transfer ratio decreases if two blocks are read from disk in a single I/O, while the traffic ratio is the same regardless of the number of I/O's used to transfer the data. The transfer ratio also has an indirect effect on the access time if there are enough transfers to create contention, particularly in multiple processor systems with shared memory. A general expression for the transfer ratio is

$$T(C) = [m_r(C) + m_w(C)*Wf + dp(C)] / N \tag{2.7}$$

which is almost proportional to the traffic ratio using constant block sizes.

These metrics are ones which have been commonly used. Of course it is possible to redefine the metrics to meet the needs of a specific memory system. For example, in Chapter 7 we want a metric to measure the file server load. Since the number of bytes transferred has a non-negligible effect on the load, we redefine the transfer ratio to include a component which is proportional to the number of bytes transferred, in addition to the fixed cost per transfer.

## 2.3. Trace Driven Simulation

One common method to calculate these metrics is to use trace-driven simulation. Memory references are gathered from a system assumed to be similar to the system being modeled. These events are then used to drive a simulation of the system under study with varying design parameters. To the extent that the traces apply to the modeled system, simulation is a relatively simple way to observe the effect of changes to the memory hierarchy. Unfortunately, it could take a large number of simulations if only a single combination of memory sizes could be simulated at a time.

In a classic paper, Mattson, et al. showed that for certain replacement policies, the miss ratios for all cache sizes could be calculated in a single pass over the reference trace [Matt70]. These policies are collectively known as *stack algorithms*. The technique depends on the *inclusion property* of these policies such that the contents of any size cache includes (i.e. is a superset of) the contents of any smaller cache. Thus the cache at any time can be represented as a stack, with the most recently referenced block on top. The upper $k$ elements of the stack are the blocks present in a cache of size $k$. The current stack level of any block is therefore the minimum cache size for which the block is resident. If a block is referenced while at level $k$, it is a "hit", and therefore resident, for all sizes $k$ and larger. The level at which the block is found is referred to as its *stack distance*. See Figure 2.1. Using stack analysis, it is possible to compute the miss ratio of equation (2.1) for all sizes by recording the hits to each level. The miss ratio for a cache size $C$ is:

$$MR_R(C) = (N - \sum_{i=1}^{C} hits(i))/N \tag{2.8}$$

where $N$ is again the total number of references. Notice that, since $hits(i)$ is never negative, this is a non-increasing function of cache size. All stack algorithms possess this characteristic, whereas non-stack algorithms may show points at which performance declines with increased cache [Bela69].

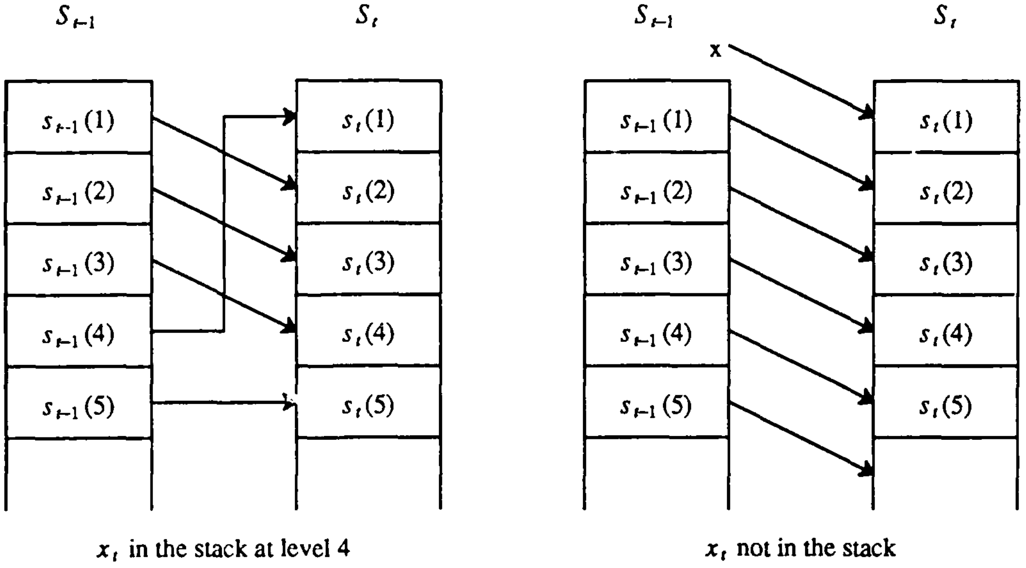


**Figure 2.1**: Examples of stack maintenance using LRU replacement. The referenced block is always "pulled" to the top of the cache stack. All blocks with smaller stack distance are pushed one level.

The simplest example of a stack algorithm is the Least Recently Used (LRU) policy. The stack always contains the blocks in order of last reference, with the most recently referenced block on the top. For any cache size $C$, the LRU block for that cache size is the block at level $C$ in the stack. When a block at level $k$ is referenced, it is not in any cache smaller than $k$, and therefore it must be fetched. The block that must be removed from any cache of size $j$, $j$ smaller than $k$, is the block at level $j$. The stack is updated by simply "pulling" the referenced block out of the stack and placing it on top. All blocks down to level $k$ are effectively "pushed" down one level. Since the referenced block was in all caches $k$ or larger, all blocks below level $k$ remain unchanged. Figure 2.1 illustrates these operations for the case where the referenced block is at stack level 4, and the case where the block is not currently in the stack.

More generally, Mattson, et al. [Matt70] showed that any stack algorithm possesses a "priority function" which imposes a total ordering on all blocks at any given time, independent of cache size. Notice that LRU imposes such an ordering based on the time of last reference.

However, in the more general case the relative priority of two blocks may change without either of them being referenced. (See, for example, Figure 2.3 where the relative positions of blocks **B** and **C** reverse between times 7 and 8). It is no longer the case that the block at level $j$ is necessarily the one to be pushed from that size cache. This complicates the stack update procedure, but only slightly. The stack can still be updated in a single pass that is similar to one pass of a bubble-sort. A single comparison at each level determines the new block at the level and the block pushed from the level. Figure 2.2 illustrates the process, which is described in the following paragraph.

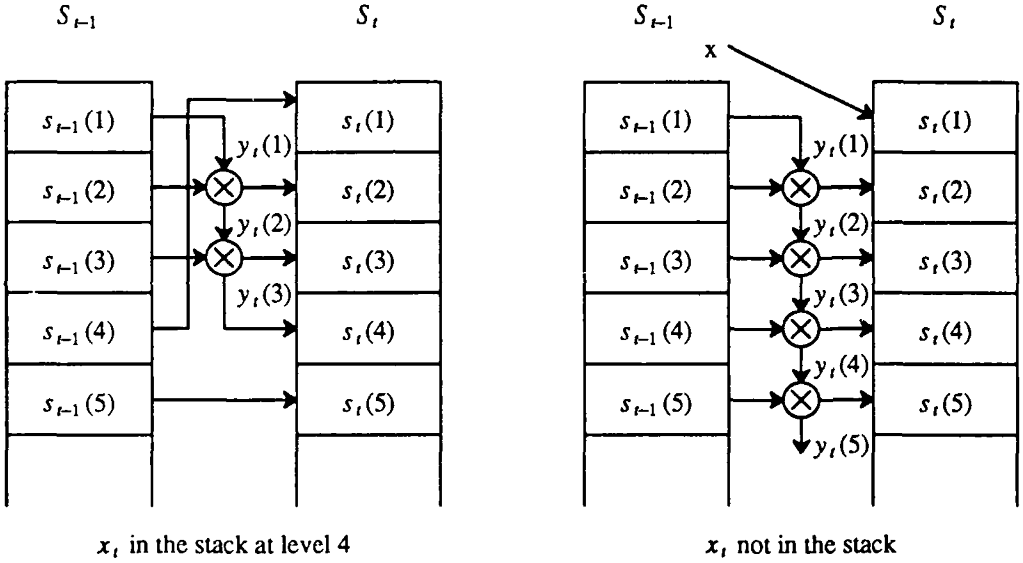

$S_{t-1}$
$S_t$
$s_{t-1}(1)$
$s_{t-1}(2)$
$s_{t-1}(3)$
$s_{t-1}(4)$
$s_{t-1}(5)$
$s_t(1)$
$s_t(2)$
$s_t(3)$
$s_t(4)$
$s_t(5)$
$y_t(1)$
$y_t(2)$
$y_t(3)$
$x_t$ in the stack at level 4
$S_{t-1}$
$S_t$
x
$s_{t-1}(1)$
$s_{t-1}(2)$
$s_{t-1}(3)$
$s_{t-1}(4)$
$s_{t-1}(5)$
$s_t(1)$
$s_t(2)$
$s_t(3)$
$s_t(4)$
$s_t(5)$
$y_t(1)$
$y_t(2)$
$y_t(3)$
$y_t(4)$
$y_t(5)$
$x_t$ not in the stack

**Figure 2.2**: Examples of stack maintenance using a stack replacement algorithm. For each level, $C$, a single comparison (indicated by a circled cross) between the prior block at the level ($s_{t-1}(C)$) and the block pushed from above ($y_t(C-1)$) determines the new block at the level and the block pushed from the level. Update continues down to the current level of the referenced block, or to the bottom of the stack if $x_t$ is not in the stack.

First, the referenced block is still pulled to the top of the stack since it must become resident in all cache sizes. Using the terminology from Mattson, et al., let $y_t(C)$ be the block pushed ("yanked?") from a cache of size $C$. To make room for the referenced block, the top block in the stack, $s_{t-1}(1)$, must be pushed from a one-block cache, becoming $y_t(1)$. Some block must also be pushed from a two-block cache — the one with the lowest priority. A single comparison between $y_t(1)=s_{t-1}(1)$ and $s_{t-1}(2)$ determines which becomes $y_t(2)$. (Ties are broken by some arbitrary rule.) Similarly, the block pushed from a three-block cache should be the lowest priority of the three blocks previously present. However, the lowest priority block can be determined in a single comparison of $y_t(2)$ and $s_{t-1}(3)$, since the third block, now $s_t(2)$, has already "won" a comparison against $y_t(2)$, and thus can not have the lowest priority. Similar logic applies for all levels down to level $k$, the original level of the referenced block; only the block currently at the level and the one pushed from above need to be compared to find the block to be pushed. The contents of all sizes larger than $k$ are again unchanged.

The stack analysis algorithm is formally presented below. This algorithm will be used as the basis for the extensions in Chapter 3. Let:

$X$ = $x_1 x_2 .. x_N$ be a trace, where $x_t$ is the reference at time $t$.

$S_t$ = the cache stack just after reference to $x_t$, with $s_t(C)$ = the block at stack level $C$. $s_0(C) = \phi$ for all $C$.

$\Delta$ = the stack distance of $x_t$, that is, $s_{t-1}(\Delta)=x_t$

$y_t(C)$ = the block pushed ("yanked") from cache of size $C$ by reference $x_t$.

$rh(C)$ = a count of the number of hits to level $C$ by time $t$.

Algorithm 2.1: General Stack Analysis Algorithm

```
1.  For $1 \le t \le N$ do                                      For all events
2.     If $x_t \notin S_{t-1}$ then $\Delta=\infty$                   If not referenced before.
3.     else do
4.        Find $\Delta$ such that $s_{t-1}(\Delta)=x_t$                Find the stack distance
5.        $rh(\Delta)=rh(\Delta)+1$                               Update the read hits
6.     If $\Delta \ne 1$                                          If the stack needs updating
7.        $y_t(1)=s_{t-1}(1)$                                  Calculate push from each level
          for $2 \le i < \Delta$ do                               Down to referenced block
             $y_t(i)=pmin[y_t(i-1),s_{t-1}(i)]$             Push is the minimum of the
                                                         block at level and push from above
          for $i \ge \Delta$ do $y_t(i)=\phi$                       No pushes below reference
8.        $s_t(1)=x_t$                                        Establish new stack.
          for $i>1$ do $s_t(i)=s_{t-1}(i)+y_t(i-1)-y_t(i)$
```

Notes:

1. In step 5, all counts that are not incremented are assumed to remain unchanged at the next time interval.
2. In step 7, *pmin* returns the block with the lowest priority, as defined by the replacement algorithm. *Pmin* is the comparison function in the circles of Figure 2.2.
3. In step 8, plus and minus have the intuitive meaning of adding a member to a set, or removing a member. In this context, adding a member that is already present, or subtracting a member that is not present, have no effect. The same is true of adding or subtracting $\phi$. Thus, the block kept at level $i$, $s_t(i)$, is either $s_{t-1}(i)$ or the block pushed from above, $y_t(i-1)$, whichever is not pushed from level $i$.

Note that, in practice, it is possible to search the stack for the referenced block and update the stack simultaneously, since the priority function can not depend on where (or even if) the referenced block is in the stack. The update stops when the referenced block is found. The block being pushed takes the place of the referenced block, which is inserted on top of the stack.

As an example, consider the application of the Least Frequently Used policy to the reference string {**AAABBCCDB**}. Using this policy, the block pushed from any cache is the one that has been used the fewest times since it was loaded into cache. Figure 2.3 shows the contents of the stack after each reference, where the number beside each block is the priority, (i.e. the number of uses of the block). Notice that a block may be pushed several levels because of a reference, as seen at time 8. Note too that blocks below the level where the referenced block is found are unchanged, even though they may have higher priorities, as seen after the last reference.

| Time | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| Reference String | A | A | A | B | B | C | C | D | B |
|  | A1 | A2 | A3 | B1 | B2 | C1 | C2 | D1 | B3 |
| Cache |  |  |  | A3 | A3 | A3 | A3 | A3 | A3 |
| Stack |  |  |  |  |  | B2 | B2 | B2 | D1 |
|  |  |  |  |  |  |  |  | C2 | C2 |

Figure 2.3: Cache contents using Least Frequently Used policy. The number beside the block is the priority, i.e. the number of references.

## 2.4. Non-Stack Algorithms

The prohibition against a priority function that depends on cache size prevents some otherwise-simple policies from being stack algorithms, such as the First-In First-Out (FIFO) rule [Matt70]. Another common technique that is not a stack algorithm is the use of *demand prefetch* or prefetch-on-miss [Smit78b]. Suppose that the prefetch policy is to fetch the following block along with any fetched block, but not to prefetch if the referenced block is already present. This is typical of a demand prefetch policy, where no fetch should take place unless the referenced block is missing. Assume an arbitrary stack algorithm for replacement. It is easy to construct counter-examples that violate inclusion, because the priority of a prefetched block depends on when it is fetched, which varies with cache size. For example, consider the examples of Figure 2.4, where the contents of a larger cache is clearly not a superset of a smaller cache after the final reference.

It is possible to construct prefetch policies that are stack algorithms. For example, the non-demand policy that always prefetches the next block, regardless of whether the referenced block is resident, is a stack algorithm. This policy is a form of One Block Lookahead, or OBL [Smit78b]. From the point of view of the stack this is equivalent to the insertion of a reference to the next block after each reference. Non-demand prefetch is not practical if the cost of a fault is high, as it is in virtual memory systems for example, because the penalty for faulting to prefetch a block that may not be needed is greater than the potential gain. Non-demand prefetch is practical

| Time | | 1 | 2 | 3 |
|---|---|---|---|---|
| Reference | | A | C | A |
| | Size | | | |
| | 1 | A | C | A |
| Cache | 2 | AB | CD | AB |
| Contents | 3 | AB | CDA | ACD |

(a)

| Time | | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Reference | | A | B | C | A | D |
| | Size | | | | | |
| | 1 | A | B | C | A | D |
| Cache | 2 | AB | BA | CD | AB | DE |
| Contents | 3 | AB | BA | CDB | ABC | DEA |
| | 4 | AB | BA | CDBA | ACDB | DACB |

(b)

**Figure 2.4**: Cache contents using one-block prefetch. Since this is not a stack algorithm, the contents of each cache size are listed separately. In both examples the next block is prefetched only if the referenced block is not present. In all cases the referenced block becomes the highest priority, followed by the prefetched block, if any. The inclusion property is violated after the last reference in both cases.

when it is possible to look for the next block in the cache and prefetch it if necessary without significantly slowing down processing the current reference. This is the case for many processor caches and file system caches.

As an aside, a demand prefetch policy that obeys inclusion has been described by Horspool and Huberman [Hors83]. Their algorithm adds the condition that the following block ($x_t$+1) is only prefetched if its stack distance is less than that of the referenced block ($x_t$). This prevents the loss of inclusion seen in Figure 2.4(a) above. It has the added benefit of reducing memory pollution, since it ensures that $x_t$+1 was referenced after the prior reference to $x_t$, increasing the chances that it will be referenced after the current reference to $x_t$. In addition, their algorithm pulls $x_t$+1 to stack level one whether or not $x_t$ is fetched, preventing the anomaly seen in Figure 2.4(b). Their algorithm also allows prefetched blocks to "age" down the stack at a rate $k$ times faster than referenced blocks, where *k* is some small constant. They refer to their class of prefetch policies as OBL/k policies, and also discuss variable-space counterparts, VOBL/K. The combination of factors decreases the miss ratio by 10-30% compared to LRU, with far fewer prefetches than OBL.

### 2.5. Extensions to Stack Analysis

There have been several important extensions to the stack analysis technique. Mattson, et. al. [Matt70] showed how the hit ratio can be computed for an arbitrary number of levels, assuming a common block size and replacement policy. Gecsei [Gecs74] showed how it could be generalized to multiple levels with different block sizes for LRU and certain related policies. Traiger and Slutz [Trai71] showed that it is possible to compute miss ratios for variable block sizes, and variable associativity, in a single pass. See also [Shed76] and [Slut72].

Coffman and Randell [Coff71] investigated the "extension problem", that is, to predict the performance of cache sizes greater than $C$, given only the misses from cache size $C$ instead of a full trace. For LRU, a trace of "pushes" and "pulls" was sufficient; for other stack algorithms, the priority ranking for the block pushed and all blocks not in the cache of size $C$ was also required. A trace of misses only was shown to be sufficient to provide good approximations to the performance of larger caches in [Smit77].

A more recent extension by Silberman [Silb83] showed that stack analysis can be applied to a "delayed-staging hierarchy" in which the processor directly accesses several levels of the memory hierarchy. When a referenced block is not in a higher-level cache, it is supplied to the processor (at the speed of the highest level cache to contain the block) and begins "migrating" into the higher caches. The time elapsed until it becomes "staged" (resident) in a higher cache is equal to the sum of the access times of the caches below it. Further, the displacement of a block in the higher level cache is also delayed, creating a situation where the stack level of a block may be a function of the size of several lower level caches, and the time since last reference of one or more *other* blocks. Silberman showed that stack analysis can be applied to this class of hierarchy by maintaining the time and cache depth of last "migration" for each block. This information is used at the time of each reference to compute the stack distance of the block for different sizes of each level, considering the delayed staging times. This idea of maintaining additional information about each block will be seen again in our write-back algorithm presented in the next chapter.

# Chapter 3

# Efficient (Stack) Algorithms for Analysis of Write-Back and Sector Memories

## 3.1. Summary

For the class of replacement algorithms known as *stack algorithms*, existing analysis techniques permit the computation of memory miss ratios for all memory sizes simultaneously, in one pass over a memory reference string. We extend the class of computations possible by this methodology in two ways. First, we show how to compute the effects of copy-backs in write-back caches. (The key observation here is that a given block is clean for all memory sizes less than or equal to $C$ blocks and is dirty for all larger memory sizes.) Our technique permits efficient computations for algorithms or systems using periodic write-back and/or block deletion. The second extension permits stack analysis simulation for sector (or sub-block) caches, in which a sector (associated with an address tag) consists of sub-sectors (or sub-blocks) which can be loaded independently. (The key observation here is that a sub-sector is present only in caches of size $C$ or greater.) Load forward prefetching in a sector cache is shown to be a stack algorithm and is easily simulated using our technique. Running times for our methods are only slightly higher than for a simulation of a single memory size using non-stack techniques.

## 3.2. Introduction

Until now, stack analysis has not been applied to some important situations, forcing the designer to fall back on the one-size-at-a-time method. One example of this is the write-back policy, also called copy-back, where a write to a block causes the block to be marked ''dirty'' in the cache, but the write to secondary storage is delayed until some later time. Write-back is particularly desirable where memory bandwidth may be a limiting resource, such as in a shared-bus multiprocessor or network file system. The alternative policy, write-through or store-through, where all modifications go directly to secondary storage, severely restricts the performance improvement due to caching. Even with write-back, in many cases a write can cause twice the memory accesses of a read; one to fetch the block prior to modification and another to rewrite the block (copy-back). However, discussions of stack analysis have either ignored writes altogether, or considered only write-through.

The problem with stack analysis of write-back is that it appears to violate the inclusion property. For example, suppose that a dirty block at level $k$ in the stack is read. It must come to the top of the stack, but it is now clean for some sizes and dirty for others. We will show that by maintaining a "dirty level" for each block, the stack analysis technique can be extended to analyze write-back. This dirty level is the smallest cache in which the block is dirty. This is the lowest level in the stack to which the block has been pushed since its last write, or infinity if the block has never been written. The dirty level is used to count the number of write-backs for each cache size by assuming that each write results in a write-back, then waiting to see where the write is avoided. If a dirty block is rewritten, then the previous write has been avoided in all dirty sizes, since both the previous and current write can be written-back with one memory access.

Stack analysis can be similarly extended to analyze sector or sub-block caches [Hill84, Lipt68]. In a microprocessor cache, access time, on-chip data path widths, and pin-counts favor small blocks, whereas the size of associative lookup circuitry and tags favor fewer, larger blocks. A possible compromise is to break each block into independent sub-blocks, any or all of which may be present. Again, we will show that stack analysis can be applied by maintaining additional data with each block.

The remainder of this chapter is divided into five sections. Section 3.3 develops the stack algorithm for write-back, followed in Section 3.4 by several simple extensions to handle deletions, periodic write-back, and cache flush. Section 3.5 similarly presents the algorithm for sector caches, including an extension for a useful form of prefetch. Section 3.6 presents a comparison of the time required to perform analysis using the stack technique. This section also shows how stack analysis makes possible other useful measurements, exemplified by a computation of the probability of a write-back as a function of memory size.

## 3.3. Write-Back Stack Algorithm

We begin by discussing the problems with write-back stack analysis, then present a general non-stack algorithm for computing the write-back ratios. We then prove that the algorithm obeys a form of inclusion, and derive a corresponding stack algorithm.

### 3.3.1. The Write-Back Problem

In write-back, a write access to the secondary storage occurs whenever a dirty block is "pushed". The main problem with write-back is maintaining the "state" (clean or dirty) of each block in the stack. A single dirty bit is sufficient in the real cache, however it clearly is not for the simulation stack. Consider a read to a dirty block at level $k$. For sizes $k$ and larger the block is still dirty, since it has not been written; for sizes 1 to $k-1$ it is clean. The inclusion property is violated since the contents of the larger cache is "different" in the sense that the block has different attributes in some larger sizes. A second problem is accounting for the "dirty pushes". Each miss from a memory of size $C$ causes a push from each smaller memory; that pushed block may be dirty. On first inspection, this suggests that counts need to be maintained and updated for every memory size from which a dirty block is pushed. We will show that a surprisingly simple technique solves both of these problems.

### 3.3.2. A Non-stack Algorithm

We begin by assuming that write-back is not a stack algorithm, and imagining a general algorithm for computing write-back miss or transfer ratios. The algorithm is based on the stack analysis algorithm from Section 2.3, but maintains a separate set of dirty blocks for each cache size in order to solve the problem of the non-inclusion of dirty bits. In addition to the symbols defined in Section 2.3, let:

$$w \quad = w_1,w_2...w_N \text{ where } w_t = \begin{cases} x_t & \text{if } x_t \text{ is a write} \\ \phi & \text{otherwise} \end{cases} .$$

$D_t(C)$= the set of dirty blocks in a memory of size $C$ after the reference to $x_t$.

$p_t(C)$ = the dirty block "pushed" from a memory of size $C$ by reference $x_t$

$$= \begin{cases} y_t(C) & \text{if } y_t(C) \in D_{t-1}(C) \\ \phi & \text{otherwise} \end{cases}$$

$dp(C)$= the number of blocks written back from a memory of size $C$ by time $t$.

Algorithm 3.1. General Non-stack Write-Back Algorithm

```
1.   For $1 \le t \le N$                                                For all events
2.     If $x_t \notin S_{t-1}$ then $\Delta = \infty$                  If not referenced before.
3.     else
4.       Find $\Delta$ such that $s_{t-1}(\Delta) = x_t$                Find the stack distance
5.       $rh(\Delta) = rh(\Delta)+1$                                    Update the read hits
6.     If $\Delta \ne 1$                                                If stack needs updating
7.       $y_t(1) = s_{t-1}(1)$                                          Calculate push set.
         for $2 \le i < \Delta$ do $y_t(i) = pmin[y_t(i-1), s_{t-1}(i)]$
         for $i \ge \Delta$ do $y_t(i) = \phi$
7a.      for $2 \le i \le \Delta$ do                                    Calculate dirty push set
           If $y_t(i) \in D_{t-1}(i)$ then                              If block is dirty
             $p_t(i) = y_t(i)$                                          Include in dirty push set
             $dp(i) = dp(i)+1$                                          Count dirty pushes
           else $p_t(i) = \phi$
8.       $s_t(1) = x_t$                                                 Establish new stack.
         for $i > 1$ do $s_t(i) = s_{t-1}(i) + y_t(i-1) - y_t(i)$
8a.      for $i \ge 1$ do $D_t(i) = D_{t-1}(i) + w_t - p_t(i)$          Establish new dirty set.
```

We define Algorithm 3.1 by adding steps 7A and 8A to Algorithm 2.1. When a block is written it must be added to each dirty set (line 8A). A block is removed from a set if and only if a dirty block is pushed from memory (line 7A). Note that if write fetch is not used then line 5 of Algorithm 3.1 must be conditioned on a read, i.e. If $w_t = \phi$ then $rh(\Delta) = rh(\Delta)+1$.

### 3.3.3. Dirty Set Inclusion Property

The inclusion property of stack algorithms states that if a block is present in memory of size $C$ then it is present in size $C+1$, and therefore in all larger sizes. This can be formally stated as $M_t(C) \subseteq M_t(C+1)$, for all $t$ and $C$. We now show that a similar condition applies to dirty sets, that is, if a block is dirty in a memory of size $C$ then it is dirty in all larger sizes.

An intuitive argument of this fact is the following. In order to become dirty a block must be written, which makes the block dirty in all sizes. A block becomes clean only when it is replaced (ignoring deletions for now). Because the replacement algorithm is a stack algorithm, the block is always pushed from a smaller cache before it is pushed from a larger one. The dirty level is therefore the maximum level to which the block has been pushed since it was written. A read may pull the block to the top of the stack, but will leave it dirty in an inclusive set of sizes. There is no way to make it dirty in some sizes without making it dirty in all sizes, therefore inclusion holds. A more formal proof follows.

**THEOREM 3.1:** $D_t(C) \subseteq D_t(C+1)$, for all $t$ and $C$.

**PROOF:** Choose an arbitrary $C$. The condition certainly applies at the start of the simulation when no blocks are in cache, therefore the dirty sets are empty. Assume it to be true at time $t-1$. Beginning with this induction hypothesis, the proof adds and subtracts blocks from each side, preserving the subset relation, and finally arrives at an expression for the dirty sets at time $t$.

$$D_{t-1}(C) \subseteq D_{t-1}(C+1)$$

Adding the possibly null block $w_t$ to both sets does not affect the subset relation.

$$D_{t-1}(C)+w_t \subseteq D_{t-1}(C+1)+w_t$$

Similarly, the relation holds if the block $p_t$ is removed from the smaller set.

$$D_{t-1}(C)+w_t-p_t(C) \subseteq D_{t-1}(C+1)+w_t$$

Finally, removing the same block from both sets preserves the subset relation.

$$D_{t-1}(C)+w_t-p_t(C)-p_t(C+1) \subseteq D_{t-1}(C+1)+w_t-p_t(C+1)$$

Note that the right-hand side is exactly $D_t(C+1)$ as computed by line 8A, while the left-hand side differs from $D_t(C)$ only by the term $p_t(C+1)$. There are three possible values for $p_t(C+1)$ none of which affect the set on the left-hand side:

a) if $y_t(C+1)$ is not dirty then $p_t(C+1)=\phi$;

b) if $y_t(C+1)$ is dirty, and $y_t(C+1)=y_t(C)$ then $p_t(C+1)=p_t(C)$. Subtracting this block twice can not affect the contents of the left-hand set.

c) if $y_t(C+1)$ is dirty, and $y_t(C+1)\neq y_t(C)$ then $p_t(C+1)\neq p_t(C)$. However, it must be true that $y_t(C+1)=s_{t-1}(C+1)$, that is, the block pushed from size $C+1$ was the block at level $C+1$. But $s_{t-1}(C+1)\notin M_{t-1}(C)$, and therefore $p_t(C+1)\notin D_{t-1}(C)$, so again it has no effect.

Removing this term gives

$$D_{t-1}(C)+w_t-p_t(C)\subseteq D_{t-1}(C+1)+w_t-p_t(C+1)$$

which is exactly equal to

$$D_t(C)\subseteq D_t(C+1)$$

as set by Algorithm 3.1 line 8A. □

With these facts we can simplify the algorithm considerably. First, $D_t(C)\subseteq D_t(C+1)$ implies that there is a minimum size at which a block is dirty (if it is dirty at all). Intuitively, this is the smallest memory from which the block has not been pushed since its last write reference, and therefore the smallest memory size in which it is still dirty. This is also the largest stack distance the block has attained since it was last written. Therefore the separate $D_t(C)$ can be replaced by a single array. Let $dl(x)$ be the *dirty level* of block $x$; infinity if the block has never been written. A block at level $k$ (i.e. $s(k)=x$) is dirty if and only if $dl(x)\leq k$. We can set the dirty level to 1 when a block is written and update it as the block is pushed.

### 3.3.4. Writes Avoided

Before defining the new algorithm, let us also reconsider the way dirty pushes are counted. In Algorithm 3.1, *dp* is updated as each block is pushed. Also, recall that the purpose of write-back is to avoid the write-back to secondary storage for each write reference which is required when using write-through. We can count the number of write-backs required in two ways. One is to count them directly. The other is to count the total number of writes, and then to subtract the number of times that no *additional* write-back is required, since the block was already dirty or is being deleted. When a write does not require a write back, we increment the count of *writes avoided.* This is analogous to the way reads are computed in the basic stack analysis algorithm, where a read is avoided for all sizes larger that the current stack distance.

Ignoring deletes for now, a write is avoided only when a dirty block is overwritten, since both the previous and current modification can be written by the next write-back. Therefore we can say that the previous write has been avoided for all sizes equal to or greater than the current dirty level. Notice that we now only care about the dirty level for the block being referenced and therefore we only need to adjust $dl$ for the referenced block. If it is found at level $\Delta$ which is below its dirty level (i.e. $\Delta > dl(x_t)$), we can reason that the block has been pushed (while dirty) from all levels between $dl(x_t)$ and $\Delta$, therefore the proper value for $dl(x_t)$ is $\Delta$. (See line 6 of Algorithm 3.2)

We now define $wa(C)$ to be the *writes avoided* at level $C$, that is, the number of writes for which the referenced block was still dirty in memory sizes $C$ and larger. The write-back stack algorithm, Algorithm 3.2, is shown below. The differences between this algorithm and Algorithm 2.1 are line 6 which adjusts the dirty level as described above, and lines 10-13 which count the writes avoided and write references, and reset the dirty level to one on a write.

For the special case of LRU, this algorithm is particularly simple. As in the standard stack analysis algorithm for LRU, updating the stack is a matter of removing the referenced block and inserting it at the top of the stack. The fact that only the referenced block affects the statistics is particularly useful for this case, since no work needs to be done while searching for the referenced block.

Algorithm 3.2: Write-Back Stack Algorithm

```
1.  For $1 \le t \le N$ do                                              For all events
2.    If $x_t \notin S_{t-1}$ then $\Delta=\infty$                      If not referenced before.
3.    else
4.      Find $\Delta$ such that $s_{t-1}(\Delta)=x_t$                   Find the stack distance
5.      $rh(\Delta)=rh(\Delta)+1$                                       Update the read hits
6.      If $dl(x_t)<\Delta$ then $dl(x_t)=\Delta$                       Set the "real" dirty level.
7.    If $\Delta \ne 1$                                                 If stack needs updating
8.      $y_t(1)=s_{t-1}(1)$                                             Calculate push set.
        for $2 \le i < \Delta$ do $y_t(i)=pmin[y_t(i-1),s_{t-1}(i)]$
        for $i \ge \Delta$ do $y_t(i)=\phi$
9.      $s_t(1)=x_t$                                                    Establish new stack.
        for $i>1$ do $s_t(i)=s_{t-1}(i)+y_t(i-1)-y_t(i)$
10.   If $w_t \ne \phi$ then                                            If this is a write.
11.     If $dl(x_t) \ne \infty$ then
          $wa(dl(x_t))=wa(dl(x_t))+1$                                   Count writes avoided.
12.     $dl(x_t)=1$                                                     Block is dirty.
13.     $W_t=W_{t-1}+1$                                                 Count of write references.
```

### 3.3.5. Dirty Push Computation

*Using Algorithm 3.2, the number of dirty pushes which have occurred by time* $t$ for a memory of size $C$ is given by

$$dp(C)=W_t-\sum_{i=1}^{C} wa(i) - |D_t(C)| \tag{3.1}$$

where the count of write references by time $t$ is

$$W_t = \sum_{i=1}^{t} (1 : w_i = x_i)$$

and the count of dirty blocks resident in the cache of size $C$ is the size of the set

$$D_t(C) = \{x : x=s_t(\Delta), \Delta \le C, dl(x) \le C\}$$

The first two terms of (3.1) are obvious, but we should elaborate on the need for the third term. It should be clear that each block which is still dirty has avoided the most recent write for all sizes in which it is still dirty and should therefore be subtracted from the count of writes. This argument applies at any point during the trace, and at the end of the simulation. Since the relevant metrics are those gathered during the trace period, regardless of any activity which occurs after the trace ends, we should consider each dirty block remaining at the end of simulation as having avoided a write. To simplify the computations, we make a final scan of the memory stack and update $wa(dl(x))$ for each dirty block $x$. We can then eliminate the third term of (3.1). Of course, the effect of this should be small if the total number of trace events is large.

Using this expression for the number of dirty pushes leads to a simple recurrence for computing the transfer ratio. Recall that equation (2.7) for computing the transfer ratio from Section 2.2 is

$$T(C) = [m_r(C) + m_w(C) * Wf + dp(C)] / N$$

Assuming write-fetch, the first two terms can be replaced by the stack analysis computation of the miss ratio given by (2.8), giving

$$T(C) = [[N - \sum_{i=1}^{C} rh(i)] + dp(C)] / N$$

Substituting (3.1) for $dp(C)$, assuming that the final scan has updated $wa$, this simplifies to

$$T(C) = [N - \sum_{i=1}^{C} rh(i)] + [W_t - \sum_{i=1}^{C} wa(i)] / N$$

$$T(C) = [(N + W_t) - \sum_{i=1}^{C} (rh(i) + wa(i))] / N \tag{3.2}$$

or

$$T(0) = (N + W_t) / N \tag{3.3}$$

$$T(C) = T(C-1) - [(rh(C) + wa(C)) / N]$$

Notice that since $rh(i)$ and $wa(i)$ are both non-negative, this function also decreases as memory size increases, just as the miss ratio does.

### 3.3.6. Warm Start

If the simulation results are gathered starting from an empty stack, the results can be biased by the fact that many of the early references will be misses in all cache sizes. In fact, until the memory contains $k$ blocks there is no chance of a hit at level $k$, producing a higher than expected miss ratio. In some situations this *cold-start* miss ratio is appropriate, for example when a single-program address trace is used to derive multi-programming metrics [East78]. In other situations, the desired metrics are those for a system in steady-state. In these cases it is common to *warm start* the simulation to reduce startup effects. A warm start consists of allowing the simulation to run until it is assumed to be in steady-state, often either for a fixed number of events or until the memory contains a fixed number of blocks, then stopping. Without changing the state of the simulation, all statistics are cleared. The simulation then resumes from its current state. The final metrics are those gathered after the warm start.

Warm start using the write-back algorithm can produce an anomaly in the transfer ratio. This is caused by the final scan of memory which considers all dirty blocks as having avoided a write — a write which may have occurred before the warm start. Suppose, for example, that the write-back simulation is warm started, and suppose that $W_t$ and $wa$ are zeroed. Then immediately after warm start, the value of $dp(C)$ calculated using (3.1) may be negative for some values of $C$, as shown in Figure 3.1, where the number in parentheses is the dirty level of the block. Of course, a "negative push" is meaningless. We can keep the numbers positive by setting $W_t$ to the number of dirty blocks in the cache at warm start, but then $dp$ is immediately non-zero for some cache sizes. Another alternative would be to zero both $wa$ and $dl$, but then it will be a long time before any dirty block could be pushed from large sizes — in conflict with the reason to warm start in the first place.

| Level | Stack | *wa* | *dp* |
|---|---|---|---|
| 1 | A(1) | 0 | -1 |
| 2 | B(4) | 0 | -1 |
| 3 | C(∞) | 0 | -1 |
| 4 | D(4) | 0 | -3 |
| 5 | E(5) | 0 | -4 |

**Figure 3.1:** The count of dirty pushes may be negative after warm start if $W_t$ and $wa$ are zeroed.

Since the third term of (3.1) increases with $C$, the second term of (3.1), the sum of $wa$, must decrease for larger $C$ if we want the computed value of $dp$ to be zero immediately after warm start. This can only happen if some $wa$ are negative. The solution we use is to zero $wa$ at warm start, then *decrement* $wa[dl(x)]$ for all dirty blocks $x$. With this solution $dp(C)$ is zero immediately after warm start for all $C$, as it intuitively should be. See Figure 3.2a. Now suppose that a total miss causes all blocks to be pushed (Figure 3.2b). The result is that $dp(C)$ is zero except for those sizes from which a dirty block is pushed — which is exactly the result obtained from a simulation of a single cache size, or a real cache.

| Level | Stack | *wa* | *dp* |
|---|---|---|---|
| 1 | A(1) | -1 | 0 |
| 2 | B(4) | 0 | 0 |
| 3 | C(∞) | 0 | 0 |
| 4 | D(4) | -2 | 0 |
| 5 | E(5) | -1 | 0 |

(a)

| Level | Stack | *wa* | *dp* |
|---|---|---|---|
| 1 | F(∞) | -1 | 1 |
| 2 | A(1) | 0 | 0 |
| 3 | B(4) | 0 | 0 |
| 4 | C(∞) | -2 | 1 |
| 5 | D(4) | -1 | 1 |
| 6 | E(5) | 0 | 0 |

(b)

**Figure 3.2**: Revised count of dirty pushes after warm start. Figure (a) is immediately after warm start with $D_t$=4, while (b) is after all blocks are pushed one level. The count of dirty pushes from each size, $dp(C)$, agrees with the results from a real cache.

Note, however, the unexpected result that the transfer ratio due to dirty pushes is no longer a monotone decreasing function of size. In fact, if the warm start of Figure 3.2(a) were followed by the unlikely event of five total misses, the resulting transfer ratio would be *increasing* with cache size. It seems that the rate of dirty pushes may be exaggerated for larger cache sizes by the fact that there are more dirty blocks in the larger cache. (There may also be a higher probability that blocks pushed from larger caches are dirty. See Section 3.5) This "error" for large sizes is bounded by the number of dirty blocks in the stack divided by the number of references after warm start. It can therefore be made arbitrarily small by increasing the number of references after warm start (which also reduces the need for warm start). In most cases, locality will cause the write-back traffic ratio to assume its normal decreasing form.

## 3.4. Extensions

In addition to write-back, several intermediate and related policies can be analyzed using our technique.

### 3.4.1. Write-Through

This policy is trivially included in the algorithm by setting $dl(x_t)$ to infinity instead of one after a write. If fact, since the total number of write requests is known, both the write-back and write-through transfer and traffic ratios are available simultaneously. It is also possible to simulate a combination of policies, provided the choice of policy is not a function of memory size. For example, some blocks could be write-through and others write-back, a scheme used in some real caches, for example the Fairchild CLIPPER processor [Cho86] and the NEC disk cache [Toku80].

Algorithm 3.3: Mixed Write-Back/Write-Through Stack Algorithm

```
 1.  For $1 \le t \le N$ do                                        For all events
 2.    If $x_t \notin S_{t-1}$ then $\Delta=\infty$                If not referenced before.
 3.    else
 4.      Find $\Delta$ such that $s_{t-1}(\Delta)=x_t$             Find the stack distance
 5.      If $w_t=\phi$ or block is write-back                      If this is not write-thru
 6.        $rh(\Delta)=rh(\Delta)+1$                               Update the read hits
 7.      If $dl(x_t)<\Delta$ then $dl(x_t)=\Delta$                 Set the "real" dirty level.
 8.    If $\Delta \ne 1$                                           If stack needs updating
         and not ( $\Delta=\infty$ and block is write-thru)        and not a write-thru miss
 9.      $y_t(1)=s_{t-1}(1)$                                       Calculate push set.
         for $2 \le i < \Delta$ do $y_t(i)=pmin[y_t(i-1),s_{t-1}(i)]$
         for $i \ge \Delta$ do $y_t(i)=\phi$
10.      $s_t(1)=x_t$                                              Establish new stack.
         for $i \ge 1$ do $s_t(i)=s_{t-1}(i)+y_t(i-1)-y_t(i)$
11.    If $w_t \ne \phi$ then                                      If a write.
12.      If $dl(x_t) \ne \infty$ then                              Update dirty pushes.
           $wa(dl(x_t))=wa(dl(x_t))+1$                             Count writes avoided.
13.      If block is write-back then                               Write to write-back block.
           $dl(x_t)=1$                                             Block is dirty.
14.      $W_t=W_{t-1}+1$
```

An example of an algorithm for such a cache is given as Algorithm 3.3. The differences between this and Algorithm 3.2 are very superficial. First, line 5 checks for a read or write-back before counting fetches; it assumes there is no fetch-on-write for write-through blocks. Line 11 treats all blocks the same, but in reality only write-back blocks will ever be dirty, so writes are only avoidable for write-back blocks. Line 13 checks that only write-back blocks become dirty.

All writes are included in $W_t$. As a simplification, both write-back and write-through are counted the same. In reality, a write-through may involve less data and therefore is less costly.

### 3.4.2. Periodic Write-Back

With large caches, there may be a very long delay before a block is removed by replacement. We have mentioned that reliability considerations may dictate that a dirty block be written before this time. Suppose that all dirty blocks are written every $n$ seconds instead. An example of this is the UNIX file system policy of writing all dirty file system buffers to disk every 30 seconds. Alternatively, suppose only certain blocks are written, for example by a policy to write a block after it has been unreferenced for $n$ seconds. These policies are all stack algorithms, provided that the write happens for all memory sizes where the block is dirty, in order to maintain inclusion in the dirty set.

A forced write-back is implemented in the algorithm by setting $dl(x)$ to infinity for each written block. It has no effect on writes avoided, except that the write which made the block dirty can not subsequently be avoided. The effect of this is to increase the calculated number of dirty pushes. Consider the third term in (3.1) for any $C$ where the block is dirty: the block was dirty and included in $D_t(C)$; it is now clean and not in the term; the net increase to $dp(C)$ is 1.

### 3.4.3. Deletions

A important consideration in file system studies is the existence of deletions in the reference string. If a file is deleted, the blocks of that file should be removed from the cache without write. With a write-back cache and short file lifetimes, it is likely that file blocks will be created and deleted without ever being written to the next level [Oust85]. Deletions also occur in processor caches when blocks are invalidated, but generally not without writing the block first if it is dirty. This case is discussed in Section 3.4.4.

Deletion of blocks from the cache was discussed by Mattson et al. [Matt70] in the context of a "call back" hierarchy, where cache blocks may be invalidated by a write directed to a lower level. The example used by Mattson is a virtual memory system in which all I/O occurs to blocks residing in an "I/O Subsystem", not the CPU memory. If an I/O is addressed to a block which is in CPU memory, that block must be invalidated. Greenburg [Gree74] also discusses deletions, and implemented an algorithm to approximate the effect of deletion. Olken [Olke81] proposes an exact algorithm, and discusses implementation using various data structures. None of these consider the effect of write back.

If a deleted block were simply deleted from the stack, the stack level for all lower blocks would be reduced. This would have the undesirable effect of calling these blocks back into a memory from which they had been pushed. Instead, what Mattson called a "marker" block is inserted in the stack replacing the deleted block.

---

| Stack Level | Initial Stack | Delete D | Reference B (above gap) | Reference F (below gap) |
|---|---|---|---|---|
| 1 | A | A | B | F |
| 2 | B | B | A | B |
| 3 | C | C | C | A |
| 4 | D | $\gamma$ | $\gamma$ | C |
| 5 | E | E | E | E |
| 6 | F | F | F | $\gamma$ |

**Figure 3.3**: A gap "jumps" down the stack. The gap ($\gamma$) is unaffected by references above it, while a reference below it causes it to "jump" to the level of the referenced block.

---

We refer to the marker blocks as *gaps* in the stack, corresponding to a vacant block in all larger caches. The next push from above the gap will replace the gap with the pushed block, since no block needs to be replaced in a cache containing a vacant block. Thus a gap will stop the sequence of stack updates, just as finding the referenced block stops the pushes in the normal case. However, since the referenced block must still be pulled to the top and blocks below the referenced block do not change stack level, the referenced block must be replaced in the stack by another gap. Thus, a reference to a block below the first gap will seem to make the gap "jump" down the stack. As an example, consider the sequence of Figure 3.3. After block D is deleted, a gap is left at level 4. A reference to block **B** above level 4 will not affect the gap. However, the reference to block **F** below level 4 will "jump" the gap to the stack level of **F**. From the point of view of the "real" cache, the gap represents the same vacant block, which was in all memory sizes 4 or larger. Since block **F** is already resident in memories of size 6 or larger, the reference to F has not fetched any block to fill the gap. Therefore the gap still exists in these sizes.

The effect of deletions on the transfer ratio is to introduce another way in which a write can be avoided, particularly evident in large cache sizes. If a block is written then deleted before it is pushed, the copy-back is avoided for all sizes greater than the current dirty level. It is therefore a simple matter to increment the appropriate $wa[dl(x_t)]$ on deletion. In addition, the count of read hits must exclude deletes, since a deleted block is never fetched. This is seen in lines 6 and 7 below.

Algorithm 3.4: Write-Back Stack Algorithm with Deletes

```
1.   For $1 \le t \le N$ do                                              For all events
2.     If $x_t \notin S_{t-1}$ then $\Delta=\infty$                            If not referenced before.
3.     else
4.       Find $\Delta$ such that $s_{t-1}(\Delta)=x_t$                       Find the stack distance
5.       If $dl(x_t)<\Delta$ then $dl(x_t)=\Delta$                          Set the "real" dirty level.
6.       If $x_t$ is a delete then
           $wa(dl(x_t))=wa(dl(x_t))+1$                                   Count writes avoided.
           $s_t(\Delta)=\gamma$                                            Store a gap in the stack.
           break                                                        Process next reference
7.       else $rh(\Delta)=rh(\Delta)+1$                                     Update the read hits
8.     $\Gamma=\min(i : s_{t-1}(i)=\gamma)$                                 Level of the first gap.
9.     $\Delta'=\min(\Delta,\Gamma)$                                        Level where pushes stop.
10.    If $\Delta' \ne 1$                                                 If stack needs updating
11.      $y_t(1)=s_{t-1}(1)$                                              Calculate push set.
         for $2 \le i < \Delta'$ do $y_t(i)=pmin[y_t(i-1),s_{t-1}(i)]$
         for $i \ge \Delta'$ do $y_t(i)=\phi$
12.      for $i>1$ do $s_t(i)=s_{t-1}(i)+y_t(i-1)-y_t(i)$                 Establish new stack.
13.    $s_t(1)=x_t$                                                       Pull reference to top.
14.    if $\Delta'=\Gamma$ then $s_t(\Delta)=\gamma$                          Jump the gap
15.    If $w_t \ne \phi$ then                                             If this is a write.
16.      If $dl(x_t) \ne \infty$ then
           $wa(dl(x_t))=wa(dl(x_t))+1$                                   Count writes avoided.
17.      $dl(x_t)=1$                                                      Block is dirty.
18.      $W_t=W_{t-1}+1$                                                  Count of write references
```

The complete, though somewhat complicated, algorithm for write-back with deletions is given as Algorithm 3.4. Let:

$\gamma$ = a gap marker in the stack.

$\Gamma$ = the level of the first gap in the stack.

$\Delta'$ = $\min(\Delta,\Gamma)$, the level at which pushes stop.

There are actually only a few changes between Algorithm 3.2 and Algorithm 3.4. First, line 6 handles a deletion by updating the count of writes avoided and replacing the block by a gap in the stack. Line 8 computes $\Gamma$, the level of the top-most gap, while line 9 determines whether the referenced block or $\Gamma$ stops the sequence of updates. Line 10 uses this value instead of $\Delta$. A subtle change in line 13 inserts $x_t$ on top of the stack even if $\Delta'=1$; this handles the case where there is a gap at the top of the stack. Finally, line 14 replaces the referenced block with a gap if it was below the first gap, implementing the "jump" of a gap described above.

### 3.4.4. Flush Back

In some situations a block should be written and removed from the cache before it is a candidate for replacement. An example is the wholesale flush of a local processor cache in a multiprocessor system. A more selective example is where an individual block is flushed from a private cache on a multi-processor bus so that another processor can acquire the block [Katz85]. Flushing the cache periodically is also used in some processor simulations to approximate multiprogramming effects [Smit82]. It should be clear that this can be simply implemented as a periodic write-back followed by a delete. The contents of *wa* is unchanged.

## 3.5. Sector Cache Simulation

We now consider the study of sub-block or sector caches, and show that they too can be simulated using stack analysis by a technique similar to that used for write-back. Although sector caches are typically not appropriate for file system caches, this discussion is included to demonstrate the generality of the proposed techniques.

### 3.5.1. Background

A typical cache consists of blocks or sectors of data, each with associated tags identifying the virtual addresses contained in the block. See Figure 3.4(a). If smaller blocks are used the total space for tags increases (Figure 3.4(b)), since each of the blocks requires its own tags. Regardless of size, each block also requires a valid bit indicating whether the block contains valid data.

An alternative arrangement is the sub-block or sector cache. In a sector cache each cache block/sector is divided into a fixed number of sub-blocks/sub-sectors. (Throughout this section we will use IEEE-proposed terminology for such caches, referring to *sectors* and *sub-sectors.*) Goodman also uses the terms *address block* and *transfer block* for sectors and subsectors respectively [Good83]. Tags are associated with the sector as a whole. See Figure 3.4(c). Transfers between the cache and secondary storage are done in units of sub-sectors. In addition, there must be a valid bit for each sub-sector to indicate whether or not the sub-sector data is present.

---

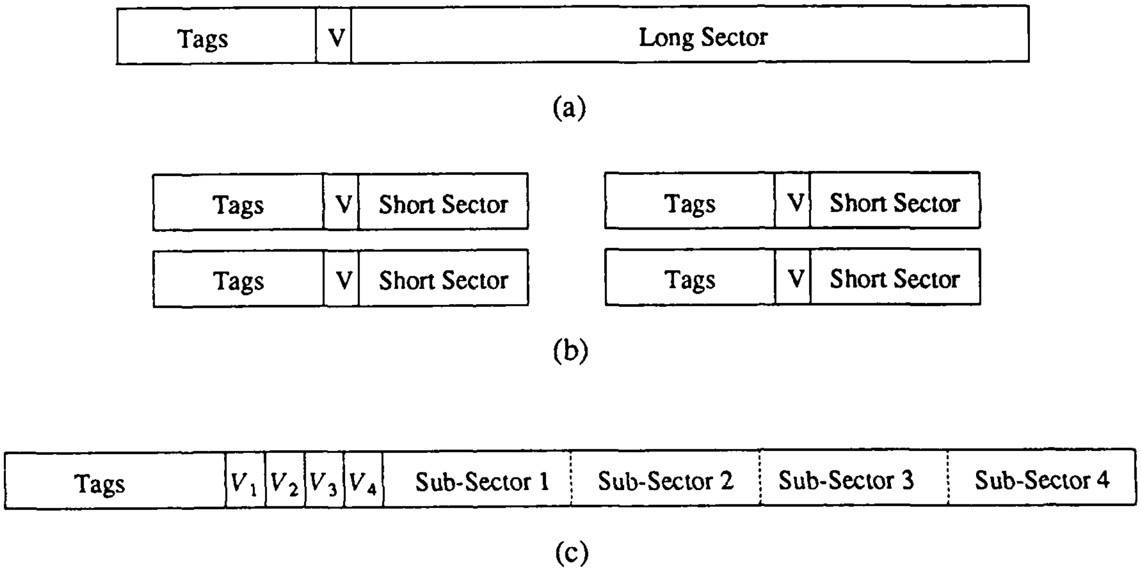

Tags
V
Long Sector
(a)
Tags
V
Short Sector
Tags
V
Short Sector
Tags
V
Short Sector
Tags
V
Short Sector
(b)
Tags
$V_1$
$V_2$
$V_3$
$V_4$
Sub-Sector 1
Sub-Sector 2
Sub-Sector 3
Sub-Sector 4
(c)

**Figure 3.4**: Alternative layouts for a block/sector in a cache. Long sectors reduce the ratio of tags to data, while short sectors reduce the fetch delay. Sub-sector caches combine the advantages of both.

Sector caches are motivated by two factors. The first is a need to reduce the number of tags to be searched. This was the motivation when it was first used in the IBM 360/85 cache [Lipt68]. The reduced number of tags also reduces the chip area needed for tags in a VLSI cache. A second reason for sub-sectors is to reduce the size of each data transfer. On a cache chip with limited pins for parallel data transfer, a large sector size would require multiple cycles, where a smaller sub-sector could be transferred in one parallel access. Similarly, on-chip data path widths favor a small sector. The smaller sub-sectors may also be used to reduce memory bus traffic when the bus is a potential constraint [Good83].

A sector cache tends to have a higher miss ratio than the same size cache with sub-sector-sized blocks because the of the rigidity in the assignment of sub-sectors. It may also have a higher miss ratio than the same size cache using sector-sized blocks because each sub-sector can cause a fault. However, misses that fetch smaller sectors may ''cost'' less than larger sectors, is some cases. At the same time, the sector cache reduces the traffic ratio compared to the non-sector cache with large blocks by not loading sub-sectors which are not needed, as would be the case if the entire sector were loaded. The performance of sub-sector processor cache was studied by Hill and Smith [Hill84].

The disk cache in the IBM 3880 Control Unit is also a form of sector cache [Gros85]. The sector size is a full track, with a variable number of sub-sectors — one for each disk record. This organization was chosen so that the cache could be a physical and logical copy of the disk contents, while offering the advantages of caching. To avoid holding up the processor waiting for a full track to be transferred, the disk is positioned to the requested record which is then transferred to both the processor and controller cache. After signaling completion of the requested I/O, the controller continues to read to the end of the track into cache, anticipating further sequential requests. This form of prefetch is called *load forward*, and is discussed in Section 3.5.3.2.

### 3.5.2. The Stack Simulation Problem

The problem with stack simulation of a sector cache is that the valid bits do not obey inclusion. For example, suppose sub-sector 1 of a sector is referenced and becomes valid. Now suppose that the sector is pushed to level *k* in the stack, then sub-sector 2 is referenced. The entire sector must be pulled to the top of the stack in order for sub-sector 2 to become valid in all cache sizes, but sub-sector 1 is valid for some sizes (*k* and larger), and invalid for others.

Our solution is to replace the valid bit with a *valid level* for each sub-sector. The valid level is the minimum memory size for which the sub-sector is still valid; infinity if the sub-sector has never been referenced. A reference to any sub-sector will pull the entire sector to the top of the stack. As in the case of write-back, there is no need to adjust the valid levels as a sector is pushed; if a sub-sector has a valid level less than the current stack level of the sector, the valid level is set to the current level, since no sub-sector can be valid in smaller cache sizes. Finally, the referenced sub-sector is assigned a valid level of one, since it must be present.

Algorithm 3.5: Sub-sector Stack Algorithm

```
1.   For $1 \le t \le N$ do                                      For all events
2.     If $(x_t,*) \notin S_{t-1}$ then $\Delta=\infty$                If sector not in stack
3.     else
4.       Find $\Delta$ such that $s_{t-1}(\Delta)=(x_t,*)$              Find the stack distance
5.       for $1 \le j \le B$ do
           $vl_t(x_t,j)=max(vl_{t-1}(x_t,j),\Delta)$               Fix valid levels
6.       $\Delta_a=vl_t(x_t,a_t)$                                Stack distance for (x,a)
7.       $rh(\Delta_a)=rh(\Delta_a)+1$                           Update the read hits
8.     If $\Delta \ne 1$                                         If the stack needs updating
9.       $y_t(1)=s_{t-1}(1)$                                  Calculate push set.
         for $2 \le i < \Delta$ do $y_t(i)=pmin[y_t(i-1),s_{t-1}(i)]$
         for $i \ge \Delta$ do $y_t(i)=\phi$
10.    $s_t(1)=(x_t,a_t)$                                     Establish new stack.
       for $i>1$ do $s_t(i)=s_{t-1}(i)+y_t(i-1)-y_t(i)$
11.    $vl_t(x,a)=1$                                          (x,a) valid in all sizes.
```

The formal algorithm is similar to the one for write-back, and is presented as Algorithm 3.5. The terms are somewhat different from those used previously.

Let:

$X$ = $(x_1,a_1),(x_2,a_2)...(x_N,a_N)$ be a series of references, where $(x_t,a_t)$ is a reference to sector $x$, sub-sector $y$ at time $t$.

$(x,*)$ = any sub-sector of sector $x$.

$B$ = number of sub-sectors per sector.

$vl_t(x,a)$ = the valid level of sub-sector $(x,a)$.

Line 5 adjusts the valid level of all sub-sectors to ensure that they are smaller than the level of the sector. Then notice that the count of hits in line 7 is based on $\Delta_a$, the valid level of the referenced sub-sector — not the stack level of the sector as a whole. For example, in Figure 3.5, a reference to sub-sector **A1** is a hit at level 4, since the sub-sector is not present in sizes smaller than 4. On the other hand, a reference to sub-sector **B2** is a hit only at level 2, since the sector as a whole is absent from size 1. However, the stack is updated to the level of the sector as a whole, $\Delta$, since the entire sector is pulled to the top of the stack.

| Stack Level | Sector | Sub-sector | | |
|---|---|---|---|---|
| | | 1 | 2 | 3 |
| 1 | A | 4 | 1 | ∞ |
| 2 | B | 2 | 1 | ∞ |
| 3 | C | 1 | 1 | 1 |
| 4 | D | 1 | ∞ | ∞ |

**Figure 3.5**: Valid levels in a sector cache. Each sub-sector can be valid for different cache sizes, or invalid in all sizes (∞).

### 3.5.3. Extensions

#### 3.5.3.1. Write Back

The first obvious extension is to consider write back with a sector cache. Since these are independent they can be combined by maintaining a dirty level in addition to the valid level. The dirty level could be associated with the entire sector if the cache writes-back entire sectors.

However since part of the motivation for the sub-sector cache is to reduce bus traffic, the dirty level is more logically associated with each sub-sector. The algorithm is similar to those already presented.

### 3.5.3.2. Load Forward

*Load forward* is a form of prefetch associated with sector caches [Hill84]. After loading a requested sub-sector, successive sub-sectors are loaded until the end of the sector. As with any prefetch, this reduces the miss ratio because of the strong probability of sequential references. However, unlike normal prefetch, there is no chance that load forward can cause memory pollution [Smit78b] by displacing a soon-to-be-referenced sector.

Load forward may be implemented as either a demand or non-demand prefetch policy. However, we will show that even with demand prefetch, load forward is a stack algorithm. The key that distinguishes load forward from other forms of prefetch is that load forward always prefetches to the end of a sector and no farther, rather than prefetching a fixed number of blocks or sub-sectors. To show that it is a stack algorithm, we again imagine a general algorithm for load forward which does not assume that inclusion holds. The algorithm uses a stack to determine which sector is replaced at each reference, but keeps a separate memory set $M_t(C)$ for all sub-blocks present in size $C$. For simplicity we ignore writes. In addition to symbols previously defined, let:

$(x,a)^+ =$ $(x,a)$ plus the set of all sectors/sub-sectors prefetched with sub-sector $(x,a)$.

$M_t(C) =$ the set of valid sub-sectors in memory of size $C$.

$= \{(x,a) : (x,a)$ in memory of size $C$ at time $t\}$

$s_t(C) =$ the set of valid sub-sectors of the sector that is at stack level $C$. Note that there can be sub-sectors which are valid at larger levels but not at $C$.

$= \{(x,a) : (x,a) \in M_t(C),\ (x,i) \notin M_t(j),\ 1 \le i \le B$, for all $j < C\}$

$y_t(C) =$ the sector pushed from size $C$.

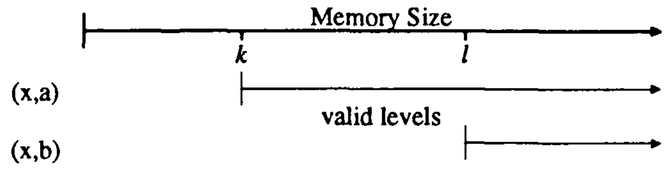
Algorithm 3.6: General Load-Forward Algorithm

```
For $1 \le t \le N$ do                                                    For all events
    If $(x_t,*) \notin S_{t-1}$ then $\Delta=\infty$                                If sector not in stack
    else
        Find $\Delta$ such that $(x_t,*) \in s_{t-1}(\Delta)$                       Find the sector distance
        Find $\Delta_a=\min(C:(x,a) \in M_t(C))$                       Find the sub-sector distance
        $rh(\Delta_a)=rh(\Delta_a)+1$                                        Update the read hits
    If $\Delta \ne 1$                                                   If the stack needs updating
        $y_t(1)=s_{t-1}(1)$                                            Calculate push set.
        for $2 \le i < \Delta$ do $y_t(i)=pmin[y_t(i-1),s_{t-1}(i)]$
        for $i \ge \Delta$ do $y_t(i)=\phi$
    for $1 \le i < \Delta_a$ do $M_t(i)=M_{t-1}(i)+(x,a)^+ - y_t(i)$            Establish new memory.
```

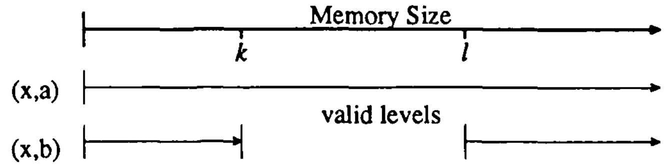
To prove that load forward is a stack algorithm we want to show that inclusion still applies, that is, $M_t(C) \subseteq M_t(C+1)$ for all $t$ and $C$. We can immediately think of a situation where this will be violated. For example, suppose $(x,a)$ prefetches $(x,b)$, and these sub-sectors are valid at the levels shown in Figure 3.6(a). Now let $(x,a)$ be referenced. For all sizes less than $k$, $(x,a)$ is fetched, prefetching $(x,b)$. For sizes greater than $k$, neither sub-sector is fetched. The valid levels, which are shown in Figure 3.6(b), violate inclusion since $(x,b)$ is not present for sizes between $k$ and $l$.

However, suppose the initial configuration is reversed, as shown in Figure 3.7(a). The result of a reference to $(x,a)$ is that $(x,b)$ is prefetched for all sizes less than $l$ (although it only needs to be accessed from secondary storage for sizes less than $k$), resulting in both sub-sectors becoming valid in all sizes. See Figure 3.7(b). Therefore a necessary condition for inclusion is that the first configuration can never occur using load forward.

An intuitive argument that this is the case is the following. The first time that sub-sector $(x,a)$ is referenced, both $(x,a)$ and $(x,b)$ will be fetched and become valid in all sizes. As sector $x$ is pushed, both simultaneously become invalid in the same sizes. To reach the configuration of Figure 3.6, either $(x,a)$ must be fetched without prefetching $(x,b)$, or $(x,b)$ must be pushed from sizes where $(x,a)$ is still valid. Both of these are impossible. To formally prove this, we state the following theorem.

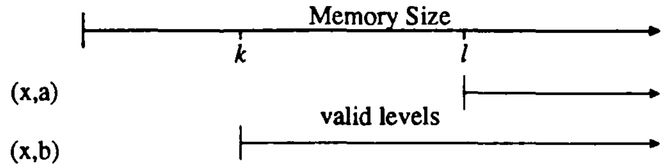

Memory Size
k
l
(x,a)
valid levels
(x,b)

(a) Before reference to (x,a)

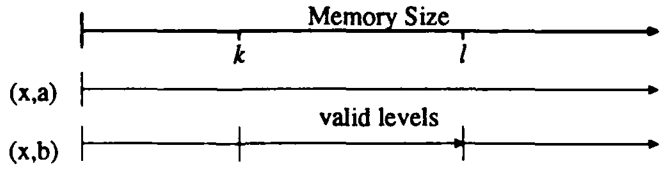

Memory Size
k
l
(x,a)
valid levels
(x,b)

(b) After reference to (x,a)

**Figure 3.6**: A situation where load-forward can violate inclusion. Initially subsector (x,a) is valid in sizes larger than *k*, while subsector (x,b) is valid above *l*. Subsector (x,b) is prefetched along with (x,a) in sizes less than *k*, but not in larger sizes where there is no fetch of (x,a). Validity of (x,b) is not inclusive.


Memory Size
k
l
(x,a)
valid levels
(x,b)

(a) Before reference to (x,a)


Memory Size
k
l
(x,a)
valid levels
(x,b)

(b) After reference to (x,a)

**Figure 3.7**: A situation where load-forward preserves inclusion. If (x,a) is only valid in sizes where (x,b) is also valid then load forward preserves inclusion.

**THEOREM 3.2:** Using Algorithm 3.6, if $(x,b) \in (x,a)^+$ and $(x,a) \in M_t(C)$, then $(x,b) \in M_t(C)$.

**PROOF:** Choose an arbitrary size $C$. The condition is certainly true at the start when the cache is empty. Assume the induction hypothesis that the condition holds at time $t$-$1$. We will show that it holds after the reference at time $t$. Consider the possible configurations of $(x,a)$ and $(x,b)$ which could lead to $(x,a)$ present at time $t$.

Case 1:

Neither $(x,a)$ nor $(x,b)$ present at time $t$-$1$, and $(x,a)$ is referenced. Both sub-sectors are fetched, and the condition holds.

Case 2:

Sub-Sector $(x,b)$ present, but not $(x,a)$, and $(x,a)$ referenced. Both are again fetched, and the condition holds.

Case 3:

Both present, and $(x,a)$ referenced. As shown earlier, both become valid in all sizes, and the condition holds.

Case 4:

Both present, and $(x,b)$ referenced. Sub-sector $(x,b)$ will become valid in all sizes. Although $(x,a)$ is unchanged, the condition still holds.

Case 5:

Both present and another sector referenced. If sector x is not pushed from size $C$, then the condition still holds. If the sector is pushed from size $C$, then neither $(x,a)$ not $(x,b)$ will be present, and the condition holds.

Case 6:

A more subtle case, where both are present and sub-sector $(x,c)$ is referenced. If neither $(x,a)$ nor $(x,b) \in (x,c)^+$ then neither is affected, and the condition holds. If $(x,b) \in (x,c)^+$, or if both are, then the condition holds. However, if $(x,a) \in (x,c)^+$, but not $(x,b)$, then the new configuration violates the condition. Similarly, if neither is present, or $(x,b)$ alone is present, there is no problem unless $(x,a)\in(x,c)^+$ and $(x,b)$ is not.

The condition is therefore true if the prefetch sets obey the transitive condition that if $(x,b)\in(x,a)^+$ and $(x,a)\in(x,c)^+$ then $(x,b)\in(x,c)^+$. This condition is satisfied by load forward since it loads the entire rest of the sector (but would not be if it loads just the next sub-sector, say).

We can now show formally that Algorithm 3.6 satisfies inclusion.

**THEOREM 3.3:** If Algorithm 3.6 is used then $M_t(C)\subseteq M_t(C+1)$ for all $t$ and $C$.

**PROOF:** It is certainly true at the start. Assume it is true at time $t-1$ for an arbitrary size $C$. Again we add and subtract blocks from both sides, which preserves the subset relation, arriving at an expression which shows that it is true at time $t$.

$$M_{t-1}(C)\subseteq M_{t-1}(C+1)$$
$$M_{t-1}(C)-y_t(C)\subseteq M_{t-1}(C+1)$$
$$M_{t-1}(C)-y_t(C)-y_t(C+1)\subseteq M_{t-1}(C+1)-y_t(C+1)$$

By an argument similar to the one used to prove Theorem 3.1, $y_t(C+1)$ can have three possible values, none of which affect the subset the left-hand side of the relation:

a) $y_t(C+1)=\phi$

b) $y_t(C+1)=y_t(C)$

c) $y_t(C+1)\neq y_t(C)$, in which case $y_t(C+1)=s_{t-1}(C+1)$, which is not in $M_{t-1}(C)$. Therefore

$$M_{t-1}(C)-y_t(C)\subseteq M_{t-1}(C+1)-y_t(C+1)$$
$$M_{t-1}(C)+(x,a)^+-y_t(C)\subseteq M_{t-1}(C+1)+(x,a)^+-y_t(C+1)$$

For any $C<\Delta_a-1$, these are exactly the computations of $M_t(C)$ and $M_t(C+1)$ from line 9 of Algorithm 3.6. For $C>\Delta_a-1$, the theorem is true by the induction hypothesis, since the contents of memory are unchanged. This leaves only the case where $C=\Delta_a-1$, in which Algorithm 3.6 computes $M_t(C)$ using line 9, but $M_t(C+1)$ is unchanged. Thus the above relation reduces to

$$M_t(C)\subseteq M_{t-1}(C+1)+(x,a)^+-y_t(C+1)$$

Now, because $C=\Delta a-1$, there is no push from size $C+1$, and $y_t(C+1)=\phi$, so the relation becomes

$$M_t(C) \subseteq M_{t-1}(C+1)+(x,a)^+$$

But $(x,a)^+ \subseteq M_t(C+1)$ by Theorem 3.2, therefore the right-hand side is $M_t(C+1)$, giving

$$M_t(C) \subseteq M_t(C+1)$$

as was to be shown. □

Because of inclusion, we can convert Algorithm 3.5 to a load forward algorithm using valid levels, as follows.

```
Algorithm 3.7: Stack Algorithm for Load Forward

1.    For $1 \le t \le N$ do                                          For all events
2.      If $(x_t,*) \notin S_{t-1}$ then $\Delta=\infty$               If sector not in stack
3.      else
4.        Find $\Delta$ such that $s_{t-1}(\Delta)=(x_t,*)$             Find the sector distance
5.        for $1 \le j \le B$ do $vl_t(x_t,j)=max(vl_{t-1}(x_t,j),\Delta)$   Fix valid levels
6.        $\Delta_a=vl_t(x_t,a_t)$                                      Find sub-sector distance
7.        $rh(\Delta_a)=rh(\Delta_a)+1$                                 Update the read hits
8.      If $\Delta \ne 1$                                              If the stack needs updating
9.        $y_t(1)=s_{t-1}(1)$                                          Calculate push set.
          for $2 \le i < \Delta$ do $y_t(i)=pmin[y_t(i-1),s_{t-1}(i)]$
          for $i \ge \Delta$ do $y_t(i)=\phi$
10.     $s_t(1)=(x,a)^+$                                              Establish new stack.
        for $i>1$ do $s_t(i)=s_{t-1}(i)+y_t(i-1)-y_t(i)$
11.     for $y \le i \le B$ do $vl_t(x,i)=1$                           $(x,a)^+$ valid in all sizes.
```

### 3.6. Experimental Results

In this section we demonstrate two advantages of write-back analysis by reporting the results of simulations using the technique. The first result shows that stack analysis can be much faster than approximating the miss or transfer ratio using several single-size simulations. The second shows that other useful statistics can be produced as a by-product of write-back stack analysis. In this case we analyze the probability that the pushed block is dirty as a function of cache size.

### 3.6.1. The Trace Data

The traces used in these comparisons consist of instruction and data addresses from the execution of programs on one of several machines. They represent a variety of different applications in three different languages. The traces are: FGO1 (IBM 370, Fortran execution, factor analysis), FGO2 (IBM 370, Fortran execution, analysis of satellite data), MVS (standard MVS operating system workload at Amdahl Corp.), LISPCOMP (VAX, LISP compiler, written in LISP), SPICE (VAX, Spice circuit simulator, written in Fortran), VAXIMA (VAX symbolic algebraic manipulation program derived from Macsyma, written in LISP), and RISC (simulated execution of a C compiler for a RISC-architecture processor). All traces except MVS represent the execution of a single program. Most have been used in previous studies [Patt83, Smit85a]. We used a 16-byte block size for all traces, as used in [Smit85a]. The simulations using these traces considered only data caching; instruction fetches were ignored because they are never writes, and to compare the write back results to those of Smith [Smit85a].

Two sets of simulation were done with each program address trace. First, each was simulated as if it were a stand-alone program. This provides a characterization of each program, but generally gives an optimistic prediction of the actual performance of the program in a multiprogramming environment [East78]. In the second experiment we used the technique proposed by Smith [Smit82] to approximate the effects of multiprogramming by writing all dirty blocks and flushing the cache after a fixed time quantum, in our case every 20,000 memory references. The simulations without flushing were warm started; the others were not, producing some variation in the number of read/write events shown in Table 3.1 for the same trace file.

By way of comparison, we also ran simulations using UNIX 4.2 BSD file system traces generated on three university research computers. The traces are identified by machine: ARPA is a VAX 11/780 used for operating system research and development and text processing; ERNIE is a VAX 11/780 used by staff and graduate students for program development and text processing; CAD is a VAX 11/750 used for computer aided design research. All three machines are also used extensively for electronic mail. These traces were analyzed in detail by Ousterhout et al. [Oust85]. The trace events show logical file creation, deletion, opens, closes, and seeks. Actual reads and writes are not recorded, however each close or seek event includes the range of bytes read or written since the last positioning event for the file. The simulator recreates reads and writes in block-size units based on this information. These traces tend to overestimate the miss ratio, since some of the simulated reads/writes were actually several small requests. For these simulations we used a block size of 4096 bytes, consistent with common UNIX 4.2BSD usage. There is no information on program paging, or file system overhead activity such as directories. All files are identified by a logical identifier; there is no data on physical location [Oust85].

Table 3.1: General Trace File Characteristics

| File | Number of Events | Types of Events | | Unique Blocks | Mean Stack Size | Dirty Blocks | Mean Dirty |
|---|---|---|---|---|---|---|---|
| | | Read | Write | | | | |
| Program Address Traces | | | | | | | |
| FGO1N | 233727 | 66.5% | 33.5% | 2675 | 970.03 | 1341 (50.1%) | 501.21 (51.7%) |
| FGO2N | 182290 | 79.4% | 20.6% | 1049 | 614.51 | 660 (62.9%) | 403.03 (65.6%) |
| MVSN | 244292 | 63.4% | 36.6% | 4972 | 2672.71 | 3499 (70.4%) | 1688.25 (63.2%) |
| LISPCOMPN | 224856 | 62.1% | 37.9% | 1764 | 1370.19 | 660 (37.4%) | 423.03 (30.9%) |
| RISCRN | 45975 | 81.3% | 18.7% | 902 | 675.92 | 500 (55.4%) | 310.02 (45.9%) |
| SPICEN | 190460 | 63.2% | 36.8% | 602 | 555.75 | 347 (57.6%) | 322.11 (58.0%) |
| VAXIMAN | 238237 | 65.1% | 34.9% | 4326 | 3035.69 | 1263 (29.2%) | 717.78 (23.6%) |
| Average | 194262 | 68.7% | 31.3% | 2327 | 1413.54 | 1181 (50.8%) | 623.63 (44.1%) |
| Program Address Traces with Flushing | | | | | | | |
| FGO1 | 234696 | 65.8% | 34.2% | 5293 | 154.91 | 3018 (57.0%) | 88.56 (57.2%) |
| FGO2 | 182839 | 79.3% | 20.7% | 4530 | 133.45 | 2106 (46.5%) | 60.90 (45.6%) |
| MVS | 244168 | 63.4% | 36.6% | 17174 | 424.75 | 8921 (51.9%) | 221.79 (52.2%) |
| LISPCOMP | 237867 | 62.1% | 37.9% | 11875 | 317.33 | 2889 (24.3%) | 78.67 (24.8%) |
| RISCR | 50336 | 81.2% | 18.8% | 4357 | 107.46 | 1186 (27.2%) | 29.60 (27.5%) |
| SPICE | 194174 | 63.3% | 36.7% | 5642 | 164.37 | 2253 (39.9%) | 66.67 (40.6%) |
| VAXIMA | 238211 | 65.4% | 34.6% | 14011 | 353.99 | 3556 (25.4%) | 91.79 (25.9%) |
| Average | 197470 | 68.6% | 31.4% | 8983 | 236.61 | 3418 (38.0%) | 91.14 (38.5%) |
| UNIX File System Traces | | | | | | | |
| ERNIE | 475471 | 74.7% | 25.3% | 85119 | 8879.06 | 69024 (81.1%) | 4021.80 (45.3%) |
| ARPA | 492040 | 72.0% | 28.0% | 93930 | 7254.39 | 81002 (86.2%) | 3471.87 (47.9%) |
| CAD | 489962 | 56.8% | 43.2% | 103488 | 11370.72 | 87180 (84.2%) | 4554.18 (40.1%) |
| Average | 485824 | 67.8% | 32.2% | 94179 | 9168.06 | 79068 (84.0%) | 4015.95 (43.8%) |

**Table 3.1:** General Trace File Characteristics This table shows the number of events considered from each trace file, and the percentage of these events which were reads or writes. The fifth column shows the number of unique blocks in the trace, while the next column gives the mean stack size. Flushing reduces the mean stack size, as do deletions from the file system traces. The final two columns show the number of blocks that are written at some time during the trace (dirty blocks), and the mean number of dirty blocks in the stack. These are also shown as percentages of the unique blocks and mean stack size, respectively.

### 3.6.1.1. General Characteristics

Table 3.1 shows the general characteristics of the traces. We processed approximately 500,000 events from each file after warm start, but the count of events in the second column only includes the data references, ignoring instruction fetches for the program address traces. The third and fourth columns show the percentage of these events that are reads or writes. We initially speculated that writes would be a more significant percentage of the file system traces. However when instruction fetches are excluded from the traces to simulate a data-only cache the fraction of writes are comparable. We conclude that writes are a factor which should not be overlooked in any cache design.

The fifth column shows the number of *unique blocks* seen in each trace file. This number is exaggerated by cache flushing because a block reloaded after a flush is considered a new block. It is clear, however, that the file system traces come from a much larger population of blocks. The next column shows the *mean stack size*, that is, the stack distance of the least recently used block averaged over all trace references. These two columns together indicate the range of interesting cache sizes for study. For example, program address traces with flushing seldom use more than a few hundred blocks, whereas 10,000 blocks may be too few for a file system simulation. Notice that the mean stack size for the program address traces without flushing is generally about half the number of blocks for the same trace, whereas the mean stack size for file system traces is less than 10% of the number of blocks. This is because nearly 90% of the file system blocks are deleted. Deletions do not decrease the stack size, but they do leave gaps which allow other blocks to be added without increasing the stack size.

The final two columns indicate the impact of write activity. The column labeled *dirty blocks* shows the number of blocks which are ever written. This figure is also shown as a percentage of the number of unique blocks. The fraction of the blocks in the cache which are dirty will obviously affect the chances that a block must be written when it is pushed. The file system traces have far more dirty blocks (84% compared to 50%). The final column shows the *mean number of dirty* blocks in the cache, shown also as a percentage of the mean stack size. Although there is a wide variation between the individual program address traces, we find that on the average, the fraction of the cache which is dirty is about the same, near 44%, for both programs and files. The reason for the relatively low value for the file system traces, compared to the fraction of written blocks, is that most blocks are deleted before they are pushed very far down the stack, and most of the deleted blocks have been written. The blocks that ''survive'' seem to be equally likely to have been written.

### 3.6.2. Run-Time Comparison

As we stated earlier, the chief advantage of stack analysis is that it allows the desired metrics to be calculated for all cache sizes in a single pass of the trace data. Although the overhead of maintaining the memory stack usually makes stack analysis take longer than the simulation of a single cache size, it should take only a fraction of the time required to produce a reasonable curve using several single-size simulations. We have used our write-back stack algorithm in the analysis of a variety of trace data, and find that this time savings does not always occur. For example, file system traces typically exhibit much poorer locality than single program address traces. This results in excessive run times using the straight-forward implementation of the stack simulator. In the next chapter we present several techniques to reduce the execution time of stack analysis by using a tree-based representation of the stack.

Table 3.2 shows the execution times for simulations using several traces of various types. The trace files are described in more detail in Chapter 4. All simulations use LRU replacement. The first column shows the time required to compute a point on the miss or transfer ratio curve using a simple simulation of a single cache size. The simple simulation maintains an LRU list to determine the block to be removed from cache if necessary, but uses a hash table to find the referenced block in the list. Therefore the running time is independent of cache size.

The second column is the time for stack analysis to compute the entire curve using a naive implementation which searches the simulation stack from the top to find the stack distance of the referenced block. This time is also expressed as a fraction of the time for simple simulation of a single size. We can see that stack analysis of file system traces is not efficient using this implementation. The third column shows the time required using the best stack implementation presented in Chapter 4. All times are in seconds for simulations of approximately 500,000 events running on a VAX 11/750. We see that the stack algorithm takes on the average 22% more time for memory address traces and twice as long for the file system traces, as compared to the single-size simulation. However, it reduces the execution time by as much as 90% for the program address traces, and at least 80% for the file system traces, when compared to the time required to approximate the miss/transfer ratio curves using ten non-stack simulations.

Table 3.2: Comparison of Execution Times (CPU seconds)

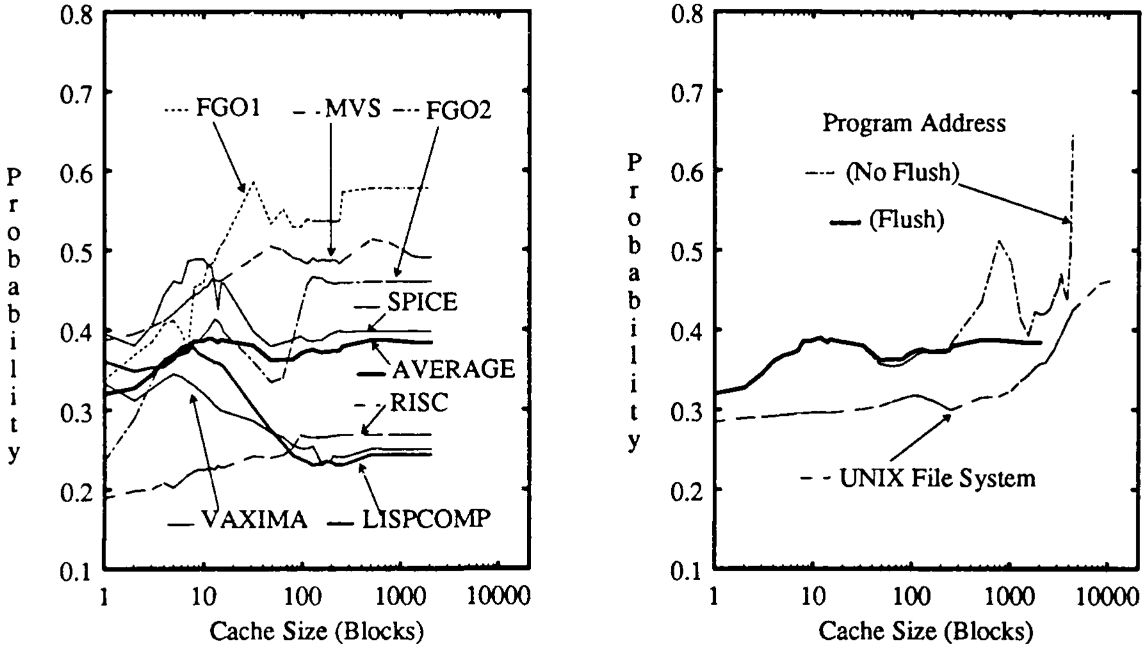
| Trace | Single Size Simulation | Simple Stack Simulation | Best Stack Simulation |
|---|---|---|---|
| | Program Address Traces | | |
| FGO1 | 351 | 487 (138%) | 429 (122%) |
| FGO2 | 309 | 357 (115%) | 357 (115%) |
| MVS | 369 | 878 (237%) | 460 (124%) |
| LISPCOMP | 328 | 512 (156%) | 428 (130%) |
| RISCR | 225 | 248 (110%) | 225 (100%) |
| SPICE | 277 | 342 (123%) | 342 (123%) |
| VAXIMA | 359 | 737 (205%) | 475 (132%) |
| AVERAGE | 316 | 508 (160%) | 388 (122%) |
| | UNIX File System Traces | | |
| ERNIE | 598 | 20,787 (3,476%) | 1,317 (220%) |
| ARPA | 612 | 19,416 (3,172%) | 1,246 (203%) |
| CAD | 622 | 31,375 (5,044%) | 1,111 (178%) |
| AVERAGE | 610 | 23,859 (3,907%) | 1,224 (200%) |

**Table 3.2:** Comparison of simulations run-time using non-stack and stack techniques. The non-stack simulation computes the miss ratio for a single cache size, while the stack simulations compute all sizes in one run. The simple stack simulation uses a linked-list implementation of the stack. The best stack simulation uses one of the alternative implementations discussed in Chapter 4.

### 3.6.3. Write-Back Probability

Before the discovery of our write back stack algorithm, an attempt was made to estimate write-back traffic in the following way. Each time a miss occurs (ignoring gaps) a block is pushed from all cache sizes. The write-back traffic should be approximately equal to the miss ratio times the probability that the block pushed from cache is dirty [Smit85a]. Smith estimated for data blocks that half of all pushes are dirty, but with wide variations between programs. He also reasoned that the probability that a push was dirty increases with cache size, since the pushed block will have been resident longer and hence have a higher probability of having been written.

Write-back stack analysis allows us to compute the probability of a dirty push directly and to validate the previous approximation. Although the probability of a dirty push is not needed directly in the computation of transfer ratios using equation (3.2), it is useful in its own right, for example as a parameter of a queuing model of a memory system. For example, see [Arch86].



(a) Program Address Traces (Flushing)

(b) Average by Type of Trace

**Figure 3.8**: Probability of a dirty push. This figure and Tables 3.3-3.6 show the probability that the block pushed from cache is dirty, as a function of cache size. The probability is generally increasing with cache size, but two LISP program traces, LISPCOMP and VAXIMA show an unexpected decreasing trend. The tables also show the percentage of references which were writes and the percentage of blocks ever written.

In Figure 3.8 we show the probability that a pushed block is dirty as a function of size. We see that on the average the projected increasing trend holds, although the probability for our traces was closer to 40%. However, the trend for individual traces is not consistent, and some show a distinct *downward* trend. Tables 3.3-3.5 show the same data, as well as the percent of references which are writes and the percent of blocks which are ever written. Notice that the two traces which show the strongest downward trend are the two LISP traces, LISPCOMP and VAXIMA. These are also the only two which have a higher percent write than percent dirty. We believe there is a relation between these observations, as explained below.

First, notice that the probability of a dirty push from a single-block cache is close to the

Table 3.3: Probability of Dirty Push - Program Address Traces With Flushing

| File Name | FGO1 | FGO2 | LISPCOMP | MVS | RISC | SPICE | VAXIMA | Avg |
|---|---|---|---|---|---|---|---|---|
| Percent Writes | 34.19 | 20.75 | 37.94 | 36.59 | 18.79 | 36.75 | 34.63 | 31.38 |
| Percent Dirty | 57.02 | 46.49 | 24.33 | 51.94 | 27.22 | 39.93 | 25.38 | 38.90 |
| Cache Size | | | | | | | | |
| 1 | 33.83 | 23.57 | 36.10 | 38.73 | 18.93 | 39.55 | 33.36 | 32.01 |
| 2 | 36.88 | 28.98 | 34.86 | 39.50 | 19.81 | 38.06 | 31.18 | 32.75 |
| 4 | 40.71 | 36.40 | 35.41 | 41.14 | 20.80 | 44.40 | 33.80 | 36.09 |
| 8 | 45.31 | 38.77 | 37.77 | 44.35 | 22.29 | 48.77 | 33.09 | 38.62 |
| 16 | 50.99 | 39.76 | 35.44 | 46.34 | 22.75 | 45.87 | 29.83 | 38.71 |
| 32 | 58.58 | 36.02 | 30.33 | 48.95 | 24.16 | 39.78 | 28.62 | 38.06 |
| 64 | 55.09 | 33.99 | 25.80 | 49.99 | 24.29 | 38.39 | 26.55 | 36.30 |
| 128 | 53.67 | 46.68 | 23.03 | 48.96 | 26.53 | 38.57 | 25.42 | 37.55 |
| 256 | 57.26 | 45.86 | 23.06 | 48.65 | 26.84 | 39.77 | 23.99 | 37.92 |
| 512 | 57.76 | 46.03 | 24.37 | 51.33 | 26.84 | 39.80 | 25.09 | 38.75 |
| 1024 | 57.76 | 46.03 | 24.37 | 50.07 | 26.84 | 39.80 | 25.06 | 38.56 |
| 2048 | 57.76 | 46.03 | 24.37 | 49.07 | 26.84 | 39.80 | 25.06 | 38.42 |

Table 3.4: Probability of Dirty Push - Program Address Traces Without Flushing

| File Name | FGO1 | FGO2 | LISPCOMP | MVS | RISC | SPICE | VAXIMA | Avg |
|---|---|---|---|---|---|---|---|---|
| Percent Writes | 33.55 | 20.63 | 37.94 | 36.62 | 18.74 | 36.80 | 34.86 | 31.31 |
| Percent Dirty | 50.13 | 62.92 | 37.42 | 70.37 | 55.43 | 57.64 | 29.20 | 51.87 |
| Cache Size | | | | | | | | |
| 1 | 33.44 | 23.50 | 36.15 | 38.77 | 18.87 | 39.59 | 33.64 | 31.99 |
| 2 | 36.46 | 28.88 | 34.91 | 39.52 | 19.65 | 38.12 | 31.46 | 32.71 |
| 4 | 40.57 | 36.25 | 35.51 | 41.16 | 20.66 | 44.45 | 34.08 | 36.10 |
| 8 | 45.29 | 38.61 | 37.81 | 44.39 | 22.24 | 48.77 | 33.44 | 38.65 |
| 16 | 50.77 | 39.53 | 35.43 | 46.38 | 22.35 | 45.83 | 30.13 | 38.63 |
| 32 | 58.48 | 35.36 | 30.22 | 48.98 | 23.65 | 39.26 | 28.94 | 37.84 |
| 64 | 53.79 | 32.54 | 25.29 | 50.13 | 22.51 | 36.72 | 26.90 | 35.41 |
| 128 | 51.06 | 54.33 | 21.83 | 48.52 | 23.49 | 36.09 | 24.63 | 37.14 |
| 256 | 51.93 | 56.27 | 20.82 | 47.85 | 32.90 | 36.40 | 22.66 | 38.40 |
| 512 | 49.40 | 65.61* | 22.80 | 53.07 | 55.60* | 35.10* | 22.25 | 43.40 |
| 1024 | 49.67 | 80.00* | 31.15* | 61.28 | 0.00 | 0.00 | 20.25 | 48.47 |
| 2048 | 44.50* | 0.00 | 0.00 | 62.24 | 0.00 | 0.00 | 18.88* | 41.87 |
| 4096 | 0.00 | 0.00 | 0.00 | 74.54 | 0.00 | 0.00 | 24.78* | 49.66 |

* Based on fewer than 500 pushes.

probability of a write, for all traces. This is certainly reasonable since most blocks are pushed from the single-block cache shortly after they are referenced. For other small cache sizes (in the range 2-15), the chance that a block has been written does increase as predicted, and most of the traces exhibit an upward trend. However, some blocks are never written, and the chance that a clean block will ever be written decreases as it is pushed down the stack.

At the other extreme, notice that without flushing (Table 3.4) the number of pushes eventually reaches zero for cache sizes which hold all the blocks of the program. Therefore, when

Table 3.5: Probability of Dirty Push - UNIX File System Traces

| File Name | ARPA | CAD | ERNIE | Avg |
|---|---|---|---|---|
| Percent Writes | 26.79 | 39.14 | 24.05 | 29.99 |
| Percent Dirty | 86.24 | 84.24 | 81.09 | 83.86 |
| Cache Size | | | | |
| 1 | 27.42 | 33.52 | 24.42 | 28.45 |
| 2 | 27.88 | 33.97 | 24.93 | 28.93 |
| 4 | 28.14 | 34.44 | 25.06 | 29.21 |
| 8 | 28.37 | 35.06 | 25.21 | 29.55 |
| 16 | 28.83 | 34.68 | 25.36 | 29.62 |
| 32 | 30.02 | 34.32 | 25.76 | 30.03 |
| 64 | 31.24 | 34.69 | 27.00 | 30.98 |
| 128 | 31.47 | 33.57 | 29.79 | 31.61 |
| 256 | 28.56 | 30.74 | 30.55 | 29.95 |
| 512 | 30.22 | 30.11 | 34.14 | 31.49 |
| 1024 | 32.47 | 29.92 | 34.62 | 32.34 |
| 2048 | 37.19 | 33.52 | 36.52 | 35.74 |
| 4096 | 48.52 | 36.19 | 40.42 | 41.71 |
| 8192 | 50.70 | 44.39 | 42.32 | 45.80 |
| 16384 | 0.00 | 28.36 | 0.00 | 28.36 |

flushing is used (Table 3.3), all of the pushes from large cache sizes are due to flushing. The probability that a block flushed from a large cache is dirty should be very close to the fraction of blocks which are dirty, as it is.

Since we can predict the probability of a dirty push from both small and large caches, we naturally expect that the trend should be from the percentage of writes to the percentage of dirty blocks. This explains the observed results, but suggests that they may be an artifact of the flushing methodology. We believe this is not the case and offer another explanation.

In Table 3.6 we classify all blocks into one of four classes: read-only, read/write (in no particular order), write-only, and write-once/read (e.g. a variable which is initialized and subsequently only read). The latter class we expect to be small in program address traces and larger in the file system traces. The table also classifies all events as to the class of block they reference. Both of the LISP traces show a surprisingly large fraction of read-only blocks. At the same time these blocks receive a relatively low fraction of references. Therefore a few "dirty" blocks are receiving most of the references, and therefore are more likely to stay near the top of the stack. Thus the blocks being pushed from larger caches are more likely to be from the large class of clean, read-only blocks. This generally explains the decline in the probability of a dirty push,

Table 3.6: Percentage of Blocks and Reference By Class

| | Blocks | | | | References | | | |
|---|---|---|---|---|---|---|---|---|
| File | Read Only | Read/ Write | Write Once/ Read | Write Only | Read Only | Read/ Write | Write Once/ Read | Write Only |
| | | | | Program Address Traces | | | | |
| FGO1 | 49.87 | 5.27 | 0.07 | 44.78 | 15.14 | 82.40 | 0.15 | 2.31 |
| FGO2 | 37.08 | 53.77 | 0.76 | 8.38 | 47.49 | 51.76 | 0.02 | 0.74 |
| MVS | 29.63 | 28.30 | 0.24 | 41.84 | 16.69 | 78.90 | 0.03 | 4.38 |
| LISPCOMP | 62.59 | 37.14 | 0.11 | 0.17 | 19.51 | 80.42 | 0.00 | 0.06 |
| RISCR | 44.57 | 53.55 | 0.00 | 1.88 | 34.02 | 64.20 | 0.00 | 1.78 |
| SPICE | 42.36 | 51.33 | 0.17 | 5.65 | 12.93 | 85.96 | 0.00 | 1.11 |
| VAXIMA | 70.80 | 28.18 | 0.16 | 0.85 | 24.05 | 75.89 | 0.01 | 0.05 |
| Average | 48.13 | 36.86 | 0.22 | 14.79 | 24.26 | 74.22 | 0.03 | 1.49 |
| | | | | Program Address Traces (With Flushing) | | | | |
| FGO1 | 42.98 | 21.31 | 0.11 | 35.59 | 16.85 | 80.91 | 0.03 | 2.21 |
| FGO2 | 53.51 | 30.50 | 0.71 | 15.27 | 48.89 | 48.70 | 0.09 | 2.32 |
| MVS | 48.06 | 34.00 | 0.52 | 17.42 | 19.76 | 75.05 | 0.12 | 5.06 |
| LISPCOMP | 75.67 | 23.36 | 0.27 | 0.70 | 21.46 | 78.32 | 0.04 | 0.18 |
| RISCR | 72.78 | 23.97 | 0.41 | 2.84 | 37.22 | 60.49 | 0.22 | 2.07 |
| SPICE | 60.07 | 25.90 | 1.13 | 12.91 | 14.63 | 83.10 | 0.10 | 2.17 |
| VAXIMA | 74.62 | 23.45 | 0.23 | 1.71 | 26.00 | 73.72 | 0.05 | 0.22 |
| Average | 61.10 | 26.07 | 0.48 | 12.35 | 26.40 | 71.47 | 0.09 | 2.03 |
| | | | | UNIX File System Traces | | | | |
| ERNIE3 | 18.91 | 7.89 | 32.66 | 40.54 | 48.77 | 21.42 | 20.56 | 9.25 |
| ARPA5 | 13.76 | 6.06 | 40.24 | 39.94 | 33.09 | 28.25 | 28.33 | 10.32 |
| CAD4 | 15.76 | 6.56 | 36.73 | 40.95 | 28.36 | 36.61 | 22.62 | 12.41 |
| Average | 16.14 | 6.84 | 36.54 | 40.48 | 36.74 | 28.76 | 23.84 | 10.66 |

**Table 3.6:** Percentage of blocks and references by read/write class. All blocks are classified into read-only, read/write, write-once/read (initialized, then read-only), and write-only classes. The table shows the percentage of blocks of each class, and the percentage of references which accessed each class of block.

independent of whether we use flushing. We can not say whether this phenomena is characteristic of LISP programs in general.

In passing we note that the UNIX file system traces also show an increasing trend in the probability that a pushed block is dirty, although not nearly as much as might be predicted based on the fact that 85% of all blocks are eventually written. The difference again is caused by deletions, leaving about half of the remaining blocks dirty. We also note that the class of write once/read is very common for file system blocks, as predicted.

Another interesting observation from Table 3.6 is the high percentage of write-only blocks in the file system and some program address traces. Because of the low percentage of references to these blocks it is likely that most were only written once. For the program traces these can be explained by the initialization of a large data area (or perhaps a program area in the MVS trace) that is not used again during the observed portion of the trace. There are several possible explanations for the file system traces. The first are files that are created or written during the trace and simply not read before the trace ended (just over two days). The second possibility are files which are written then deleted without being read. A file which is entirely overwritten is also considered to have been deleted. Common examples of this second type are editor recovery files, object files containing errors, and *rwho* status files which are written every three minutes (and rarely read during the evening hours). A third possibility are files which are truely write-only, such as accounting or log files. However our analysis shows these to account for less than 5% of the blocks.

## 3.7. Conclusions

In this chapter we have shown how stack analysis can be extended to two important new areas. The ability to collect transfer ratios for all memory sizes in a single pass reduces simulation time by as much as 90% compared to running 8-10 individual simulations, making this metric much more reasonable to collect. The transfer ratio is increasingly important in the study of shared-memory systems, particularly for file systems. Equally important, the ability to easily simulate sector caches, including writes and a form of prefetch, opens up a variety of new cache designs to efficient analysis.

# Chapter 4

# Improving the Efficiency of Stack Simulation

## 4.1. Summary

We saw in the previous chapter that stack analysis can perform much worse than conventional single-size simulation on traces that show relatively poor locality, such as file system and database traces. In this chapter we build on the work of several others who have suggested efficient techniques to reduce the running time of stack simulation from $O(N*L)$ to $O(N*log(S))$, where $N$ is the number of trace events, $L$ is the mean stack distance, and $S$ is the maximum interesting cache size. The key to these techniques is replacing the linked-list implementation of the simulation stack with a tree structure. First we review these techniques and present a number of practical optimizations. We also present a new *hybrid* structure which performs well against all types of traces. Finally we compare the performance of each of the techniques against a variety of real trace data, including program address traces, logical file system traces, and disk address traces. We find that the alternative data structures can reduce the stack analysis time by over 90% for file system traces, but that a simple linked list is adequate for most program address traces. Our hybrid technique performs better than all others for many of the traces, but the improvement is minor.

## 4.2. Introduction

A problem with stack analysis is that it is frequently slow when applied to file system, database, and certain program address traces. In fact, we saw in the previous chapter that stack analysis may be 1-2 orders of magnitude slower than single-size simulations for the same trace data. These data have poorer locality than seen in most program address traces. This lower locality shows up in a high mean "stack distance", that is, the depth in the stack where the referenced block is found. In addition, file system traces may have 15-20% of references that are not in the stack at all. This is a natural consequence of the fact that new files are continually being created and deleted.

The original stack analysis technique determines the stack distance by searching for the block from the top of the stack, giving execution times that are O($N*L$), where $N$ is the number of events and $L$ is the mean stack distance. Thus high mean distances can lead to large simulation times using the stack technique.

Bennett and Kruskal [Benn75] observed this problem in database traces, and presented a technique, for the special case of LRU replacement, which greatly improved performance. By constructing a tree on top of the reference string they were able to determine the stack distance in time that is O(*log (inter-reference time)*). However, the tree required a leaf for every event in the trace -- usually in the millions. Olken extended this technique by periodically compressing the tree, to create an algorithm that ran in bounded space, and time that is O($N*log$ $(S)$), where $S$ is the maximum cache size [Olke81]. He also presented a dynamically balanced (AVL) tree structure for maintaining the simulation memory stack with similar bounds. He evaluated the techniques against artificial traces and found that the AVL tree performed best.

Our interest is in the performance of these techniques against *real* data in order to discover which technique is appropriate to different types of data. We therefore extend the existing results in two ways. First, we notice that even the file system traces show frequent accesses near the top of the stack. However the tree methods are designed to work best for accesses deep in the stack and may in fact perform worse for the frequent accesses near the top. We propose a *hybrid* of the linked list and Bennett & Kruskal's (B&K) tree to exploit the locality that is present.

Second, we compare the techniques against real trace data of several types. We find that the tree-based structures reduce simulation time by as much as 90% in some cases. However, this improvement is not consistent across trace files. We suggest situations where the various techniques are most applicable. For example, we find that the simple linked list performs best for most instruction address traces we tried.

In Section 4.3 we review the various data structures that have been suggested for maintaining the simulation memory stack. In Section 4.4 we present our hybrid technique whose performance is relatively insensitive to the stack distance. Section 4.5 summarizes all the techniques. Finally, Section 4.6 presents results of the application of each technique to a wide variety of trace data, including program address traces, file system traces, and disk address traces.

### 4.3. Review of Stack Implementations

This section reviews and compares several alternative implementations of the simulation memory stack. The discussion of each implementation includes a fairly detailed table of the factors that determine the run time of the technique. We do this to emphasize the fact that constant factors and terms that appear insignificant "in the limit", may actually be the dominant factor in real simulations. Thus, although we may characterize some technique as "$O(N*log(S))$", for example, we know it is not necessarily better than a technique which is $O(N*S)$.

In analyzing the implementations, we use the abstract description of a simulator shown in Figure 4.1. The actions taken for each event are independent of whether the reference is a read or a write, and deletions are ignored for now. LRU replacement is assumed because of its simplicity, and because most of the implementations are only applicable to LRU replacement. The actions taken for each event are the following:

| | |
|---|---|
| *get_event* | Read a trace event. Returns the identity of the block referenced and the type of action. This routine is the same for all implementations, so it is not discussed further. |
| *find* | Locate the referenced block in the memory stack. Returns a pointer to the block. |
| *distance* | Determines the stack distance of the block. If the block is not in the stack, it returns MAXINT. |
| *stats* | Updates statistics, including miss counts. Also assumed to be the same for all implementations. |
| *remove* | Removes the block from the stack in preparation for a pull to the top. |

*insert*    Place the block in the stack at level 1.

```
while not EOF do begin
    get_event(block, action);
    ptr = find(block);
    k = distance(block);
    stats(block, action, k, ptr);
    if (k ≠ MAXINT) then remove(ptr);
    insert(ptr);
end;
```

**Figure 4.1**: Abstract Stack Analysis Algorithm

### 4.3.1. Linked List

The "obvious" implementation of the stack is a linked list with the top of the stack at the front. Finding a block requires a linear search of the list, however the stack distance is determined as a side effect of the find. The referenced block can be pulled to the front in constant time by simply changing pointers.

The average time required to process an event using this implementation is shown in Table 4.1. This analysis, as well as the others in this section, makes use of the definitions in Table 4.2. The mean distance $L$ includes only those references found in the stack. An average reference examines $L$ blocks in the list before finding the block, or a maximum of $S$ blocks if the block is not in the stack. The stack distance is determined in the process. If the block is in the stack, it is deleted. In all cases the block is inserted on top of the stack.

**Table 4.1**: Time complexity of Linked List implementation

| Routine | Time |
|---|---|
| find | $P_h L C_L + P_m S C_L$ |
| level | - |
| delete | $P_h C_{DL}$ |
| insert | $C_{IL}$ |

**Table 4.2:** Definitions of terms used in Time Analyses

$N$ = the number of trace events
$L$ = the mean stack distance
$S$ = the mean stack size
$I$ = the mean inter-reference time, where each event takes one time unit
$P_h$ = the probability that a block is in the stack
$P_m$ = the probability of a miss from the stack = $(1 - P_h)$
$C_L$ = the cost of traversing a node in the linked list
$C_{IL}$ = the cost of inserting a node into the linked list
$C_{DL}$ = the cost of deleting a node from the linked list
$C_H$ = the cost to look up a block in the hash table
$C_{IH}$ = the cost of inserting a block in the hash table
$C_{BK}$ = the cost to traverse one level in Bennett & Kruskal's (B&K) tree
$C_{IBK}$ = the cost to insert a node in the B&K tree
$C_{DBK}$ = the cost of deleting a node from the B&K tree
$C_{COMPACT}$ = the amortized cost to compact the B&K tree
$C_{AVL}$ = the cost to traverse one level in the AVL tree
$C_{IAVL}$ = the cost to insert a node in the AVL tree, which includes the possible cost of a rotation
$C_{DAVL}$ = the cost of deleting a node from the AVL tree, which includes the cost of possibly doing $\log(S)$ rotations

Taken as a whole, the algorithm appears to be O($N$), that is, linear in the number of trace events. However, $S$ and $P_h$ can easily be related to $N$. For example, in any stack simulation $S$ is initially a function of $N$, since the stack starts out empty and grows while previously-unreferenced blocks are added. In most program address traces the number of unique blocks in the trace is small compared to $N$ (see Table 4.10) and the stack quickly approaches this bound. At the same time, the probability that the entire stack is searched unsuccessfully, $P_m$ approaches zero. In the long-run, the time to process the trace is O($N$*$L$).

On the other hand, $P_m$ never vanishes in many file system traces since files are constantly being created and deleted. We have found as many as 20% of the events refer to a new block, and therefore will be a miss. Consequently, the $P_m S C_L$ term may dominate the time complexity. While $S$ is certainly bounded (by the size of the backing store), this size is rarely approached even for long traces. Also, we have observed that more files tend to be created than deleted, making $S$ a slowly-growing function of $N$. The long-run simulation time is thus O($N^2$).

One way to reduce this is to place a bound on the size of the stack. This is often done in any case as a practical matter, since it is more convenient to keep statistics in fixed-size tables.

Although this makes the running time linear again, the time to search the stack on a miss may still be considerable.

### 4.3.2. Hash Table

The futile search for missing blocks can easily be avoided by maintaining a *hash table* of all blocks currently in the stack. This technique was proposed and evaluated by Bennett and Kruskal for database traces [Benn75]. With it, we can find a block in nearly constant time regardless of whether it is present, assuming we keep the hash table less than 80% full [Knut68]. Of course, the hash table requires additional space. For our implementation we define the hash table to have one and a half times the number of elements in the maximum stack.

**Table 4.3:** Time complexity of Linked List with Hash Table

| Routine | Time |
|---|---|
| find | $C_H$ |
| level | $P_h L C_L$ |
| delete | $P_h C_{DL}$ |
| insert | $C_{IL} + P_m C_{IH}$ |

Table 4.3 presents an analysis of the time for this technique. Although finds can now be done in constant time, the stack distance is not determined in the process. The easiest way to determine the distance is to search from the top of the stack after using the hash table to verify that the block is present. An equivalent method, if the list is double-linked, is to locate the block using the hash table, then walk the list to the top of the stack. In either case, the net effect is to trade the hash table search and possible insertion for the $P_m S C_L$ term.

The time to perform the simulation is now $O(NL)$ for any type of trace. For traces with very low mean stack distance this may be quite adequate, and it is certainly easy to implement. On the other hand, it is easy to see how even this implementation of stack simulation could be slow for file system and database traces; in Table 4.10 we found stack distances of 400 or more for these traces.

One way to further reduce this time is to implement a set associative cache. If the addresses are well distributed across sets, the mean stack distance within each set will decrease in proportion to the number of sets. At the same time, the miss ratio tends to increase [Smit78a,Smit82]. However, for small numbers of sets, the miss ratios will be very close to the miss ratio of a fully associative cache.

Another method, proposed by Franta [Fran77], is to divide the linked list into $\lceil \sqrt{S} \rceil$ lists of approximately $\sqrt{S}$ nodes each. The first list forms an index into the others, sorted on time of last reference. After finding the last reference time of a block using the hash table, the block can be located in the lists by examining no more than $O(\sqrt{S})$ nodes. The stack distance is determined in the process. Franta shows that the stack can be updated by moving no more than $\sqrt{S}$ nodes. Thus the time complexity for this method is $O(N\sqrt{S})$. Since later methods are much more efficient we do not consider this method further.

### 4.3.3. Bennett and Kruskal's Algorithm

Bennett and Kruskal observed the fact that database traces had a high mean stack distance, and stack analysis of these traces took a long time [Benn75]. They also noticed that, for LRU replacement, the stack distance of a referenced block is identical to the number of other unique blocks referenced since the prior reference to the block. They used this as the basis for a different representation of the stack.

First, they allocate an array with one bit for each event in the trace, all initially set to zero. As the trace is processed, the bit corresponding to the current event, $b_t$, is turned on, and the bit corresponding to the prior reference to the current block, $b_p$, is turned off. At any point in time a bit is on if the corresponding reference is the most recent reference to some block. Therefore the stack distance for a reference is the number of 1-bits between $b_t$ and $b_p$. To efficiently determine this count, Bennett and Kruskal construct a fixed-structure tree over the reference bits, which we shall refer to as a B&K tree. Each node in the tree contains the number of 1-bits in leaves of the subtree rooted at that node. See Figure 4.2. They use a hash table with an entry for each block to locate the prior reference, $b_p$. The stack distance for a reference is found by ascending the tree from $b_p$. For each node that contains $b_p$ in its left subtree, add the count of bits in the right subtree. This determines the number of blocks above it in the stack; add one to give the distance. To update the tree, $b_p$ must be reset, $b_t$ set, and the counts adjusted up the tree. The tree ascension for both distance determination and count adjustment can be done simultaneously and need only go as far as the lowest common ancestor of the prior and current bits, since the number of 1-bits is unchanged above this level, and all right-hand subtrees above this level are zero ($b_t$ being the right-most 1-bit). As an example, see Figure 4.2. This example, as well as our implementation, uses a binary tree, although an m-ary tree would work. Block **D** is seen to give a stack distance of 6 (1+0+1+3+1), while **A** has distance 4 (0+0+3+1). Figure 4.3 shows the resulting tree after a reference to block **A**.

The time to determine the stack distance and update the tree is $O(E(\lceil \log(I) \rceil))$, where $I$ is the inter–reference time, assuming one clock tick per reference, and $E(.)$ is the expected value. We will write this as $O(log(I))$, with the expected value implied. Unfortunately the space required can be enormous, and is unbounded for arbitrary length traces.

#### 4.3.3.1. Bounded Space Bennett and Kruskal's Algorithm

If we are only interested in cache sizes less than some bound, say $S$, then Olken showed that Bennett and Kruskal's technique can be modified to run in bounded space [Olke81]. Since the number of 0-bits is irrelevant to the distance computation, the leaves of the B&K tree can be compressed at any time without affecting the algorithm, as shown in Figure 4.4, which is a compressed version of Figure 4.3. The compression requires either an LRU list of the blocks or a reverse pointer in the tree leaves in order to update both the tree and the hash table. The compression takes time which is $O(S)$, but if it can be done no more frequently than every $S$ events then compression only increases the time per event by a constant factor, independent of $N$ or $S$. Olken made this guarantee by constructing a tree with $2S$ leaves and keeping only the most recent $S$ unique blocks when compressing. In addition, as suggested by Bennett and Kruskal, if a reference is made to the same block several times in succession the tree is not updated, further increasing the number of events between compressions. The time to find the stack distance and update the tree is bounded by $O(\log(S))$, regardless of the inter-reference time. However, the

| Memory Stack | | |
|---|---|---|
| Stack Distance | Time of Block | Last Reference |
| 1 | F | 15 |
| 2 | E | 12 |
| 3 | B | 9 |
| 4 | A | 5 |
| 5 | C | 2 |
| 6 | D | 1 |

Fixed Binary Tree

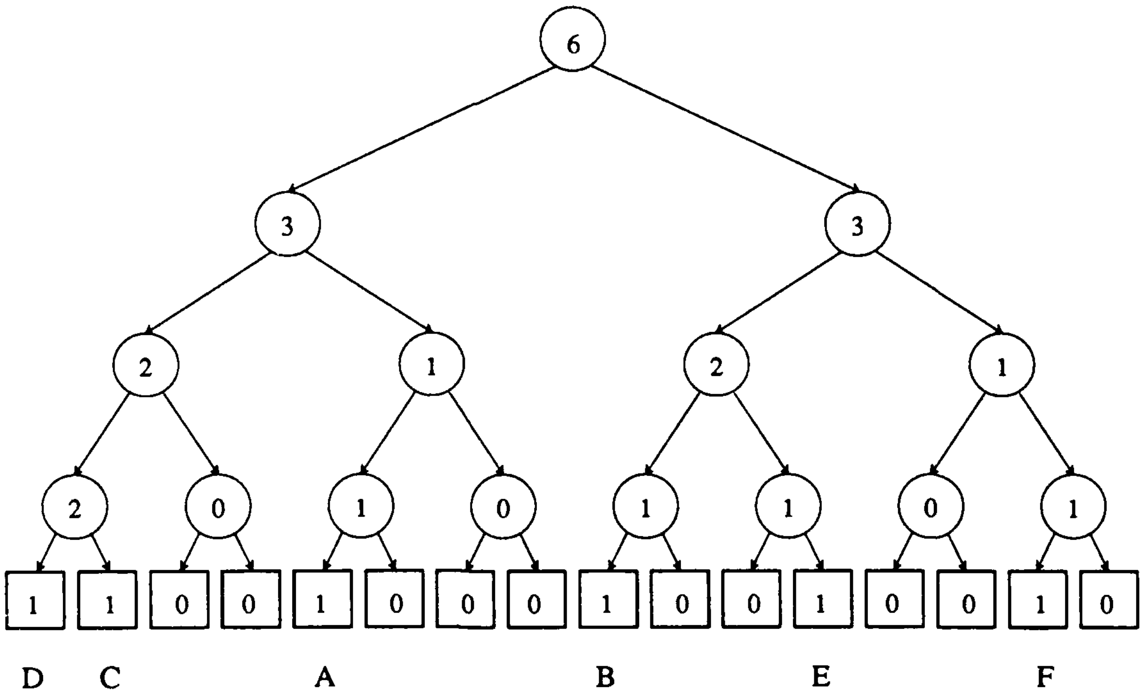

6
3
3
2
1
2
1
2
0
1
0
1
1
0
1
1
1
0
0
1
0
0
0
1
0
0
1
0
0
1
0
D
C
A
B
E
F

**Figure 4.2:** Bennett and Kruskal's (B&K) tree representation. The corresponding memory stack is shown above.

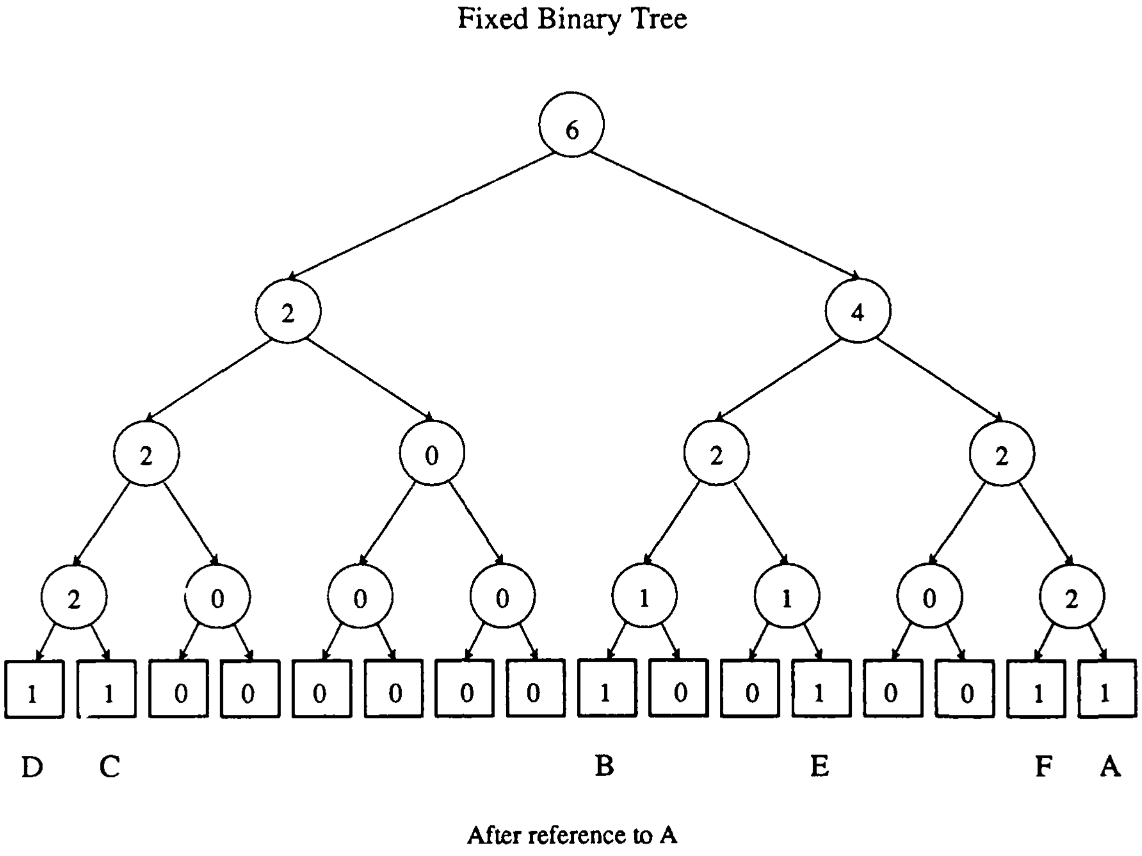

Fixed Binary Tree
6
2
4
2
0
2
2
2
0
0
0
1
1
0
2
1
1
0
0
0
0
0
0
1
0
0
1
0
0
1
1
D
C
B
E
F
A
After reference to A

**Figure 4.3**: Example of update to B&K tree. The figure shows the changes to Figure 4.2 after a reference to block A.

mean inter-reference time, $I$, is generally smaller than the bound on cache size, $S$, so we consider the update time to be O(log($I$)).

The time to process an event is analyzed in Table 4.4, with several terms proportional to $log(I)$. Thus the time to process a trace is generally O($Nlog(I)$), as in Bennett and Kruskal's algorithm. Space is O($S$). In our implementation, the hash table and tree require about $8S$ words, although the size of the internal nodes of the tree could be reduced if the algorithm were made more complicated. The algorithm is moderately easy to implement.

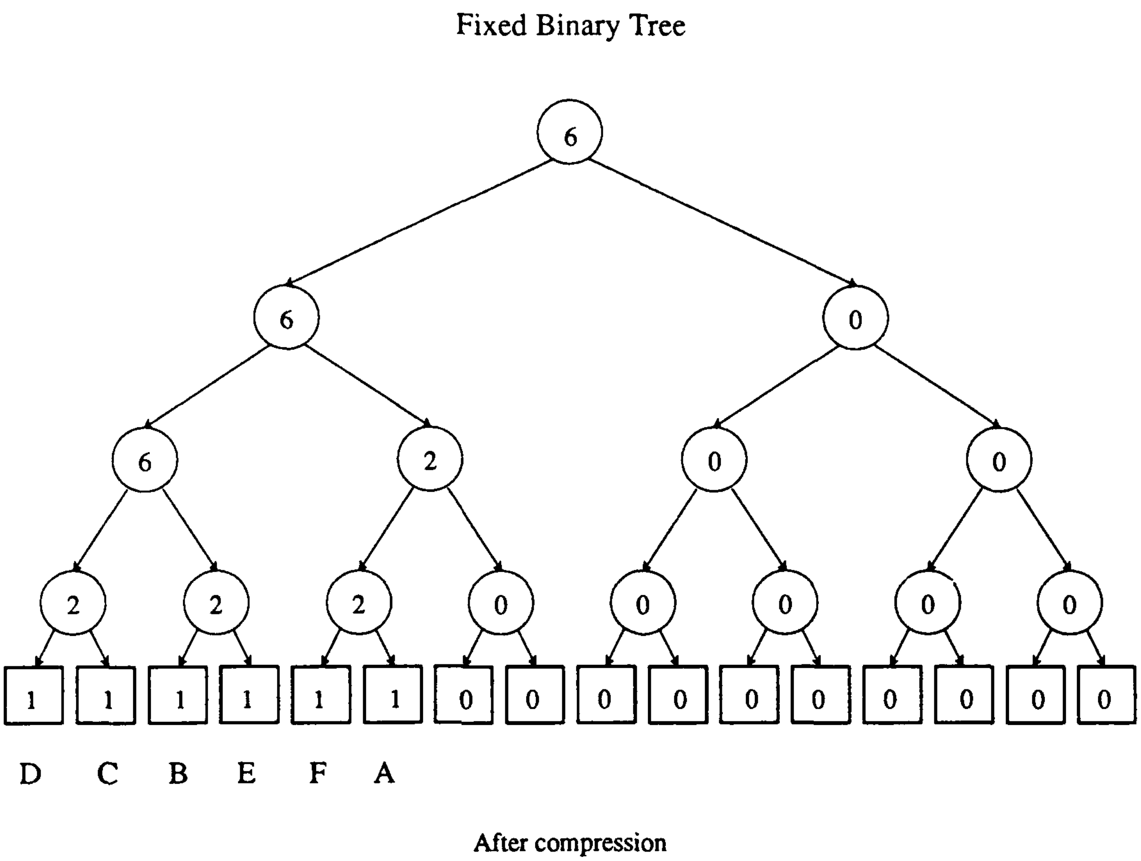

Fixed Binary Tree
6
6
0
6
2
0
0
2
2
2
0
0
0
0
0
1
1
1
1
1
1
0
0
0
0
0
0
0
0
0
0
D
C
B
E
F
A
After compression

**Figure 4.4**: B&K tree of Figure 4.3 after compaction. The right-hand subtree is now available to hold future references.

**Table 4.4**: Time complexity of bounded Bennett & Kruskal's tree

| Routine | Time |
|---|---|
| find | $C_H$ |
| level | $P_h \log(I) C_{BK}$ |
| delete | $P_h \log(I) C_{DBK}$ |
| insert | $P_h \log(I) C_{IBK}$<br>$+P_m (C_{IH} + \log(2S) C_{IBK})$<br>$+C_{COMPACT}$ |

### 4.3.4. AVL Tree

Bennett and Kruskal's technique achieves efficiency using a tree to reduce the number of nodes that must be traversed for a reference, particularly ones deep in the stack. The B&K tree is a sparse, fixed-structure tree with blocks at the leaves. Olken showed that we can make the same improvement in the order of the algorithm by representing the memory stack as a binary search tree of blocks [Olke81], with the time of last reference as the key. All blocks in the left subtree of any node have lower last-reference times, and all blocks in the right subtree have greater time of last reference. The binary tree requires at most *S* nodes, one for each block in the maximum-size stack, which is fewer than the 4**S* nodes required for a B&K tree with 2**S* leaves. Figure 4.5 shows an example of the binary tree representation, where block **D** is at the top of the stack. The memory stack order is a reverse inorder traversal of the tree, i.e. **DECAGHFB**. As with the B&K tree, this structure is only usable with LRU replacement. Any other policy requires a traversal of all nodes down to the referenced block, eliminating the advantage of the tree structure.

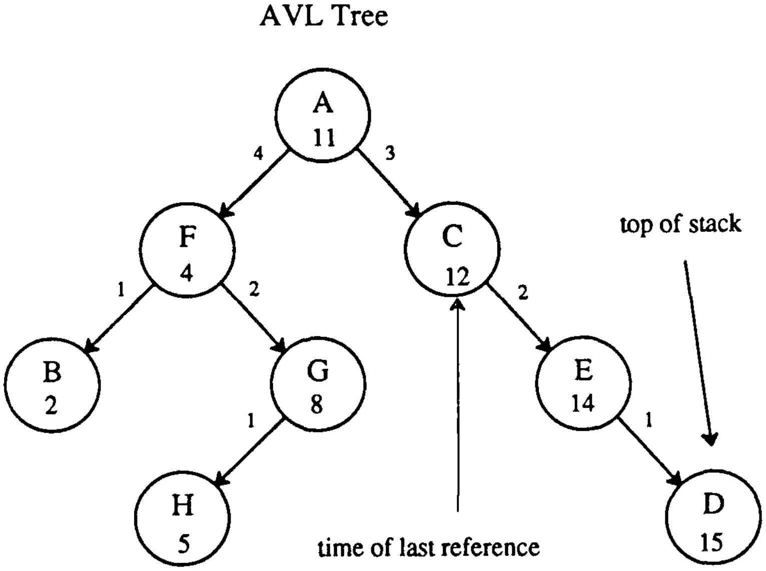


**Figure 4.5**: Binary tree representation of memory stack. The number in the circle is the time of the last reference to the block, which is used as the sort key for the tree. The number beside each branch is the size of the corresponding subtree. This is used in the computation of stack distance.

Notice that a block is lower in the memory stack than one of its tree ancestors if it is in the ancestor's left subtree. When this is the case, it is also lower than all blocks in the ancestor's right subtree. Therefore the stack distance of a block can again be determined by examining the blocks on the path from the block to the root. In this case we must store the total number of nodes in right subtrees in each node rather than just the number of leaves.

Using Figure 4.5 as an example, we can determine the stack distance of node **F** as follows: it is below nodes **G** and **H** since they are in its right sub-tree; it is above **B**; it is below **A** and the three nodes to its right. Since it is below a total of 6 nodes it is at level 7 in the stack. In the worst case, the number of nodes examined is the height of the tree.

Unfortunately, the way the keys of the tree are chosen, as the last reference time, all insertions in the tree are at the extreme right, so the tree will quickly degenerate to a linked list. Olken proposes to use a generalization of the AVL tree [Adls66] to ensure that the tree remains balanced and the height is close to minimum.

In an AVL tree each node contains a ''balance indicator'', which is the difference in height between the right and left subtrees. The heights are allowed to differ by at most one. If an insertion or deletion causes them to differ by more than one, the node is unbalanced. This is corrected by a ''rotation'' of nodes. See [Knut73] for a more complete description. It can be shown that only a single rotation is necessary to restore balance after an insertion, whereas $\log(S)$ rotations may be needed after a deletion [Knut73]. The height is generally very close to $\log(S)$, where $S$ is the number of nodes in the tree.

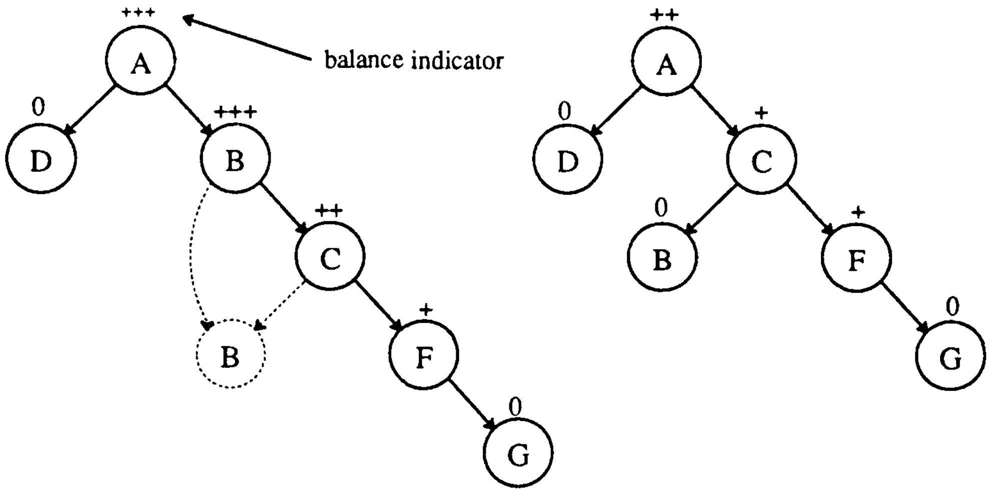
An AVL tree can be generalized to a k-balanced tree, where the balance indicator is allowed to range between $\pm k$ [Fost73]. Small values of $k$ (up to about 4) are effective in reducing rotations with modest effect on tree height. This is desirable because rotations are relatively expensive compared to the cost of other tree operations. As suggested by Olken, we use $k=2$ in our simulations, which reduces rotations by 17%. See Table 4.5.

We made one improvement to Olken's algorithm to further reduce the frequency of rotations caused by the fixed insertion point. Luccio and Pagli described the use of 2-rotations, and showed that they reduced the number of rotations due to random insertions and deletions [Lucc76]. A 2-rotation is a rotation around the grandchild of an unbalanced node instead of around a child, and is illustrated in Figure 4.6. We found that they have an even greater effect when the tree insertion point is always at the far right. First, we noticed that often the node above the unbalanced node is within one of being unbalanced itself, as seen in Figure 4.6 (a). Although the normal rotation around node C will restore balance, the A node immediately becomes unbalanced again at the next insertion. Instead of doing the normal single rotation in these cases, we do a 2-rotation around F. See Figure 4.6 (b). This corrects both unbalanced nodes, and leaves the tree "heavy" to the left. This has two advantages. First, more insertions may be made before another rotation is needed. In addition, it keeps nodes near the top of the stack on a shorter path to the root, since we expect these to be the most frequently referenced. Figure 4.7 shows the sequence of tree configurations while processing the reference sequence **{ABCDEFADBG}** starting with an empty stack.

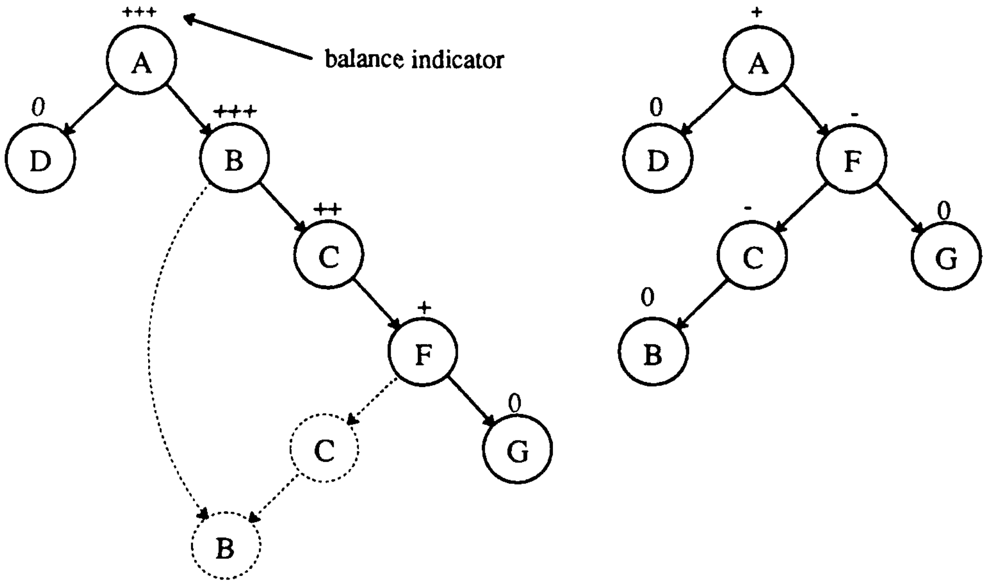
Table 4.5: Effect of AVL Balance Schemes

| Method | Balance (k) | Rotations per event | Mean Depth of Reference |
|---|---|---|---|
| AVL | 1 | 0.607 | 10.88 |
| 2-Balance | 2 | 0.507 | 11.61 |
| w/ 2-rotate | 2* | 0.323 | 11.22 |

Even with these changes, rotations are fairly frequent. Table 4.5 shows the frequency of rotations we observed for the AVL tree, 2-balance tree, and 2-balance tree with double rotations, for a typical trace. Also included is the mean stack depth for a tree with 4096 nodes. This is the tree depth at which a referenced block is found, which is more important than the actual height of the tree. The mean tree height can not be determined analytically for the common AVL tree, nor for this variant. Notice however that the mean reference depth is less than 12, which is log($S$), in all cases, showing that our efforts to reduce rotations have not significantly increased the reference depth. Note too that the 2-rotations actually decrease the mean reference depth compared to the 2-balance tree, as predicted.

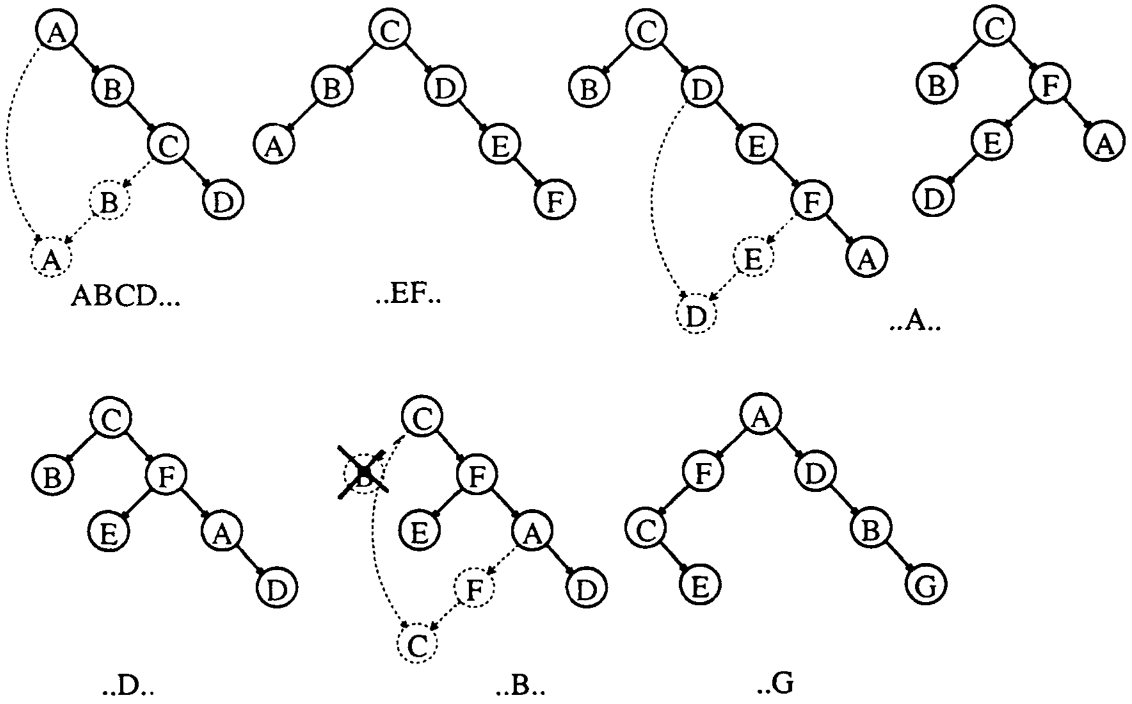

+++
balance indicator
A
0
D
+++
B
++
C
B
+
F
0
G
++
A
0
D
+
C
0
B
+
F
0
G

(a) Standard Single Rotation


+++
balance indicator
A
0
D
+++
B
++
C
+
F
C
0
G
B
+
A
0
D
-
F
-
C
0
G
0
B

(b) 2-Rotation

**Figure 4.6**: Single and 2-rotations in AVL tree. In (a), block B rotates around C, which becomes the right child of A. The tree remains right-heavy and will be unbalanced after the next insertion. With 2-rotations (b) both B and C rotate around F. The tree is left-heavy, permitting several insertions before rebalancing.



ABCD... ..EF.. ..A..

..D.. ..B.. ..G

**Figure 4.7**: AVL tree changes due to reference string {**ABCDEFADBG**}. The first reference to **D** and the second reference to **A** cause a 2-rotation on insertion. The last reference to **B** results in a rotation when the block is deleted (indicated by the **X** through block **B**).

The time per event using the AVL tree algorithm is shown in Table 4.6. We assume that the upper bound on reference depth is $\log(S)$, giving a time to process a trace of $O(N\log(S))$. This is generally worse than the modified Bennett and Kruskal's algorithm. However the storage is somewhat less, since the tree contains only as many nodes as there are unique blocks in the trace. The implementation is much more complicated than either the linked list or Bennett and Kruskal's algorithm.

**Table 4.6**: Time complexity of AVL tree implementation

| Routine | Time |
|---|---|
| find | $C_H$ |
| level | $P_h \log(S) C_{AVL}$ |
| delete | $P_h C_{DAVL}$ |
| insert | $C_{IAVL}+P_m C_{IH}$ |

We have not done any experiments with other types of balanced trees. In particular, the self adjusting "splay" data structure [Slea85] may be a promising alternative. It has an advantage that it does not require a "subnodes" or "balance" field for each block. The subnodes field in the AVL tree needs to be updated on the path to the root for each insertion/deletion, so there is a potential savings of two log($S$) operations per trace event. However, our results reported in Section 4.6 imply that the complexity of balanced trees may be unnecessary.

### 4.3.5. Deletions

A important consideration in file system studies is the existence of deletions in the reference string. If a file is deleted, the blocks of that file should be removed from the cache without write. With a write-back cache and short file lifetimes, it is likely that file blocks will be created and deleted without ever being written to the next level [Oust85].

If a deleted block were simply deleted from the stack, the stack level for all lower blocks would be reduced. This would have the unrealistic effect of calling these blocks back into a memory from which they had been pushed. Instead, what Mattson called a "marker" block is inserted in the stack replacing the deleted block. We refer to the marker blocks as "gaps" in the stack, corresponding to a vacant block in all larger caches. The next push from above the gap will fill it with the pushed block. Thus a gap will stop the sequence of pushes, just as finding the referenced block stops the pushes in the normal case. However, since the referenced block must still be pulled to the top, it may have to be replaced in the stack by another gap. Thus, a reference to a block below the first gap will seem to make the gap "jump" down the stack to the point of the reference. This has been previously discussed in Chapter 3.

The occurrence of deletions introduces two problems in the implementation. The first is the representation of the "gaps" in the memory data structure so that all other blocks maintain their proper stack distance. The simplest approach is that proposed by Mattson et al. [Matt70] -- to leave a gap block in the stack. This is equally applicable to the linked list or tree structures. A more complicated approach is to maintain a count of the gaps adjacent to each node and include this count in the computation of stack distance [Olke81]. For example, when searching the linked list, instead of adding one for each node traversed, add one plus the number of adjacent gaps. This technique can save many traversal operations when the expected number of gaps is large. It also saves storage when the number of gaps is large and the maximum stack size is not bounded. We have implemented the simple approach, since the stack size is bounded.

The second problem is to locate the first ("highest") gap, which determines the range of pushes. As seen previously in Figure 3.3, a gap above the referenced block will jump, while a gap below the block remains. When using a linked list, one possibility is simply to watch for a gap while searching for the referenced block. In the case of a hash-table miss it is still necessary to locate the highest gap by searching the stack. For the AVL or B&K tree this requires a tree traversal. This operation is O(log($S$)) for either tree, thus the B&K tree is slowed down to within a constant factor of the AVL tree. Regardless of the structure a search may be inefficient if the first gap is deep in the stack. Instead, we maintain a list of gaps, sorted by last reference time (and therefore in stack sequence) as suggested by Olken [Olke81]. Locating the first gap is a constant-time operation. When a block is deleted, or a gap jumps, we must search the list to locate the proper position for the new gap. We believe that this is acceptable because we expect deletions to be relatively few, and tend to be toward the top of the stack. If this is not the case, a heap can be used to locate the first gap, which would have performance which is O(log(*number of gaps*)).

Since deletions often cluster, for example when sequential pages of a file are deleted, it is often advantageous to check prior stack blocks. If they are gaps, then the deleted block can be inserted in the deletion list at that point. This works as long as blocks are deleted in the order they were referenced, for example block number sequence within a file.

A final optimization is used in the case of the AVL tree implementation. As discussed earlier, when the first gap is above the referenced block, the gap appears to jump to the location vacated by the referenced block. If we delete and insert both the block and the gap from the tree, there would be a possibility of rebalancing the tree *four* times. However, since none of the blocks below the first gap have changed position (except the referenced block), much of this manipulation is unnecessary. Instead, we delete the gap from its location in the tree, *replace* the block with the gap, and insert the block at the top of the stack. A similar technique is used when adjusting the counts in the B&K tree.

### 4.4. Hybrid Method

The tree methods achieve their efficiency by reducing the search time for references deep in the stack. However, as we shall see in the next section, all real traces are characterized by a combination of many references near the top of the stack and occasional reaches deep in the stack -- if this were not the case then caching would not be as effective as it is. In this section we propose a *hybrid* data structure that performs well for both types of references.

We have already seen a form of hybrid technique in the suggestion that the B&K tree not be updated on a hit to the top of the stack. In fact, since a re-reference to the top of the stack is a common special case, it may be worthwhile to check for this case before accessing the hash table, and certainly before ascending the tree to determine the stack distance. This applies equally well to the AVL tree technique.

Now consider the work involved in other references near the top of the stack, say the block at stack level two. This block is near the bottom of the AVL tree; determining the stack distance requires examining nearly $log(S)$ nodes, even with 2-rotations. The bounded space B&K tree does somewhat better; a reference to stack level two may look at only two nodes, since exactly one leaf separates the prior reference from the next bit (See Figure 4.3). On the other hand, the algorithm may have to look at up to $log(2*S)$ nodes if the two are in opposite subtrees of the root. The average over all possible positions is 3 nodes (half look at 2 nodes, one-fourth look at 3 nodes, one-eighth look at 4 nodes, etc). This is three times the single node examined in the simple linked list, after subtracting the test for the top of the stack in both cases. In addition, considering the addition, subtraction, and array indexing involved in the tree search and update, it is certainly more costly to use the tree compared to the linked list. We therefore consider a hybrid of the list and tree, using each where they are most beneficial.

Since the linked list is presumably quicker for short searches, we keep the upper $B$ elements of the stack in a linked list and the remainder of the stack in a tree. Because of its relative simplicity, we use the B&K tree. A single bit per block indicates whether the block is in the list or the tree. The referenced block is located using a hash table, and the stack distance determined by the appropriate method for the structure in which the block resides. To update the stack after a reference to a block in the list only requires that it be moved to the front of the list. If the block is in the tree, it is removed and added to the list, while the least recently used block on the list is added to the tree. The tree is updated to the common ancestor of the new bit and the referenced block just as in the previous B&K tree method. The tree is compressed as needed, but since the tree is not updated for every reference this is very infrequent.

As $B$ varies from zero to $S$ the algorithm varies from pure B&K to pure linked list. We would like to know if there is an optimal value for $B$, or at least a reasonable bound. A simple approach would seem to be to run both tree and list techniques for traces with varying stack distance, and choose the value for $B$ where the expected time to walk the list is the same as the expected time to walk and update the tree, i.e. the point where the stack distance *vs* time lines cross. This overlooks several factors. First, for every "hit" in the list, the frequency of tree compaction is reduced, effectively making the tree operations slightly less costly. Thus $B$ should be somewhat less than the crossover point. Second however, this assumes that the expected height in the tree is the same when the list is present. This is may or may not be the case. For example, suppose that the stack distance distribution was uniform over (1,100). A list of two entries would have little effect on the expected height in the tree since the references in the tree would still be uniform over (3,100). Now suppose that the stack distance is either 2 or 98, with equal probability. Again the mean stack distance is 50, but a two-node list will substantially vary the height in the tree; all references in the tree have a stack distance of 98 and require climbing at least $\log_2(98)$ levels in the tree.

This illustrates that the performance depend on the full stack distance distribution, not just the mean, $S$. In general, the stack distance distributions have declining hazard rate (i.e. the longer it has been since a reference, the longer it is likely to be), which implies that the height in the tree will increase with an increase in the length of the list.

In addition to the two factors mentioned, the optimal value of $B$ is certainly implementation dependent. Before attempting to analytically determine the optimal value, we ran simulations with varying stack distances and values of $B$ to determine the effect of the value of $B$ on simulation run-time. We used synthetic traces instead of real traces as input to the simulations in order to control both the mean and form of the stack distance distribution. We chose an exponential distribution as a model for the distance distribution because it is simple to generate and easy to control with a single parameter.

Figure 4.8 shows the time to perform simulations as a function of the value of $B$, for several mean stack distances. Note that the linked list does improve performance in all cases, as shown by the decreased time going from $B=1$ to $B=10$. We also see that the curves are relatively flat over a large range of values. Based on these simulations we chose a value of $B=80$ for use in the comparisons which follow. Because real traces tend to be hyperexponential (see Section 4.6.5), with higher weight toward the top of the stack, we would expect the optimal value for real traces to be less than this. However, these results do support our belief that the performance of the hybrid technique is relatively insensitive to the value of $B$ over a wide range, and that the effort involved in finding the true optimal value is disproportionate to the expected gain.

Implementation of the method is only slightly more complicated than the B&K tree technique. Space required is less than the B&K tree since it does not require tree nodes for the blocks in the linked list. This savings is usually small compared to the bound on the stack size. Time is proportional to the mean distance in the list and the interarrival time in the tree, both of which are a function of the number of blocks in the linked list. The time complexity is presented in Table 4.7, which makes use of the following new terms:

$B$ = the number of blocks in the linked list

$P_{HLIST}(B)$ = probability of a hit in the list of size $B$.

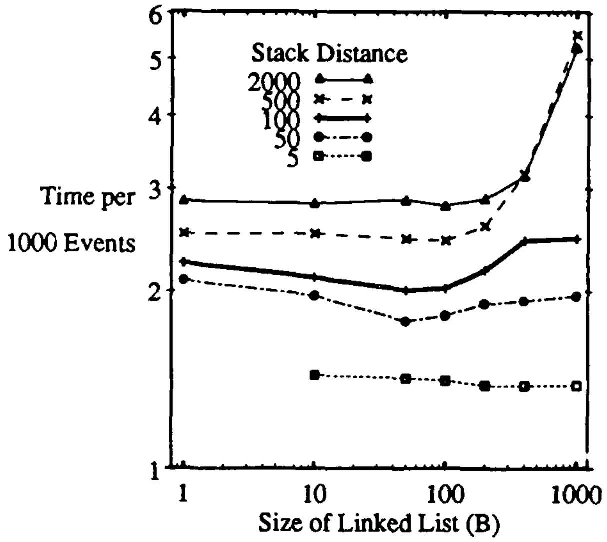


**Figure 4.8**: Simulation time for varying values of B. This figure shows the time to process 1000 events as a function of B for varying stack distance. The flatness of the curves indicates that the performance is relatively insensitive to B.

$P_{hBK}(B)$ = probability of a hit in the B&K tree, when the list is of size $B$.

$L(B)$ = the mean stack distance in the list of size $B$.

$I(B)$ = the interarrival time of events to the B&K tree.

Also the cost of compaction is a function of $B$. We believe that the dominant terms are $O(log(I(B)))$, but this certainly depends on the stack distance distribution.

**Table 4.7**: Time complexity of Hybrid implementation

| Routine | Time |
|---|---|
| find | $C_H$ |
| level | $[P_{hLIST}(B)C_{LIST}\,L(B)]+[P_{hBK}(B)\log(I(B))C_{BK}]$ |
| delete | $P_{hLIST}(B)C_{DL}+P_{hBK}(B)[C_{DL}+C_{DBK}\log(I(B))]$ |
| insert | $C_{IL}+P_{hBK}(B)[C_{IL}+C_{IBK}\log(I(B))]$<br>$+P_m[C_{IL}+C_{DL}+C_{IBK}\log(2S)+C_{IH}]$<br>$+C_{COMPACT}(B)$ |

## 4.5. Summary of Implementations

Table 4.8 summarizes the methods discussed in this chapter. The linked list is the simplest to implement, but theoretically the most time consuming. The tree algorithms have a lower expected complexity, but are also more difficult to code. This additional code results in a higher constant factor, which affects the running time in practice. In the next section we will compare the performance of these algorithms against actual traces.

**Table 4.8:** A comparison of simulation stack implementations

| Method | Time Complexity | Space Complexity | Coding Difficulty |
|---|---|---|---|
| Linked List | $O(NS)$ | $O(S)$ | simple |
| with Hash Table | $O(NL)$ | $O(S)$ | moderately simple |
| B&K Tree | $O(N\log(I))$ | $O(S)$ | moderately simple |
| AVL Tree | $O(N\log(S))$ | $O(S)$ | hard |
| Hybrid List/Tree | $O(N\log I(B))$ | $O(S)$ | medium |

## 4.6. Experimental Performance

Having reviewed the known techniques, and proposed a new hybrid technique, we now seek to compare the methods. Our interest is not simply finding the "best" algorithm, but rather to determine which method is best suited to a given situation. To do this we compared the methods against a wide variety of *real* trace data.

An alternative would have been to compare the methods using artificial traces with known distance distributions. This is the method used by Olken, who concluded that the AVL tree was best. It is possible with this approach to estimate the constant factors for very simplified equations for predicting execution time based on the dominant factors from Table 4.8. The equations depend on three different characteristics (stack size, stack distance, inter-arrival time) which are controllable. However it is not certain that these dominate in all cases, or even in most cases. For example, the fraction of hits to level one affects the number of times the B&K tree is compressed. Similarly, the fraction of hits to levels less than $B$ affects the Hybrid method. Also, the fraction of total misses (references which are not counted in the mean stack distance) affects performance; misses are very expensive in B&K trees but very cheap in the linked list with hash.

It would be difficult to design experiments to test all factors, and having done so we would still not know what method to use for a given problem.

We therefore conducted simulation experiments using the previously-described techniques and data structures against a variety of real trace data. This section presents the results of those simulations. After summarizing the simulator and the trace data we compare the performance of each of the data structures described earlier using each of the trace files. Finally, we characterize the types of trace data and show where the various techniques may be applicable.

### 4.6.1. The Simulator

All of the simulation experiments were done using a common simulator with "plug in" modules for each of the data structures. For comparison we also implemented a non-stack version that simulates a single cache size. This simulator maintains an LRU list of blocks to determine the block to replace if necessary. However, it uses a hash table to locate the referenced block in the (doubly-linked) list, so its performance is independent of stack distance or stack size. All of the programs were coded in C with little tuning specific to a technique. All simulations were done on a lightly-loaded VAX 11/750.

The simulator gathered a number of statistics that are not shown, increasing the overhead time of the simulation. All times were measured from a warm-start point when the stack contained a trace-dependent number of blocks. That number was calculated to be the 99.9% point on the cumulative stack distance distribution of references that would be found in an infinite cache. In other words, 99.9% of references that would ever be found were in the stack someplace. The time spent reading trace events varied by type of trace and was not counted in the timing.

### 4.6.2. The Trace Data

Three different types of traces were used in these simulations: program address traces, disk address traces, and file system traces. The program address and file system traces were described in Chapter 3. An additional two sets of program address experiments simulated a multiprogramming reference stream. The reference streams were created by multiplexing four trace files in a round-robin fashion, flushing the cache and switching every 20,000 references. The trace labeled MULT1 consists of the files FGO1, FGO2, SPICE, and MVS; MULT2 consists of the files RISC, LISPCOMP, VAXIMA, and MVS.

The disk address traces are physical disk address traces from two IBM systems: SLAC is an IBM 370/168 at the Stanford Linear Accelerator Center used primarily for text editing, timesharing and remote job entry; CROC is an IBM 370/168 at Crocker Bank running TSO and small batch jobs. The CROC trace is from a system using the OS/MVS operating system, while SLAC used SVS. These traces were used by Smith to investigate disk caching [Smit85b]. They consist of physical seek addresses (device/cylinder/track) for all disk I/O. The record within a track is also identified, but neither the size of the record nor of the data transferred is recorded. We have therefore assumed a full-track transfer in all cases (usually around 16K) and defined the block size to be one track. Since all results are presented in terms of blocks rather than bytes, the precise block size is immaterial. The trace does not distinguish reads from writes so no analysis of write-back is possible. Also file creation or deletion is not observable at the physical level. Table 4.10 presents the general characteristics of these traces. For convenience, the characteristics of the other traces are repeated here as well. The number of unique blocks in the disk traces is less than the number present in the file system traces because there are no deletions of physical disk addresses. The stack sizes are large, for the same reason.

### 4.6.3. Trace File Characteristics

Tables 4.9 and 4.10 summarize the traces used in this study. Several interesting comparisons can be made between the types of traces, which will be important in Section 4.6.5. Table 4.9 shows the number of events processed from each trace file and the frequency of events of each type. We processed approximately 500,000 events from each file after warm start, but ran out of events in some cases. The event types include instruction fetches for the program address traces, which were ignored. Table 4.10 presents the same data, but counting only read and write events. We initially speculated that writes would be a more significant percentage of the file system traces. In Table 4.9, considering all event types, this is in fact the case. However when instruction fetches are excluded from the traces to simulate a data-only cache the fraction of writes are comparable. We conclude that writes are a factor that should not be overlooked in any cache design.

Table 4.9: Types of Trace Events

| File | Number of Events | Types of Events Read | Write | Instruction |
|---|---|---|---|---|
| Program Address Traces | | | | |
| FGO1n | 500000 | 31.1% | 15.7% | 53.3% |
| FGO2n | 500000 | 28.9% | 7.5% | 63.5% |
| mvsn | 500000 | 31.0% | 17.9% | 51.1% |
| lispcompn | 472998 | 29.5% | 18.0% | 52.5% |
| riscrn | 458205 | 8.2% | 1.9% | 90.0% |
| spicen | 401235 | 30.0% | 17.5% | 52.5% |
| vaximan | 500000 | 31.0% | 16.6% | 52.4% |
| Program Address Traces with Flushing | | | | |
| FGO1 | 500000 | 30.9% | 16.0% | 53.1% |
| FGO2 | 500000 | 29.0% | 7.6% | 63.4% |
| mvs | 500000 | 31.0% | 17.9% | 51.2% |
| lispcomp | 500000 | 29.5% | 18.0% | 52.4% |
| riscr | 500000 | 8.2% | 1.9% | 89.9% |
| spice | 409600 | 30.0% | 17.4% | 52.6% |
| vaxima | 500000 | 31.1% | 16.5% | 52.4% |
| Multi-Program Address Traces with Flushing | | | | |
| mult1 | 500000 | 30.7% | 14.6% | 54.7% |
| mult2 | 500000 | 25.2% | 13.8% | 61.0% |
| **File** | **Number of Events** | **Types of Events Read** | **Write** | **Deletions** |
| UNIX File System Traces | | | | |
| arpa5 | 500000 | 58.9% | 22.8% | 16.7% |
| cad4 | 499206 | 46.6% | 35.3% | 16.2% |
| ernie3 | 492583 | 61.3% | 20.8% | 14.4% |
| IBM Disk Address Traces | | | | |
| croc | 500000 | 100.0% | 0.0% | 0.0% |
| slac | 500000 | 100.0% | 0.0% | 0.0% |

Table 4.9: Types of trace events. This table shows the total number of events taken from each trace file, and the percentage of each type. Only reads, writes and deletions were considered by the simulator; instruction fetches in the program address traces were ignored in order to simulate a data-only cache.

The number of unique blocks and average stack size, shown in Table 4.10, give an indication of the potential cost for linked-list stack processing. It is clear that the file traces come from a much larger population of blocks. Flushing exaggerates the number of blocks for the program address traces with flushing since a block that is reloaded after a flush is counted as a new block.

The number of unique blocks is certainly also a function of the block size. We have chosen

Table 4.10: General Trace File Characteristics

| File | Number of Events | Types of Events Read | Types of Events Write | Unique Blocks | Mean Stack Size | Dirty Blocks | Mean Dirty |
|---|---|---|---|---|---|---|---|
| Program Address Traces | | | | | | | |
| FGO1N | 233727 | 66.5% | 33.5% | 2675 | 970.03 | 1341 (50.1%) | 501.21 (51.7%) |
| FGO2N | 182290 | 79.4% | 20.6% | 1049 | 614.51 | 660 (62.9%) | 403.03 (65.6%) |
| MVSN | 244292 | 63.4% | 36.6% | 4972 | 2672.71 | 3499 (70.4%) | 1688.25 (63.2%) |
| LISPCOMPN | 224856 | 62.1% | 37.9% | 1764 | 1370.19 | 660 (37.4%) | 423.03 (30.9%) |
| RISCRN | 45975 | 81.3% | 18.7% | 902 | 675.92 | 500 (55.4%) | 310.02 (45.9%) |
| SPICEN | 190460 | 63.2% | 36.8% | 602 | 555.75 | 347 (57.6%) | 322.11 (58.0%) |
| VAXIMAN | 238237 | 65.1% | 34.9% | 4326 | 3035.69 | 1263 (29.2%) | 717.78 (23.6%) |
| Average | 194262 | 68.7% | 31.3% | 2327 | 1413.54 | 1181 (50.8%) | 623.63 (44.1%) |
| Program Address Traces with Flushing | | | | | | | |
| FGO1 | 234696 | 65.8% | 34.2% | 5293 | 154.91 | 3018 (57.0%) | 88.56 (57.2%) |
| FGO2 | 182839 | 79.3% | 20.7% | 4530 | 133.45 | 2106 (46.5%) | 60.90 (45.6%) |
| MVS | 244168 | 63.4% | 36.6% | 17174 | 424.75 | 8921 (51.9%) | 221.79 (52.2%) |
| LISPCOMP | 237867 | 62.1% | 37.9% | 11875 | 317.33 | 2889 (24.3%) | 78.67 (24.8%) |
| RISCR | 50336 | 81.2% | 18.8% | 4357 | 107.46 | 1186 (27.2%) | 29.60 (27.5%) |
| SPICE | 194174 | 63.3% | 36.7% | 5642 | 164.37 | 2253 (39.9%) | 66.67 (40.6%) |
| VAXIMA | 238211 | 65.4% | 34.6% | 14011 | 353.99 | 3556 (25.4%) | 91.79 (25.9%) |
| Average | 197470 | 68.6% | 31.4% | 8983 | 236.61 | 3418 (38.0%) | 91.14 (38.5%) |
| Simulated Multiprogramming Address Traces | | | | | | | |
| MULT1 | 226737 | 67.8% | 32.2% | 8138 | 222.40 | 3928 (48.3%) | 109.48 (49.2%) |
| MULT2 | 194957 | 64.7% | 35.3% | 12666 | 314.05 | 3976 (31.4%) | 98.67 (31.4%) |
| Average | 210847 | 66.3% | 33.8% | 10402 | 268.23 | 3952 (38.0%) | 104.08 (38.8%) |
| UNIX File System Traces | | | | | | | |
| ERNIE | 475471 | 74.7% | 25.3% | 85119 | 8879.06 | 69024 (81.1%) | 4021.80 (45.3%) |
| ARPA | 492040 | 72.0% | 28.0% | 93930 | 7254.39 | 81002 (86.2%) | 3471.87 (47.9%) |
| CAD | 489962 | 56.8% | 43.2% | 103488 | 11370.72 | 87180 (84.2%) | 4554.18 (40.1%) |
| Average | 485824 | 67.8% | 32.2% | 94179 | 9168.06 | 79068 (84.0%) | 4015.95 (43.8%) |
| IBM Disk Address Traces | | | | | | | |
| CROC | 500000 | 100.0% | 0.0% | 22366 | 11048.99 | 0 ( 0.0%) | 0.00 ( 0.0%) |
| SLAC | 500000 | 100.0% | 0.0% | 19656 | 12554.24 | 0 ( 0.0%) | 0.00 ( 0.0%) |
| Average | 500000 | 100.0% | 0.0% | 21011 | 11801.62 | 0 ( 0.0%) | 0.00 ( 0.0%) |

**Table 4.10:** General Trace File Characteristics This table shows the number of events considered from each trace file, and the percentage of these events which were reads or writes. The fifth column shows the number of unique blocks in the trace, while the next column gives the mean stack size. Flushing reduces the mean stack size, as do deletions from the file system traces. The final two columns show the number of blocks that are written at some time during the trace (dirty blocks), and the mean number of dirty blocks in the stack. These are also shown as percentages of the unique blocks and mean stack size, respectively.

block sizes appropriate to the type of trace data; had we used the same size for all traces the difference would have been much greater. The miss ratio is also a function of block size. Studies have shown that the miss ratio decreases with increasing block size over a moderate range for both processor and disk caching, although performance may not improve due to the increased time necessary to transfer larger blocks [Smit82,Oust85].

### 4.6.4. Distance Distributions

The mean stack distance is an important component of the timing estimates from Section 4.3. The mean of the distance distribution is theoretically undefined, since some references are misses with an infinite distance. In Table 4.11 we snow the mean distance of those references that were found in the stack, along with the coefficient of variation (i.e. the standard deviation divided by the mean). We also show the fraction of total misses, and the mean distance if misses are assigned the value of the largest allowable cache. The latter gives an upper bound on the work required in a linked list without hashing.

In Table 4.11 we can see that the mean stack distance is an order of magnitude lower for the program address traces, with or without flushing, when compared to the file system traces. Because of the high sequentiality of instruction references we would expect the difference to have been even greater if we had simulated a mixed instruction/data cache. The coefficient of variation is greater than 1 for all of the traces. Incidentally, this wide variation and a generally decreasing hazard rate (not shown) suggests that a mixture of exponentials (hyperexponential) might be a reasonable candidate for the stack distance distribution [Ferr78].

The fraction of misses, shown in the fourth column, affects the improvement possible with hashing. While slightly higher for the disk traces, it is much higher for the UNIX file system traces. This is to be expected, since the UNIX traces are of logical activity and new files are constantly being created. File creation and deletion was certainly also present on the IBM system where the disk address traces were created, but was not visible at the physical device level.

Periodically flushing the cache reduces the mean distance for the "hits", since blocks do

Table 4.11: Stack Distance Distribution Summary

| File | Stack Distance | | Misses | Mean |
|---|---|---|---|---|
| | Mean | Cf. Var. | | with misses |
| Program Address Traces | | | | |
| FGO1N | 16.93 | 2.35 | 2403 (1.02%) | 100.99 |
| FGO2N | 11.96 | 3.07 | 889 (0.48%) | 51.86 |
| MVSN | 70.54 | 3.51 | 4460 (1.82%) | 218.83 |
| LISPCOMPN | 39.42 | 3.07 | 1252 (0.56%) | 84.82 |
| RISCRN | 27.15 | 2.88 | 710 (1.54%) | 153.26 |
| SPICEN | 14.50 | 3.48 | 538 (0.28%) | 37.60 |
| VAXIMAN | 51.20 | 3.35 | 3814 (1.60%) | 181.54 |
| Average | 33.10 | 3.22 | 2009 (1.02%) | 118.41 |
| Program Address Traces with Flushing | | | | |
| FGO1 | 15.05 | 2.22 | 5293 (2.26%) | 60.92 |
| FGO2 | 8.78 | 2.26 | 4530 (2.48%) | 59.33 |
| MVS | 26.73 | 2.65 | 17174 (7.03%) | 168.97 |
| LISPCOMP | 18.89 | 2.60 | 11875 (4.99%) | 120.24 |
| RISCR | 9.71 | 1.95 | 4357 (8.65%) | 186.23 |
| SPICE | 7.57 | 2.19 | 5642 (2.90%) | 66.89 |
| VAXIMA | 23.35 | 2.36 | 14011 (5.88%) | 142.50 |
| Average | 15.73 | 2.40 | 8983 (4.55%) | 115.01 |
| Simulated Multiprogramming Address Traces | | | | |
| MULT1 | 15.30 | 2.64 | 8138 (3.61%) | 308.81 |
| MULT2 | 21.48 | 2.63 | 12666 (6.50%) | 552.36 |
| Average | 18.39 | 2.63 | 10402 (4.93%) | 430.58 |
| IBM Disk Address Traces | | | | |
| CROC | 295.41 | 3.50 | 20318 (4.06%) | 949.23 |
| SLAC | 498.70 | 3.08 | 17608 (3.52%) | 1058.15 |
| Average | 397.06 | 3.24 | 18963 (3.79%) | 1003.69 |
| UNIX File System Traces | | | | |
| ERNIE3 | 312.21 | 3.21 | 80672 (16.97%) | 2680.98 |
| ARPA5 | 241.13 | 3.30 | 91878 (18.67%) | 2881.43 |
| CAD4 | 414.14 | 3.13 | 101440 (20.70%) | 3221.77 |
| Average | 322.49 | 3.20 | 91330 (18.79%) | 2928.06 |

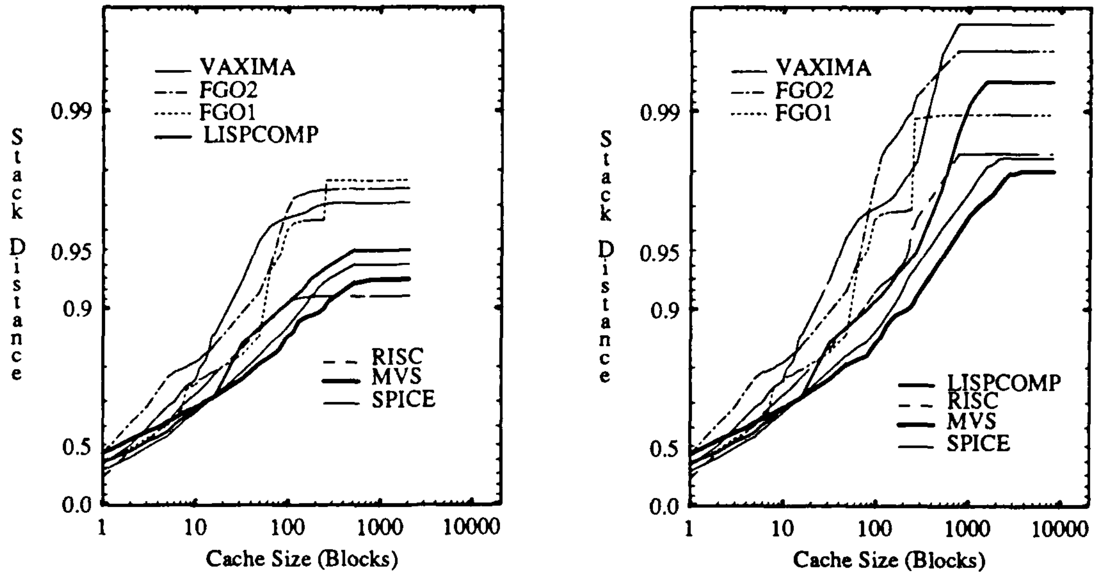
**Table 4.11:** Stack distance distributions. This table reports the mean and coefficient of variation of the stack distance distribution for each trace, ignoring all first references to blocks (which have an infinite distance). It also reports the number and percent of these first references (misses), and the mean distance if these misses were assigned a distance of the largest allowable stack size. The high mean distance and very high percent of misses explains the slow stack simulation times for disk and file system traces.

not have a chance to be pushed very far before being flushed. At the same time, the fraction of misses, of course, increases. The net result is that the mean with misses sometimes increases and sometimes decreases.

Although the performance of a linked-list depends only on the mean stack distance, the other techniques are sensitive to the full stack distance distribution. Figure 4.9 shows the cumulative stack distance distributions for all of the traces. Note that all of the distance distributions are highly skewed; only 15-30% of all references have stack distances above the mean of the "hits" which is plotted on each curve. We would expect instruction/data cache to be even more localized. It is obvious from these graphs (particularly Figure 4.9 (d)) that the different types of traces have very different distributions. For example, an average of 40% of program addresses refer to the top of the memory stack, whereas the disk and file system traces have only about 10% consecutive references. This is partially due to the fact that both of the latter types are buffered I/O streams; repeated references were handled from a buffer and were not reflected in the trace. Altogether 80% of the program addresses have a stack distance less than 10, leaving few opportunities for improvement using the tree techniques. Similarly, although they have few references to the top block, the disk traces show strong locality with nearly 50% of stack distances less than 10. Conversely, over 80% of the file system references have a stack distance greater than 10; nearly 50% are over 100. We would expect the tree techniques to favor the file system traces.

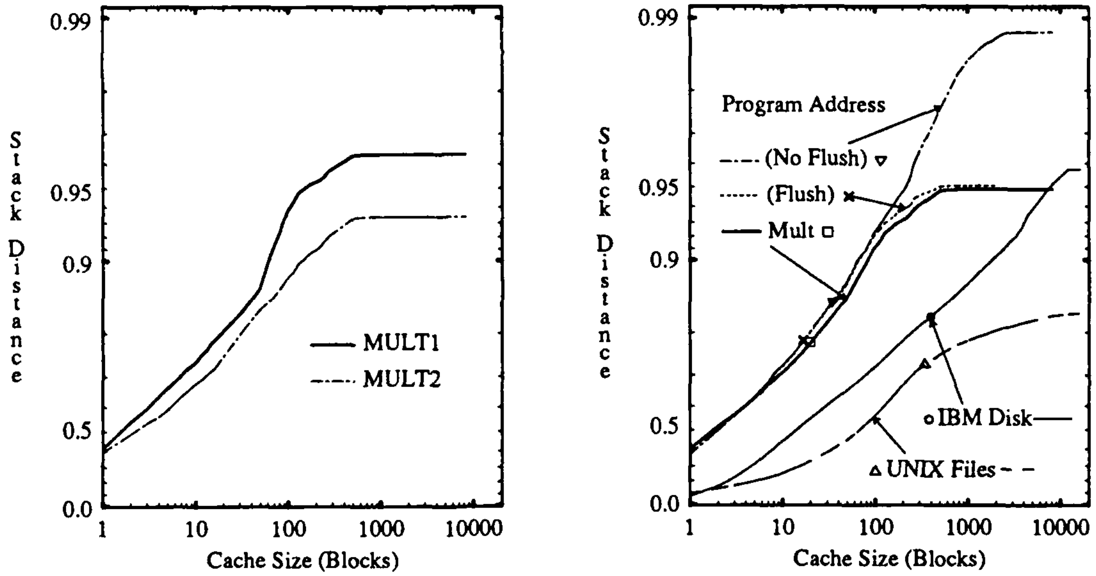
These graphs also show the range of interesting cache sizes for the trace types. If this number is small then there is little advantage to the tree techniques since the worst-case search of a list will still be small. The distribution of stack distances for the program address traces grows very little above 200; although the curve with flushing continues to climb until 3000, there are fewer than 5% of hits above 200. At the same time, the file system traces have 25% of their distances above 200, and the curve is still climbing above 5000! Clearly the file system traces demand larger simulation stacks, again favoring the tree structures.

Finally, figure 4.9 illustrates an advantage of stack analysis. Notice that all of the curves, created as a by-product of stack analysis, are continuous curves, not approximations using just a few points. This permits the observation of interesting anomalies such as the jump in the FGO1 curves at 256 blocks which would not otherwise be obvious. Stack analysis is not only efficient, it produces more precise curves than are reasonably possible using non-stack methods.



(a) Program Address Traces (Flushing)

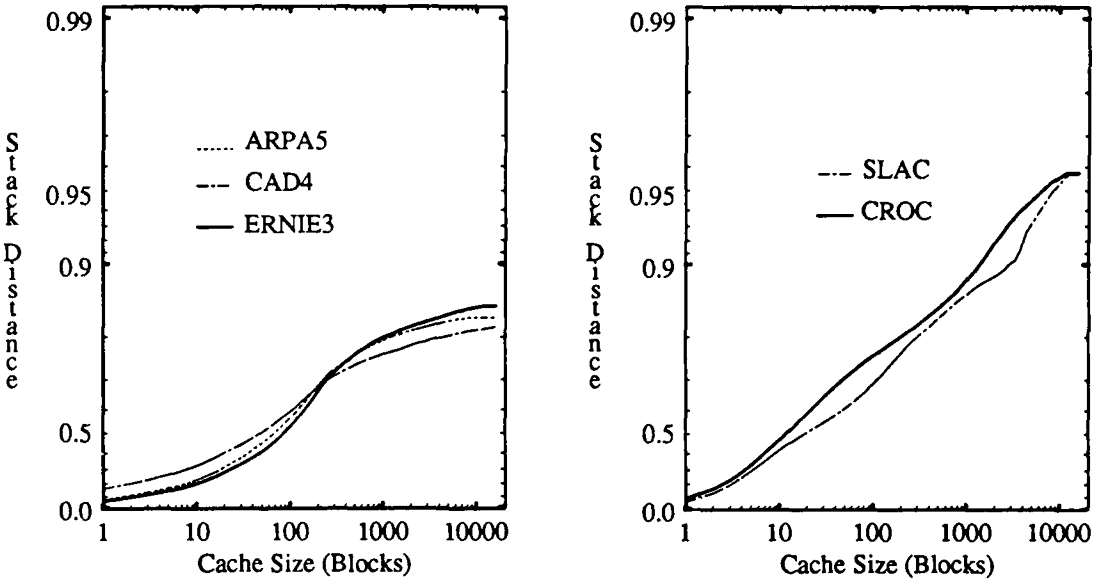
(b) Program Address Traces (No Flushing)



(c) Multi-Program Address Traces

(d) Average by Type of Trace

**Figure 4.9**: Stack distance distributions for various traces. For example, about 65% of references in the MVS trace of frame (a) have stack distances less than 10, while 7% (100-93%) missed in all sizes. Notice that the y-axis is an inverted log scale with most of the weight at the bottom. The symbols in frame (d) represent the means of the distributions.



(e) UNIX File System Traces (f) IBM Disk Address Traces

**Figure 4.9**: (Cont.)

### *4.6.5. Comparison of Methods*

We finally compare the proposed data structures under a wide range of conditions that are typical of actual use. In Section 4.5 we predicted that the tree methods would perform best for "high" mean stack distances. We would like to determine the range of distances for which this is the case.

The comparisons involved all of the traces and each of the data structures. Table 4.12 shows the mean time in seconds per 1000 read or write events averaged over all the events in the trace. (The RISC times seem out of line with the others because it had so few data references compared to instruction fetches that the overhead dominates.) The best time in each row is noted with an asterisk. Several entries are marked as "N/A" because times for these simulations are not available. Table 4.13 shows the mean number of "fundamental operations" per event for each trace and method. The fundamental operation varies with the method: examining a node in the linked list is the operation for both linked-list methods; ascending the tree one level toward the root for the AVL tree; ascending the tree toward the common ancestor for the fixed tree.

Table 4.12: Time in seconds per 1000 Read/Write Events

| Type | File | Single Size | Linked List No Hash | Linked List w/ Hash | Tree B&K | Tree AVL | Hybrid List/Tree |
|---|---|---|---|---|---|---|---|
| | FGO1n | 1.50 | 2.09 | 1.88 | 2.11 | 2.49 | 1.84* |
| | FGO2n | 1.70 | 1.96* | 1.97 | 2.18 | 2.56 | 1.98 |
| Program | lispcomn | 1.46 | 2.28 | 2.04 | 2.05 | 2.44 | 1.90* |
| Address | mvsn | 1.51 | 3.60 | 2.46 | 2.03 | 2.38 | 1.88* |
| Traces | riscrn | 4.91 | 5.40 | 5.21 | 5.08 | 5.62 | 4.94* |
| | spicen | 1.46 | 1.80* | 1.84 | 2.01 | 2.45 | 1.82 |
| | vaximan | 1.51 | 3.09 | 2.25 | 2.07 | 2.58 | 1.99* |
| | Average | 2.01 | 2.89 | 2.52 | 2.50 | 2.93 | 2.34* |
| | FGO1 | 1.43 | 1.88* | 1.92 | 2.11 | 2.45 | 2.03 |
| Program | FGO2 | 1.86 | 2.02* | 2.06 | 2.25 | 2.54 | 2.19 |
| Address | lispcomp | 1.53 | 2.10 | 1.91* | 2.10 | 2.40 | 2.01 |
| Traces | mvs | 1.56 | 2.48 | 1.99* | 2.12 | 2.46 | 2.03 |
| with | riscr | 5.03 | 5.02* | 5.16 | 5.59 | 5.69 | 5.48 |
| Flushing | spice | 1.53 | 1.80* | 1.81 | 2.10 | 2.43 | 1.93 |
| | vaxima | 1.51 | 2.31 | 2.00* | 2.20 | 2.59 | 2.03 |
| | Average | 2.06 | 2.52 | 2.41* | 2.64 | 2.94 | 2.53 |
| Multi- | mult1 | 1.59 | 2.18 | 2.04* | 2.29 | 2.58 | N/A |
| Program | mult2 | 1.75 | 2.69 | 2.35* | 2.48 | 2.77 | N/A |
| | Average | 1.67 | 2.44 | 2.19* | 2.38 | 2.67 | N/A |
| Disk | croc | 1.15 | 21.92 | 5.65 | N/A | 2.66* | N/A |
| Address | slac | 1.20 | 17.19 | 8.20 | N/A | 2.80* | N/A |
| | Average | 1.18 | 19.56 | 6.93 | N/A | 2.73* | N/A |
| UNIX | arpa5 | 1.24 | 39.46 | 5.28 | 2.63 | 3.27 | 2.53* |
| File | cad4 | 1.27 | 64.04 | 7.58 | 2.33 | 2.97 | 2.27* |
| System | ernie3 | 1.26 | 43.72 | 6.46 | 2.76* | 3.44 | 2.77 |
| | Average | 1.26 | 49.07 | 6.44 | 2.57 | 3.23 | 2.52* |

**Table 4.12:** Comparison of simulation time for all techniques. The table shows the time required to process 1000 read or write events for each data structure and trace file. The best time for each trace is flagged with an asterisk. The linked-list implementations give acceptable performance for program address traces. The addition of a hash table gives the greatest improvement in performance for the disk and file system traces, but the use of a tree structure provides another 2-3 fold improvement. The Hybrid technique performs the best for both program address traces (without flushing) and file system traces, but the improvement is minimal.

First we examine the effectiveness of hashing, comparing the two linked-list implementations. Hashing makes the largest difference in the file system traces, cutting the time by better than 80%. The disk traces show nearly as good an improvement. Hashing does improve performance for program address traces, but never by more than 35%, for MVS without flushing. Although flushing introduces many more misses, hashing does not do as well with flushing as without. The reason for this is that flushing also decreases the penalty for a miss. As seen in Table 4.13, the average search distance without the hash table is only 26 with flushing. Hashing

Table 4.13: Fundamental Operations per Event

| Type | File | Linked List | | Tree | | Hybrid |
|---|---|---|---|---|---|---|
| | | No Hash | w/ Hash | B&K | AVL | List/Tree |
| Program Address Traces | FGO1N | 31.91 | 16.93 | 3.91 | 6.06 | 10.23 |
| | FGO2N | 14.85 | 11.96 | 3.25 | 5.30 | 7.43 |
| | LISPCOMN | 44.92 | 38.80 | 4.29 | 6.37 | 8.91 |
| | MVSN | 116.70 | 67.87 | 4.18 | 6.26 | 7.45 |
| | RISCRN | 35.19 | 27.15 | 4.27 | 6.68 | 8.57 |
| | SPICEN | 15.40 | 14.50 | 3.64 | 5.69 | 6.74 |
| | VAXIMAN | 86.86 | 48.91 | 4.56 | 7.50 | 8.61 |
| | Average | 49.40 | 32.30 | 4.01 | 6.27 | 8.28 |
| Program Address Traces with Flushing | FGO1 | 17.84 | 15.05 | 3.77 | 4.09 | 10.29 |
| | FGO2 | 11.41 | 8.78 | 3.05 | 3.74 | 7.10 |
| | LISPCOMP | 30.13 | 18.89 | 3.85 | 4.62 | 8.90 |
| | MVS | 51.80 | 26.73 | 3.64 | 4.46 | 7.91 |
| | RISCR | 17.32 | 9.71 | 3.62 | 4.21 | 7.93 |
| | SPICE | 11.71 | 7.57 | 3.38 | 4.17 | 6.54 |
| | VAXIMA | 39.83 | 23.35 | 4.15 | 5.11 | 9.21 |
| | Average | 25.72 | 15.73 | 3.64 | 4.34 | 8.27 |
| Multi-Program | MULT1 | 22.61 | 15.30 | 4.17 | 4.12 | N/A |
| | MULT2 | 39.77 | 21.48 | 4.41 | 4.65 | N/A |
| | Average | 31.19 | 18.39 | 4.29 | 4.39 | N/A |
| Disk Address | CROC | 725.95 | 289.61 | N/A | 13.28 | N/A |
| | SLAC | 912.51 | 493.68 | N/A | 13.61 | N/A |
| | Average | 819.23 | 391.64 | N/A | 13.44 | N/A |
| UNIX File System | ARPA5 | 1477.78 | 241.12 | 7.77 | 11.44 | 13.23 |
| | CAD4 | 2417.57 | 414.13 | 7.01 | 10.72 | 10.52 |
| | ERNIE3 | 1602.50 | 312.20 | 8.12 | 11.83 | 13.72 |
| | Average | 1832.62 | 322.48 | 7.63 | 11.33 | 12.49 |

**Table 4.13:** Fundamental operations per event. The table shows the number of "fundamental operations" per event for each technique and trace file. The fundamental operations are: walking one list element (linked list with or without hash), walking one level in the tree (B&K or AVL tree), or both (hybrid).

only reduces it to 16, at a cost of maintaining and searching the hash table. Looking at the difference between the first two columns of Tables 4.12 and 4.13, we find that hashing only seems to be effective if it saves examining at least 8-10 nodes, on the average.

Comparing the tree methods, the bounded-space Bennett and Kruskal tree is consistently faster that the AVL tree in our implementation. This is opposite from the result obtained by Olken [Olke81], who used recursive implementations of both methods. Our result is consistent with the operations required by the two tree methods, seen in Table 4.13. The number of operations is consistently higher with the AVL tree, as was predicted. The number reported for the AVL tree is the mean depth of a reference. This number is well below $\log(S)$, where $S$ is the maximum stack size, found in Table 4.10, indicating that our rotations tend to leave the tree leaning to the left with the top of the stack closer to the root, as intended.

Comparing the linked list methods to the B&K tree, we see in Table 4.12 that the B&K tree is invariably better for the file system traces, reducing time by an average of 60%. The tree method is rarely better than the best linked list method for program address traces, happening only for MVS and VAXIMA with no flushing which have mean distances of 48-67, and RISC. This is true in spite of the fact that the number of fundamental operations is considerably less for the fixed tree for all of these traces. Obviously, and not unexpectedly, the tree operations are more time consuming.

Finally, Tables 4.12 and 4.13 also include the results of the use of the hybrid method, with the number of blocks in the linked list, $B$, equal to 80. From Figure 4.9 we can see that at this size 90% of the program addresses and 50% of the file system references will be found in the linked list. Because of this, the number of operations went up compared to the B&K tree. However even for the program address traces the tree reduced the number of operations compared to the linked list. The method performs better than either the linked list or the B&K tree for a large number of the traces, both program address and file system. However, the improvement was marginal in all cases. In addition, for the program address traces without flushing, the simple linked list is still the best technique.

### 4.7. Conclusions

In this chapter we have seen that the tree-based methods are quite effective in reducing the simulation time by as much as 90% for file system and other traces. Of the two tree methods, the bounded-space B&K tree is the clear choice over the AVL tree since it is both faster and simpler. The hybrid technique is the fastest all-around technique, performing well for all types of data. However, it is not clear that the added complexity is required since the improvement is minor. We conclude that the basic linked list is the best method for simple program address traces, and the B&K tree is the best for file system and disk cache studies.

# Chapter 5

# One-Pass Analysis Techniques for a Class of Multi-processor Cache Consistency Protocols

## 5.1. Summary

We have seen that *stack analysis* permits the computation of performance metrics for all cache sizes in one pass of a memory reference string. In this chapter we extend this technique to apply to multi-processor caches using a class of cache consistency protocols supported by the IEEE Futurebus. We show that, given certain reasonable restrictions, the miss and transfer ratios of all cache sizes are efficiently obtainable in one pass. The key to this result is that the state of each block in all cache sizes can be maintained using a small number of variables. The state of a block is a combination of three attributes: validity, ownership or "dirtyness", and exclusivity. We show that there is a single threshold cache size for each attribute at which the value of the attribute changes. For example, below a certain size the block is clean; above that size it is dirty. Using the state information we can determine the cache sizes in which there is a memory access or other bus activity, and thereby compute the miss and transfer ratios for all sizes. For certain of the protocols this result applies to $K$ caches whose sizes vary independently. For the rest of the protocols, the results hold for all sizes of $K$ caches provided that each cache is the same size. Finally, the chapter presents simulation algorithms for several consistency protocols from the class. These algorithms demonstrate the techniques for maintaining the state information and computing the misses and bus transfers for all cache sizes. The application of these techniques to multi-processor file system caches is discussed in the next chapter.

## 5.2. Introduction

There is a trend in computer architecture toward the use of shared-memory multi-processors rather than a single high-speed processor to achieve a given performance level [Smit84]. Current memory speeds and bus throughput limitations generally requires that each of these processors maintain a local cache of its working set of memory blocks. In order to maintain the impression of a single shared memory, some cache consistency protocol is needed. An analysis of the design of these systems needs to consider all these factors, including memory speeds, cache sizes, and different consistency protocols, as they apply to the expected workload.

One method for performing such an analysis is trace-driven simulation, where a trace of references from a "real" system is used as input events to a simulation of the system under study. In previous chapters we have discussed the stack analysis technique for analyzing all sizes of a single cache in one pass. The ability to derive performance metrics for all sizes at once may be even more important in the study of multi-processor caches. If the cache sizes can vary independently then the number of simulations increases as the power of the number of processors. The ideal technique would be one that computes the miss ratio and bus traffic for all sizes of each cache in one pass. Nearly as valuable would be a technique to compute the miss ratio for all sizes simultaneously, assuming all caches have the same size. A third possibility would be to consider a K-processor system where K-1 of the sizes are fixed and one cache varies over all sizes.

This chapter will show that some consistency protocols fall into the first category, but most do not. However, it will also show that all of the protocols of a particular class, referred to as the MOESI class [Swea86], can be efficiently simulated for certain combinations of cache sizes. These relationships include both of the latter categories above.

The one-pass techniques for analyzing multi-processor caches are applicable to processor caches as well as file system caches in a distributed system. This chapter focuses on processor caches because the consistency protocols are fairly well defined and the state space is smaller than that required for file system caches.

In Section 5.3 we briefly review stack analysis as it applies to single-processor caches, then generalize the definition to apply to a wider class of problems. Section 5.4 defines some metrics of interest in the analysis of multi-processor systems. Section 5.5 presents a class of cache consistency protocols developed by Sweazey and Smith, called MOESI protocols, which we use as the basis of the development of stack algorithms. In Section 5.6 we begin the development of a stack algorithm for the simulation of the two-processor case. We introduce some initial restrictions that permit stack analysis to be applied. We then show that a subset of the MOESI protocols can be simulated with no restrictions on sizes of each cache. In Section 5.7 we show that when there is a fixed relation between the two cache sizes, stack analysis applies to the entire MOESI class of protocols for an arbitrary number of processors. Finally, in Section 5.8 we present an abstract algorithm for maintaining the state of a block in each cache. We also discuss the applicability of the techniques to other classes of consistency protocols.

## 5.3. One-Pass Algorithms

### 5.3.1. Review of Stack Algorithms

A replacement algorithm has been called a *stack algorithm* if the cache could be represented as a stack, where the top $k$ blocks are the contents of the cache of size $k$. An equivalent characterization is that these algorithms possess an *inclusion property* since the contents of any cache *includes* the contents of any smaller cache. For any block, its position in the stack, or *stack distance*, determines the smallest cache where a reference to the block will not "miss", and is therefore the key information needed to compute the miss ratio.

Stack analysis has been extended to cases where the stack distance alone is not sufficient to determine if a reference is a miss. One example of this is the sub-block cache technique presented in Chapter 3. We showed that sub-block validity satisfies a form of inclusion (if the sub-block is present in a cache of size $k$ then it is present in all larger sizes), hence there is a minimum cache size where the sub-block is valid. By maintaining a *valid level* for each sub-block, the miss ratio can easily be determined.

The concept of stack analysis was further extended in Chapter 3 to consider writes, and particularly the effect of writes on memory accesses. Whereas a read miss causes a single memory access at the time of the reference, a write may cause two accesses — one to read the block prior to modification (write fetch) and a second one to rewrite the block (copy-back). The write fetch occurs when the block "misses" in the cache, just like a read, but the copy-back occurs independently of whether the block was in the stack and may be delayed until some later time. There are two key observations that permit the number of write accesses to be computed for all cache sizes. First, if the block being written is present in the cache and already dirty then it was modified previously, and hasn't been copied back yet. Since the current modification must eventually be written back, both the current and previous modification can be copied back with one access. Thus one copy-back access is avoided whenever the block is dirty. The second key observation is that the state of the block, whether clean or dirty, also obeys a form of inclusion (i.e. if the block is dirty in size $k$ then it is dirty for all larger sizes). Therefore by maintaining a *dirty level* for each block we can determine exactly those cache sizes where a copy-back is avoided, and hence compute the total number of write accesses for each size.

This illustrates several ideas important to the generalization that follows. First, the metrics for a particular cache size can be computed by knowing just the state of the referenced block in that size. In the basic stack algorithm the relevant state was simply valid or invalid. For the write-back analysis there are three relevant states: invalid, valid but unmodified, and valid dirty. This illustrates the second point, that the state is a combination of several components or characteristics, for example validity and "dirtyness". A third point is that inclusion may (or may not) apply independently to each *attribute*, where an attribute is a specific value of some characteristic. Consider the set of blocks that have a certain attribute for a cache of size $k$; the attribute obeys inclusion if this set *includes* the set of blocks with the same attribute for any smaller cache size. In the case of write back analysis, inclusion applies to both the valid attribute and the dirty attribute. A consequence of inclusion is that there is a threshold size for the attribute, as exemplified by the dirty level and valid level presented earlier.

### 5.3.2. Efficient One-Pass Algorithms

We can generalize this notion of a stack algorithm to apply to a wider variety of memory analyses. First, we define a *cache management algorithm* as the complete set of rules used to manage a cache or set of caches. These include, but are not restricted to, the placement and replacement policies, the write policy, fetch and prefetch policies, and cache consistency protocols. See [Smit82] for a full discussion of cache management algorithms.

In our generalization we would like to capture the advantage of stack algorithms that they can be analyzed in one pass. Of course, anything can be done in one pass given enough time and space. Therefore we say that a cache management algorithm is an *efficient one-pass algorithm* for a cache metric if the metric can be computed for all cache sizes up to $S$ in time which is $O(N*S)$ and space which is $O(S)$, where $N$ is the length of the trace. The intent of this definition is to identify those algorithms that can be efficiently simulated both in terms of time and space. The early stack algorithms fit this definition, as do write-back and sector cache techniques and the delayed-staging techniques [Silb83].

The motivation for a time bound is fairly obvious, and also fairly loose. In Chapter 4 we reviewed techniques for certain cache management algorithms to reduce this to $O(N*logS)$, in particular when the LRU replacement algorithm is used. The bound on space is less obvious, but is motivated by the desire to simulate large caches. If the simulation required space which is $O(S^2)$, say, then simulation of caches of even a few thousand blocks would exceed the size of most machines available for simulation. In addition, a large space requirement often leads to long simulation run time. We therefore concentrate on the space bound in the discussion to follow.

The bound on space places some limits on the structure of the simulation state space. For example, suppose that the state of a given block could change arbitrarily from one cache size to another. In the extreme the simulator would have to maintain the state of the block in each of the $S$ sizes. Since there can be $S$ distinct blocks in the largest cache, the total space required is $O(S^2)$. Therefore to meet the space constraint, the state can not change arbitrarily with size. The only allowable possibility, then, is for the state to remain the same throughout a range of cache sizes, and for the number of ranges to be bounded. (By *bounded* we mean bounded by some small constant, independent of $S$ or $N$. If the number of ranges is bounded only by $S$ then we say that the number is *unbounded*). Each range of sizes is referred to as a *region*. The number of regions is certainly bounded for the basic stack algorithms, with two regions: invalid and valid. Clearly if each of the characteristics of the state obeys inclusion then one threshold value per characteristic is sufficient to identify all regions. Thus inclusion of all characteristics that determine the state is a sufficient condition for the algorithm to meet the space bound. It is not a necessary condition however. It is possible for there to be several thresholds for an attribute, provided the number of regions is still bounded.

For multi-processor caches, the space limit allows a separate stack per cache. However, the information about each block must still be bounded. If there are $K$ caches, the number of stack entries is at most $K*S$, and space is still $O(S)$. Space is also required for storing the metrics being accumulated. This is discussed in the next section.

## 5.4. Metrics for Multi-processor Simulations

Before discussing the application of stack analysis to multi-processor systems we should briefly discuss the expected results of such an analysis. For single-processor systems, two metrics are most common: the *miss ratio* and the *transfer ratio*.

The *miss ratio* is the ratio of requests that are not satisfied by the cache to total requests. One of the desirable traits of a stack algorithm is that the miss ratio always decreases with increasing cache size — performance never gets worse by adding memory. For a multi-processor system we define the miss ratio in exactly the same way. We assume that bus requests from other processors do not count as a ''miss'' that slows down the processor. This is approximately true if the cache is dual-ported. A write back from a local cache may or may not be counted as a miss since it may be possible to buffer the write and perform it in parallel with other processing [Smit79].

The miss ratio does not measure the slowdown that may occur due to queuing while waiting for a path to memory. This is generally negligible on a single processor system, but in a shared memory multi-processor system the path to memory may be a critical resource, and queuing delays may be important [Arch86]. Also, some methods to reduce the miss ratio, such as prefetch or increased block size, may actually increase the number of memory accesses. Therefore we need a metric to measure memory and bus traffic.

The *transfer ratio* was defined in Chapter 2 to be the ratio of memory accesses with and without cache. It was originally used by Smith in studies of prefetch policies [Smit78b]. Goodman applied the metric to multi-processor systems, defining it as the ratio of bus transfers with and without cache [Good83]. With cache there is a bus data-transfer for each read miss, prefetch, write fetch, and write back, as well as bus actions at other times to affect cache consistency.

Both the miss and transfer ratios may be a function of each cache size in a multi-processor system, in other words, a K-dimensional surface. Therefore it might appear that just maintaining the metrics would require space which is O($S^K$). There are cases where this is true, as well as others where the metrics can be maintained in O($S$) space. There are also cases where the metrics require O($S^K$) even though the cache states themselves can be represented in O($K$*$S$) space, as we will show in Section 5.7. Not only do these violate our definition of efficient one-pass algorithms, it also creates an unmanageable result space for more than two or three caches. In many situations this is simplified by fixing the size of some caches or assuming that all caches are the same size. In this case the metrics can also be maintained in O($K$*$S$) space. For now we will assume that the cache sizes are independent in order to explore the limits of one-pass analysis.

### 5.5. Cache Consistency Protocols

There have been many protocols described to maintain the consistency of local caches in a multi-processor system [Good83, Fran84, Papa84, Rudo84, Katz85, Arch86]. The protocols of interest all rely on a single interconnection bus and *snooping* caches that see all bus requests. Consistency is assured by each cache maintaining the state of each block that it contains, and updating the state in response to its own requests and the bus actions of other caches.

There are two general strategies to maintaining consistency between caches — invalidation and write broadcast. In the invalidation schemes, such as the Berkeley and Illinois protocols [Katz85,Papa84], a write causes the block to be invalidated in all other caches. Any cache needing the block must re-read it to get the updated value. In the write broadcast schemes, such as the Dragon and Firefly protocols [McCr84, Arch86], the modified block is written through to the bus if it is shared so that all interested caches can get the updated value. A write to a cache that does not contain the block (a write miss) must be handled as a special case, since the cache may need to obtain the current value prior to the modification. This can done by performing a Read to get the current value followed by the Write, or by invalidating all other caches after receiving the current value.

A write-through cache can be considered to be using a form of write-broadcast since the write-through simultaneously updates main memory and updates (or invalidates) all other cached copies. Therefore no other consistency controls are needed when write-through is used on a multi-processor bus. Similarly, there are no consistency problems when blocks are not cached. Both these alternatives requires a bus action for every Write (and Read when no caching is done), and therefore are not practical when the bus bandwidth may be a limiting factor. For a more complete discussion of multi-processor protocols and tradeoffs, see [Arch86] and [Swea86].

In addition to the specific protocols mentioned earlier, Sweazey and Smith suggest a class of compatible protocols, referred to as MOESI protocols [Swea86], compatible with the IEEE Futurebus control lines and signals [P896]. Theirs is a class of protocols, not a single protocol, because it incorporates a number of alternative actions in different situations. The MOESI class of protocols includes several of the individual protocols mentioned. In addition it is a compatible class in the sense that caches using the class can choose different alternatives with no effect on the consistency assurances, only on performance. This allows a wide mixture of devices with different protocols on the same bus. In fact, the protocol could vary from cache to cache, from block to block within a cache, or even from time to time for the same block.

The MOESI protocols are so called because each block in a cache is in one of five states, represented by the letters MOESI:

M - The block has been *modified*, and this is the only cached copy of the block. The copy in memory is not current.

O - The block has been modified but there are valid copies of the block in other caches. The cache that has the modified version is the *owner* of the block. The copy in memory is not current.

E - The block has not been modified and is *exclusive* to a single cache. The copy in memory agrees with the cached copy.

S - The block is potentially *shared* between caches and the block is unmodified or some other cache is the owner (i.e. has the block in state O). If all cached copies are in the S state then the copy in memory is current.

I - The block is *invalid* (i.e. not present) in the cache.

There are three characteristics of a block which combine to give these states: validity, exclusivity, and ownership. If a block is invalid then the other characteristics are meaningless. For valid blocks, the other characteristics can be taken in any of four combinations, as shown in Figure 5.1. Exclusivity determines whether a cache has the only copy of a block or if the block is shared. Ownership is another way of expressing whether the block has been modified; if the block has been modified in a cache then that cache "owns" the block and is responsible for propagating the modification to others. Figure 5.1 illustrates the characteristics and pair-wise relationships between the states.

The fact that the validity characteristic obeys inclusion forms the basis for the original stack analysis technique. In Chapter 4 we showed that ownership (or "dirtyness") also obeys inclusion in a single cache. We have no assurance that either of these properties holds for multiprocessor protocols, but this would be one way to prove that they are one-pass algorithms. There are no prior results about the inclusion property of exclusiveness or *sharing*, but intuitively a block is more likely to be shared in a larger cache compared to a smaller one since it has been present longer. We will show that this is in fact the case.

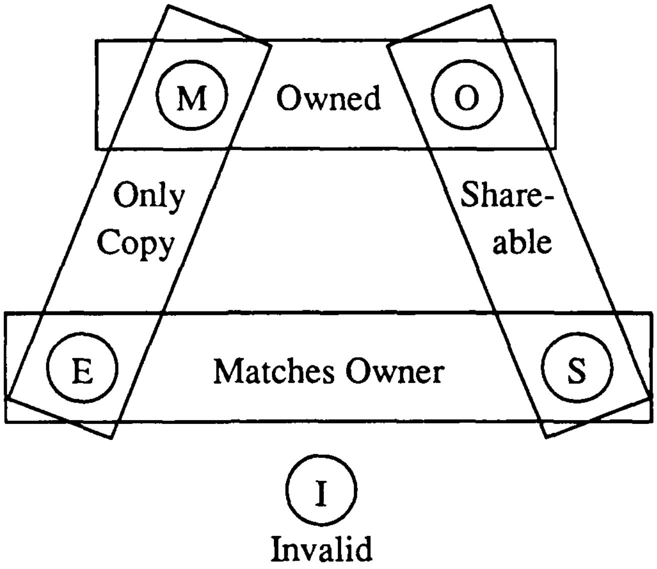


**Figure 5.1**: Relationships between MOESI states

The MOESI protocols are represented by Tables 5.1 and 5.2, which are transition matrices showing the actions taken and the result state if a particular *Event* is received by a cache in a particular state. Table 5.1 presents the actions for local requests to Read, Write, Pass (Copy-Back), or Flush the block. The cache receiving the request from its processor is known as the *master* cache for that request; all other caches are *slaves*. The actions may include performing a bus action to read or write data or give commands to other caches. The result state can depend on the state in other caches. For example, the entry {CS:S/E} for a Read to the I state means that the result state is S if a slave cache signals {CS} (i.e. it has the block), and E otherwise.

The matrix contains several alternatives for some states and events, any of which are acceptable. It is this choice of rules that makes this a class of protocols. These alternatives are numbered in braces and referred to as *rules* throughout the paper. For example, the second alternative for a Write to the O state is referred to as {M,CC,IM}{5}, or simply rule {5}.

| Table 5.1: MOESI protocols: Result State and Bus Signals - Local Requests | | | | |
|---|---|---|---|---|
| Event: note: From State | Read 1 | Write 2 | Pass 3 | Flush 4 |
| O | O | CS:O/M, {4}<br>CC,IM,BC,W<br>or<br>M,CC,IM {5} | CS:S/E, {15}<br>CC,BC?,W<br>or<br>S,CC,BC?,W {16} | I,BC?,W |
| M | M | M | E,CC,BC?,W | I,BC?,W |
| S | S | CS:O/M, {6}<br>CC,IM,BC,W<br>or<br>M,CC,IM {7}<br>or<br>S,IM,BC,W* {8}<br>or<br>S,IM,W* {9} | -- | I |
| E | E | M | -- | I |
| I | CS:S/E, {1}<br>CC,R<br>or<br>S,CC,R* {2}<br>or<br>I,R** {3} | M,CC,IM,R {10}<br>or<br>Read>Write {11}<br>or<br>I,IM,BC,W*,** {12}<br>or<br>I,IM,W*,** {13}<br>or<br>Read>Write {14} | -- | -- |

| Table 5.2: MOESI Protocols: Result State and Bus Signals - Bus Requests | | | | | | |
|---|---|---|---|---|---|---|
| Event: note: From State | CC,BC? 5 | CC,IM 6 | BC? 7 | CC,IM,BC 8 | IM 9 | IM,BC 10 |
| O | O,CS,DI,DK {17}<br>or<br>S,CS,DI {18} | I,DI,DK | CS:O/M,<br>DI,DK | S,SL,CS {23}<br>or I {24} | O,DI,DK,CS? | O,SL,CS |
| M | O,CS,DI,DK {19}<br>or<br>S,CS,DI {20} | I,DI,DK {21}<br>or<br>I,DI {22} | M,DI,DK,CS? | - | M,DI,DK,CS? | M,SL,CS? |
| S | S,CS | I | S,CS | S,SL,CS {25}<br>or I {26} | I | S,SL,CS {27}<br>or I {28} |
| E | S,CS | I | E,CS? | - | I | E,SL,CS? {29}<br>or I {30} |
| I | I | I | I | I | I | I |

## Notes on Tables 5.1 and 5.2

Format: result state (M, O, E, S, I),
bus signals (CC, IM, BC, BS, SL, DI, DK, CS)
action (R,W)

* Write-Through Cache

** No Cache

CC = cache master signal

IM = "intent to modify" used on address cycle to signal a write

BC = "broadcast" - signals intent to broadcast data write

CS = "cache status" - issued by slave that will retain a copy of the block. CS? = don't care, CS:O/M = If CS then O else M. Similarly for CS:S/E.

DI = response by slave signaling reflection to cause main memory to be updated.

DI,DK = response by slave signaling intervention

SL = response by slave or 3rd party signaling connect on write

BS = "busy" - aborts transaction

W = issue write on bus

R = issue read on bus

Any transaction of the form CS:O/M can be replaced by O

Any transaction of the form CS:S/E can be replaced by S

1: Read by local processor

2: Write by local processor

3: Push of dirty block by local processor, and keep a copy

4: Push dirty block and discard copy

5: Read by cache master on bus (includes write-through cache), or Pass by cache master.

6: Read for modify by cache master (i.e. write miss by write-back cache) or address only invalidate signal. Invalidates other caches.

7: Read by processor without cache, or Flush by cache master.

8: Broadcast write by cache master

9: Write by processor without cache, or write past write-through cache

10: Broadcast write by non-cache processor or write past write-through cache

- not a legal case. Error condition.

Table 5.2 presents the actions by a slave for bus requests from other caches. In this case the event shows exactly those bus signals out of {CC,IM,BC} that must be on for the column to be selected (i.e. CC,IM means signals CC and IM *on* and BC *off*). In two cases the BC signal is immaterial (BC?). Following the tables is a page of notes briefly explaining the bus signals and other symbols used in the tables. In particular, the numbered notes at the bottom give the meaning of certain bus signal combinations. See [Swea86] for a complete description of the MOESI class.

## 5.6. One-Pass Algorithms for MOESI Protocols

### 5.6.1. Initial Assumptions

We can immediately think of some situations in a multi-processor system where the MOESI protocols would not be one-pass algorithms, as described below, but these are correctable with some reasonable assumptions.

#### 5.6.1.1. Synchronous Reference Streams

Clearly the relative order of requests from different processors can affect the value of the metrics. For example, suppose that a block is written twice by processor one and once by processor two at nearly the same time. Also suppose the consistency protocol is one that invalidates the block in all other caches on a write (e.g. rules {5}, {7}, and {10}). If both writes from the first processor occur before the write from the second processor, there is no bus activity required for the second write. This is illustrated in Figure 5.2(a) which shows the state of a block in caches one and two after each request. On the other hand, if the writes from processor one are separated in time by the write from processor two then each of the writes requires a bus action, as shown in Figure 5.2(b).

| Time/Request: | Cache State | Bus Actions |
|---|---|---|
| Initial: | 1:I | |
| | 2:I | |
| Write 1: | 1:M | {CC,IM} |
| | 2:I | |
| Write 1: | 1:M | None |
| | 2:I | |
| Write 2: | 1:I | |
| | 2:M | {CC,IM} |

(a)

| | | |
|---|---|---|
| Initial: | 1:I | |
| | 2:I | |
| Write 1: | 1:M | {CC,IM} |
| | 2:I | |
| Write 2: | 1:I | |
| | 2:M | {CC,IM} |
| Write 1: | 1:M | {CC,IM} |
| | 2:I | |

(b)

**Figure 5.2**: The order of events can affect bus activity. There are only two bus transactions under the first ordering, but three under the second.

Since a cache miss may delay one processor while the other continues, the relative order of the requests can be affected by the size of each cache and by the other requests in the reference string. All of this information is therefore needed to know the precise state of a block, which requires unbounded space. From this we can conclude that, in principle, the MOESI protocols (at least the subset used above) are not one-pass algorithms.

We can avoid this problem by making the assumption that the processor requests are synchronous, in the sense that the relative order of the requests is fixed in advance and therefore independent of the cache sizes. A simple argument explains why this is a reasonable assumption. First, it should be clear from the example that slight changes in the sequence are just as likely to increase the number of bus requests as to decrease them. Therefore the metric will be relatively insensitive to small changes in the ordering. Second, if the processors do not share variables then the ordering is immaterial. If they do share variables and are processing the same workload then they will tend to have the same fault rates, and therefore will naturally stay relatively synchronized. If they are not processing the same workload but do share variables, for example if one of the processes is a server of the other, then the processes will tend to synchronize themselves periodically. In each case the request streams will naturally tend to remain synchronized in the order captured in the trace. Therefore there is little error introduced by assuming that this is a fixed ordering.

### 5.6.1.2. Replacement Priority

In a multi-processor cache there are two types of requests that the cache sees; local requests from its processor and bus requests from other caches. In general, a local read or write request that loads or retains a block will assign the block the highest replacement priority, bringing the block to the top of the stack. (The only time this is not the case in the MOESI protocols are the rules marked with "**" which apply only to processors without caches.) Now suppose that a bus request from another processor also assigned a block the highest priority. The block would therefore move to the top of the stack. However, this would have the effect of loading the block in response to a bus request, something which is never done in these protocols. We therefore make the assumption that bus requests, while they may change the state of a block, will not change the replacement priority (except perhaps downward by invalidating the block). This is also a reasonable assumption; the fact that another cache is using a block may be reason to discard a block but never a reason to want to keep it.

### 5.6.1.3. Consistent Rule Usage

The MOESI protocols permit the free choice of the possible rules for a particular state and action at any time. Although this is valid, it introduces situations that prevent their being one-pass algorithms. For example, suppose a cache alternates between Table 5.1 rule {1} which checks for the presence of the block in another cache and rule {2} which does not. This could result in an unbounded number of regions where the block is S or E, as shown in Figure 5.3 below. The figure shows a sequence of actions and the state of the block in different cache sizes after each action. For example, the third line shows that the block is Invalid for small sizes and

Exclusive beyond the size marked by "E".

---

```
  Time     Rule                Result  States

Initial:                    1:I------------------>
Read:     {CS:S/E}{1}       1:E------------------>
Later:                      1:I-------------E---->
Read:     {S}{2}            1:S-------------E---->
Later:                      1:I----------S--E---->
Read:     {CS:S/E}{1}       1:E----------S--E---->
Later:                      1:I-------E--S--E---->
Read:     {S}{2}            1:S-------E--S--E---->
```

**Figure 5.3**: Unbounded regions result from inconsistent rule usage. The figure shows the state of a single block for increasing cache sizes. It assumes that the block is only in cache one. The cache sometimes tests the {CS} signal on a read miss and enters the E state. Other times it assumes the block is shared. A Read does not affect the state in cache sizes where the block is present. The "Later" lines show the state after the block has been pushed from some sizes by references to other blocks. Repetition of the sequence can lead to an arbitrary number of E and S regions.

---

We avoid this problem by assuming that a cache consistently chooses the same rule in a particular situation for a given block. This means that a cache using invalidation on a block will always use invalidation, never switching to broadcast. This is reasonable since the protocol (i.e. the set of rules to be applied) is generally fixed in advance. This does not require that all caches follow the same rules, nor that a cache use the same rules for all blocks, both of which would be serious restrictions. For example, it allows some blocks to be write-back and others write-through, as provided in some real caches [Cho86]. On the other hand it is unlikely that a cache would change rules for a given block, although it does prohibit some adaptive protocols that switch based on historical reference patterns.

### 5.6.2. A Simple Example - the Berkeley Protocol

This section begins the analysis of the MOESI protocols to determine which of the protocols are one-pass algorithms, and under what conditions. This is done by considering progressively more complex examples in order to demonstrate some of the techniques used to show that a protocol is a one-pass algorithm. We first consider the simple subset of the MOESI protocols known as the Berkeley protocol, shown in Table 5.3, and proposed for use in a multi-processor system at UC Berkeley [Katz85]. The protocol performs an invalidation bus action on Write unless the cache contains the only copy. As originally designed the protocol uses a V (valid) state for any unmodified block, instead of the separate E and S states. Sweazey and Smith have mapped this to the S state to avoid adding a new state. Therefore the E state is not used in Table 5.3.

Table 5.3: Berkeley Protocol

| Action: Current State | Read | Write | Pass | CC | CC,IM |
|---|---|---|---|---|---|
| M | M | M | S,CC,BC?,W | O,CS,DI,DK | I,DI,DK |
| O | O | M,CC,IM | S,CC,BC?,W | O,CS,DI,DK | I,DI,DK |
| S | S | M,CC,IM | -- | S,CS | I |
| I | S,CC,R | M,CC,IM,R | -- | I | I |

We begin the analysis by considering a two-processor system and looking at the state of a particular block in each local cache, and allowing the cache sizes to vary independently. We must also assume that the state of a block could be a function of both cache sizes. We therefore represent the state as a two-dimensional graph for each cache, where the x-axis is the size of cache one and the y-axis is the size of cache two. We will refer to a particular arrangement of states within an individual cache as a *configuration* of states. In general, we will be trying to show that there are only a few possible configurations, and therefore they can be represented in bounded space. Before generalizing, we look at the effect of a few specific requests using the Berkeley protocol.

Consider the sequence of actions shown in Figure 5.4. We refer to the initial configuration where the block is Invalid in all sizes as the I configuration. The state of the two caches is called <I,I>. There are three other configurations present in the figure. The S configuration (fifth frame) contains just the S state and possibly an invalid region for small sizes. In the M configuration (fourth frame, among others), the block is in the Modified state for the largest sizes but may have an Invalid and/or unmodified region for small sizes. Note, however, that sharing is not inclusive, since the S state is a shared state and M is an exclusive state. We will address this in

Section 5.7.2.1. The O configuration (fifth frame) is similar, and occurs when the block is in fact shared. In this case the other cache must be in the S configuration; it may not be modified for any size. Note too that the M and O states are mutually exclusive (i.e. a block can not be in states M and O at the same time for different sizes). In order to enter the M state the block must be written, invalidating the other caches for all sizes. Any reference from another cache will miss in all sizes, converting the M states to O.

We now claim that the four configurations introduced above (I,S,M,O) are the only possible configurations for the Berkeley protocol. To show this, we look at the configurations that would result from each possible action, given any combination of configurations. Each pair of configurations is illustrated in Figure 5.5. Table 5.4 was constructed by starting with the <I,I> state and considering all possible actions. We have already seen that a write to cache one yields <M,I>, while a read gives <S,I>. We next explore these two configurations. There is no need to explore the symmetric <I,S> or <I,M> configurations which result from accesses to cache two, since both caches are identical and interchangeable. The table shows that there are only five reachable pairs of configurations. Since only one cache at a time is ever dirty, we assume it is cache one and only consider a Pass from the dirty cache.

Table 5.4: Possible Configurations - Berkeley Protocol

| Action: Initial State: | Read 1 | Write 1 | Pass 1 | Read 2 | Write 2 |
|---|---|---|---|---|---|
| I,I | S,I | M,I | -- | I,S | I,M |
| S,I | S,I | M,I | -- | S,S | I,M |
| S,S | S,S | M,I | -- | S,S | I,M |
| M,I | M,I | M,I | S,I | O,S | I,M |
| O,S | O,S | M,I | S,S | O,S | I,M |

This figure shows the state of a single block in each of two caches. The size of each cache is allowed to vary independently, and the state may vary according to the size of each cache. Shaded regions show the state in all combinations of cache sizes. The figure shows the progression of states in response to various actions.

Initially, neither cache contains the block, so that it is invalid for all sizes. The state of the two caches is <I,I>.

The first action is a Write request from processor one. The block is read into cache one in state M for all sizes. The state is now <M,I>.

Later, the block has been pushed from some sizes of cache one by references to other blocks. It is therefore invalid for some cache one sizes but still M in all larger sizes. Note that any configuration can have this region of invalid sizes caused by pushes. Unless otherwise stated, all *remaining examples will show the resulting configuration after the block has been pushed from some sizes.*

Processor one Reads the block. The block is loaded in state S into those sizes where it is invalid, but remains in the M state for the larger sizes. As the block is pushed, the invalid region grows, but the boundary between S and M remains unchanged.

Processor two Reads the block. This causes a {CC} bus request to be seen by cache one, which changes the M states to O, without affecting other states. The new cache states are <O,S>.

Processor two Writes the block. This causes a {CC,IM} bus request, which invalidates the block in cache one. The resulting state is <I,M>, similar to the <M,I> seen before.

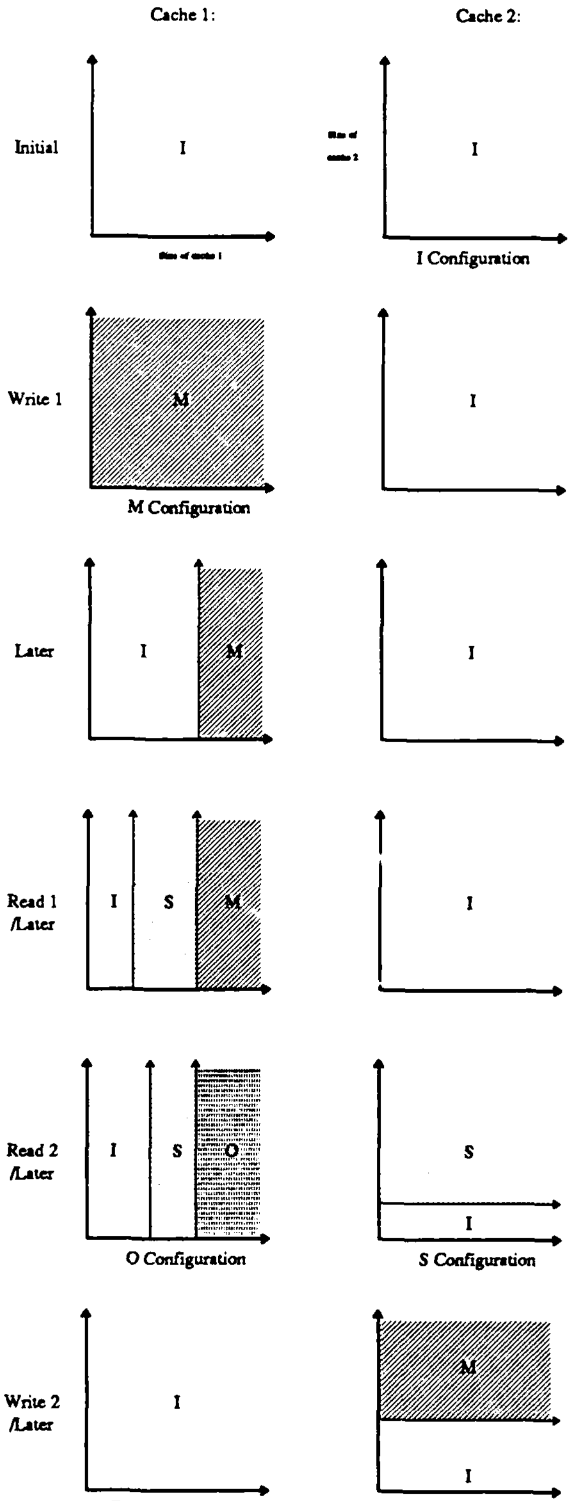

Cache 1:
Cache 2:
Initial
I
Size of cache 2
I
Size of cache 1
I Configuration
Write 1
M
I
M Configuration
Later
I
M
I
Read 1 /Later
I
S
M
I
Read 2 /Later
I
S
O
S
I
O Configuration
S Configuration
Write 2 /Later
I
M
I

**Figure 5.4**: Examples of actions using the Berkeley Protocol

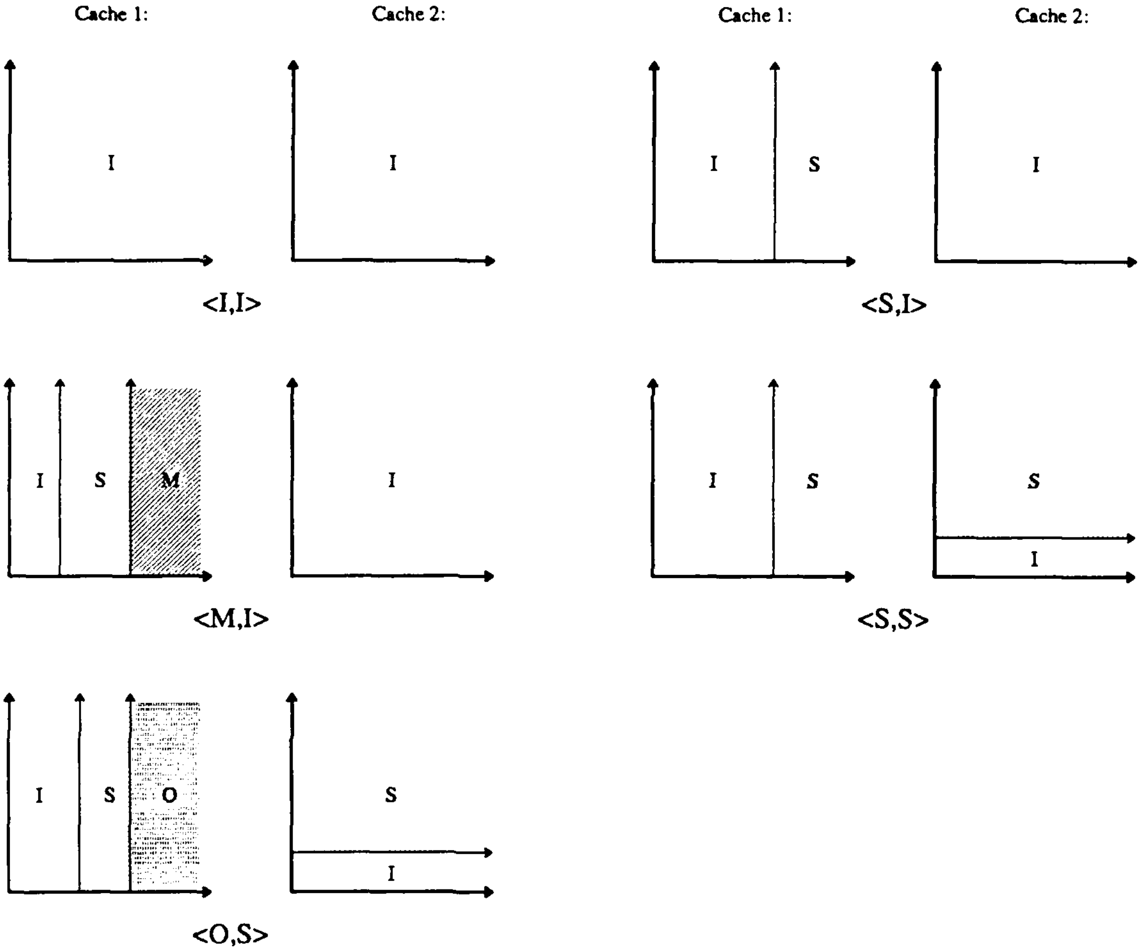


**Figure 5.5**: Examples of possible pairs of configurations.

Looking again at the configurations in Figure 5.5 above we can see that each region within a cache is bounded by a single straight line, that is, by a single cache size, independent of the size of the other cache. This makes it trivial to represent each configuration with a fixed set of values, one for each boundary. It is even simpler in this case, for several reasons. First, since each cache uses a stack for the replacement policy, the upper bound of the I region is simply the level of the block in the stack, so no additional variable is needed. Second, only one cache at a time is in the M or O configuration so there is no need for a dirty level for each cache. Also, states M and O are mutually exclusive. We can therefore represent the state of the block in all caches by two stacks (one per cache) and a tuple <*Dirty cache*, *Dirty level*, *Owned flag*> for each block, containing the identity of the cache where the block is dirty, the smallest cache size where it is dirty, and a flag indicating whether the dirty region is in state M (not owned) or O (owned). Since there are a maximum of $S$ distinct blocks in each of two caches, the space required for stacks and tuples is $O(2*S)$. A careful examination shows that this result easily extends to $K$ processors since invalidation ensures that only one can be dirty at a time.

The space required to accumulate the simulation metrics for this protocol is also $O(K*S)$. The miss ratios for each cache are independent of the other cache sizes, so one array per cache is sufficient. The contribution of each cache to the transfer ratio is also independent of the sizes of other caches, so again one array per cache is sufficient. The transfer ratio for any combination of cache sizes could be computed if necessary by summing the contributions of each cache at the appropriate size.

### 5.6.3. A Second Protocol

Now consider a slightly more complex example. The only difference between this and the Berkeley protocol is that it does not invalidate other caches on a write except for a write miss. If the block is shared then write broadcast is used to update the other copies. There is still no E state. See Table 5.5. This particular protocol does not match any of the common protocols, although it is a valid subset of the MOESI class. It is presented because it illustrates a situation where the state can be a function of both cache sizes.

Table 5.5: Second Protocol

| Action: Current State | Read | Write | Pass | CC | CC,IM | CC,IM,BC |
|---|---|---|---|---|---|---|
| M | M | M | S,CC,BC?,W | O,CS,DI,DK | I,DI,DK | -- |
| O | O | CS:O/M,<br>CC,IM,BC,W | S,CC,BC?,W | O,CS,DI,DK | I,DI,DK | S,SL,CS |
| S | S | CS:O/M,<br>CC,IM,BC,W | -- | S,CS | I | S,SL,CS |
| I | S,CC,R | M,CC,IM,R | -- | I | I | I |

The state after a Write and a Read in cache one and a Read in cache two is identical to that seen in Figure 5.4.

Processor one Writes the block. In cache one, the block becomes M for all sizes where it was invalid, and invalid in cache two for these same sizes. In the sizes where the block is valid in one, its new state depends on the value of {CS} — it is O where valid in cache two and M where it is invalid. The original valid levels in cache one and two are labeled $s_x$ and $s_y$ respectively. The resulting state is called <MO,IS>.

The invalid region grows as the block is pushed. A push in one cache does not affect the other; the bottom of the O region in cache one stays at $s_y$, the left boundary of the S region in cache two remains at $s_x$.

Processor one Reads the block and it becomes state S in its invalid sizes. The state in cache two is unaffected, since it is invalid.

Processor two Writes the block. The resulting state is <IS,MO>, and $s_y$ changes to the current valid level in cache two. Although the block was valid below level $s_x$ in cache one, the $s_x$ level is unchanged.

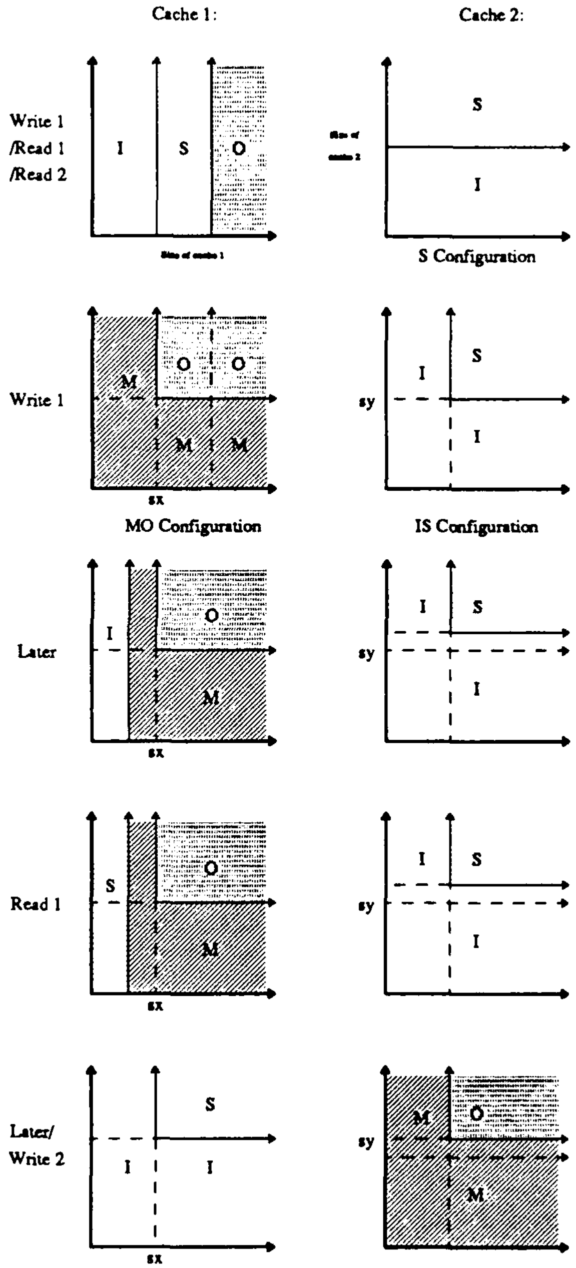

Cache 1:
Cache 2:
Write 1
/Read 1
/Read 2
I
S
O
S
I
S Configuration
Write 1
M
O
O
M
M
sx
I
S
sy
I
MO Configuration
IS Configuration
Later
I
O
M
sx
I
S
sy
I
Read 1
S
O
M
sx
I
S
sy
I
Later/
Write 2
S
I
I
sx
sy
M
O
M

Figure 5.6: Actions using the second protocol.

Consider the sequence of actions using this second protocol, as illustrated in Figure 5.6. The configurations when one or both of the caches are totally Invalid are similar to the Berkeley protocol. However, a Write to a shared configuration leads to a new combination of configurations. We see in frame two that there is a "shared" region of either O in cache one or S in cache two, each bounded at the bottom left by the point $(s_x, s_y)$. We call the first an MO configuration, the latter an IS configuration.

Also, in the last frame note that although the block was valid in cache one below level $s_x$ before the action, the $s_x$ level is unchanged due to the invalidation rule used by cache two on a write miss. However if the "valid level" in cache one was to the right of $s_x$ then $s_x$ would take that value, since a Write to the S state uses the {CS} signal to see if the block is present elsewhere. Thus the new value of $s_x$ is the maximum of itself and the current valid level in cache one.

We again claim that the configurations presented so far are sufficient to describe all possible states of two caches for the Second protocol. Table 5.6 below shows the possible configurations and transitions. We again start with the <I,I> configuration and explore all possible actions and resulting configurations.

Table 5.6: Possible Configurations — Second Example Protocol

| Action: Starting State: | Read 1 | Write 1 | Pass 1 | Read 2 | Write 2 |
|---|---|---|---|---|---|
| I,I | S,I | M,I | -- | I,S | I,M |
| S,I | S,I | M,I | -- | S,S | I,M |
| M,I | M,I | M,I | S,I | O,S | I,M |
| S,S | S,S | MO,IS | -- | S,S | IS,MO |
| O,S | O,S | MO,IS | S,S | O,S | IS,MO |
| MO,IS | MO,IS | MO,IS | S,IS | O,S | IS,MO |
| S,IS | S,IS | MO,IS | -- | S,S | IS,MO |

It is apparent that the MO and IS configurations can not be represented by a single level of cache size. However, since there is only one O or S region, bounded on the lower left, the number of variables required to describe each configuration is bounded. Since the number of possible configurations is also bounded, the state definitions meet the space requirements of a one-pass algorithm. In fact, the number of variables required is very small. In addition to the stack for each cache to maintain the valid levels, it only needs the tuple <*Dirty cache*, *Dirty level*, $(s_x, s_y)$> for each block, where $(s_x, s_y)$ is interpreted as the smallest combination of sizes where the block may be present in both caches (i.e. "shared"). The valid levels may be below or above these levels, as can the dirty level. The MO configuration is present only in the dirty cache, and the O configuration is actually an MO configuration in which one of the shared levels ($s_x$ or $s_y$, depending on the cache) is zero. The S state is used for any unmodified block, regardless of whether it is inside or outside the shared region. For example in the fourth frame of Figure 5.6 the S region extends below $s_y$.

Unfortunately the space required to accumulate metrics for this protocol is $O(S^2)$ for two caches. Consider the last configuration of Figure 5.6. A Read to cache one will "hit" for all cache sizes above $s_x$ *and* above $s_y$. Since $s_x$ and $s_y$ can vary independently between zero and $S$, the simulator needs to be able to accumulate misses for any point in 2-space. This obviously extends to K-space for $K$ processors. Thus this protocol is not an efficient one-pass algorithm by our definition.

### 5.6.4. A Complex Example - the Dragon Protocol

We now consider a more complex example for which we can not even represent the state within the one-pass constraints. This protocol, shown in Table 5.7, is essentially the Dragon protocol [McCr84] as presented in [Swea86]. The main difference between this and the previous protocol is the use of alternative rule {CS:S/E,CC,R}{1} from Table 5.1 for a Read miss. This introduces the exclusive E state. Also, a Write miss is implemented as a Read followed by a Write, requiring two cycles but no invalidation is done.

Table 5.7: Dragon Protocol

| Action: Current State | Read | Write | Pass | CC | CC,IM,BC |
|---|---|---|---|---|---|
| M | M | M | E,CC,BC?,W | O,CS,DI,DK | -- |
| O | O | CS:O/M,<br>CC,IM,BC,W | S,CC,BC?,W | O,CS,DI,DK | S,SL,CS |
| E | E | M | -- | S,CS | -- |
| S | S | CS:O/M,<br>CC,IM,BC,W | -- | S,CS | S,SL,CS |
| I | CS:S/E,<br>CC,R | Read>Write | -- | I | I |

Again consider a sequence of actions in a two processor system, shown in Figure 5.7. The initial configuration is similar to that seen for the Berkeley protocol except that the E state is used instead of S. There are several new configurations as well.

When processor two reads the block in the second frame, the {CC} bus request sets the shared characteristic for all sizes, changing the state in cache one from E to S and M to O. In cache two, the state depends on the size of cache one: for sizes where the block is invalid in cache one there is no {CS} signal, and the state in cache two becomes E; for sizes where it is valid the {CS} is generated by cache one and the state in two is S. The current valid level in cache one forms a ''sharing level'' in cache two. The configuration is called ES.

When processor one reads the block in the next frame, the block is similarly either exclusive or shared in its invalid sizes, depending on the size of cache two. The invalid region in cache one is guaranteed to completely cover the exclusive region in cache two (otherwise it wouldn't be exclusive). Therefore the {CC} request again causes all exclusive states in cache two to become shared. This configuration is called ESO.

Initially, only cache one contains the block. We refer to this state as <M,I>.

Processor two reads the block. It becomes shared (or owned) in cache one. In cache two, the block is either exclusive in all sizes or shared in all sizes, depending on the size of cache one. The new state is <O,ES>.

Later, the block has been pushed and therefore invalid in some sizes of cache two. Cache one is unchanged.

Processor one reads the block. The block is either exclusive or shared, depending on the size of cache two. The new state is <ESO,S>.

Later the block has been pushed from some cache one sizes and from some cache two sizes. Since the block has not been referenced in cache two, the valid level is at least at the level of the top of the "exclusive" region in cache one.

Processor one reads the block. It again is either exclusive or shared in the cache one region where it was invalid. There is no single pair of cache sizes that define the limits of the "exclusive" region.

There are actually three overlapping shared regions, defined by the points S1, S2, and S3. The points and regions apply to both caches, although only the region defined by S3 is relevant in cache two since the other shared points are in an invalid region. Valid and dirty levels are defined by $V_x$, $V_y$, and $D_x$.

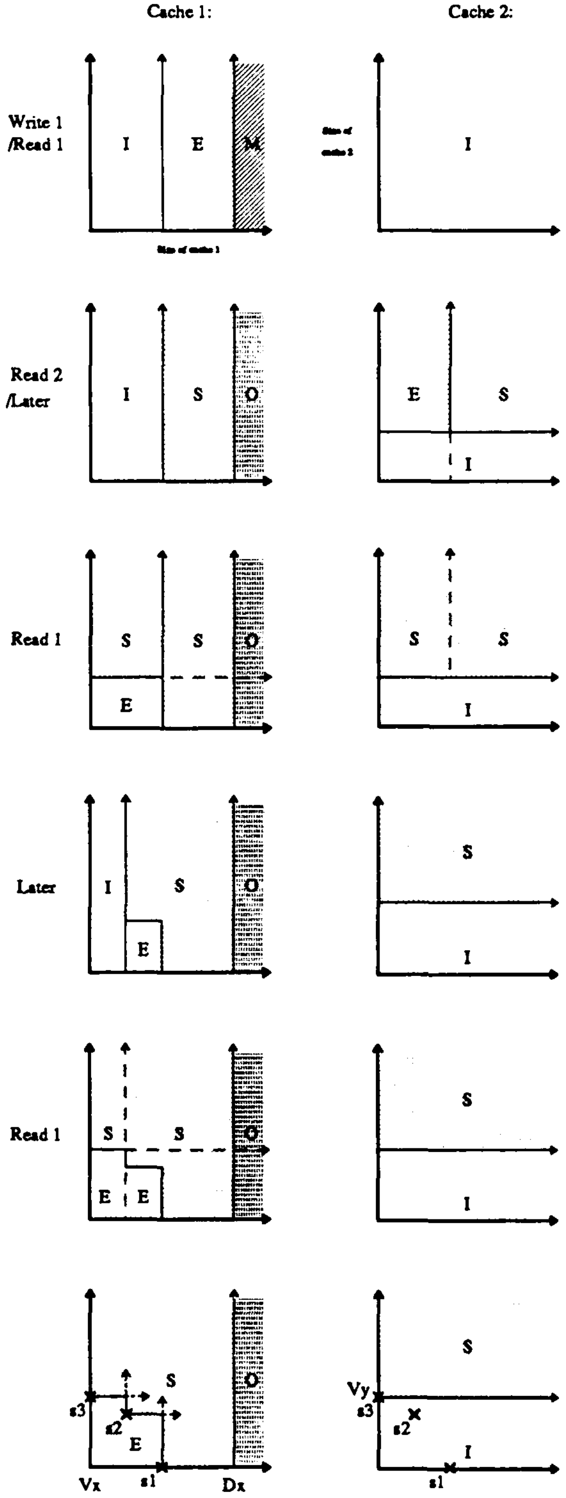


**Figure 5.7**: Actions using the Dragon protocol.

When processor one reads the block for the last time there is no longer a single pair of cache sizes that define the limits of the "exclusive" region. The shared attribute does not obey inclusion, that is, there is not a unique threshold size beyond which the block is shared. There are actually *three* overlapping regions of sharing, defined by the points $S1$, $S2$, and $S3$. These points and regions apply to both caches, although only the region defined by $S3$ is valid in cache two. The validity and dirty attributes are still inclusive; the block is valid for sizes greater than or equal to $V_x$ and $V_y$ and dirty for size $D_x$ and larger.

The block is only potentially shared above $Si$, not necessarily shared. For example, in Figure 5.7 frame 4, cache two contains the only cached copy of the block for small sizes of cache one, but the state is still S. This occurs because the push of a clean page causes no bus action to change the state in other caches. For this reason the shared points do not change as the valid levels increase.

The last configuration can no longer be represented with a small number of variables per block. In fact, repeated reads by cache one, combined with appropriate pushes, could create an arbitrary number of such regions. The worst case would be the following sequence: Read 1, Read 2, push to level $S$ in 1, Read 1, push to level $S-1$ in 1 and level 1 in 2, Read 1, push to level $S-2$ in 1 and level 2 in 2, Read 1.... This would require $S$ sharing points extending along the minor diagonal. Since the number of variables required per block to represent this configuration is bounded only by $S$ we will call this type of configuration an ***unbounded configuration***, in contrast to the ***bounded configurations*** we have seen so far. Although this means that this is not a one-pass algorithm by the definition, we will show that it is a one-pass algorithm for certain common cases. First we need to show that there are only a small number of possible configurations, bounded or unbounded.

Some further analysis shows that there are only a few general combinations of configurations for the Dragon protocol. When the block is valid in only one cache then the combination is either <E,I> or <M,I>, which we have seen before. If the block is shared, then there are $n$ overlapping shared regions defined by points $S1 \cdots Sn$, where point $Si$ has the coordinates $(Si_x, Si_y)$, and $1 \le n \le S$. For convenience they are numbered from bottom to top as seen in Figure 5.7. Assuming that the exclusive region is in cache one, the levels obey the following conditions:

$$0 = Sn_x \le V_x < Sn-1_x < \cdots < S2_x < S1_x$$

$$0 \le S1_y < S2_y < \cdots < Sn-1_y < Sn_y \le V_y$$

$$V_x \le D_x, D_y = \infty \text{ or } V_y \le D_y, D_x = \infty$$

depending on whether cache one or two is dirty. There are symmetric conditions if cache two has the exclusive region.

The simplest case of sharing, shown in Figure 5.8, satisfies these conditions with a single sharing point, $S1$. The final state in Figure 5.7 satisfies the condition with three points. We can show that all combinations of configurations of two caches using the Dragon protocol are either non-shared or meet this condition by once again considering all possible actions beginning with the <I,I> state. This is shown in Table 5.8. Any state that meets the above condition is called simply a Dragon or <D> state.

Table 5.8: Possible Configurations — Dragon Protocol

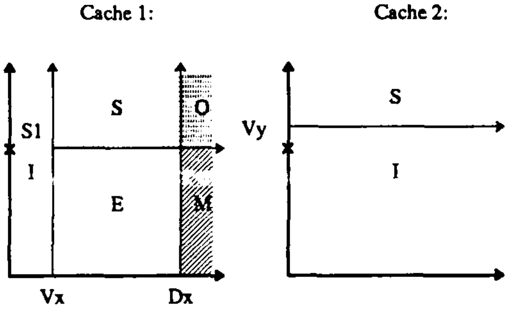
| Action: Starting State: | Read 1 | Write 1 | Pass 1 | Read 2 | Write 2 |
|---|---|---|---|---|---|
| I,I | E,I | M,I | -- | I,E | I,M |
| E,I | E,I | M,I | -- | D | D |
| M,I | M,I | M,I | E,I | D | D |
| D | D | D | D | D | D |

To show that the last line is true we must show that any action, starting from the <D> state, leads back to a Dragon state. We begin with the simple case of Figure 5.8(a), which is the result of a Read in two, a Write in one, and a Read in one, with some intermediate pushes. We must consider the result of each possible action applied to this case. The roles of cache one and two are interchangeable, so only one is actually shown. We see that any Read creates a second sharing point, $S2$, except a read to cache one when $V_y = S1_y$. A Write just moves $S1$. The new values of $S1$, $S2$, $Vx$, $Vy$, and $Dx$ or $Dy$ are given below the graphs for each action. Notice that they are independent of $Dx$ or $Dy$, which are unchanged after a Read and O after a Write.

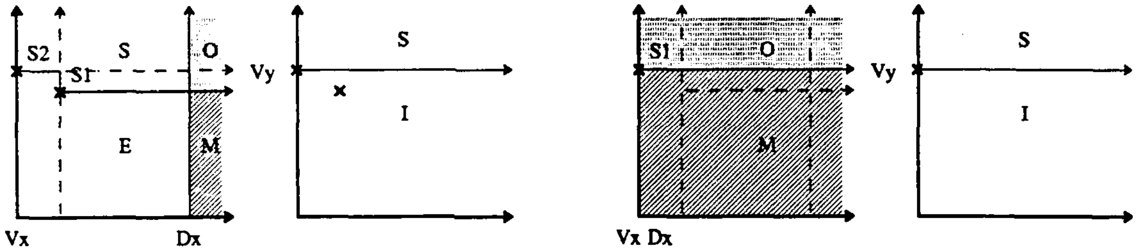
$0 = S1_x \le \infty$

$V_x \le D_x$ or $V_y \le D_y$

$0 \le S1_y \le V_y$

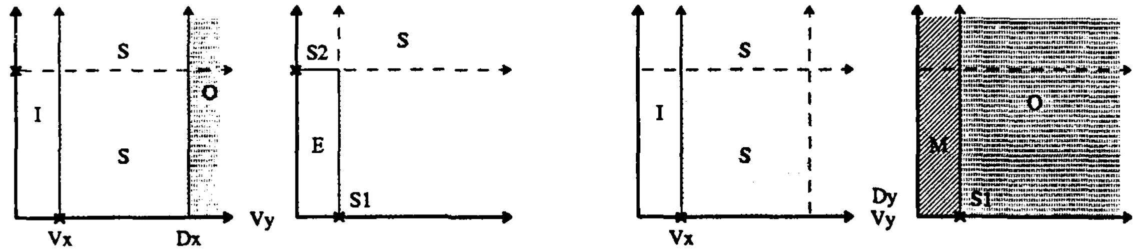

Cache 1:
Cache 2:
S1
S
O
I
E
M
Vy
Vx
Dx
S
I

a) Initial Configuration


S2
S
S1
O
Vy
E
M
Vx
Dx
S
I
S1
O
M
Vy
Vx Dx
S
I

b) Read 1: $S2=(0,V_y)$
$S1=(V_x,S1_y)$
$V_x=0$

c) Write 1: $S1=(0,V_y)$
$V_x=D_x=0$

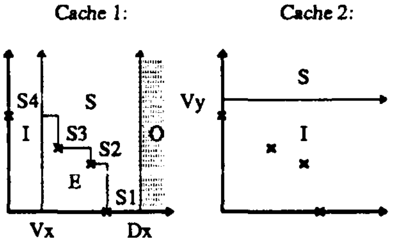

S
I
S
O
Vx
Dx
Vy
S2
S
E
S1
S
I
S
Vx
Dy
Vy
O
M
S1

d) Read 2: $S2=(0,V_y)$
$S1=(V_x,0)$
$V_y=0$

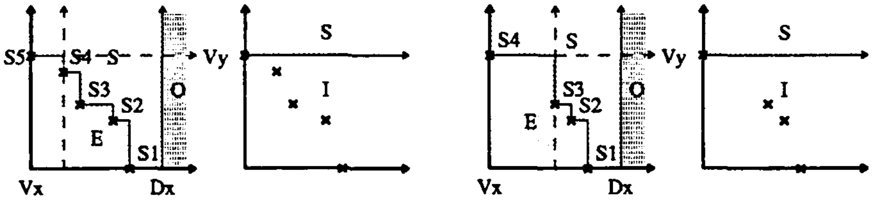
e) Write 2: $S1=(V_x,0)$
$V_y=D_y=0$

**Figure 5.8**: The simple Dragon configurations produce others. Each action, applied to the initial configuration, produces a state meeting the Dragon conditions. The effect of each action on the state variables is shown.

$0 = S4_x < S3_x < S2_x < S1_x \le \infty$
$V_x \le D_x$ or $V_y \le D_y$
$0 \le S1_y < S2_y < S3_y < S4_y \le V_y$

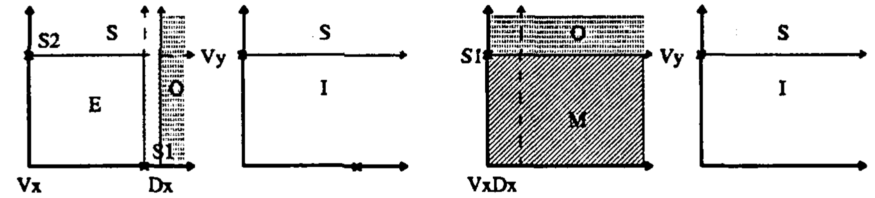

Cache 1:
Cache 2:
S4
S
I
S3
S2
O
E
S1
Vx
Dx
Vy
S
I

a) Initial Configuration


S5
S4
S
S3
S2
E
S1
O
Vx
Dx
Vy
S
I
S4
S
S3
S2
E
S1
O
Vx
Dx
Vy
S
I

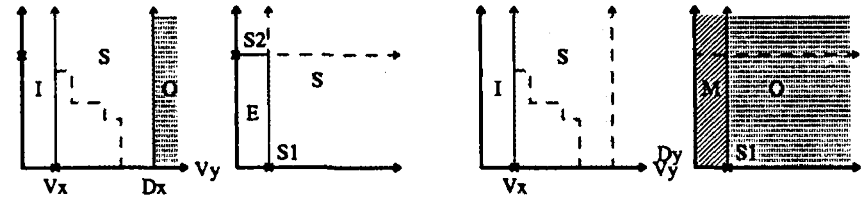
b) Read 1, $S4_x \le V_x < S3_x$:
$S5=(0,V_y)$
$S4=(V_x,S4_y)$
$V_x=0$

c) Read 1, $S3_x \le V_x < S2_x$:
$S4=(0,V_y)$
$S3=(V_x,S3_y)$
$V_x=0$


S2
S
E
O
S1
Vx
Dx
Vy
S
I
O
S1
M
VxDx
Vy
S
I

d) Read 1, $V_x \ge S1_x$:
$S4=S3=(\infty,\infty)$
$S2=(0,V_y)$
$S1=(V_x,S1_y)$
$V_x=0$

e) Write 1:
$S4=S3=(\infty,\infty)$
$S1=(0,V_y)$
$V_x=D_x=0$


I
S
O
Vx
Dx
Vy
S2
S
E
S1
I
S
Vx
Dy
Vy
M
O
S1

f) Read 2:
$S4=S3=(\infty,\infty)$
$S2=(0,V_y)$
$S1=(V_x,0)$
$V_y=0$

g) Write 2:
$S4=S3=S2=(\infty,\infty)$
$S1=(0,V_y)$
$S1=(V_x,0)$
$V_y=D_y=0$

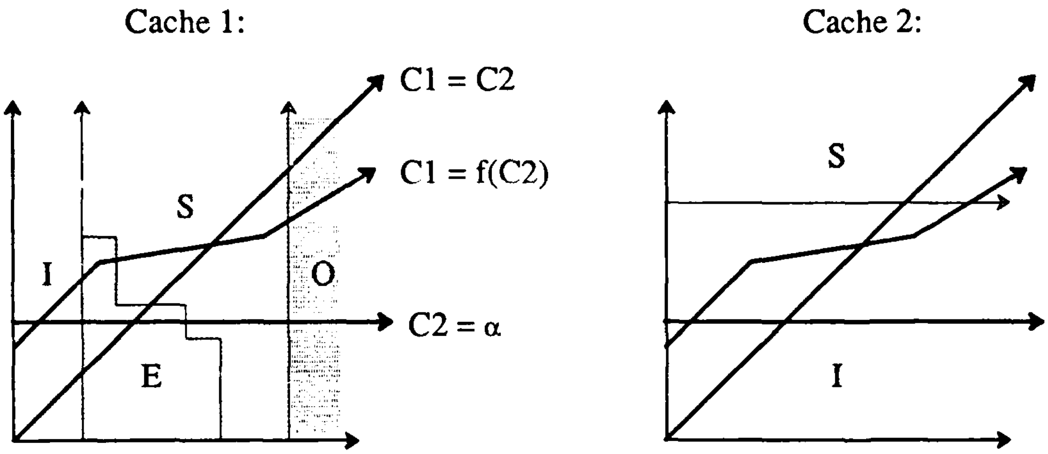
**Figure 5.9**: The general shared Dragon configuration produces others.

Now consider the more general case shown in Figure 5.9. Again, we show all possible actions applied to the initial case given in a), and demonstrate that the result fits the general conditions. In the case of a Read to cache one there are several subcases to consider, depending on the value of $V_x$. In each case the prior value is shown as a vertical dashed line. If $V_x$ is less than the x-coordinate of the next-to-last sharing point ($S3$ in this case), then the result is the addition of a new point. If $V_x$ is between $S2$ and $S3$ then the number of points stays the same. If $V_x$ is between $S1$ and $S2$, or less than $S1$, one or more of the sharing points go away.

In each case notice that there is a distinct pattern to the new values of the sharing points. Note too that these new values all satisfy the condition of the general Dragon state. A Read to cache two always yields two sharing points, with values that satisfy the condition with the roles reversed. A Write to either cache results in the simplified sharing example of the previous figure. The Pass is not shown since it only changes the values of the dirty level, and thus also preserves the condition. We have therefore shown that any action applied to a general Dragon state gives another Dragon state, and therefore we have seen all possible configurations. Keep in mind, however, that the general Dragon configuration can not be represented in bounded space, and therefore the Dragon protocol can not be simulated in one-pass for independent cache sizes.

### 5.6.5. Related Memory Sizes

In three examples we have seen one protocol that was a one-pass algorithm for independent cache sizes and two that were not. However, all are one-pass algorithms when certain relations exist between cache sizes. Consider the general Dragon state shown in Figure 5.10. Any non-decreasing (in x and y) line through the two-cache state graphs will intersect the region boundaries exactly once. This is true because all state boundaries are single horizontal or vertical lines, except the "sharing" boundary which always progresses downward to the right. Therefore the state of any pair of sizes along the line can be represented by a fixed set of threshold values. This applies to the state graphs for the two prior protocols as well. Some interesting cases are $C1=\alpha$, $C2=\alpha$, and $C1=\alpha C2$, where $C1$ and $C2$ are the cache sizes and $\alpha$ is a constant.


Cache 1:

Cache 2:

**Figure 5.10**: Possible cache size relationships. Each line intersects a region boundary exactly once, so the state of any point along the line can be represented with a fixed number of boundary values.

An important special case is $C1=C2$, which is the case we will examine in the remainder of the paper. However, the reader should be aware that the algorithms presented could be generalized to other relationships if necessary. Using this relationship, we can simplify the two dimensional state graphs to one dimension, similar to the single-cache graphs used earlier. Since the graphs are no longer two-dimensional, we revert to a numerical subscript instead of $x$ and $y$ to identify the cache. For the $C1=C2$ line in Figure 5.10, we get the following graphs:

```
1:I----->E---->S----------->O----->  {State in cache one}
         ^     ^            ^
         V1    S1           D1

2:I----------------->S----------->  {State in cache two}
                     ^
                   V2,S2
```

**Figure 5.11**: Configurations using equal cache sizes. $V_i$ are the valid levels, $S_i$ are the sharing levels, and $D_i$ is the dirty level. Taken together they determine the state in any size.

The values of $V_1$, $V_2$, and $D_1$ are identical to the values of $V_x$, $V_y$, and $D_x$ from Figure 5.10. The value of $S_1$, the sharing level in cache one, is the point where the line crosses the E-S boundary, projected onto the x-axis. The value of $S_2$ should be the same, but since this is in the invalid region in cache two, $S_2$ is defined equal to $V_2$.

When $C1=C2$ the space required for metrics is $O(S)$. For any situation there is a single size, $Vi$, above which a Read "hits", so the miss ratio can be accumulated in a linear array per cache. Similarly, a bus transfer occurs for some range of sizes and can be accumulated in a linear array. This extends to require $O(K*S)$ space for $K$ caches.

## 5.7. The MOESI Protocols

We are now ready to analyze the full MOESI protocol, using the facts established in the prior sections. For the case where all caches have the same size we claim that the full class of MOESI protocols, with a few additional restrictions presented below, are one-pass algorithms. Furthermore, they are one-pass algorithms for $K$ processors. As stated earlier, one way to ensure that the state is representable in limited space is if each characteristic that makes up the state obeys an inclusion property. We have seen forms of this in the examples, where validity, ownership, and sharing all appear to possess inclusion. Let us assume that inclusion holds, and define the variables needed to hold the threshold value for each characteristic. We will then show that this assumption is violated in certain cases, but these can be corrected either by restricting the protocol or expanding the state space.

### 5.7.1. State Variables

We define the following variables to maintain the state of each block in each cache.

| | |
|---|---|
| $V_i$ | is the *valid level* (stack distance) of the block in cache $i$, which is the smallest size in which the block is present. |
| $D_i$ | is the *dirty level* of the block in cache $i$, which is the smallest size in which the block is dirty. |
| $S_i$ | is the *sharing level* of the block in cache $i$, which is the smallest size in which the block *may be* shared, i.e. present in another cache. |

```
1:I----->E----->S----->O----->
         ^      ^      ^
         Vi     Si     Di

1:I----->E----->M----->O----->
         ^      ^      ^
         Vi     Di     Si

1:I----->S----->O----->
         ^      ^
       Vi,Si    Di

1:I----->M----->O----->
         ^      ^
       Vi,Di    Si

1:I----->O----->
         ^
      Vi,Si,Di
```

**Figure 5.12**: Possible configurations assuming inclusion. The state in any cache size is determined as a combination of attributes, which change at the indicated threshold sizes.

The state of a block for a particular cache size is determined by comparing the cache size to these levels. For any cache there are three factorial possible configurations for a block based on different orderings of these levels. However, where $S_i$ or $D_i$ is less than $V_i$ we assume they are equal, leaving the five configurations shown in Figure 5.12. In addition there are relationships between states in different caches. For example, if a block is in an exclusive state for some sizes of cache $i$ (i.e. it is valid in sizes where it is not shared, $V_i<S_i$), the block can not be in an exclusive state in any other cache for those sizes ($V_j \geq S_i, j \neq i$). See for example Figure 5.11, where $V_1<S_1$, so $V_2 \geq S_1$. We will show formally that these conditions hold for the MOESI protocol, but first we need to introduce some further restrictions.

### 5.7.2. Further Restrictions

In Section 5.6.1 we restricted the protocol to correct a situation where there were several disjoint cache size regions with the same state. Any time there are several regions with the same state then one of the state attributes does not obey inclusion. Frequently if there can be two regions with the same state then there can be an unbounded number with that state, and therefore the state can not be maintained in bounded space. We will now consider several other cases in the protocols where inclusion is violated, and will introduce some additional restrictions to prevent these anomalies.

### 5.7.2.1. Berkeley Protocol

When the fourth frame of Figure 5.4 is reduced to one dimension, the cache one configuration is:

```
1:I------>S------>M--------->
          <shared><exclusive>
          ^       ^
          V1      D1            S1?
```

**Figure 5.13**: The Berkeley protocol violates sharing inclusion. There is no unique threshold level for $S_1$.

In this configuration the block has the shared attribute for sizes smaller than some threshold, not larger as assumed by the definition of $S_i$. The problem is that the Sweazey/Smith mapping of the Berkeley protocol uses the S state to indicate any unmodified block, regardless of whether the block is actually sharable with other caches. The anomaly stems from rule {S,CC,R}{2} which enters state S on a read miss without testing {CS} to see if the block is actually present in another cache. The situation leading to the configuration above is a Write followed by a Read from the same processor. Since each Write (which is the only action that enters the M state) invalidates the block in all other caches, the block can never be in another cache, and therefore should be in the E state rather than S. Any Read in another cache would cause a {CC} bus request and transform it to an O state.

There would not be an inclusion problem if the protocol used rule {1} since the S region would become an E region, but it would no longer be the same protocol. For example, the Berkeley protocol performs an invalidation {CC,IM} bus request for the S region (even though it may be unnecessary), while the MOESI class never causes a bus action for a Write to the E state.

The S region in the Berkeley protocol example above is therefore one in which the block is exclusively held, but should be treated as a shared block in order to conform to a particular protocol. To solve this problem we introduce a new state called $E^S$; it is an exclusive state which behaves like a shared state. In terms of the state variables, it is in sizes between $V_i$ and $S_i$, but less than $D_i$. This also describes the E state (See Figure 5.14). To distinguish the two states, we also define a new state variable, $E^S$, which if true, indicates that a block is in state $E^S$ instead of E in the region described above.

Now we need to integrate the new state into the transition matrices without affecting the behavior of the system. In any such system we can introduce a new state as long as it is *equivalent* to an existing state. Two states are equivalent if their actions are the same, and the result states are the same or equivalent. Because the tables contain choices of actions, we must also insist that equivalent states always make the same respective choices.

We define the $E^S$ tables entries so that the $E^S$ state is equivalent to the S state. The state is reached by replacing rule {S,CC,R}{2} of Table 5.1 with the equivalent rule {CS:S/$E^S$,CC,R}{2'}. The rule is equivalent because it performs the same actions and the system enters one of two equivalent states, so the performance is unchanged. Next we define the actions for $E^S$ to be equivalent to S. The applicable portions of the revised entries from Tables 5.1 and 5.2 are shown in Table 5.9. Note that some S rules remain in the S state. The corresponding $E^S$ rules transition to S if the respective E rule goes to S; otherwise it remains in state $E^S$. This choice ensures that the $E^S$ states transition to shared or remain exclusive exactly with the E rules do. Thus the effect on the $S_i$ state variable is the same for E and $E^S$ states.

Table 5.9: Revised MOESI protocols: Result State and Bus Signals - Local Requests

| Event: note: From State | Read 1 | Write 2 | Pass 3 | Flush 4 |
|---|---|---|---|---|
| S | S | CS:O/M, {6}<br>CC,IM,BC,W<br>or<br>CS:S/E, {7}<br>CC,IM,BC,W<br>or<br>M,CC,IM {7}<br>or<br>S,IM,BC,W* {8}<br>or<br>S,IM,W* {9} | -- | I |
| $E^s$ | $E^s$ | CS:O/M, {6'}<br>CC,IM,BC,W<br>or<br>CS:S/E, {7'}<br>CC,IM,BC,W<br>or<br>M,CC,IM {8'}<br>or<br>$E^s$,IM,BC,W* {9'}<br>or<br>$E^s$,IM,W* {10'} | -- | I |
| E | E | M | -- | I |
| I | CS:S/E, {1}<br>CC,R<br>or<br>CS:S/$E^s$,CC,R* {2'}<br>or<br>I,R** {3} | M,CC,IM,R {10}<br>or<br>Read>Write {11}<br>or<br>I,IM,BC,W*,** {12}<br>or<br>I,IM,W*,** {13} | | |

Table 5.9: (cont) MOESI Protocols: Result State and Bus Signals - Bus Requests

| Event: note: From State | CC,BC? 5 | CC,IM 6 | BC? 7 | CC,IM,BC 8 | IM 9 | IM,BC 10 |
|---|---|---|---|---|---|---|
| S | S,CS | I | S,CS | S,SL,CS {25}<br>or I {26} | I | S,SL,CS<br>or I |
| $E^S$ | S,CS | I | $E^S$,CS | S,SL,CS {26'}<br>or I {27'} | I | $E^S$,SL,CS<br>or I |
| E | S,CS | I | E,CS? | -- | I | E,SL,CS?<br>or I |

```
E^S false   1:I----->E----->S------>O------>   rule {1} used
E^S true    1:I----->E^S---->S------>O------>  rule {2'} used
                     ^        ^       ^
                     V1       S1      D1
```

**Figure 5.14**: Whether state E or $E^S$ depends on the variable $E^S$. The value is set based on which read rule was used.

Using these rules, the $E^S$ variable, defined above to distinguish the $E^S$ state from the E state, is set if rule {2'} is used, and not set if rule {1} is used. This reinforces the prior restriction that rules must be consistently used, otherwise a single variable is not sufficient. The only time that the $E^S$ variable needs to be reset is if rules {7} or {7'} from Table 5.9 is used. There are no cases where E and $E^S$ states could both occur at the same time.

To be completely general there would have to be an $E^S$ variable for each cache and block. However, if a cache consistently used one rule or the other for all blocks, then a single variable would be sufficient. In the examples of Section 5.8, it is assumed that only the Berkeley protocol uses the revised rule {2'}, so no $E^S$ variable is needed; knowledge of the protocol alone is sufficient to distinguish the $E^S$ and E states.

A similar problem could arise because the definition of the MOESI protocols permits any rule of the form {CS:O/M} to be replaced by {O}. In this case a block that is in the exclusive M state should be treated as being in the O state. A similar introduction of an $M^O$ state could be used if needed to implement this condition.

### 5.7.2.2. Invalidation

Several of the Write rules cause the bus action {CC,IM} which invalidates the block in all other caches. If invalidation rule {M,CC,IM}{5} and write broadcast rule {CS:O/M,CC,IM,BC,W}{6} are used in the same protocol, then both sharing and validity attributes may violate inclusion. We can see this in the first example of Figure 5.15, in which write broadcast is used in sizes where the block is in the S state, but invalidation is used in the larger caches where the block is in the O state. This situation does not occur if the invalidation rules are used for smaller size states, for example if rules {CS:O/M,CC,IM,BC,W}{4} and {M,CC,IM}{7} are mixed, as seen in the second example.

```
Initial     1:I------->S------->O------->
            2:I------------>S----------->

Write 1     1:M------->M--->O-->M------->  Using {5} and {6}
            2:I------------>S-->I------->

Write 1     1:M------->M--->M-->O------->  Using {4} and {7}
            2:I------------>I-->S------->
```

**Figure 5.15**: Violation of inclusion using invalidation rules. Sharing inclusion is violated if rules {5} and {6} are used, but not with {4} and {7}.

A similar problem occurs with the rules that optionally invalidate a slave cache in response to bus requests, such as rules {24} and {28}. The block can be invalidated in larger sizes while still valid in smaller sizes.

We correct both of these by making the restriction that if an invalidation rule is used for the "larger" state (such as the O state) then it must be used for "smaller" states as well. For the Write rules, if the invalidation action {CC,IM} is used for the O state then it must be used for the S and I states as well. This is a reasonable restriction since there is little advantage in determining invalidation action based on the current state. All of the popular protocols obey this restriction. Table 5.10 shows the valid combinations of Write rules under this restriction, along with the names of protocols using each combination. The "Preferred" protocol is taken from the top rules in each block of the MOESI tables.

Table 5.10: Possible Combinations of Write Rules

| Name | Preferred | Dragon | Illinois*† |
|---|---|---|---|
| O | CS:O/M,CC,IM,BC,W {4} | | -- |
| M | M | | M |
| S | CS:O/M, {6}<br>CC,IM,BC,W | | M,CC,IM {7} |
| $E^S$ | CS:O/M,<br>CC,IM,BC,W | | M,CC,IM |
| E | M | | M |
| I | M,CC,IM,R<br>{10} | Read>Write<br>{11} | M,CC,IM,R<br>{10} |

Table 5.10: Possible Combinations of Write Rules (Cont.)

| Name | Berkeley | Write-Thru | Write-Thru |
|---|---|---|---|
| O | M,CC,IM {5} | -- | -- |
| M | M | -- | -- |
| S | M,CC,IM {7} | S,IM,BC,W {8} | S,IM,W {9} |
| $E^S$ | M,CC,IM | $E^S$,IM,BC,W {8'} | $E^S$,IM,W {9'} |
| E | M | (E,IM,BC,W) | (E,IM,W) |
| I | M,CC,IM,R {10} | I,IM,BC,W {12} | I,IM,W {13} |

* Requires rule {20}, eliminating the O state.

† Requires rule {22}.

#### 5.7.2.3. Write-Through Rules

Several rules in Table 5.1 are marked with "*" to be followed by write-through caches. These include Write rules, as well as the Read rule {CS:S/$E^S$,CC,R}{2'}. Although it would be a valid MOESI protocol to choose Read rule {1} in a write-through cache, this would not be sensible, since a write-through cache has no reason to care whether it has the only copy of a block, since it uses write-through regardless. (For this reason the E state is never used in a real write-through cache.) In addition, the use of rule {1} could violate inclusion of the dirty attribute. For example, consider the following sequence.

```
Initial     1:I------------------------------>
            2:I----------------S------------->

Read 1/     1:I----->E--------->S------------->
Later       2:I------------------>S----------->

Write 1     1:I----->M--------->S------------->
            2:I------------------>S----------->
```

**Figure 5.16**: Violation of dirty inclusion using write-through. The Read 1 uses rule {1}, creating an E region. The Write uses write-through rules {8} and {12} for the S and I regions, but the rules never perform write-through to an E state. There is no unique dirty level.

The assumed inclusion property of the dirty attribute is violated after the write. So is the intent of write-through, since the write to the E region is kept in the cache. This problem can be solved in either of two ways. First, we can insist that Read rule {2'} be used by any write-through cache, so the E state can never occur. This is the obvious solution for a real cache, since it always writes the block and therefore does not need to distinguish shared from exclusive states. As a more general solution we can add the rule {E,IM,BC,W} for a Write in the E state, which satisfies the intent of write-through. This was done in Table 5.10 and used in the algorithms of Section 5.8 in order to "decouple" the Read and Write rules.

### 5.7.2.4. Pass/Copy-Back

The Pass rule {CS:S/E,CC,BC?,W}{15} for the O state can cause a violation of inclusion for the sharing attribute, as seen in the following example:

```
Initial     1:I----->E------>S------->O------->
            2:I-------------------------->S--->

Pass 1      1:I----->E------>S------->E-->S--->
            2:I-------------------------->S--->

            1:I----->E------>SPass--->E-->S--->
```

**Figure 5.17**: Pass can violate sharing inclusion. The block is written where dirty, and sharing is verified. Sharing is not verified where the block is clean. The out-of-place S region is referred to as an SPass region.

The problem is caused by the fact that this Pass rule is not just a ''write the block'' rule, but is actually a ''write the block and verify whether any other cache contains it'' rule. However, Pass does not verify sharing for unmodified blocks, so the S region remains, even though we can see in the example that the region should be E. We will refer to this region as an *SPass* region — a portion of the E region left in the S state after a Pass. Note that we could eliminate this problem by verifying sharing for the S region as well. However, in a real cache, this would require a bus action for clean as well as dirty blocks. If the Pass is used as part of a higher-level protocol, for example one that forces a copy-back of all dirty blocks at specific points in time, this new rule could tremendously increase the number of blocks to be handled. We therefore reject this option. The first question is whether there is a need to distinguish the SPass state from the surrounding E states. The answer is clearly yes; a Write would perform no bus action in the E state but would in the SPass (i.e. S) state.

The next question is whether the SPass region occurs a bounded number of times. Unlike some of the anomalies, this one can occur a bounded number of times — at most one time in at most one cache per block. To show this, first recognize that the SPass region is formed from a dirty region, and the block can not be dirty in any cache or size after a Pass. (We will verify shortly that only one cache at a time can be dirty, so no other cache can have a dirty region.) In order for there to be a second SPass region the block must be written, either in the same or another cache. Therefore, consider all possible actions to the configuration of Figure 5.17. Figure 5.18 shows that a Read to cache one preserves the SPass region; all other actions combine the region with the surrounding sizes. Therefore there can be at most one region at any time.

---

```
Initial        1:I----->E------>Spass--->E-->S--->
Read 1         1:E----->E------>Spass--->E-->S--->  {Unchanged}
Write 1        1:M----->M------>M------->M-->O--->  {Gone}
Read 2         1:I----->S------>S------->S-->S--->  {Gone}
Write 2 {11}   1:I----->S------>S------->S-->S--->  {Gone}
                            or
        {10}   1:I------------------------->S--->  {Gone}
```

**Figure 5.18**: Effect of actions on SPass region. Consider each possible action on the initial state. The write to cache two must consider both rules {10} and {11} on a write miss. In each case the SPass region either disappears or remains, but never repeats.

---

To keep track of the region we define a state variable *SPass=(Start,End)* for each block. It is set by the Pass and cleared as shown above. Note that since it only exists in a clean cache that was dirty, in practice we could reuse the dirty fields. However, for clarity, we will describe it as a separate variable in the algorithms that follow.

### 5.7.3. Proof of One-Pass Algorithms

Given these additional changes and restrictions, we can formally prove that the MOESI protocols are one-pass algorithms when all caches are the same size. To do this we must show that each characteristic of the state satisfies a form of inclusion (with the bounded exceptions for $E^S$ and SPass noted above), and that the variables provided are sufficient to maintain the state. We also need to show that the time bound is satisfied, which we will do by example in the algorithms of Section 5.8.

#### 5.7.3.1. Dirty Inclusion

**Theorem 5.1:** If a block is dirty in a cache of size $C$ then it is dirty in the same cache of size $C+1$, and therefore in all larger caches. We refer to this an an inclusion condition; the set of dirty blocks in a cache of size $C+1$ *includes* all dirty blocks in size $C$.

**Proof:** By induction on references.

Basis: The inclusion condition holds before the block is written, since it is clean in all sizes.

Induction: Assume the condition holds at time t. Consider the effect of all possible actions on the dirty state of the block.

a). The only action that converts a block from a clean state (I, S, or E) to a dirty state (M or O) is a Write. For all valid Write rules except for the write-through rules and rule {7}, the block is made dirty for all states, therefore for all sizes. The write-through rules never make the block dirty, and must be consistently used. In all cases the inclusion condition is maintained.

b). The only action that makes a block clean in the cache master is the Pass, which makes it clean for all dirty states, therefore inclusion is maintained.

c). The only bus action that makes a block clean is {CC,IM,BC}, {CC,IM}, or {CC} when rule {20} is used. All these actions make it clean for all dirty states. The result obeys inclusion, since it is clean in all sizes.

□

**Corollary 5.1.1:** For each block and cache $i$, there exists a size $D_i$ such that for all cache sizes $C$, the block is dirty in size $C$ if and only if $C \geq D_i$.

**Proof:** This follows directly from Theorem 5.1, since there must be a minimum dirty size. If the block is clean in all sizes then $D_i = \infty$.

### 5.7.3.2. Validity Inclusion

**Theorem 5.2:** *If a block is valid in a cache of size $C$ then it is valid in the same cache of size $C+1$, and therefore in all larger sizes.*

**Proof:** By induction on references.

Basis: The condition holds before the block is first referenced, since it is invalid in all sizes.

Induction: Assume the condition holds at time t. Consider the effect of all possible actions that affect the validity of the block.

a). The only time when a block becomes valid is in the cache master as a result of a Read or Write request. In both cases, if the block becomes valid it does so for all invalid sizes, and the block remains valid for all valid states/sizes. Therefore the block becomes valid in all sizes.

b). A block may become invalid in three ways: by replacement in a single cache size; by a Flush from all sizes; or because of a bus action. Replacement is assumed to be a stack algorithm which itself obeys inclusion.

c). The only action that invalidates a block in the cache master is a Flush, which invalidates the block for all states and therefore all sizes. Inclusion is preserved.

d). The {CC,IM} bus action invalidates the block for all states in all slave caches. By restriction, the bus action occurs for inclusive sizes. The cache master does not perform the {CC,IM} action for all sizes, and in particular not for exclusive states, but the slave still becomes invalid in all sizes because the block can not be in any slave cache when it is exclusively in the master cache.

e). If a slave cache invalidates itself when it receives a {CC,IM,BC} bus action in the S state (rule {26}), then it must also do so for the O state (rule {24}), by the restriction of Section 5.7.2.2. The bus action occurs in all sizes where the block might be present in another cache, so the block remains valid in an inclusive set of sizes in the slave cache.

f). The {IM,BC} and {IM} bus actions may invalidate a slave cache in all clean states, which by Theorem 5.1 are in smaller sizes than dirty states. By the restriction of Section 5.7.2.2, the slave cache must maintain inclusion in its use of invalidation. These bus actions are the result of a write-through cache and occur for I, E, and S states, which are the only states permitted in a write-through cache. (The E state is not normally present in a write-through cache, but is theoretically allowed as discussed in Section 5.7.2.3.)

☐

**Corollary 5.2.1:** For each block and cache $i$, there exists a size $V_i$ such that for all cache sizes $C$, the block is valid in size $C$ if and only if $C \geq V_i$.

**Proof:** This follows directly from Theorem 5.2, since there must be a minimum valid size. If the block is absent from all sizes then $V_i = \infty$.

### 5.7.3.3. Single Dirty Cache

***Theorem 5.3:*** A block may be dirty (or owned) in only one cache at a time, regardless of cache sizes.

**Proof:** There are two parts to the proof. We first show that there is only one dirty cache for any particular cache size, then show that this extends to all sizes.

The first part is clearly true, otherwise the protocol would not be correct. However, we can easily verify this by examining the possible transitions. Suppose the block is dirty in one cache for a particular size. In order to make it dirty in a second cache, the second processor must Write the block. No other action can change the state to dirty. Consider the state of the block in the second cache before the write:

| | |
|---|---|
| O or M | The block is already dirty, which is a contradiction of the assumption that only one was initially dirty. |
| E | No other cache has the block, which is a contradiction of the assumption that the block is dirty in another cache. |

I or S These are the only possible states in the second cache. Consider the result of all possible write rules. If write-through is used then the result is an unmodified state, either I or S, so again only one cache is dirty. Otherwise there is a bus action {CC,IM,BC} or {CC,IM} which either invalidates the block or makes it clean in all other caches. The result is that the block is only dirty in the written cache.

Now consider the minimum cache size for which the block is dirty in any cache. By the preceding result, there can not be two caches with the block dirty for this size. But by Theorem 5.1, the block is also dirty in this cache for all larger sizes. Therefore the block must be clean for all sizes in all other caches, which completes the proof.

□

**Corollary 5.3.1:** For each block there exists a size $D$ and a dirty cache $DC$ such that $D_{DC}=D$ and $D_j=\infty$ for all $j \neq DC$.

**Proof:** This corollary says that there is no need to keep separate dirty levels, $D_i$, for each cache, since only one can be dirty at a time. This follows directly from Theorem 5.3, where $DC$ is the cache in which the block is dirty. If the block is clean in all caches then $DC$ is arbitrary, since $D=\infty$.

### 5.7.3.4. Sharing Inclusion

**Theorem 5.4:** Except for a possible SPass or $E^S$ region, if a block is sharable in a cache of size $C$ then it is sharable in a cache of size $C+1$, and therefore in all larger sizes. This is known as sharing inclusion.

**Proof:** By induction on reference.

Basis: It is true before the first reference to a block.

Induction: Assume it is true at time t. Consider all actions that affect the sharability of a block:

a) Read: The only state in the cache master where sharability changes on a Read is I. (The I state is defined to be neither shared nor exclusive.) The block becomes E for sizes where the block is not present in any other cache, and S where it is present in some other cache. By Theorem 5.2 there is a valid level, $V_i$, for each other cache. Therefore there must be a smallest $V_i$, which is the smallest size where the block is shared, and therefore there is a single threshold between E and S. In the other caches, the {CC} bus action makes the block shared in all exclusive states, therefore it is shared in all valid sizes, and inclusive.

The only other time that the shared attribute changes in a slave cache is if the slave becomes invalid, either by direction ({CC,IM} action) or by choice ({CC,IM,BC} or {IM,BC} actions). The restrictions of Section 5.7.2.2 ensure that the slave becomes invalid for inclusive sizes. Since the block was shared in inclusive sizes (by the induction hypothesis), and becomes non-shared for inclusive sizes, the shared sizes are still inclusive. Since we have considered all bus actions which could affect sharability in slaves, we will only consider the master cache for the remaining actions.

b) Write: We need to consider each valid combination of write rules, given in Table 5.10.

Dragon

The new state is exclusive in the master for all sizes where it is in an exclusive (E or M) state. For shared states, which are in larger sizes than the exclusive states by the induction hypothesis, the new state depends on the {CS} bus signal. The state is therefore exclusive for all sizes less than the smallest valid level in a slave cache. Since this validity is inclusive by Theorem 5.2, the shared states are inclusive.

Preferred

This is similar to the Dragon protocol, except that all sizes where the block is invalid produce an exclusive state, since all other caches are invalidated.

Berkeley

The new state is always exclusive, therefore the condition holds.

Write-Through

The state is unchanged, therefore the condition remains true by the induction hypothesis.

Illinois

The O state is unreachable using this protocol. For all other states the resulting state is the exclusive M state, therefore the condition holds.

c) Pass: This is the condition that creates *SPass*, which has already been discussed. We have already shown that the region only occurs once.

This completes the proof.

□

**Corollary 5.4.1** For each block, and for each cache $i$ there exists a size $S_i$ such that for all cache sizes $C$, the block is shared in size $C$ if and only if $C \geq S_i$.

**Proof:** Follows directly from Theorem 5.4, since there must be a maximum size where the block is exclusive, if it is ever shared.

### 5.7.3.5. Single Exclusive Cache

**Theorem 5.5:** A block may be in an exclusive state in only one cache at a time.

**Proof:** There are again two parts to the proof. Again the first part is to show that only one cache may have the block in an exclusive state for a given cache size. The second part extends this to all sizes.

The first part could again be assumed from the correctness of the protocols, but it is easy enough to verify. Consider an arbitrary cache size, and assume that some cache has the block in an exclusive state. Consider the possible action/state combinations by which a second cache could attain an exclusive state (E or M) from a non-exclusive state (I,S, or O):

Read/I The new state is E only if there is no cache hit in another cache, which violates the assumption that another cache has the block.

Write/I,O, or S

The new state is M only if the block is not present in another cache at the end of the action either by invalidation or testing {CS}. This insures that only one cache has the block.

Pass/O The new state is M only if the block is not in another cache, which violates the assumption.

To show the second part, we again consider the smallest cache size where some cache has the block in an exclusive state. By Theorem 5.2, the block is present in all larger caches, therefore the block could not be exclusively present in some other cache for any larger size. This completes the proof.

□

**Corollary 5.5.1** If $V_i<S_i$ then $V_j \geq S_i$ for all $j \neq i$.

**Proof:** This is essentially a restatement of Theorem 5.5 using the terms from Corollary 5.4.1. The antecedent implies that cache $i$ has the block exclusively in sizes from $V_i$ to (but not including) $S_i$. Therefore no other cache may have the block present in these sizes.

**Corollary 5.5.2** For each block there exists an $S$ such that $V_i \leq S$ , $V_j \geq S$ for some $i$ and all $j \neq i$.

**Proof:** This again says that individual sharing levels are not needed; a single variable $S$ is sufficient. This follows from Corollary 5.5.1, since there is at most one cache with $V_i<S_i$. Since the other caches can not have an exclusive region, all valid states must be shared. If there is no cache with an exclusive region, then $S=0$, and the Corollary is still true.

### 5.7.4. Variables (Revisited)

These proofs permit us to further refine the variables necessary to maintain the state of a block in all caches:

$V_i$ is the *valid level* (stack distance) of the block in cache $i$. This is maintained by the simulation stack for cache $i$, except as discussed in Section 9.

$DC$ is the identification of the cache that contains the block in a dirty state, or that contains an *SPass* region.

$D$ is the *dirty level* of the block in cache $DC$, which is the smallest size in which the block is dirty.

$S$ is the *sharing level* of the block, which is the smallest size in which the block *may be* shared, i.e. present in more than one cache. Only one cache may have a $V_i$ below $S$.

$SPass$ is a pair $(start, end)$ giving the range of an SPass region in cache $DC$. If there is none, $SPass=(\infty,\infty)$

The maximum space required for $K$ caches of maximum size $S$ is $(K+5)*S$, which is O($S$). It therefore meets the space constraint of a one-pass algorithm.

## 5.8. Formal Algorithms

Given the proof that the MOESI protocols satisfy the space constraint, we could now present algorithms for maintaining the state of the variables for each of the rules in the MOESI action tables. Although this is possible, the results are neither very understandable nor very practical. Instead, we have grouped the rules into common protocols for each request type, and will present the algorithm for each. We will also show that the algorithms satisfy the time constraint of a one-pass algorithm.

Figure 5.19 shows the basic one-pass multi-processor analysis algorithm for the MOESI protocols. The following variables are used in the algorithm.

$N$ = the number of events in the trace.

$X = x_1,x_2, \cdots x_N$, where $x_t$ is the identity of the block referenced at time $t$.

$Action = action_1, action_2, \cdots action_N$, the action at time $t$, which may be Read, Write, Pass, or Flush.

$i$ = the processor making the reference at time $t$, which is the master cache for the action.

$Stack_i$ = the memory stack used by the replacement algorithm for cache $i$.

$Protocol_i$ = the protocol used by processor $i$. Values can be Preferred, Berkeley, Dragon, Illinois, or Write-Through. The protocol variable could also be a function of the block, since a cache does not need to use the same protocol for every block.

Other variables are those described in Section 5.7.4. To simplify the algorithms, all variables are assumed to be global.

```
General One-Pass Algorithm

For $1 \le t \le N$ do                              For all events
  $V_i$ = stack distance of $x_t$ in $Stack_i$
  Locate variables $S, D, DC, Spass$ for $x_t$
  $S_i = max(S, V_i)$                               Share level in cache i
  If $Protocol_i$ = Berkeley then                   Use $E^S$ state?
    $S_i = V_i$                                     All valid are shared
  If $i = DC$ then $D_i = D$                        Find dirty level
    else $D_i = \infty$

  If $D_i < V_i$ then                               The block has been pushed
    Call FlushRoutine($V_i - 1$)                    Update copy-back stats
    $D = V_i$                                       Correct dirty level

  If $i = DC$ then                                  Correct the SPass region
    If $SPass.end < V_i$ then                       If it is invalid
      $SPass = (\infty, \infty)$
    Else If $SPass.start < V_i$ then                Or partially invalid
      $SPass.start = V_i$

  Case of $action_t$:
   Read:                                            Read action.
    Call ReadRoutine()
   Write:                                           Write action.
    Call WriteRoutine()
   Pass:                                            Pass action.
    Call PassRoutine()
   Flush:                                           Flush action.
    Call FlushRoutine($\infty$)

  Update $Stack_i$                                  Update stack
End
```

Figure 5.19: General One-Pass Multi-processor Algorithm.

The algorithm in Figure 5.19 first establishes the values of $V_i$, $D_i$, and $S_i$ for the master cache. If the block is dirty in cache $i$, and the valid level is now larger than the dirty level, then the block must have been written to memory by copy-back for all intervening cache sizes. In a real cache, the write backs happened at some prior time, but since we are interested only in a count of the number of writes we can safely delay the accounting until this next reference. The algorithm calls the Flush routine to update the counts. It also adjusts SPass if the block has been pushed into or beyond the SPass region. The algorithm then calls an appropriate routine for the current action, and updates the memory stack.

Considering the time required for the algorithm, we see that there is only a single outer loop executed $N$ times. Therefore each step can require no more than O($S$) time to meet the overall bound of O($N*S$). The value of $V_i$ can be determined by walking the stack in time O($S$), as can the stack update [Matt70]. Both of these can potentially be done in much less time in certain cases [Olke81 and Chapter 4]. Since there are at most $K*S$ distinct blocks in all caches, the variables for block $x_t$ can be located in time O($K*S$), and probably much less using a hash table, for example. Therefore the time bound of the algorithm is met if each of the subroutines takes no more than O($S$) time.

### 5.8.1. Read Algorithm

The processing necessary for a Read is very simple, but illustrates the logic required to develop an algorithm from the other rules in Tables 5.1 and 5.2. First, consider the cache sizes for which the master performs a memory access or bus request. In this case a read and bus action occur only for sizes where the block is invalid (i.e. for sizes less than $V_i$). The algorithm counts these actions for the sizes in which they occur by passing the size range to the routines Read and Bus.

Next there is a loop to perform the processing of the {CC} bus request in each slave cache, where applicable. The {CC} bus action makes the block shared in any slave cache that has exclusive access. Consider the two cases illustrated below to see the effect this has on the state variables. In the first case, the valid levels in all slave caches are greater that S, meaning that there are sizes where the master cache has the only cached copy, but the state indicates that the block is shared. Because there is no bus action on a cache hit, there is no way for the master cache to know that the sharing level has changed. Therefore it remains at the same level.

In the second case, the valid level of some other cache is below S, implying by Corollary 5.5.1 that $V_1$ is greater than S. The new state of the block in the master cache will be E for all sizes up to the smallest size where some other cache responds with the {CS} signal, which is the minimum of all $V_j\, j \neq 1$. The two cases can be combined by setting S to the minimum of S and all $V_j\, j \neq i$.

```
Case 1:      1:I------->E------->S------->     S < all Vj
             2:I------------------->S---->
             3:I---------------------->S->
                        ^        ^  ^  ^
                        V1       S  V2 V3

Read 1       1:E---------------->S------->     S unchanged
             2:I------------------->S---->
             3:I---------------------->S->
               ^                 ^  ^  ^
               V1                S  V2 V3

Case 2:      1:I------------------->S---->     S > some Vj
             2:I------->E------->S------->
             3:I---------------------->S->
                        ^        ^  ^  ^
                        V2       S  V1 V3

Read 1       1:E------->S---------------->     S moves to Vj
             2:I------->S---------------->
             3:I---------------------->S->
               ^        ^              ^
               V1      V2,S            V3
```

**Figure 5.20**: The effect of Reads on the sharing level. The value of S after a Read depends on the valid level in the other caches.

The algorithm then considers the effect of the {CC} if rule {20} or {18} is used in some slave cache, for example if it uses the Illinois protocol. If the slave cache contains the block in a dirty state then it becomes clean for all sizes where the {CC} occurs. Since these protocols do not permit a block to be in a shared dirty state, the block must be invalid in the master cache whenever it is dirty in a slave, which ensures that the {CC} occurs for all dirty sizes. Therefore the block becomes clean in all sizes, and the algorithm sets $D$ to infinity.

If the cache master is not the dirty cache then a read absorbs any SPass region (See Section

5.7.2.5). In the master cache, the block becomes valid for all sizes ($V_i$=1). The complete algorithm is shown in Figure 5.21. All of the operations take constant time with the exception of determining $V_j$, which takes O($S$) time. Since this is inside a loop that is repeated K-1 times, the time required for the routine is O($K$*$S$), which meets the bound.

```
ReadRoutine()

Read($0 \rightarrow V_i$); Bus($0 \rightarrow V_i$)          State I
For $j \neq i$ do                                  Update other caches
  {CC}
    $S = min(S, V_j)$                              Change exclusive to shared
    If $Protocol_j$= Illinois                      Rule {20} used
      and $j=DC$ then                              and block is dirty in j
        $D=\infty$                                 Block is clean
If $i \neq DC$ then $SPass=(\infty,\infty)$                Read to non-SPass cache
$V_i=1$                                          Block valid in all sizes
End
```

**Figure 5.21**: Read Algorithm.

### 5.8.2. Write Algorithm

Since the write algorithm varies so widely with protocol, the write routine in Figure 5.22 does little more than select the appropriate routine for the write protocol. These are described individually below. The rules for each protocol are those given in Table 5.11.

```
WriteRoutine()

Case $WriteRule_i$:                    Which write protocol
  Preferred:                          Preferred
    Call PreferredRoutine()
  Berkeley:                           Berkeley
    Call BerkeleyRoutine()
  Dragon:                             Dragon
    Call DragonRoutine()
  Illinois:                           Illinois
    Call IllinoisRoutine()
  WriteThrough:                       write-through
    Call WriteThruRoutine()
End
```

**Figure 5.22**: Main Write algorithm.

### 5.8.2.1. "Preferred" Protocol Algorithm

As with the Read routine, we first consider the rules and states for which a memory access or bus action takes place. For this protocol a memory read occurs for invalid sizes (less than $V_i$), and a memory write for all shared sizes ($S_i$ and larger), along with a bus action for each. The notation ($S_i \rightarrow$) means that a write occurs for all sizes $S_i$ and larger. A memory write also occurs for any SPass sizes.

Now consider the effects of bus actions on other caches. For invalid sizes, below $V_i$, the {CC,IM} bus action will invalidate the block in all other caches, so the resulting valid levels are at least $V_i$. In shared regions, above $S_i$, {CC,IM,BC} usually has no affect except to make the block clean in any dirty cache. There is no reason to do this directly in the algorithm since it happens indirectly when DC is changed later. However optional invalidation rules may affect the valid levels in slave caches as well.

Because the S and O rules contain {CS:O/M}, the algorithm must adjust $S$ as was done in the Read routine. Finally, the block is made dirty and valid in all sizes of cache $i$. The complete algorithm is shown in Figure 5.23. Again time is $O(K*S)$ based on the need to determine $V_j$.

### 5.8.2.2. Berkeley Protocol Algorithm

The Berkeley protocol performs a memory read for invalid sizes, a bus action for I, $E^S$ and S states (below $D_i$) and the O state (above $S_i$ and $D_i$). For all non-exclusive states (I, S, $E^S$, and O), the {CC,IM} bus action invalidates the block in all other caches. Since these include all sizes where the block might be present in another cache, the algorithm simply sets $V_j$ to infinity. Because the block is now exclusively in cache $i$, the sharing level is also infinite. The remainder of the processing is similar to the Preferred routine above. The time depends on the time required to invalidate the block in other caches. This can certainly be done in $O(S)$ time, and probably in constant time. Therefore the time bound is met.

```
PreferredRoutine()
Read($0 \rightarrow V_i$); Bus($0 \rightarrow V_i$);                 I state
Write($S_i \rightarrow$); Bus($S_i \rightarrow$ )                    O, S & $E^S$ states
Write($SPass.start \rightarrow SPass.end$)                            SPass region
Bus($SPass.start \rightarrow SPass.end$)
For $j \neq i$ do                                                     Update other caches
    {CC,IM}                                                           I state
        $V_j = max(V_i, V_j)$                                         Invalidate for all I sizes
    {CC,IM,BC}                                                        S & O states
        If S:I used in $j$ then                                       Optional invalidation rules
            $V_j = max(D_j, V_j)$                                     Invalidate to top of S
        If O:I used in $j$ then
            $V_j = \infty$                                            Invalidate in all sizes
$S = min(S, V_j, j \neq i)$                                           Shared in valid sizes
$V_i = 1$                                                             Valid in all sizes
$D = 1; DC = i$                                                       Dirty cache is i
$SPass = (\infty, \infty)$                                            SPass vanishes
End
```

**Figure 5.23**: Write algorithm for Preferred protocol.

```
BerkeleyRoutine()
Read($0 \rightarrow V_i$);                                           I state
Bus($0 \rightarrow D_i$ )                                             I & S states
Bus($max(D_i, S_i) \rightarrow$ )                                     O state
For $j \neq i$ do                                                     Update other caches
    {CC,IM}                                                           O,S,I states
        $V_j = \infty$                                                Invalidate for all I sizes
$SPass = (\infty, \infty)$                                            SPass vanishes
$S = \infty$                                                          Shared in valid sizes (none)
$V_i = 1$                                                             Valid in all sizes
$D = 1; DC = i$                                                       Dirty cache is i
End
```

**Figure 5.24**: Write algorithm for Berkeley protocol.

### 5.8.2.3. Dragon Protocol Algorithm

The rules for this protocol are similar to the Preferred protocol, with the exception that a Write miss is implemented as a Read, to make the block valid in all sizes, followed by the Write. The algorithm is therefore similar to the Preferred algorithm, without the {CC,IM} processing. Since the Read routine does not affect anything except the invalid sizes, we call it directly to implement the {Read>Write}. The resulting algorithm is shown in Figure 5.25. Time required is O($K$*$S$).

```
DragonRoutine()
Call ReadRoutine()                                  I state
Write($S_i \rightarrow$); Bus($S_i \rightarrow$ )   O & S states
Write($SPass.start \rightarrow SPass.end$)          SPass region
Bus($SPass.start \rightarrow SPass.end$)
For $j \neq i$ do                                   Update other caches
  {CC,IM,BC}                                        S & O states
    If S:I used in $j$ then                         Optional invalidation rules
      $V_j = max(D_j, V_j)$                         Invalidate to top of S
    If O:I used in $j$ then
      $V_j = \infty$                                Invalidate in all sizes
$S = min(S, V_j, j \neq i)$                         Shared in valid sizes
$D = 1; DC = i$                                     Dirty cache is i
$SPass = (\infty, \infty)$                          SPass vanishes
End
```

**Figure 5.25**: Write algorithm for Dragon protocol.

### 5.8.2.4. Illinois Protocol

The Illinois protocol uses invalidation and exclusive modified states. Thus it invalidates in all sizes. It also uses rule {22} to write the block back to main memory when a modified cache sees the {CC,IM} bus request. Because of the exclusive modified states, if the block is dirty in a slave cache then it must be invalid in all sizes of the master cache. Thus the {CC,IM} processing below makes the block clean in all sizes. (The operation is superfluous, but is included as a reminder that a main-memory access is required.) The algorithm presented below is applicable if all caches use the Illinois protocol. If the slave and master caches could use different protocols then each write routine that uses the {CC,IM} bus action should include the code that tests for a slave using the Illinois protocol and makes the block clean. The total time required is $O(K^*S)$.

```
IllinoisRoutine()
Read(0→$V_i$); Bus(0→$V_i$);          I state
Bus($S_i$→ )                          S state
For $j \neq i$ do                     Update other caches
  {CC,IM}                             I state
    $V_j=\infty$                      Invalidate for I or S sizes
    If $WriteRule_j$= Illinois        Protocol used by slave
      and $j=DC$ then                 Cache has dirty block
        $D=\infty$                    Block is written back
$V_i=1$                               Valid in all sizes
$D=1$;$DC=i$                          Dirty cache is i
End
```

**Figure 5.26**: Write algorithm for Illinois protocol.

### 5.8.2.5. Write-Through Protocol Algorithm

This protocol is particularly simple, since a memory write and bus action occur for all states and sizes. The {IM,BC} bus action may invalidate the block in other caches.

```
WriteThruRoutine()
Write(0→); Bus(0→)                    I & S states
For $j \neq i$ do                     Update other caches
  {IM,BC}                             S,I states
    If E:I used in $j$ then           Optional invalidation rules
      $V_j=min(S,D_j)$                Invalidate to top of S
    If S:I used in $j$ then
      $V_j=D_j$                       Invalidate in clean sizes
$SPass=(\infty,\infty)$               SPass vanishes
End
```

**Figure 5.27**: Write algorithm for Write-Through protocol.

### 5.8.3. Pass Algorithm

There are two event types in Table 5.9 that have not been mentioned much — the Pass and Flush. The Pass event is used to copy-back a dirty block, but retain a copy, and so is applicable only to the dirty cache. Figure 5.28 defines an algorithm for simulating a Pass. A memory write and bus action occurs for all dirty sizes in cache $i$, above $D_i$. As described in Section 5.7.2.5, an SPass region is formed in the current S region, bounded by $S_i$ and $D_i$. Then the sharing level is adjusted because of the {CS:S/E} rule to be the smallest valid level in any other cache. Finally, the dirty level is infinite, making the block clean in all sizes. The variable $DC$ is unchanged, identifying the cache containing an SPass region. Time is O($K*S$) because of the need to determine $V_j$.

```
PassRoutine()

If $i=DC$                                              Only applicable to dirty cache
    Write($D_i \rightarrow$); Bus($D_i \rightarrow$)   M & O states
    $SPass=(S_i, D_i)$                                 SPass region
    $S=min(V_j, j \neq i)$
    $D=\infty$                                         No longer dirty
End
```

**Figure 5.28**: Pass algorithm.

### 5.8.4. Flush Algorithm

Another event from Table 5.9, the Flush, is used to purge a block from cache, after writing it to memory if necessary. Figure 5.29 defines an algorithm for simulating a Flush. It can be used in two ways, either as a direct action to invalidate the block in all sizes, or when the block is invalidated through a push by the replacement algorithm. To distinguish the cases, there is a parameter giving the upper bound on the invalidation. By restriction, the block becomes invalid in all smaller sizes. If the block is dirty, a write and bus action occurs above $D$. The block then becomes clean and invalid in all flushed sizes. Constant time is required.

```
FlushRoutine(high)

If $i=DC$ then                      If the block is dirty
    Write($D \rightarrow high$);    Write in dirty region
    $D=high$                        No longer dirty
$V_i=high$
End
```

**Figure 5.29**: Flush algorithm.

## 5.9. Other Protocols

### 5.9.1. Firefly Protocol

The Firefly protocol as defined in Table 5.11 is almost a MOESI protocol, differing only by the rule {CS:S/E,CC,IM,BC,W} which is not in the MOESI transition Tables 5.1 and 5.2. The Firefly protocol uses write broadcast and assumes that it updates main memory along with all shared caches. Therefore the master cache remains unmodified, and there is no shared modified O state. It also uses a Read followed by a Write (Read>Write) for a write miss.

Table 5.11: Firefly Protocol

| Action: Current State | Read | Write | CC | CC,IM,BC |
|---|---|---|---|---|
| M | M | M | S,CS,DI | -- |
| E | E | M | S,CS | I |
| S | S | CS:S/E, CC,IM,BC,W | S,CS | S,SL,CS |
| I | CS:S/E, CC,R | Read>Write | I | I |

It can easily be shown that its attributes do not always obey the inclusion assumptions of the state variables. Consider the sequence shown in Figure 5.30. The second Write uses write-broadcast for the larger shared sizes, resulting in a "clean" block, but makes the block dirty in the smaller exclusive sizes. The result is that there is no value for $D_1$ that satisfies inclusion (i.e. dirty for all sizes $D_1$ and greater).

```
Write 2/        1:I--------------------------------->
 Later          2:I---------------->M--------------->

Read 1/         1:E---------------->S--------------->
 Later          2:I---------------------->S--------->

Write 1         1:M---------------->E--->S--------->
                2:I---------------------->S--------->
                  ^                 ^     ^
                 V1,D               U    S,V2
```

**Figure 5.30**: Violation of the dirty level caused by Firefly protocol. The new variable $U_1$ shows the upper limit of the dirty region.

The solution to this problem is to define a new state variable, $U_i$, to contain the upper-bound of the dirty region (or the level at which the block becomes *unmodified*). Note that $U_i$ can only be in the dirty cache, and must be less than $S_i$ since there is no O state. To ensure that there is only one dirty region ($D_i$ to $U_i$), consider all possible actions applied to the first state of Figure 5.31. In each case the dirty region moves or disappears, but never repeats.

It is interesting to see that the bound $U_i$ has no real effect on the performance of the system. This is because the region is always exclusive, and the exclusive states E and M never cause misses or bus actions. The only difference in their effect, either in Table 5.11 or Tables 5.1 and 5.2, is the additional {DI} or {DI,DK} signals for the M state. These could affect real performance, since data is obtained from cache rather than main memory. It has no affect in the algorithm below because the algorithm simply counts bus actions.

```
From Figure 5.30 1:I-------->M------>E--->S--------->
 /Later          2:I------------------------>S------>
                             ^       ^    ^  ^

                           V1,D      U    S  V2

Read 1           1:E-------->M------>E--->S--------->
                 2:I------------------------>S------>
                   ^         ^       ^    ^  ^

                  V1         D       U    S  V2

Write 1          1:M-------->M------>M--->E->S------>
                 2:I------------------------>S------>
                   ^                      ^  ^

                  V1,D                    U  V2,S

Read 2           1:I-------->S------>S--->S--------->
                 2:E-------->S-------------->S------>
                   ^         ^

                  V2        V1,S                     D=U=∞

Write 2          1:I-------->S------>S--->S--------->
                 2:M-------->S-------------->S------>
                   ^         ^

                  V2,D      V1,S,U
```

**Figure 5.31**: There is a single dirty region for the Firefly protocol. All actions lead to a configuration of states where the dirty region does not repeat.

Since the Firefly protocol does not use the O state, it uses rule {S,CS,DI}{20} instead of {O,CS,DI,DK}{19} for a {CC} action to the M state. The other choice of rules could lead to situations where there are an unbounded number of dirty regions. Therefore any cache using the Firefly protocol must use rule {20}.

A final problem occurs if a Firefly cache is on the same bus with a write-through cache using rule {I,IM,W}{13}. From Table 5.2 note that the {IM} bus request invalidates unmodified blocks but retains modified ones. In the Firefly cache the block may be dirty in small sizes and clean in larger ones. This implies invalidation in larger sizes, violating the inclusion of the valid level. Therefore we must prohibit the mixture of Firefly and write-through caches in order to make this a one-pass algorithm.

The Read algorithm for the Firefly protocol is the same as given before. The write algorithm is given below. Firefly uses a Read>Write, which again is simply a call of the Read algorithm since only the invalid states will be changed. The {CC,IM,BC} requests in the S state have no effect in other caches unless the optional invalidation rules are used. Time is O($K*S$).

```
FireflyRoutine()
Call ReadRoutine();                          I state
Write($S_i \rightarrow$); Bus($S_i \rightarrow$ )        S state
$U=max(S,V_i)$                               Block is clean where shared
For $j \neq i$ do                            Update other caches
   {CC,IM,BC}                                S state
      If S:I used in $j$ then                Optional invalidation rules
         $V_j=max(D_j,V_j)$                  Invalidate to top of S
      If O:I used in $j$ then
         $V_j=\infty$                        Invalidate in all sizes
$S=min(S,V_j,j \neq i)$
$D=1$;
$DC=i$                                       Dirty cache is i
End
```

**Figure 5.32:** Write algorithm for Firefly protocol.

### 5.9.2. Write Once

The earliest cache consistency protocols was the Write Once protocol [Good83], so called because it uses a write through on the first write from a shared state to notify other caches to invalidate the block. Sweazey and Smith describe a version of Write Once as an extension to the MOESI protocols. The one rule which is not part of the MOESI class defined in Tables 5.9 and 5.10 is the rule {E,CC,IM,W} for a Write in the S state, which performs the write-through and invalidation. In addition, rule {S,CS,DI}{20} is used for the {CC} bus action to perform a copy-back to update memory when another cache reads the block. Thus, any modified state is an exclusive state.

Table 5.12: Write Once Protocol

| Action: Current State | Read | Write | CC | CC,IM |
|---|---|---|---|---|
| M | M | M | S,CS,DI | I,DI,DK<br>or<br>I,DI |
| E | E | M | S,CS | I |
| S | S | E,CC,IM,W | S,CS | I |
| I | S,CC,R | M,CC,IM,R<br>or<br>Read>Write | I | I |

This protocol can easily be shown not to be a one-pass algorithm using the following sequence. Consider only a single cache, with the block invalid in all other caches. As in other examples the result of each action is shown some time after the action, after the block has been pushed from some sizes.

```
Read          :I------------------------->S----->
Write         :I------------------->M---->E----->
Read          :I-------------->S---->M---->E----->
Write         :I-------->M---->E---->M---->E----->
```

**Figure 5.33**: Write-once violates dirty inclusion.

Repeated sequences of Read followed by Write can give an arbitrary number of distinct regions of M and E. The problem is that, although E is a valid state, a read miss does not test {CS} to distinguish shared from exclusive states, resulting in the violation of sharing and dirty inclusion. The $E^S$ state does not help in this case because the protocol does distinguish and act differently for the E and S states. One possible solution is to use the protocol as described, but restricted to the rule {Read>Write}. This takes two separate actions for a write, however. The solution we propose is to use Read rule {1}, {CS:S/E,CC,R}, which takes advantage of the {CS} information available on read.

This only solves one inclusion problem with this protocol. Consider the following situation, where the initial configuration is obtainable by a Read to processor 2 followed by a Read to processor 1.

```
Initial     1: I---------->E------->S---------->
            2: I---------------------->S------->

Write 1     1: M------------------->E---------->
            2: I------------------------------->
                              or
Write 2     1: I------------------------------->
            2: M---------------------->E------->
```

**Figure 5.34**: Write-once violates dirty inclusion.

Clearly this violates the inclusion of dirty states. The problem now is that the result state is dirty where it was invalid (in small sizes) and clean for larger S sizes. The first question is whether there can again be an arbitrary number of such regions. In this case we argue that the E region is anomalous, but it can not repeat. It is formed from an S region, which is only formed when the other cache reads the block. However, the {CC} bus action from the Read will make the block clean, restoring inclusion. Therefore the region occurs at most once, and only in the cache with dirty states.

To solve this problem we define a new state $M^E$, an exclusive state which "should be" dirty, but isn't. After a Write, we let the dirty level be 1, but keep track of the $M^E$ level at which the block becomes clean again. It turns out, however, that the value of $M^E$ is immaterial; the only difference in the handling of M and E states are in the {DI} or {DI,DK} signals generated on bus requests. The algorithm as coded does not count these separately since there is already a bus action at these times. The algorithm uses the $M^E$ only as a reminder of the anomaly. The resulting Read and Write routines are given in Figure 5.35.

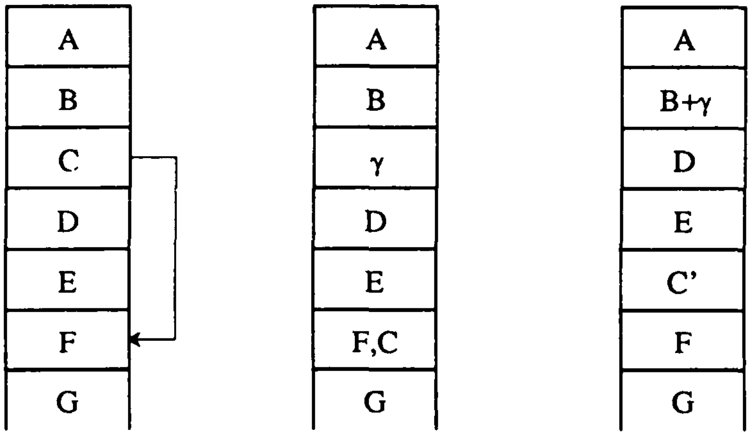
```
ReadRoutine()

Read($0 \rightarrow V_i$); Bus($0 \rightarrow V_i$)          State I
For $j \neq i$ do                                  Update other caches
   {CC}
    $S = min(S, V_j)$                             Change exclusive to shared
    If $j = DC$ then
       $D = \infty$                                 Block is copied back
       $M^E = \infty$
$V_i = 1$                                         Block valid in all sizes
End


WriteRoutine()

Read($0 \rightarrow V_i$); Bus($0 \rightarrow V_i$);          I state
Write($S_i \rightarrow$); Bus($S_i \rightarrow$ )              S state
For $j \neq i$ do                                  Update other caches
   {CC,IM}                                        I state
      $V_j = \infty$                                Invalidate for I or S sizes
$M^E = max(V_i, S)$                               Bottom of S region
$S = \infty$                                      Invalid in all others
$V_i = 1$                                         Valid in all sizes
$D = 1; DC = i$                                   Dirty cache is i
End
```

**Figure 5.35**: Write-Once protocol algorithms.

## 5.10. Implementation Issues - Partial Invalidation

In all of the common consistency protocols which use invalidation, the {CC,IM} bus request is sent for all shared states of the cache master. This ensures that the block is invalidated for all cache sizes in each slave, since it doesn't exist in any other cache when the master is in an exclusive state. Invalidation in all sizes is easily implemented in the simulation, as discussed by Olken [Olke81] and in Chapter 4. The invalidated block is removed from the simulation stack and is replaced by a "gap" to maintain the stack distance of all lower blocks. The gap is absorbed by the first push from above.

In the full suite of MOESI protocols, however, we have permitted combinations of invalidation with write broadcast, {CC,IM,BC,W}, thus permitting invalidation for only some sizes. The Preferred protocol is an example where this occurs. This situation can no longer be implemented using just a gap. For example, consider the stack shown in Figure 5.36(a). Suppose that block C is invalidated in sizes smaller than 6. We would like to move block C to level 6 in the stack, leaving a gap at level 3, as shown in Figure 5.36(b). To do this directly would require the creation of a gap in the stack and some mechanism for two blocks to have the same stack level.



Invalidate C to 6 (a)

Desired Stack (b)

Practical Stack (c)

**Figure 5.36**: Implementation of partial invalidation. Block C is to be invalidated in all caches smaller than six. This requires that both blocks F and C reside at stack level six, and that a gap be inserted at level three.

One way to do this is to keep a count of the gaps adjacent to each block, as suggested by Olken [Olke81]. We can then move block C to level 6 in the stack and mark it such that it will not be counted in the stack distance of blocks below it. Therefore block F also has stack distance 6. This is shown in Figure 5.36(c). Since block C has a higher LRU priority than block F in the caches where it is valid, it is important that it remain higher in the stack than F. This method if fairly simple, but unfortunately is difficult to do using the tree-based methods of determining the stack distance [Olke81]. The effect of this should be minimal, since a single processor in a multi-processor system should typically execute a single program for a long time: chapter 4 showed that a simple linked list works as well as a tree implementation for single program address traces.

### 5.11. Conclusions

In this chapter we have shown that an important class of cache consistency protocols are one-pass algorithms, permitting the performance of all sizes of cache to be computed efficiently in a single trace-driven simulation. The MOESI class of protocols is applicable to multiprocessor caching systems communicating over a backplane bus, and includes many of the common protocols for maintaining cache consistency in this environment. The fact that multiple caches can be efficiently simulated, not just in a few cases but for a wide range of consistency protocol, is an exciting new result which greatly expands the range of design options which can be easily explored. In the next chapter we show that a similar result holds for network file systems.

# Chapter 6

# A Class of File System Consistency Protocols

## 6.1. Summary

The efficient use of high-performance workstations with a central file server demands the caching of files at each client workstation. This introduces a cache consistency problem — how to ensure that all clients see the same view of the file system. In this chapter we introduce a class of protocols that provide consistency while permitting concurrent readers and writers. The class includes several alternatives that may be chosen independently for any file and client. We discuss the tradeoffs involved in these choices by describing four specific protocols from the class: a simple protocol, one optimized for private files, one optimized for read-only files, and one for volatile files. We also show that, with few restrictions, the protocols can be efficiently simulated using one-pass techniques.

## 6.2. Introduction

Building on the discussion of Chapter 5, this chapter presents a class of consistency protocols for use by a network file system. Archibald et al. [Arch86] showed that the cache consistency protocol makes a non-trivial difference in performance of multi-processor systems. We believe, and Chapter 7 confirms, that this is the case for file systems as well. Formally defining the consistency protocols in this chapter achieves four things: 1) it defines protocols that provide similar consistency assurances, thus improving their comparability; 2) the enumeration of possibilities makes it easier to verify that all reasonable alternatives are considered; 3) formality assists in the correct design of the simulation; and 4) we assure ourselves (and the reader) that the protocols *are* efficient one-pass algorithms.

The file system envisioned for these protocols is one commonly used to support high-performance workstations, and consists of three components: the file server, file system clients (i.e. the workstations), and a network connecting them. The server is the main repository for all data, as well as providing other file services discussed in a moment. The client workstations each contain a local file system representative which responds to requests from local processes, caches data, and forwards requests to the server when necessary. The interconnection is assumed to be a local area network, but is not restricted to a particular media or technology. It therefore might be a broadcast bus, such as an ethernet, or a token ring network.

These components are roughly equivalent in function to the main memory, client caches, and backplane bus supported by the MOESI protocols — with some important differences. First, higher-level operations on files, such as open and close, often impose additional load on the server. Second, the lack of a reliable broadcast media such as the backplane bus, leads to further server participation to maintain global state. Finally, consistency based on open/close provides opportunities to reduce this load, but introduces a new consistency problem — concurrent update.

We discuss each of these differences in the following section. Section 6.4 then reviews several existing file system consistency protocols. Section 6.5 defines the class of consistency protocols, beginning with a discussion of the basic approach, the attributes comprising the state, and the actions and transitions for both clients and servers. The next section presents four specific protocols from the class and discusses tradeoffs between them. Finally, Section 6.7 shows that the class constitutes one-pass algorithms.

## 6.3. Differences Between File System and Processor Caches

There are three basic differences between file system caches and processor caches that affect the design of consistency protocols for the former. First, the blocks of a file system are organized into higher-level objects (i.e. files) which are not present in processor caches. Additional file-level requests, such as Open, Close, and Delete, may affect cache consistency. In particular, the Open performs three functions: binding a file name to a specific file object, specifying a type of access (read or write), and verifying authority to access the file. These are usually done by the server — at least for the first access — and impose server load over and above that of simply providing data. At the same time, the Open offers opportunities to reduce consistency overhead by performing consistency controls for all blocks of a file at one time.

A second difference between multi-processor and network file systems is in the nature of the interconnection. A backplane bus provides synchronized, reliable delivery of messages to all connected caches. Only one client can use the bus at a time, therefore all clients see each request simultaneously and act on it before acting on any other requests (whether bus or local requests). Further, the cache master *knows* that all other caches have seen the request (because of an acknowledgement signal, {AI}, not discussed in Chapter 5). Further, it knows that any "interested" cache will respond, either with the {CS} status or the data, as appropriate. Together these assurances permit the consistent, decentralized maintenance of state in each client.

Contrast this with a network file system, even one based on a broadcast-capable local area network. Although the network may permit all clients to see a broadcast, and only one client to access the network at a time, the medium is not guaranteed to be reliable. Acknowledgements are needed to ensure that all clients act in the same order, but there is no "cheap" acknowledgement mechanism (i.e. no {AI} line). Protocols exist for broadcast media such as ethernet requiring far less than one acknowledgement per client, but they still impose a heavy burden on the network and each client [Chan84]. A token ring can provide assurances that each other client is "alive" by setting a bit as the message passes. However the client file system is rarely involved in this acknowledgement so this is no assurance that the file system has acted on the request. Without hardware support to check the cache contents, the interface can not supply important feedback such as "I have the block" ({CS} signal) or "I have a modified copy of the block" ({DI} signal). Therefore both network media require either a costly reliable broadcast protocol, or the centralized maintenance of the state. The latter is a common choice, using the server as a centralized control point.

A third difference between the multi-processor bus and a network file system is the location of the most likely system bottleneck. A multi-processor system is likely to be limited by bus throughput as the number of processors increases [Arch86], while current single-server network file systems are more likely to be server-limited [Lazo86, Rama86]. (However, with increases in processor power and file server efficiency, file systems may also become limited by network bandwidth [Nels87].) In addition, the penalty from a cache miss may be higher in the file system case. Nevertheless, the conclusion is the same for both systems — the protocols should attempt to satisfy as many requests as possible within cache.

As we will see in Chapter 7, caching can be as effective at avoiding open/close requests as read/write requests. In order to avoid opens the client must retain enough information to satisfy all three functions of the open: name resolution, consistency control, and access control. There are obvious cases where the server need not be contacted for consistency control, such as when the file is already open. The protocols in Section 6.5 identify many others. A name cache is required to avoid the name resolution function. This is easy in principle, but needs to consider the additional burden of maintaining the consistency of names as well as data. Access rights can be similarly cached. Revocation should also be provided, but revoking access to data already cached in the client is problematic (i.e. once the data is present in the client it is nearly impossible to effectively deny access to it). Name and access-rights caching are assumed to be possible, and are not discussed further.

### 6.4. Concurrent Access

The higher-level open and close operations offer the possibility of performing consistency controls on the entire file, rather that individual blocks as was done for processor caches. This has the advantage of reducing the amount of global state that the server is required to maintain. Also, since files are often read or written in their entirety [Oust85, Floy86a], the cost of consistency control can be amortized over several reads or writes. For example, if a cached copy is to be invalidated in some cache because of a write to another cache, the entire file can be invalidated in a single action rather than waiting for each block to be written.

Consistency controls based on open/close introduces a problem which does not exist with a backplane bus — concurrent access. The purpose of consistency controls are to coordinate sharing, and specifically write sharing. There is clearly no consistency problem with shared files when all caches access the file for read only. There are two types of write sharing. The first is *sequential sharing* where there are multiple users and at least one writer, but not simultaneously.

This is the type of sharing on a multi-processor bus, since the bus prohibits simultaneous requests. The need here is to ensure that subsequent readers get the most recent copy. The second type is *concurrent write sharing*, where multiple clients have the file open at the same time and at least one is writing.

There are several possible solutions to concurrent write sharing. One is simply to disallow it; if a request would result in concurrent sharing then the request is blocked until the concurrent reader or writer has finished. This classic readers/writers locking is common in some file systems, but introduces the potential for deadlock [Eswa76].

Another way to prevent concurrent write sharing is to use *immutable files* where each write open implicitly creates a new *version* of the file while concurrent readers continue to access the current version. The temporary inconsistency and possible additional space are tolerable for most files, with the notable exception of databases and large append-mostly log files where update-in-place is more practical. Note however that consistency controls are still needed to ensure that subsequent opens from other clients see the new version. There is also a risk of losing updates if there are concurrent writers.

The UNIX operating system allows concurrent readers and writers. Consistency is maintained by performing all file I/O through a common *buffer cache* [Thom78]. Thus a write is immediately available in the cache for the next reader. Actually, this only guarantees consistency at the operating system interface. Since most programs use standard file access routines (*stdio*) which buffer data in user space, consistency is not guaranteed at the programming interface. For example, when a program writes a word it is buffered in the program until the buffer is flushed to cache, either by a full buffer, a close, or an explicit flush. The cache continues to show the "old" value. Similarly, even after the buffer is flushed, other programs may continue to have the old value buffered. (A similar problem occurs in multi-processor caches when variables are buffered in registers — something which can occur inadvertently due to compiler optimization.) The consistency visible to the user program is actually something like:

> A reader will *probably* get the value written by the most recent write, and if all writes stop, *eventually* all readers will see the same value.

This definition assures a form of consistency, but with no guarantee of synchronization. If programs need a stronger guarantee of real-time consistency (i.e. really reading the last value written) then they must use explicit synchronization. UNIX provides optional file locks which may be used for this purpose.

A simple solution for a network file system is to prohibit more than a single cached copy when there is concurrent write sharing. All clients access this single copy using Remote Procedure Calls (RPC) [Birr84], for example. This is essentially the mechanism used in the Sprite network file system [Nels87]. Like single-processor UNIX, this does not really ensure consistency at the user program level. In addition to *stdio* buffering, there are communication delays and variable delays in each client which prevent real-time consistency. However, it does meet the loose definition of consistency defined above. Again, explicit synchronization should be used if real-time guarantees are needed.

Another possible solution is to use a write broadcast protocol, similar to several in the MOESI class. If there is an underlying reliable broadcast protocol that guarantees that all clients eventually see all messages, in the same order, then the loose definition of consistency is assured. Further, it is assured even if there are concurrent writers since the reliable broadcast serializes the individual writes.

The obvious way to achieve reliable broadcast is to require an acknowledgement for each message from each client. Chang and Maxenduk have presented a reliable broadcast protocol that assures reliable ordered delivery and requires far fewer than one acknowledgement per client (on the order of 3-4 overhead messages per broadcast message, for 10-30 clients) [Chan84]. The protocol requires storage in each client proportional to the number of broadcast participants, and introduces a delay before a client can act on a request. This is significant overhead if all messages are sent using reliable broadcast. However, it is perhaps acceptable if confined to just broadcast writes.

A different solution that permits concurrent access, including multiple writers, is the Caching Ring proposed by Kent [Kent86]. He proposes a token ring network with special "snooping" interface hardware that maintains a directory of cached file system blocks. Since the interface sees all messages in real time it can reliably maintain the state just like a multi-processor cache with a backplane bus. His proposal assures consistency by invalidating the cache when a write from another cache is seen, but it could be extended to perform write update. The overhead on a token-ring network is minimal, but it requires special hardware and the delay increases with the number of clients.

### 6.5. Existing Schemes

Existing network file systems use a variety of approaches to consistency control, some of which are reviewed below. A survey of some earlier network file systems is presented in [Svob84].

#### 6.5.1. Cedar

The Cedar file system caches whole files on local disk [Schr85]. Files are shared by copying them to the file server; once there the file is *immutable*. Each such copy atomically creates a new version of the file. Any concurrent readers continue to use the version that was copied to their local disk. An open can specify any existing version, or can request the "latest" version. This ensures consistency at the granularity of the open/close (i.e. the client can be assured that a version is current at the time of an open, although the version may become outdated while it is open). Immutability and full file caching simplify the consistency mechanism and eliminate the need for any locks or long-term state in the server.

#### 6.5.2. Andrew

Andrew is a distributed system being developed by Carnegie-Mellon University. The Andrew file system also caches whole files on local disks [Saty85, Morr86]. The first implementation used a polling scheme which contacted the server on every open to verify that the cached version was correct. They found that this polling accounted for much of the server traffic, and that the cached version was correct most of the time. The current implementation uses a *call-back* approach where the server notifies each caching site when a file is written. A file is written back to the server on close so that other clients get the new copy on their next open. Andrew also supports read-only file systems that never require verification of cached versions. The consistency guarantees are similar to Cedar.

### 6.5.3. SUN Network File System (NFS)

The SUN NFS caches in memory at both the client and the server, and is designed primarily to support diskless workstations [Sand85]. The server is *stateless*, maintaining no information about open files, which makes consistency control very difficult. However a stateless file server has the advantage of simplifying recovery since the server can fail and recover without affecting the client state. Each client "trusts" its cached data for three seconds. After that, the client verifies the file-update time (a form of file version information) with the server and flushes cached blocks if the file has changed. Blocks are written through (actually write-behind) to the server. This technique gives a high probability of consistency to sequential sharing, but no assurances for concurrent write sharing. Users are warned to avoid it. UNIX advisory file locks are provided if more control is needed.

### 6.5.4. Sprite

Sprite is an operating system being developed at UC Berkeley for a workstation environment [Nels87]. The Sprite file system also caches at both clients and servers, and provides consistency control for concurrent writers. All opens are sent to the server, which detects an out-of-date version due to sequential sharing, or concurrent write sharing. In the latter case, all cached copies are invalidated and caching is permitted only in the server as long as the condition prevails. During this time all reads and writes are sent to the server using the Sprite remote procedure call (RPC). The consistency assurance is the same as provided by UNIX, although the write is delayed by the time of RPC. The method is costly when a concurrent write occurs, both in terms of latency and load on the server, but this is assumed to be a rare event.

## 6.6. A Class of Consistency Control Protocols

This section describes and formally defines a class of consistency protocols that permit concurrent write sharing. As with the MOESI class, the class is defined as a transition matrix giving the actions required to respond to a request and the resulting new state, depending on the current file state.

### 6.6.1. Basic Approach

The basic approach to consistency is similar to the MOESI class; the protocol ensures that no more than one cache contains dirty blocks of a file at one time, and that no client reads a file when there is a newer version elsewhere. Consistency is assured by each client maintaining the local state of each cached file, while the server maintains a global state, including the location of all cached copies. The protocol class provides consistency using *either* invalidation or write broadcast. The mechanism for reliable broadcast is not specified.

The server state consists of sufficient information to ensure consistency, but may not precisely match the client state. For example, if a file is modified in some cache, it may not be essential to know if the file is open or closed; the server must contact the client in either case if another client requests the file.

Client file systems receive requests from local processes and respond using cached information if possible. When necessary the client forwards requests to the server. Because accessing the server is slow compared to local processing, and the server is a likely bottleneck, these requests are minimized.

The server is responsible for interacting with other clients to assure consistency, for example to invalidate other cached copies. The new state of the file can be a function of the global state maintained by the server, or it may depend on information provided by another client (e.g. the resulting state may be different if the modified file mentioned earlier is actually open).

The state affected by consistency controls is the state of the file as a whole. However, the basic cache unit is still the block, and the blocks of a file can be individually valid or invalid, clean or dirty, within the constraints of the file state (e g blocks may be dirty only if the file is dirty). In general the consistency actions on open and close use the file state, while the actions on read or write depend on the block state. In effect, the consistency mechanism is superimposed on the normal write-back mechanism used to count reads and writes. The consistency protocols make use of periodic write-back and deletions, but these are stack algorithms as discussed in Section 3.4. Since these algorithms are the same as presented earlier they are not discussed further.

### 6.6.2. State Attributes

The state of a file in each client cache, and the system as a whole, can be represented as a combination of five attributes. The first three are similar to those used to define the MOESI states. Two additional attributes are needed because of the granularity of consistency, based on open/close.

#### 6.6.2.1. Valid/Invalid

A file may be **Valid** in the cache, or it may be **Invalid**. Any file that is not in the cache and not "known" to the processor (i.e. the name is not cached) is invalid, in which case the remaining attributes are irrelevant. Conversely, a file may be valid even if no blocks are present, for example if the file is accessed remotely as discussed in Section 6.6.2.5.

#### 6.6.2.2. Modified/Unmodified

The portions of the file that are cached may be **Modified**, and therefore not in agreement with the copy in the server, or they may be **Unmodified**. The file is assumed modified whenever it is opened for write and write-back is permitted, regardless of whether a write has occurred. Only one client may contain a modified version of the file, therefore this client may also be referred to as the "owner" of the file. If broadcast is being used then it is assumed that the server also receives the broadcast, hence the client-cached copies are not modified.

#### 6.6.2.3. Exclusive/Shared

Since most files are not shared (see Chapter 7), one way to reduce server accesses is to give a client **Exclusive** access to the file. Just as in the MOESI states, a file that is exclusive is present in exactly one client cache. If a file is **Shared** then its blocks may be present in more than one cache. If a file is shared and open for write then the client must use write-through broadcast to update the other copies. The exclusive states allow writes to be avoided, along with other operations such as open and close. In the latter case the server may not be aware of the precise state of the file in an exclusive cache.

#### 6.6.2.4. Open/Closed

A file may be valid but **Closed**, or it may be **Open for Read** or **Open for Write**. This specifies whether the client is actively accessing the file and whether the file may be modified during the access. The server uses this information to detect concurrent writes.

#### 6.6.2.5. Cachable/Remote

In most cases a file is **Cachable** in the client. In some cases the client may choose not to cache the file, or the protocol may prevent a client from doing so. In this case the file is accessed **Remotely** from the server, or from another client, using read through and write through. This attribute is only significant if the file is open; a closed remote file is considered invalid.

### 6.6.3. States

These attributes would seem to give 48 possible states, but many of these are meaningless. For example, if the file is invalid, none of the other attributes is relevant. Also, a Remote file must be Open and Unmodified, while a Cachable file Open for write must be Modified. There are actually 13 meaningful combinations of the attributes, listed in Table 6.1. Figure 6.1 depicts the relationships between the attributes which form the states. Two states, OpM and M, are shared, even though they can only exist in one cache. The shared attribute simply means that the client must notify the server of state changes, where their exclusive counterparts do not.

Table 6.1: Possible States

| Attributes | | | | | State | |
|---|---|---|---|---|---|---|
| V/I | E/S | R/W/C | M/U | C/R | Name | Abbr |
| I | X | X | X | X | Invalid | I |
| V | E | W | M | C | Exclusive Open Write | ExOpW |
| V | E | R | U | C | Exclusive Open Read | ExOpR |
| V | E | R | M | C | Exclusive/Modified Open Read | ExOpM |
| V | E | C | M | C | Exclusive Modified | ExM |
| V | E | C | U | C | Exclusive Unmodified | ExR |
| V | S | W | U | C | Shared Open Write | OpW |
| V | S | R | M | C | Open Read Modified | OpM |
| V | S | R | U | C | Shared Open Read | OpR |
| V | S | C | M | C | Modified | M |
| V | S | C | U | C | [illegible]/Unmodified | R |
| V | X | W | U | R | Remote Open Write | RemOpW |
| V | X | R | U | R | Remote Open Read | RemOpR |

Table 6.1: Possible file states. The state is a combination of the attributes Valid/Invalid, Exclusive/Shared, Open Read/Open Write/Closed, Modified/Unmodified, and Cachable/Remote

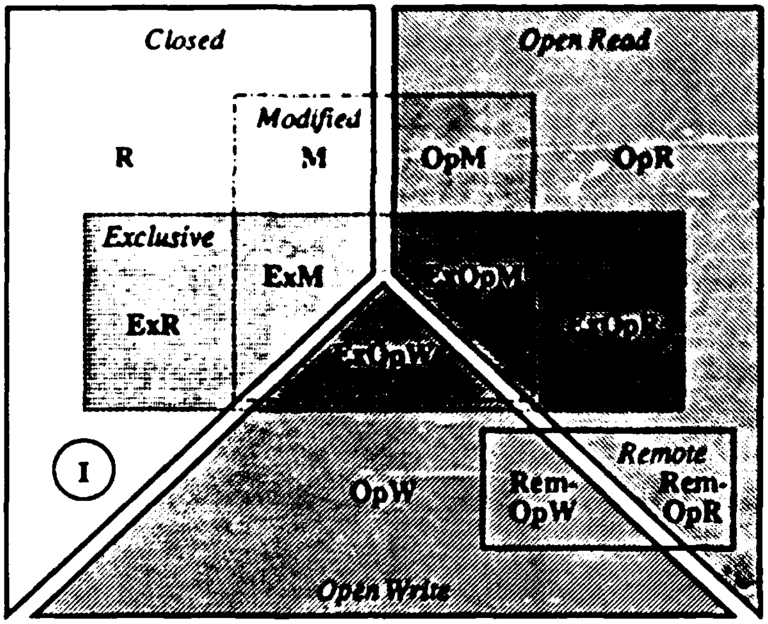


Figure 6.1: Relationships between attributes and states. The three largest areas represent the Open/Closed components. A small area shows the two Remote states; all others are Cachable. States that are not in the Modified area are Unmodified, and states that are not Exclusive are Shared. The invalid (I) state is included with Closed states since all Open files are presumed to be "known" and therefore valid.

### 6.6.4. Client Actions

The requests received by a client from local processes are described below. These requests obviously require other parameters such as the file name and/or identification, but these are not pertinent to the discussion and are therefore omitted.

| | |
|---|---|
| *Open(R)* | Open the file for reading. |
| *Open(W)* | Open the file for reading and writing. |
| *Close(*)* | Close the last open on the file. In actuality, the client file system will receive all closes from local processes, but since only the last close (or the last Write close) can affect the global state, the table only shows the actions for these. |
| *Close(W)* | Close the last open for writing. There must be read opens remaining (otherwise the request would be a *Close(*)*). |
| *WriteBack()* | Write all dirty blocks of the file to the server. Although this could occur at any time, it only affects the state if the file is not open for write (otherwise the client could immediately make a block dirty, and the notification would be meaningless). If blocks are being written individually by the replacement algorithm, *WriteBack()* is called at or after the push of the last dirty block of a file. This operation is similar to Pass used in the MOESI protocols. |
| *Flush()* | Remove all blocks of the file from cache. Again, it is only meaningful if there are no cachable opens outstanding. This action is also used when the last block of the file is removed by replacement. |

Individual Read and Write requests do not affect the state, so they are omitted.

Table 6.2 defines the client actions for this class of protocols. Each entry shows the server requests required, if any, and the resulting state for each given request when the file is in a particular state. For example. *Open.S(R)* is a server open request described in the next section. Where several states are given separated by slashes (/), the choice is determined by the result of the

Table 6.2. Client Actions to Cache Requests

| Client State | Open(R) | Open(W) | Close(*) | Close(W) | WriteBack() | Flush() |
|---|---|---|---|---|---|---|
| I | Open.S(R),<br>ExOpR/OpR/<br>RemOpR | Open.S(W),<br>ExOpW/OpW/<br>RemOpW | | | | -- |
| ExOpR | ExOpR (1)<br>or<br>Open.S(R),<br>ExOpR | ExOpW<br>or<br>Open.S(W),<br>ExOpW | ExR | | Write.S(),<br>OpR | -- |
| ExOpW | ExOpW<br>or<br>Open.S(R),<br>ExOpW | ExOpW<br>or<br>Open.S(W),<br>ExOpW | ExM<br>or<br>Close.S(*),<br>M | ExOpM<br>or<br>Close.S(W),<br>OpM | | -- |
| ExOpM | ExOpM<br>or<br>Open.S(R),<br>ExOpM | ExOpW<br>or<br>Open.S(W),<br>ExOpW | ExM<br>or<br>Close.S(*),<br>M | | Write.S(),<br>OpR | -- |
| ExR | ExOpR<br>or<br>Open.S(R),<br>ExOpR | ExOpW<br>or<br>Open.S(W),<br>ExOpW | | | ExR<br>or<br>Write.S(),R | I<br>or<br>Flush.S(),<br>I |
| ExM | ExOpM<br>or<br>Open.S(R),<br>ExOpM | ExOpW<br>or<br>Open.S(W),<br>ExOpW | | | Write.S(),<br>OpR (3)<br>or<br>Close.S(*),Write.S(),<br>R (4) | Close.S(*),<br>Write.S(),<br>Flush.S(),<br>I or<br>Close.S(*),<br>Write.S(),I |
| OpR | OpR<br>or<br>Open.S(R),<br>OpR | Open.S(W),<br>ExOpW/OpW/<br>RemOpW | Close.S(*),<br>R<br>or<br>OpR (2) | | | Close.S(*),<br>Flush.S(),<br>I (3) |
| OpM | OpM<br>or<br>Open.S(R),<br>OpM | Open.S(W),<br>ExOpW | Close.S(*),<br>M<br>or<br>OpM (2) | | Write.S(),<br>OpR | Close.S(*),<br>Flush.S(),<br>I (3) |
| OpW | OpW<br>or<br>Open.S(R)<br>OpW, | OpW<br>or<br>Open.S(W)<br>ExOpW/OpW | Close.S(*),<br>R (5) or<br>Close.S(W),<br>OpR | Close.S(W),<br>OpR (6) | -- | -- |
| R | Open.S(R),<br>ExOpR/OpR | Open.S(W),<br>ExOpW/OpW/<br>RemOpW | | -- | R | I<br>or<br>Flush.S(),I |
| M | Open.S(R),<br>OpM | Open.S(W),<br>ExOpW | -- | -- | Write.S(),R | Write.S(),<br>Flush.S(),<br>I |
| RemOpR | RemOpR<br>or<br>Open.S(R), (7)<br>OpR/RemOpR | Open.S(W),<br>RemOpW | Close.S(*),<br>I | -- | -- | -- |
| RemOpW | RemOpW<br>or<br>Open.S(R), (7)<br>ExOpR/RemOpW | RemOpW<br>or<br>Open.S(W),<br>ExOpW/RemOpW | Close.S(*),<br>I | Close.S(W),<br>RemOpR | -- | -- |

NOTES ON TABLE 6.2:

1. Assumes a *namei* cache is available in the client to bind names to identifiers. If the name is not cached then it must be resolved by the server using *Open.S(R)*.

2. The *Write.S()* is superfluous, but notifies the server that the file is unmodified. It can piggyback on the *Close.S(*)*. This sequence is used to relinquish Exclusive ownership, but retain a copy of the file.

3. By not telling the server that the file is fully closed the client can avoid notifying the server of subsequent opens for read. The file must be "closed" when flushed to maintain proper state in the server.

4. Since the server doesn't know the precise state, the *Close.S(*)* is necessary to relinquish exclusive access.

5. Since a cache in the OpW state must use write-through, it normally reverts to R when closed. In the unlikely case that the server is not party to write-through updates then OpW would go to M on close. In this case, other caches would have to locate the owner, and the owner, not the server, would service read faults from other clients. We choose to avoid this complexity by assuming that the server sees all broadcast writes.

6. There are read opens remaining, otherwise it would be a Close(*). Since OpW uses broadcast, the file is not modified.

7. It is possible that the client has the only outstanding opens, in which case remote processing is not required. The server will so indicate in response to *Open.S(.)* if queried.

server request. As an example, if an invalid (I) file is opened for write (*Open(W)*), the client requests the open from the server using *Open.S(W)*, which returns the new state (ExOpW, OpW, or RemOpW). When two server requests are listed together (e.g. *Write.S(), Flush.S()*) it is assumed that both are sent to the server in a single message. Actions connected by 'or' are alternatives, any one of which could be chosen. The choice could be different for any file, any cache, or even at different times for the same file. It is this freedom of choice that makes this a compatible class of protocols. The numbers is braces in Table 6.2 refer to notes following the table that amplify on certain of the alternatives.

### 6.6.5. Server Actions

Now consider the server actions based on requests it receives from the clients. These requests generally correspond to the requests received by the client, so we use a ".S" suffix to distinguish them.

| | |
|---|---|
| *Open.S(R)* | Open the file for reading. |
| *Open.S(W)* | Open the file for writing. |

| | |
|---|---|
| *Close.S(*)* | The client no longer has any open instances of the file. An Exclusive cache asserting *Close.S(*)* relinquishes exclusive control, but the file is still assumed to be modified. |
| *Close.S(W)* | The client no longer has the file open for write (but still has it open for reading). An Exclusive cache asserting this relinquishes exclusive control, but is assumed to be still reading the file. |
| *Write.S()* | The client is writing the last dirty block of the file. This is only meaningful if the file is no longer open for writing. The request may piggyback on the data for the last dirty block, or on a *Close.S(*)*. An Exclusive cache that asserts *Write.S()* relinquishes exclusive control, but is assumed to be reading the file unless the request is combined with *Close.S(*)*. |
| *Flush.S()* | The client no longer contains any blocks of the file. This is only meaningful if the file is no longer open. The request may piggyback on a write of the last block, or on a close message. |

The following directives are sent to clients by the server.

| | |
|---|---|
| *CallBack()* | Write all dirty blocks and invalidate all cached blocks. The file is assumed to be no longer cachable. |
| *WriteBack()* | Write all dirty blocks, but the blocks may remain in cache. |
| *Redirect()* | Informs a cache that all future actions should be directed to another cache (either a client or the server). The cache performs CallBack actions. |

Table 6.3 defines the server actions when a request is received from a client. The current server state is the union of the states in the clients (i.e. if any client has the file open for write then the server state reflects this, even though other clients may only be reading the file.) The server performs consistency control actions, which may include notifying other clients, and returns the new state to the requester. If the file is held exclusively by another client, the server may be unsure of the exact state, and therefore queries that client using requests *Open.E(R)* or

Table 6.3: Server Actions to Client Requests

| Server State | Open.S(R) | Open.S(W) | Close.S(*) | Close.S(W) | Write.S() | Flush.S() |
|---|---|---|---|---|---|---|
| I | ExOpR,<br>Ret(ExOpR)<br>or<br>OpR,Ret(OpR) | ExOpW<br>Ret(ExOpW)<br>or<br>OpW,Ret(OpW) | -- | -- | -- | -- |
| Ex**<br>{1} | Open.E(R), {2}<br>server-state,<br>Ret(client-state) | CallBack(owner),<br>RemOpW,<br>Ret(RemOpW)<br>or<br>Open.E(W), {2}<br>server-state,<br>Ret(client-state) | M | OpM | OpR | I |
| OpR | OpR,<br>Ret(OpR) | RemOpW,<br>Invalidate(all),<br>Redirect(all open),<br>Ret(RemOpW) {3}<br>or<br>OpW,Ret(Opw) {4} | OpR/R<br>{5} | -- | -- | OpR<br>{6} |
| OpM | WriteBack(owner),<br>OpR<br>Ret(OpR)<br>/ {7}<br>OpM,<br>Ret(OpM) | CallBack(owner),<br>RemOpW,<br>Ret(RemOpW)<br>or WriteBack(owner),<br>OpW,<br>Ret(OpW)<br>/ {7}<br>ExOpW,<br>Ret(ExOpW) | M | -- | OpR | -- |
| OpW | OpW,<br>Ret(OpR) | RemOpW,<br>Redirect(all open),<br>Ret(RemOpW)<br>{3}<br>or<br>OpW,Ret(OpW) {4} | OpW/OpR/<br>R {8} | OpR/OpW<br>{8} | -- | OpW<br>{6} |
| R | OpR,Ret(OpR)<br>or<br>ExOpR,<br>Invalidate(all),<br>Ret(ExOpR) | OpW,Ret(OpW)<br>or<br>ExOpW,<br>Invalidate(all),<br>Ret(ExOpW) | -- | -- | -- | R/I<br>{9} |
| M | OpR<br>WriteBack(owner),<br>Ret(OpR)<br>or<br>ExOpR,<br>CallBack(owner),<br>Invalidate(all),<br>Ret(ExOpR) | OpW<br>WriteBack(owner),<br>Ret(OpW)<br>or<br>ExOpW,<br>CallBack(owner),<br>Ret(ExOpW) | -- | -- | R | M/R/I<br>{9} |
| RemOpR<br>{10} | RemOpR,<br>Ret(RemOpR) | RemOpW,<br>Ret(RemOpW) | RemOpR/OpR/<br>I {11} | -- | -- | -- |
| RemOpW | RemOpR,<br>Ret(RemOpR) | RemOpW,<br>Ret(RemOpW) | RemOpW/OpR/<br>I {11} | RemOpW/<br>OpR {11} | -- | -- |

NOTES ON TABLE 6.3:

1. The server response is the same for all Exclusive states, since the precise state is not known. The Open request is assumed to be from a client other than the exclusive owner.

2. The Exclusive client determines both the global state and the state returned to the caller. See Table 6 6

3. Force remote processing at the server.

4. Use broadcast write.

5. If other caches have the file open then the state is OpR, otherwise the state is R.

6. The flush must be from other than the cache with the file open. The state is unchanged.

7. If the request is from other than the owner then the owner must writeback the file. If it is from the owner then the state is unchanged.

8. If other caches have the file open then the state is OpW or OpR, otherwise the state is R, since the cache was using write through.

9. If no other caches have file then I, else R or M, as appropriate.

10. The server should never be in this state. If the file is Non-cachable, there should be a writer somewhere

11. If other opens remain after the final write open then the state becomes OpR, otherwise I. A close of a reader does not affect the state.

*Open.E(W)*. This request, described in the following section, returns the new ''server-state'' and ''client-state''. Following the table are a series of notes explaining the numbers in braces. Client state transitions due to server requests are shown in Table 6.4.

### 6.6.6. Exclusive Cache Actions

When a cache has the file exclusively, the server may be unaware of the exact state of the file. When the server receives a request from another cache, it can assume the worst (i.e. that the file is open for writing in the exclusive cache), and make the file remotely accessed. Alternatively, it can inform the exclusive client of the request and let it decide how the file should be handled. The exclusive client may release the file if it isn't open, permitting the other client exclusive access; it may retain exclusive access, requiring the other client to process the file remotely at the exclusive site; or it may allow the file to become shared. Table 6.5 shows the options available to the exclusive client when a request is received. The client returns the state to

Table 6.4 Client Actions to Server Requests

| Client State | CallBack() | WriteBack() | [illegible] |
|---|---|---|---|
| ExOpW | Write()<br>Flush()<br>RemOpW | | |
| ExOpM | Write()<br>Flush()<br>RemOpR | | |
| ExOpR | Flush()<br>RemOpR | | |
| ExM | Write()<br>Flush()<br>I | | |
| ExR | Flush()<br>I | | |
| OpW | Write()<br>Flush()<br>RemOpW | | |
| OpM | Write()<br>Flush()<br>RemOpR | Write()<br>OpR | |
| Op | | | Flush()<br>RemOpR |
| M | Write()<br>Flush()<br>I | Write()<br>R | |
| R | | | |
| RemOpW | | | |
| RemOpR | | | |

be given to the original caller, as well as the new global state of the file

Another option, not shown, is for the exclusive client to force the file to be remotely processed at itself, or even at the originating client. This is advantageous when one site is the primary user of the file. This option could also reduce the load on the server by centralizing access at another site. Since concurrent write sharing is assumed to be rare, these alternatives should not make a major difference in performance, and thus are ignored.

Table 6.5. Exclusive Client Actions

| State | Open Ex(R) | Open Ex(W) |
|---|---|---|
| ExOpW | OpW<br>Ret(OpR,OpW) (1)<br>or<br>RemOpR,<br>Write(),Flush(),<br>Ret(RemOpW,RemOpR) | OpW,<br>Write(),<br>Ret(OpW,OpW) (1)<br>or<br>RemOpW,<br>Write(),Flush(),<br>Ret(RemOpW,RemOpW) |
| ExOpR | OpR,<br>Ret(OpR,OpR) (1) | OpR,<br>Ret(OpW,OpW) (1)<br>or<br>RemOpR,<br>Flush(),<br>Ret(RemOpW,RemOpW) |
| ExOpM | OpR<br>Write(),<br>Ret(OpR,OpR) | OpR,<br>Write(),<br>Ret(OpW,OpW) (1)<br>or<br>RemOpR,<br>Write(),Flush(),<br>Ret(RemOpW,RemOpW) |
| ExR | R,<br>Ret(OpR,OpR) (1)<br>or<br>I,<br>Flush(),<br>Ret(ExOpR,ExOpR) (3) | R,<br>Ret(OpW,OpW) (1)<br>or<br>I,<br>Flush(),<br>Ret(ExOpW,ExOpW) (3) |
| ExM | R,<br>Write(),<br>Ret(OpR,OpR) (1)<br>or<br>I<br>Write(),Flush(),<br>Ret(ExOpR,ExOpR) (3) | R,<br>Write(),<br>Ret(OpW,OpW) (1)<br>or<br>I<br>Write(),Flush(),<br>Ret(ExOpW,ExOpW) (3) |
| I | I,<br>Ret(ExOpR,ExOpR) | I,<br>Ret(ExOpW,ExOpW) (3) |

NOTES ON TABLE 6.5:

1. Allow the file to be shared using write through broadcast when appropriate.
2. Remote to server.
3. Pass exclusive control to caller.

## 6.7. Selected Protocols

It would not be particularly enlightening to individually discuss each option in the transition matrices. Instead, several protocols have been extracted from the class to demonstrate representative alternatives. The following describes four selected protocols from the class defined in the previous section. These protocols illustrate most of the key options in the class, and discuss some of the design tradeoffs involved. These protocols will be analyzed in Chapter 7.

### 6.7.1. Sprite Protocol

The first protocol is essentially that used by the Sprite file system [Nels87]. Its main attribute is simplicity and the heavy reliance on the file server to maintain consistency. Clients notify the server on every open and close, and therefore no name caching is required in the clients. The open returns the current version number of the file; if the client has a prior version cached then it invalidates the cache and rereads the file. The clients use write-back, and the write to the server occurs on replacement or when another client subsequently reads the file. (The actual Sprite file system forces a write-back every 30 seconds, but this policy is implemented on top of the consistency protocol.)

A unique feature of the Sprite file system is its handling of concurrent writes. If one or more clients has the file open simultaneously, and at least one is writing, the server immediately flushes the file from all of the caches and requires all clients to access the file remotely from the server. Consistency is ensured since there is only a single copy, at the server. This technique is also used in the next two example protocols.

Tables 6.6 and 6.7 define the Sprite protocol. The protocol uses invalidation to handle sequential write sharing, therefore all write opens are exclusive, but revert to non-exclusive on close. No other exclusive states are used, therefore the server knows to begin remote processing if another client opens the file while it is in an exclusive state. Concurrent write sharing is satisfied by calling all cached copies to the server.

#### 6.7.1.1. Variation: Notification of Write Flush

When a client writes and flushes the last dirty block of a file it could notify the server of this fact, saving a write-back or call-back if another client accesses the file. This action is essentially "free" since it can "tag along" on the write, and so is always advantageous. The simulation of Chapter 7 uses this variation. The client could also notify the server when a clean block is

Table 6.6: Client Actions to Local Requests - Sprite Protocol

| State | Open(R) | Open(W) | Close(*) | Close(W) | WriteBack() | Flush() |
|---|---|---|---|---|---|---|
| I | Open.S(R), OpR/ RemOpR | Open.S(W), ExOpW/ RemOpW | -- | -- | -- | -- |
| ExOpW | Open.S(R), ExOpW | Open.S(W), ExOpW | Close.S(*), M | Close.S(W), OpM | -- | -- |
| OpR | Open.S(R), OpR | Open.S(W), ExOpW/ RemOpW | Close.S(*), R | -- | -- | -- |
| OpM | Open.S(R), OpM | Open.S(W), ExOpW | Close.S(*) M | -- | Write.S(), OpR | -- |
| R | Open.S(R), OpR | Open.S(W), ExOpW/ RemOpW | -- | -- | R | I or Flush.S(),I |
| M | Open.S(R), OpM | Open.S(W), ExOpW | -- | -- | Write.S(),R | Write.S(), Flush.S(), I or Write.S(),I |
| RemOpR | Open.S(R), RemOpR | Open.S(W), RemOpW | Close.S(*), I | -- | -- | -- |
| RemOpW | Open.S(R), RemOpW | Open.S(W), ExOpW/RemOpW | Close.S(*), I | Close.S(W), RemOpR | -- | -- |

Table 6.7: Server Actions to Client Requests - Sprite Protocol

| State | Open.S(R) | Open.S(W) | Close.S(*) | Close.S(W) | Write.S() | Flush.S() |
|---|---|---|---|---|---|---|
| I | OpR, Ret(OpR) | ExOpW Ret(ExOpW) | -- | -- | -- | -- |
| ExOpW {1} | CallBack(owner) RemOpW, Ret(RemOpR) | CallBack(owner), RemOpW, Ret(RemOpW) | M | OpM | OpR | I |
| OpR | OpR, Ret(OpR) | RemOpW, Invalidate(all), Redirect(all open), Ret(RemOpW) | OpR/R | -- | -- | OpR |
| OpM | WriteBack(owner), OpR Ret(OpR) / OpM, Ret(OpM) | CallBack(owner), RemOpW, Ret(RemOpW) / ExOpW, Ret(ExOpW) | M | -- | OpR | -- |
| R | OpR,Ret(OpR) | ExOpW, Invalidate(all), Ret(ExOpW) | -- | -- | -- | R/I |
| M | OpR WriteBack(owner), Ret(OpR) | ExOpW CallBack(owner), Ret(ExOpW) | -- | -- | R | M/R/I |
| RemOpR | RemOpR, Ret(RemOpR) | RemOpW, Ret(RemOpW) | RemOpR/OpR/ I | -- | -- | -- |
| RemOpW | RemOpR, Ret(RemOpR) | RemOpW, Ret(RemOpW) | RemOpW/OpR/ I | RemOpW/ OpR | -- | -- |

Note 1: Assumed not to be from the owner, otherwise the state is unchanged.

Table 6.8: Client Actions to Local Requests - Exclusive Protocol

| State | Open(R) | Open(W) | Close(*) | Close(W) | WriteBack() | Flush() |
|---|---|---|---|---|---|---|
| I | Open.S(R),<br>ExOpR/OpR/<br>RemOpR | Open.S(W),<br>ExOpW/OpW/<br>RemOpW | -- | -- | -- | -- |
| ExOpR | ExOpR | ExOpW | ExR | -- | -- | -- |
| ExOpW | ExOpW | ExOpW | ExM | ExOpM | -- | -- |
| ExOpM | ExOpM | ExOpW | ExM | -- | Write.S(),OpR | -- |
| ExR | ExOpR | ExOpW | -- | -- | ExR | I |
| ExM | ExOpM | ExOpW | -- | -- | Close.S(*),Write.S(),<br>R | Write.S(),<br>Flush.S(),<br>I or<br>Write.S(),I |
| OpR | OpR | Open.S(W),<br>ExOpW/<br>RemOpW | Close.S(*),<br>R | -- | -- | -- |
| OpM | OpM | Open.S(W),<br>ExOpW | Close.S(*)<br>M | -- | Write.S(),<br>OpR | -- |
| R | Open.S(R),<br>ExOpR/OpR | Open.S(W),<br>ExOpW/<br>RemOpW | -- | -- | R | I |
| M | Open.S(R),<br>OpM | Open.S(W),<br>ExOpW/<br>RemOpW | -- | -- | Write.S(),R | Write.S(),<br>Flush.S(),<br>I |
| RemOpR | RemOpR | Open.S(W), | Close.S(*),<br>I | -- | -- | -- |
| RemOpW | RemOpW | RemOpW | Close.S(*),<br>I | Close.S(W),<br>RemOpR | -- | -- |

Table 6.9: Server Actions to Client Requests - Exclusive Protocol

| State | Open.S(R) | Open.S(W) | Close.S(*) | Close.S(W) | Write.S() | Flush.S() |
|---|---|---|---|---|---|---|
| I | ExOpR,<br>Ret(ExOpR) | ExOpW<br>Ret(ExOpW) | -- | -- | -- | -- |
| Ex** | Open.E(R),<br>server-state,<br>Ret(client-state) | Open.E(W),<br>server-state,<br>Ret(client-state) | M | OpM | OpR | I |
| OpR | OpR,<br>Ret(OpR) | RemOpW,<br>Invalidate(all),<br>Redirect(all open),<br>Ret(RemOpW) | OpR/R | -- | -- | OpR |
| OpM | WriteBack(owner),<br>OpR<br>Ret(OpR)<br>/<br>OpM,<br>Ret(OpM) | CallBack(owner),<br>RemOpW,<br>Ret(RemOpW)<br>/<br>ExOpW,<br>Ret(ExOpW) | M | -- | OpR | -- |
| R | OpR,Ret(OpR) | ExOpW,<br>Invalidate(all),<br>Ret(ExOpW) | -- | -- | -- | R/I |
| M | OpR<br>WriteBack(owner),<br>Ret(OpR) | ExOpW<br>CallBack(owner),<br>Ret(ExOpW) | -- | -- | R | M/R/I |
| RemOpR | RemOpR,<br>Ret(RemOpR) | RemOpW,<br>Ret(RemOpW) | RemOpR/OpR/<br>I | -- | -- | -- |
| RemOpW | RemOpR,<br>Ret(RemOpR) | RemOpW,<br>Ret(RemOpW) | RemOpW/OpR/<br>I | RemOpW/<br>OpR | -- | -- |

flushed, but this is never worthwhile since it requires an extra message for each file, with a potential savings only if the file is called back.

### 6.7.2. Exclusive Protocol

Several studies have shown that a vast majority of files are not shared [Floy86a,Kent86], a fact we substantiate in the next chapter. The next protocol, called the Exclusive protocol, reduces server interaction for these files by using a call-back technique similar to that used by Andrew. The first open on a file must contact the server, which grants the client exclusive access to the file. The client may then repeatedly close and reopen the file in any mode without notifying the server. Instead, the server will notify the client if another client wants to access the file. Once the file becomes shared all opens and closes require the server. However, since invalidation is used to maintain consistency, any writer gets exclusive access (provided there are no other opens). Concurrent writers are handled using the Sprite remote processing scheme.

This protocol saves opens and closes for single-user files, at the cost of a call-back message if the file becomes shared. A call-back scheme clearly requires fewer messages for files that are truly exclusive. For shared files, call-back generally results in fewer overhead messages than polling when the ratio of reads to writes is high [Ruan87].

Tables 6.8 and 6.9 define the Exclusive protocol. All opens from an invalid state and all shared opens require the server; all opens and closes to exclusive files are satisfied within the client. Writes use invalidation, and thereby obtain exclusive ownership. The first reader also receives exclusive access, which is retained until another cache requests the file and the server calls back the file. Because a cache never voluntarily relinquishes ownership in this protocol, the shared dirty states (OpM, M, OpW) are not used.

#### 6.7.2.1. Variation: Lazy Invalidation

In two situations the server matrix, Table 6.7, contains *Invalidate(all)* to remove cached copies from other clients. For clients known to be closed (i.e. in the R state), these could be done *actively*, by contacting the clients immediately, or using *lazy invalidation* by waiting until the client's next open to inform it that the copy is invalid. Active invalidation frees up cache space, at the cost of an extra message. Lazy invalidation saves the message, and is most advantageous for high-use files that are likely to be reopened soon.

Lazy invalidation blurs the definition of the exclusive states since a client may have exclusive access while another client continues to cache a prior version. Since the other cache *will be* invalidated the next time the client opens the file, the file can be considered invalid for the purposes of consistency control. Note, however, that the blocks of the file remain valid, taking up room in the cache.

### 6.7.3. Read-Only Protocol

Of the files that are shared, many are read-only and therefore require no consistency control. The following protocol is optimized for the assumption that the file is read-only, paying a penalty in extra messages if the assumption is wrong. If the ratio of reads to writes is high enough, even files that are occasionally written can benefit from the protocol.

When a file is first opened (for read) the cache is given shared read access to the file (OpR state). The client may then repeatedly reopen the file for read without notifying the server. Since the client can open the file at any time, the close provides no useful information to the server, and it too is avoidable. This is represented in the protocol by showing the client remaining in the OpR state after *Close()*; this matches the server's perception. In fact, the client should keep track of whether the file is actually open for reasons which will become clear in a moment.

Now suppose that the read-only assumption is wrong and the file is opened for write. The server must certainly be notified, and the server must actively invalidate all cached copies (except of course that of the caller). Active invalidation allows the server to determine if any of the clients is really open. If it is open then concurrent write sharing exists. It solves this by using the Sprite technique of caching the file only at the server. (An alternative not shown in the protocol, applicable if temporary inconsistencies are tolerable, would be to permit the read to continue on the current version and to invalidate their cached copy when it is closed). If no concurrent opens

Table 6.10: Client Actions to Local Requests - Read-Only Protocol

| State | Open(R) | Open(W) | Close(*) | Close(W) | WriteBack() | Flush() |
|---|---|---|---|---|---|---|
| I | Open.S(R), OpR/ RemOpR | Open.S(W), ExOpW/ RemOpW | -- | -- | -- | -- |
| ExOpW | ExOpW | ExOpW | ExM | ExOpM | -- | -- |
| ExOpM | ExOpM | ExOpW | ExM | -- | Write.S(), OpR | -- |
| ExM | ExOpM | ExOpW | -- | -- | Write.S(), OpR | Write.S(), Flush.S(), I |
| OpR | OpR | Open.S(W), ExOpW/ RemOpW | OpR | -- | -- | Close.S(*), Flush.S(), I |
| RemOpR | RemOpR | Open.S(W), RemOpW | Close.S(*), I | -- | -- | -- |
| RemOpW | RemOpW | RemOpW | Close.S(*), I | Close.S(W), RemOpR | -- | -- |

*Table 6.11: Server Actions to Client Requests - Read-Only Protocol*

| State | Open.S(R) | Open.S(W) | Close.S(*) | Close.S(W) | Write.S() | Flush.S() |
|---|---|---|---|---|---|---|
| I | OpR, Ret(OpR) | ExOpW Ret(ExOpW) | -- | -- | -- | -- |
| Ex** | Open.E(R), server-state, Ret(client-state) | Open.E(W), server-state, Ret(client-state) | M | OpM | OpR | I |
| OpR | OpR, Ret(OpR) | RemOpW, Invalidate(all), Redirect(all open), Ret(RemOpW) | OpR/R | -- | -- | OpR |
| RemOpR | RemOpR, Ret(RemOpR) | RemOpW, Ret(RemOpW) | RemOpR/OpR/ I | -- | -- | -- |
| RemOpW | RemOpR, Ret(RemOpR) | RemOpW, Ret(RemOpW) | RemOpW/OpR/ I | RemOpW/ OpR | -- | -- |

exist the writer is given exclusive write access.

When the writer closes the file there are two reasonable actions, based on mutually exclusive assumptions. The first is that the write was a "fluke"; the file really is read-mostly and there will soon be other clients requesting to read the file. Under this assumption the client should immediately write the file to the server and relinquish ownership, thereby saving the write-back which would otherwise occur as soon as another client opens the file. The client retains a read open to avoid subsequent opens for read. This option is not specifically shown in Table 6.10. It is implemented at a higher level by performing a *WriteBack()* immediately after the close.

The other assumption is that the write is conclusive evidence that the file is *not* read-only, in which case there is a chance that the writer will rewrite or delete the file in the near future. If this assumption is correct then the client should delay the write-back and retain exclusive control until the server requests it. Both these alternatives are analyzed in Chapter 7. Of course, since these are both valid protocols, different files could use different options.

The Andrew file system implements a form of this protocol using *read-only file systems*. The server does not maintain a list of caching sites (a *call-back list*) for files from these file systems, so no consistency controls exist. Special operations are needed to invalidate copies of the files if they must be changed.

### 6.7.4. Write Broadcast Protocol

For files that are shared *and* frequently written, none of the options discussed so far is efficient, particularly if the file is also widely read. This is because all of them use full-file invalidation; each time the file is written the readers must re-read their working set of blocks even if only a small portion changes. The best alternative for this case is the use of write broadcast, allowing each reader to retain its cached working set, while forcing the writer to broadcast changes. This saves at least one read per reader, at the cost of a single broadcast message that is assumed to be seen by all readers.

The following describes a write broadcast subset of the class of protocols. It does not fully specify the underlying reliable broadcast protocol, but is based on some assumptions about that protocol. First, the protocol is assumed to be reliable and guarantee ordered delivery of messages. This permits concurrent write sharing without pulling the file to the server. Second it assumes that no such system could economically function if all writes required acknowledgements from all other clients. To reduce acknowledgements, and to reduce the burden on disinterested clients, we assume that the writer knows which of the other clients are interested in the data, and that the write broadcast is processed and acknowledged only by these clients. The type of underlying protocol required is more properly termed a *multi-cast* protocol, but is consistent with existing proposals for reliable local area network broadcast [Cher87, Chan84]. We will continue to use the term broadcast to reinforce the assumption that the data is transmitted only once (assuming no errors). Kent makes a similar assumption that the group of recipients is known in advance even though the Caching Ring provides reliable broadcast [Kent86].

The protocol assumes that it is not advantageous to broadcast when there are no other interested clients. This is because the write of a block of data is generally more costly than a call-back message (See Chapter 7), and it is possible that the write-through broadcast can be avoided if the file is rewritten or deleted. For this reason the protocol uses elements of the Exclusive protocol when the file is not shared. A cache having exclusive write access (ExOpW state) operates as a write-back cache.

Table 6.12: Client Actions to Local Requests - Write Broadcast Protocol

| State | Open(R) | Open(W) | Close(*) | Close(W) | WriteBack() | Flush() |
|---|---|---|---|---|---|---|
| I | Open.S(R),<br>ExOpR/OpR | Open.S(W),<br>ExOpW/OpW | -- | -- | -- | -- |
| ExOpR | ExOpR | ExOpW | ExR | -- | -- | -- |
| ExOpW | ExOpW | ExOpW | ExM | ExOpM | -- | -- |
| ExOpM | ExOpM | ExOpW | ExM | -- | Write.S(),<br>OpR | -- |
| ExR | ExOpR | ExOpW | -- | -- | ExR | I |
| ExM | ExOpM | ExOpW | -- | -- | Write.S(),<br>OpR | Write.S(),<br>Flush.S(),I |
| OpR | OpR | Open.S(W),<br>ExOpW/OpW | OpR | -- | -- | Close.S(*),<br>Flush.S(),I |
| OpW | OpW | OpW | Close.S(W),<br>OpR | Close.S(W),<br>OpR | -- | -- |

Table 6.13: Server Actions to Client Requests - Write Broadcast Protocol

| State | Open.S(R) | Open.S(W) | Close.S(*) | Close.S(W) | Write.S() | Flush.S() |
|---|---|---|---|---|---|---|
| I | ExOpR,<br>Ret(ExOpR) | ExOpW<br>Ret(ExOpW) | -- | -- | -- | -- |
| Ex** | Open.E(R),<br>server-state,<br>Ret(client-state) | Open.E(W),<br>server-state,<br>Ret(client-state) | -- | -- | OpR | I |
| OpR | OpR,<br>Ret(OpR) | OpW,<br>Ret(OpW) | OpR/R | -- | -- | OpR |
| OpW | OpW,<br>Ret(OpR) | OpW,<br>Ret(OpW) | OpW/OpR/<br>R | OpR/OpW | -- | OpW |

When there are multiple simultaneous users, all write opens are forced to contact the server in order to get the set of other caching sites. Shared writes enter the OpW state which must use write-through broadcast on a write. Readers are only required to contact the server once to register their membership in the set; all subsequent closes and read opens are immaterial for consistency control, since writes are broadcast whether or not the file is open. Writers contact the server on close only for compatibility with other protocol of the class, e.g. when there are no write opens the file can be treated as read-only. If all caches use this write broadcast protocol then these closes are superfluous. Caching sites can contact the server on flush in order to remove themselves from the set of caching sites, or they can do this as a part of acknowledging the first write after flush. We simulated the former case.

### 6.8. One-Pass Analysis

The class of file system consistency protocols has a much larger state space than the MOESI protocols (13 states compared to 5) and it is not at all obvious that this class defines one-pass algorithms. However, with familiar assumptions and restrictions, it can be shown to be.

#### 6.8.1. Compatible States

The key to showing that these are one-pass algorithms is the following: of the five components of the state, two are generally independent of cache size (Open/Close and Remote/Cachable) and the remaining three are identical to those comprising the MOESI class, and still obey inclusion. The first assertion is easy to show; the latter takes the rest of this section.

When showing inclusion we need to consider all states which can coexist at the same time in different cache sizes. We therefore begin with the following definition:

> **Definition:** Two states are *incompatible* if it is impossible for a file to be in both states at the same time for different cache sizes. States that have not been shown to be incompatible are said to be *potentially compatible*, although this does not guarantee that they could coexist in different cache sizes. For example, in the MOESI class the S and M states are incompatible, assuming inclusion holds; all others are compatible. (The S state implies that the sharing level is less than the dirty level; the M state implies the opposite relation, thus they

are incompatible.)

It seems obvious that if a file is open then it is open in all cache sizes, since the open and close occur regardless of cache size. The same can be said about the type of open (read or write). However there is one situation where the open attribute is not independent of cache size, brought about by the Read-Only and related protocols. If the client neglects to tell the server that the file is closed, the server's perception is that the file is still open in all sizes. However, if the client "pushes" the file from cache (i.e. removes it by replacement) then the file becomes invalid for some sizes while seemingly open for larger sizes. This is solved as mentioned earlier by having the client record whether the file is really open or not, as well as whether the server thinks it is open. The real attribute (closed) should be used to determine the client state, in which case the open states are independent of cache size.

Thus we can easily partition the state space into three sets of potentially compatible states: those where the file is open for read, those where it is open for write, and those where it is closed. The states in different partitions are incompatible.

One more simple partitioning is possible. The decision to make a file non-cachable (i.e. Remote) is based on concurrent write sharing, which also occurs in all cache sizes. However, there is an alternative response to concurrent write — to use write broadcast. We therefore make the restriction that if the server chooses remote processing for one state it chooses it for all potentially compatible states. For example, in Table 6.3 a cache can enter a Remote state in response to an *Open.S(W)* from the states {Ex*, OpR, OpM, and OpW}. If remote processing is chosen for one then it must be chosen for all of these states. This ensures that the remote attribute is independent of cache size. Therefore the remote states are incompatible with all other state, as well as one another. This leads to the conclusion that a client in the RemOpW state is in that state for all sizes.

We have thus partitioned the states into five sets of potentially compatible states: {RemOpR}, {RemOpW}, {OpR, OpM, ExOpR, ExOpM}, {R, M, ExR, ExM}, and {OpW, ExOpW}. These are shown by the heavy lines in Figure 6.1.

### 6.8.2. Assumptions and Restrictions

It can now be shown that the remaining three attributes obey inclusion by considering only states within potentially compatible sets. First, the key assumptions and restrictions from the MOESI protocols must be applied, namely that the request streams are synchronous, that replacement priority is not affected by other caches, and that rules are applied consistently within a cache. In addition, we have already introduced a restriction to ensure that the Remote attribute is independent of cache size. Several additional restrictions are needed, most having direct analogs within the proof of the MOESI protocols.

#### 6.8.2.1. Invalidation

In the discussion of MOESI protocols we saw that misuse of invalidation could violate inclusion of validity (i.e. valid in some sizes and invalid in larger sizes). To ensure that validity inclusion is preserved, we must ensure that invalidation is either used, or not used, for all cache sizes. There are three places where invalidation occurs. First, a client can invalidate a file using *Flush()*, but this will always occur independent of cache size. Second, the server may invalidate clients using *Invalidate()* or *CallBack()*. In many cases there is an alternative action that does not use invalidation, usually involving write broadcast. We therefore make the restriction that invalidation must be used from all compatible states. It turns out that this is the same restriction needed to guarantee that the Remote characteristic is independent of cache size. The third place where invalidation may occur is within an exclusive cache in response to *Open.E()* requests. If the exclusive cache invalidates itself, using *Flush()*, then it must do so for all compatible states. In this case, the compatible states are the pairs {ExOpR,ExOpM} and {ExR,ExM}.

#### 6.8.2.2. Write Broadcast

The write broadcast used when a client is in the OpW state is similar to the write broadcast used by the Firefly protocol in that it assumes that the server sees the broadcast, and therefore the client cache is unmodified. Also like Firefly, this means that the Modified attribute is not inclusive in the resulting configuration of states. Consider the sequence of actions shown in Figure 6.2. Initially the block is dirty in cache one. When client two opens the file, the server asks for a write-back in those sizes where the block is dirty, and places cache two in the OpW state. However, the server knows that cache two is the only cache site for small sizes, so it grants exclusive write access to cache two for those sizes. Because OpW is not a modified state, the Modified attribute is not inclusive. This is even more apparent after cache two closes the file, when the file is clearly modified where write-back was used and clean where write-through broadcast was used.

In Section 5.9.1 we showed that the Firefly protocol is a one-pass algorithm despite the lack of inclusion since it requires only one additional variable to show the upper bound of the dirty sizes. For the file system protocols we do not even need the additional variable; the sharing level is also the upper bound of the dirty sizes. (This is not the case for Firefly because sharing is reevaluated on every write. See Figure 5.30.)

---

```
  Time/Action        States

Initial:          1:I------>M------------->
                  2:I--------------------->

Open(W):2         1:I------>R------------->
                  2:ExOpW-->OpW----------->

Close():2         1:I------>R------------->
                  2:ExM---->R------------->
                    ^       ^
                    D       S
```

**Figure 6.2**: Dirty inclusion is violated by write broadcast. However, the file is never dirty when it is shared, so the sharing level is also the upper bound on dirty sizes.

---

#### 6.8.2.3. Related Memory Sizes

In the MOESI protocols there were three possible situations with respect to the memory bound necessary for a one-pass algorithm:

a) The state space and the statistics were bounded.

b) The state space was bounded, but not the statistics.

c) Neither the state space nor statistics were bounded.

In the general case with independent cache sizes, neither was bounded and the algorithms were one-pass only when there was a relation between memory sizes, such as all caches the same size.

For the file system protocols we also find different situations. The Sprite protocol, like the Berkeley protocol, is of type a). An informal argument for this is the fact that the state never depends on the size of any other cache, since all read states are assumed shared and all write states are known to be exclusive.

The rest of the protocols fall at least into situation b). A simple analysis has not uncovered any examples of the latter situation. Intuitively we do not believe them to exist, since the protocols do not poll the status of unmodified caches on each open (equivalent to the {CS} bus signal), therefore the state does not achieve the "stair-step" effect seen in the Dragon protocol. A thorough search, which might uncover one, is not warranted since even type b) situations are not *one-pass algorithms for independent sizes.*

We therefore again restrict the protocols to situations where all cache sizes are the same. Even if the protocols support independent cache sizes, the K-dimensional result surface (with K greater than 100!) would be incomprehensible and justifies the simplification.

### 6.8.3. Proof of One-Pass Algorithm

Given the preceding assumptions and restrictions, the proof that the class defines one-pass algorithms is straight-forward, but tedious, and is therefore presented only in outline. The proof involves showing that each component of the state obeys inclusion, and therefore that the state is representable with a few variables. We have already seen that two of the attributes are independent of cache size, and therefore inclusive. By partitioning the state space it is easy to show by induction that the actions and result states for each request preserves inclusion for all potentially compatible states. Therefore the algorithms are one-pass.

## 6.9. Conclusions

In this chapter we have discussed the differences between processor caches and file system caches which affect the design of cache consistency protocols. We have defined a class of file system protocols in the same way that the MOESI protocols are defined, and described four subset protocols. Although the resulting state space and protocol definitions are very complicated, this formality makes it easy to conceive a variety of protocols. We have also shown that the class constitutes one-pass algorithms. We will use this in the next chapter to simulate the four example protocols, and will show that one-pass analysis greatly decreases the time required for simulation.

# Chapter 7

# Evaluation of Client Caching in a Network File System

## 7.1. Summary

This chapter ties together the preceding work by applying all of the techniques in a study of client caching in a network file system. The study makes use of very complete traces of UNIX file system activity collected from a university timesharing system. An initial analysis of the traces finds that although the files are predominantly user, temporary, and spool files, a few system files account for 40% of all reads. Nearly half of all writes are to temporary files (in */tmp*). Analysis of sharing finds little concurrent sharing, but nearly two-thirds of all opens are to the 5% of files which are accessed by two or more users. Over half of the opens are to read-only files.

A large number of options are simulated using one-pass techniques, including consistency control protocols, write policies, and fetch policies. The one-pass technique produces the miss and traffic ratios for *all* cache sizes in only 5-10% more time than required to simulate a single size, even more improvement than reported for single-processor simulations in Chapter 4. Simulations show that the choice of consistency protocol makes a large difference in performance. For example, the Read-Only protocol has half the miss ratio and half the server load (transfer ratio) of the Sprite protocol. Write broadcast improves the miss ratio, but requires a reliable broadcast protocol. The miss and transfer ratios for the best realizable protocols are nearly double that of an optimal protocol, although the absolute difference is only 5-10%.

A study of write policies confirms that 25% of writes are avoided by delaying write-back by 30 seconds; an additional 25% are avoided by waiting 5 minutes. Simply making temporary files write-back and performing buffered write on all others reduces writes by 30%.

The study of fetch policies shows that an optimistic policy that assumes a file will be processed sequentially, therefore prefetching one block ahead until shown to be wrong, is a good policy which reduces the miss ratio without affecting the transfer ratio. Another good policy is to prefetch an entire file on open, as long as the file size is below 100K. Prefetching larger files on open dramatically increases the transfer ratio, and increases the miss ratio for small caches.

### 7.2. Introduction

This chapter presents results of the application of nearly all of the discussed techniques to the analysis of caching in a network file system. The purpose is both to demonstrate the synthesis of the techniques and to present the actual results of the analysis.

The remainder of this section reviews some related studies. This is followed in Section 7.3 by an analysis of file sharing, which is one of the key factors in the performance of the consistency algorithms. Section 7.4 actually compares the four consistency protocols from Chapter 6, followed by a comparison of write policies in Section 7.5, and fetch policies in Section 7.6.

#### 7.2.1. Relation to Other Work

There have been very few prior studies which relate to this analysis. Smith presented a very comprehensive study of the placement and parameters for disk caches, using traces from large IBM mainframes [Smit85b]. He concluded that disk cache is a very effective way to reduce costly disk accesses. Ousterhout et al. [Oust85] studied caching in a file server, including the effects of cache size, write policy, and blocksize. They similarly reported that very large block caches can be quite effective in reducing disk accesses, and that delaying writes is a major source of this reduction. Nelson et al. used the same traces for a few simulations using the Sprite protocol, and reported that client caches are similarly effective at reducing server load by 75% with 1-8 Mb caches [Nels87]. They also reported that consistency control added little to the data traffic. Kent confirmed many of the Ousterhout results for a different set of UNIX trace data [Kent86]. Both were from a university environment. In addition, Kent simulated a set of caches using his Caching Ring network interface.

Using a queuing network model, Lazowska et al. simulated a number of parameters of a network file system, including server and/or client caching and faster disk and server processor [Lazo86]. Ramakrishnan and Emer performed a similar study concentrating on server options

[Rama86]. Both of these studies concluded that the server processor is the most likely critical resource as the number of clients increases, an observation confirmed in at least one real system [Morr86].

A final study by Ruan and Tucker used analytic techniques to compare several parameters of file replication in a distributed system [Ruan87]. They concluded that demand replication (i.e. demand-fetch caching) is more efficient than remote access or static replication (in terms of the number of messages required) in most instances. They also concluded that it is more efficient to invalidate when a file is written (call-back) rather than verifying validity on each open (polling), when there is a high read-to-write ratio.

Our study differs in several respects from its predecessors. Of the trace-driven simulation studies it is the first to report in any detail on client caching. It is the only study to report the effects of consistency control, and considers all server traffic, not just data traffic. This also distinguishes it from the queuing models by Lazowska and Ramakrishnan.

### 7.3. The Traces

The data used in this study consisted of traces of virtually all file system activity from a VAX 11/780 running the UNIX time sharing system in a university setting [Zhou86]. The machine, called *ucbarpa*, was one of the same ones which generated the less complete traces used in Chapter 4. The traced events include all file creation, deletion, open, close, and renaming. It also includes all process creation (*fork*) and exit, and program execution (*exec*). Each event includes the time (with 10ms resolution), the identification of the relevant process and the user's identity (*userid*). File creation, deletion, opens and renames, and program executions contain the full path name of the referenced file. They also contain the file size and, for opens and creates, the number of bytes transferred. The number of bytes actually loaded when a program is executed is, unfortunately, *not* traced. Table 7.1 summarizes the traced events.

Because the traces were so complete, the volume of trace data was enormous (roughly 40MB per day), which limited the practical length of a trace session. The data available con-

Table 7.1: Traced Events

| Event | Associated Data |
|---|---|
| (all) | file id, process id, user id, activity type, file type, time |
| Open/Create | file name, references, bytes, mode, file size |
| Close | file size |
| Read/Write | starting location, bytes |
| Delete | file name |
| Rename | old/new file names |
| Exec | command name |
| Fork | process CPU time |
| Exit | process CPU time |

Table 7.2: Number of Events and Percent of Each Record Type

| Trace file | final9 | final9a | final9b | final10 |
|---|---|---|---|---|
| Duration of trace | 96 hr | 24 hr | 24 hr | 29 hr |
| Size of trace file | 177Mb | 46Mb | 37Mb | 44Mb |
| Total records: | 3,200,550 | 844,002 | 648,001 | 763,911 |
| Open records: | 15.67 | 15.03 | 16.62 | 15.93 |
| Create records: | 1.26 | 1.08 | 1.37 | 1.98 |
| Close records: | 16.93 | 16.11 | 18.00 | 17.91 |
| Delete records: | 1.48 | 1.90 | 1.50 | 1.99 |
| Read records: | 38.87 | 34.61 | 44.45 | 33.34 |
| Write records: | 18.12 | 23.93 | 9.95 | 18.75 |
| Rename records: | 0.06 | 0.04 | 0.10 | 0.20 |
| Exec records: | 2.08 | 1.94 | 2.18 | 2.76 |
| Fork records: | 1.55 | 1.68 | 1.64 | 1.95 |
| Vfork records: | 1.21 | 1.01 | 1.27 | 1.62 |
| Exit records: | 2.76 | 2.68 | 2.91 | 3.56 |

sisted of one trace lasting four days (called *final9*) and one lasting just over one day (*final10*). A few experiments were made using the entire *final9* trace, but because of its size, most of the simulations were made against one-day extracts. Each of these traces is summarized in Table 7.2. The results for the experiments were very comparable in all cases. Therefore only the results for *final10* are shown.

Whenever possible, the *user id* defined the user actually causing the request. In addition to the real users, UNIX systems have a distinguished user (*root*) permitted special privileges and usually associated with system activity. In cases where a user "assumed the identity" of root to perform special operations, such as placing mail in another users' mailbox, the trace records this fact but continues to attribute the work to the original user. Even so, unattributed root activity is roughly half of all activity. It is interesting to note that Smith observed this same result in traces of IBM file system activity [Smit85b].

A final question which must be addressed is the applicability of using time sharing traces to study distributed systems. There are two answers to this. The simple answer is that no distributed file system traces were available for the study. Gathering equivalent data from a network file system would have been enormously expensive, requiring instrumentation of all clients from some representative server and reliable collection of the data. The second answer, albeit unsubstantiated, is the belief that the file system workloads are similar in the two environments, assuming the user workload stays constant. Similar assumptions were made by Ousterhout and Kent in their studies. This is a worthwhile topic for future research.

Each file and activity was categorized by type during post-processing. The activity type was assigned according to the command name in the *exec* requests to load a program. The activity types correspond to the three major productive activities of a user (software development, text processing, and mail handling), miscellaneous activities (user interface shells, status checking, user programs, login programs, and other), plus system daemon activity. Table 7.3 summarizes these types. The use of the system can be characterized by looking at the resources consumed by each type, shown in Table 7.4.

The file types fall into six major categories (user, system, spool, temp, admin, other). These are summarized in Table 7.5. Within each type were numerous subtypes. The complete file type assignment is more extensive than that made by Satyanarayanan [Saty81] using file name appendages, or Floyd, who used three categories (user, system, network) [Floy86a]. The file types were assigned by considering the file name and the activity type of the requesting process. Pattern matching on the file name and path succeeded in many cases (e.g. a file in */tmp* matching *ex*... is an editor temporary file; a user file ending in *.o* is an object file). When this failed, as it often did for user files, the file type was assigned based on usage (e.g. a file written by a mail process is a mail file). Table 7.6 characterizes the usage by file type. The heavy use of spool files is affected by the fact that the system measured was, at the time, also a print server for the main

Table 7.3: Summary of Activity Types

| Type | Description | Examples |
|---|---|---|
| Mail | Mail-related commands | mail, mh, msgs, talk |
| Text | Text-processing commands | troff, eqn, tbl, text edit |
| Soft | Software development | cc, ld, lisp, software edit |
| Other | Miscellaneous commands | ls, cd, sort, dump |
| Start | Initial commands for a session | login, rlogin, getty |
| Status | Status checking commands | users, ruptime, du |
| System | System daemons | init, named, rwhod, lpd |
| User | Any command from a user directory | a.out |
| Shell | Command shells | csh, sh |

Table 7.4: Summary by type of Activity

Characterization of Final9 Trace

| Activity Type | Processes | CPU Time | Open/R | Open/W | I/O | Bytes Read | Written |
|---|---|---|---|---|---|---|---|
| Grand Total | 88,556 | 187,668 sec | 408,081 | 133,846 | 1,824,132 | 1,411 mb | 380 Mb |
| Other | 29.81% | 24.06% | 5.43% | 6.18% | 30.37% | 26.41% | 32.65% |
| Start | 7.61% | 12.82% | 5.19% | 3.99% | 6.82% | 5.96% | 0.06% |
| Status | 8.04% | 3.13% | 16.83% | 0.19% | 7.52% | 9.17% | 0.07% |
| System | 27.21% | 14.03% | 48.25% | 50.95% | 22.13% | 20.32% | 15.11% |
| Mail | 13.06% | 10.78% | 15.16% | 27.96% | 10.57% | 20.25% | 21.15% |
| Text | 1.45% | 9.30% | 0.93% | 2.81% | 4.45% | 5.16% | 15.15% |
| Soft | 3.63% | 5.35% | 3.40% | 4.70% | 3.36% | 8.37% | 10.33% |
| User | 1.36% | 4.01% | 1.70% | 0.57% | 11.31% | 2.67% | 5.15% |
| Shell | 7.84% | 16.53% | 3.11% | 2.65% | 3.48% | 1.68% | 0.32% |

Characterization of Final10 Trace

| Activity Type | Processes | CPU Time | Open/R | Open/W | I/O | Bytes Read | Written |
|---|---|---|---|---|---|---|---|
| Grand Total | 27,368 | 50,143 sec | 91,415 | 45,425 | 397,955 | 415 mb | 148 mb |
| Other | 33.45% | 18.70% | 8.38% | 11.48% | 26.80% | 16.85% | 29.72% |
| Start | 6.80% | 10.09% | 6.43% | 3.58% | 8.70% | 5.61% | 0.04% |
| Status | 6.15% | 1.79% | 12.55% | 0.13% | 4.50% | 6.00% | 0.04% |
| System | 28.20% | 23.98% | 45.08% | 51.99% | 28.62% | 33.92% | 27.28% |
| Mail | 10.69% | 10.82% | 17.87% | 22.81% | 14.16% | 19.60% | 18.30% |
| Text | 1.28% | 8.13% | 0.94% | 2.08% | 3.44% | 3.26% | 8.80% |
| Soft | 2.79% | 5.23% | 2.73% | 2.84% | 3.34% | 7.70% | 7.84% |
| User | 2.36% | 2.84% | 2.03% | 1.70% | 7.54% | 5.05% | 7.61% |
| Shell | 8.29% | 18.43% | 3.98% | 3.39% | 2.90% | 2.00% | 0.36% |

**Table 7.4**: Characterization by type of activity, showing the number of processes of each type and the amount of processor time, number of opens for read and write, logical I/O requests, and number of bytes read and written. The top line shows the total number, while subsequent lines give the percent for each activity type. For both traces the System activity is very high, as is Mail activity. The Other type is also high, but a few file-transfer commands account for most of the I/O, while a majority of the Other processes consume few resources.

Table 7.5: Major File Types

| Type | Usage | Examples |
|---|---|---|
| User | All user files | /a/os/james/.login |
| System | Global, mostly read-only files. Libraries. | /usr/lib/libm.a (math library)<br>/usr/lib/aliases (mail aliases) |
| Spool | Producer-consumer files. | /usr/spool/ipd/dfA145ernie.Berkeley.EDU (printer spool file)<br>/usr/spool/rwho/whod.arpa (rwho status)<br>/usr/msgs/2431 ("messages" bulletin board)<br>/usr/spool/mail/james (mailbox) |
| Temp | Temporary files | /tmp/Ex14034 (editor temporary)<br>/usr/tmp/trtmp05474 (troff temporary) |
| Admin | Administrative and accounting log files | /etc/passwd (password file)<br>/etc/utmp (log of active users) |
| Other | Trace files and unknown | |

Table 7.6: Summary by Type of File

Characterization of Final9 Trace

| FILE TYPE | FILES | DELETES | OPEN/R | OPEN/W | I/O | BYTES READ | WRITTEN |
|---|---|---|---|---|---|---|---|
| Grand Total | 55,875 | 47,210 | 408,081 | 133,846 | 1,824,132 | 1,411 mb | 380 mb |
| Unknown | 0.77% | 0.02% | 0.01% | 0.01% | 14.85% | 11.43% | 5.67% |
| User | *32.11%* | *23.66%* | *13.55%* | *15.19%* | *25.59%* | *11.09%* | *26.99%* |
| Sys | 1.35% | 0.20% | 32.22% | 10.31% | 16.73% | 42.51% | 5.09% |
| Spool | 35.25% | 40.69% | 40.26% | 50.44% | 17.10% | 7.45% | 17.48% |
| Temp | 29.88% | 35.26% | 1.70% | 18.66% | 9.21% | 12.06% | 44.22% |
| Admin | 0.64% | 0.16% | 12.27% | 5.38% | 16.52% | 15.47% | 0.54% |

Characterization of Final10 Trace

| FILE TYPE | FILES | DELETES | OPEN/R | OPEN/W | I/O | BYTES READ | WRITTEN |
|---|---|---|---|---|---|---|---|
| Grand Total | 20,069 | 15,207 | 91,415 | 45,425 | 397,955 | 415 Mb | 148 Mb |
| Unknown | 1.52% | 0.07% | 0.01% | 0.00% | 0.26% | 0.26% | 0.10% |
| User | 40.21% | 27.95% | 16.45% | 20.89% | 34.03% | 17.00% | 37.37% |
| Sys | 1.76% | 0.05% | 41.45% | 8.69% | 20.28% | 43.88% | 0.97% |
| Spool | 32.42% | 40.99% | 25.11% | 50.33% | 18.30% | 8.19% | 17.37% |
| Temp | 23.37% | 30.66% | 2.51% | 14.64% | 11.49% | 16.77% | 43.17% |
| Admin | 0.72% | 0.27% | 14.47% | 5.45% | 15.65% | 13.90% | 1.02% |

**Table 7.6:** Characterization by type of file, showing the number of files of each type, the number of files deleted, opens for read and write, logical I/O requests, and bytes read and written. The top line gives the total for all files, while subsequent lines show the percent for each file type. There are a large fraction of User files, along with Temp and Spool files, but System files account for a large fraction of the opens.

laser printer for the department. Also, the trace period was near the end of a semester when the use of this printer was heavy.

## 7.4. File Sharing

An important factor in the performance of cache consistency protocols is the amount and type of file sharing which is present. Two types of sharing were discussed in Chapter 6: concurrent write sharing and sequential sharing. Both have an impact on performance.

Concurrent access exists when two or more users have a file open at the same time; if one is a writer then it is concurrent write sharing. Kent investigated sharing and found that only 4% of opens are concurrent accesses. He did not distinguish concurrent write sharing.

The results of our analysis are shown in Table 7.7. It shows that 3.2% of all opens were actually shared, and only 2% were concurrent write shared. The percent of I/O and bytes transferred during concurrent write sharing are similarly small. More significant is the fact that over half of the opens and 90% of the shared I/O is attributable to a single file, */etc/utmp*, which keeps track of users logged on. It is written by every user when signing on and off, and read periodically by most users. This is a prime candidate for some other mechanism, such as a server or database. The SUN NFS provides a separate file for each workstation (along with separate accounting files in */usr/adm*) to circumvent the problem. The simulations continued to treat it as a single shared file.

Table 7.7: Concurrent Sharing

| | Files | Opens | I/O | Bytes |
|---|---|---|---|---|
| Total | 19,207 | 136,840 | 397,955 | 564Mb |
| Shared | 111 (0.5%) | 4,396 (3.2%) | -- | -- |
| Concurrent Write | 16 (0.08%) | 3,023 (2.2%) | 14,153 (3.5%) | 11.3 Mb (2.0%) |
| utmp | 1 | 1,548 (1.1%) | 12,952 (3.2%) | 10.7 Mb (1.9%) |

**Table 7.7:** Concurrent write sharing exists in the traces, but involves a very small number of files, and few I/O's or bytes. Most of it is attributable to the *utmp* file. The number of files (19207) is less than the number of files reported in Table 7.6 because it ignores 862 files which were deleted but never accessed.

Sequential sharing can be analyzed in terms of the number of files that are *ever* used by multiple users. Floyd analyzed sharing and reported that fewer than 10% of files are shared, concluding that sharing is not a significant issue. This ignores the fact that consistency control must occur for each open, so it is the impact of the files, not the number of files, which is important.

Porcar studied sequential sharing in large IBM mainframes, and reported that 7-13% of files were shared, accounting for 27-46% of all opens [Porc82]. His figures did not include temporary files, which were almost 90% of the files he observed.

Table 7.8 shows an analysis of potential sharing in terms of files, opens, and data transfers. Several observations are important. First, although less than 5% of files are shared, they account for nearly two-thirds of all opens. On the fortunate side, most of the shared opens are to read-only files, or to files which are written initially and subsequently never modified (and therefore essentially read-only). Note too that the vast majority of writes and written bytes are to single-user files. However, over half of all write opens are to shared files.

We can conclude from this that most files will benefit from a protocol suited for non-shared (private) files, and that a read-only assumption is appropriate for most of the rest. However, the remaining large amount of write sharing increases the importance of the consistency protocol. Also, these traces were from a university UNIX environment, on a system used to support few, if any, group projects. We would expect the sharing in this envoronment to be less than that found in many commercial environments.

## 7.5. The Simulator

Most of the analysis which follows is based on simulations of a network of clients accessing a single server. The simulation used the one-pass techniques discussed in Chapter 6 to simulate a separate cache for each client. By actual measurement of several simulations, the one-pass technique took only 5-10% longer than a similar simulation of a single set of cache sizes. This is much better than the 20-100% additional time reported in Chapter 4, due to two factors. First, multi-processor file system simulations are time-consuming even when dealing with a single set of cache sizes. Opens and closes were ignored in the single-processor simulations of Chapter 4,

Table 7.8: Potential Sharing

| FILE-TYPE | FILES | OPEN/R | OPEN/W | READS/ | BYTES | WRITES/ | BYTES |
|---|---|---|---|---|---|---|---|
| GRAND-TOTAL | 19,207 | 91,415 | 45,425 | 254,698 | 415.9Mb | 143,257 | 148.2Mb |
| SINGLE-USER FILES | 18,284<br>(95.19%) | 22,352<br>(24.45%) | 20,530<br>(45.20%) | 97,275<br>(38.19%) | 185.3Mb<br>(44.55%) | 112,978<br>(78.86%) | 130.7Mb<br>(88.19%) |
| READ-ONLY SHARED | 275<br>(1.43%) | 41,995<br>(45.94%) | 1,681<br>(3.70%) | 72,931<br>(28.63%) | 157.7Mb<br>(37.93%) | 0<br>(0.00%) | 0<br>(0.00%) |
| WRITE-ONCE SHARED | 246<br>(1.28%) | 9,149<br>(10.01%) | 256<br>(0.56%) | 9,928<br>(3.90%) | 2.1Mb<br>(0.52%) | 246<br>(0.17%) | 58Kb<br>(0.04%) |
| WRITE-ONLY SHARED | 4<br>(0.02%) | 0<br>(0.00%) | 3,063<br>(6.74%) | 0<br>(0.00%) | 0<br>(0.00%) | 3,061<br>(2.14%) | .1Mb<br>(0.09%) |
| READ/WRITE SHARED | 379<br>(1.97%) | 17,490<br>(19.13%) | 19,888<br>(43.78%) | 74,564<br>(29.28%) | 70.7Mb<br>(17.01%) | 26,972<br>(18.83%) | 17.3Mb<br>(11.67%) |
| ALL SHARED | 923<br>(4.81%) | 69,063<br>(75.55%) | 24,895<br>(54.80%) | 157,423<br>(61.81%) | 230.6 Mb<br>(55.45%) | 30,279<br>(21.14%) | 17.4 Mb<br>(11.81%) |

**Table 7.8:** *Sequential or potential sharing, showing the activity for single-user and shared files. The shared files are divided into read-only, written-once (therefore essentially read-only), write-only, and files which are both read and written multiple times. Although most files are single-user, most opens and reads are to shared files. Of these, a large fraction are to read-only files, decreasing the need for consistency controls.*

but must be used to simulate the consistency protocol. This involves looking at the state and potentially invalidating the file in all other caches. All of this adds to the overhead of simulation, decreasing the dominance of determining the stack distance. The second difference was that each cache now receives a single-user reference stream, reducing both the size of the stack and increasing the locality. The mean stack size for these simulations was 977 blocks and the mean stack distance was 28, thereby further reducing the cost of stack analysis.

Individual workstation request streams were created by mapping each user on the trace to a separate client. All unattributable root activity was assigned to a single client. This is a different approach from that taken by Kent, who spread the root activity randomly among the user clients. We prefer our approach for two reasons. First, the two key influences on cache performance are locality and sharing; a random allocation creates an unwanted bias to both by increasing sharing and decreasing locality. Second, the residual root activity is predominantly associated with servicing the printer (40%), handling arriving mail (20%) creating status files (16%). All of these are likely candidates for special servers; none are likely to be heavy users of individual workstations. Other heavy activity types include login processing prior to password validation (7%) and log-file maintenance (7%).

Several of the protocols assume that names are cached in the clients along with data. Rather than fix the size of the name cache, or separately simulate it, the simulator assumed that the name of a file could have been cached whenever any data block of the file is cached. (i.e. it assumed that the name was cached for all sizes where some block of the file was valid). This simple assumption simulates a name cache which grows in proportion to the size of the data cache, without the complexity of maintaining the name cache itself.

The simulation computed the average miss and transfer ratio across all clients. The miss ratio, as discussed earlier, is computed as the number of read, write fetch, and open requests which required file server interaction, divided by total file system requests.

The transfer ratio was presented earlier as a simple ratio of messages, assuming all messages have equal cost. Several studies have shown that a large fraction of the server cost is due to copying data to and from the network interface. For example, Cheriton reported server CPU times of 2.3ms to process a request which transfers no data, and 5.7ms to process a request and return 1K of data [Cher83]. Lazowska et al. estimated the server CPU service time to be 2.4ms for a request and 2.6ms per 1K of response [Lazo86]. None of these figures include the CPU time required to service a disk access, estimated by Lazowska et al. to be roughly 10ms per 1K of data.

The open request is also a potentially-expensive operation. Preliminary measurements of the Sprite file system show that an open requires 4.4ms of server CPU in addition to the 3.5ms required to process any request [Welc87]. These figures also ignored the cost of disk I/O in servicing the open.

To make the transfer ratio better reflect both the network traffic and the server load, we weighted all server requests as shown in Table 7.9. These weights are rough approximations of the server cost of each operation based on data from the above references. Each request ''cost'' 3ms; each open takes an additional 2ms. (Since the Sprite file system is still under development and has not been tuned, we were more optimistic in estimating this cost). Each I/O included a factor for processing the request plus 2.5ms per 1K of data transferred.

Table 7.9: Weighting Factors for Transfer Ratio

| Request | Open | I/O | All other requests |
|---|---|---|---|
| Weight | 5 | 3+2.5*data length (in K) | 3 |

## 7.6. Consistency Policies

As mentioned earlier, one of the key results of this chapter is a comparison of cache consistency protocols. The following sections present results for each of the example protocols from Chapter 6. For each protocol we show graphs of the miss and (weighted) transfer ratio. In each such graph (for example, Figure 7.1), the upper curve is the actual ratio. The lower curves divide the region under the curve into the operations making up the metric. For example, in the left graph of Figure 7.1, the miss ratio for an 8Mb cache is about 30%; of this, just over half (18%) is due to opens and under half (12%) due to reads. In addition, Figure 7.5 compares the miss and transfer ratios for all of the protocols.

### 7.6.1. Sprite Protocol

The key feature of the Sprite protocol described in Section 6.7.1 is that it involves the server on every open and close. Figure 7.1 shows the miss and transfer ratios resulting from simulation of this policy. Note that the miss ratio is limited by the fact that all opens involve the server. The cost of opens and closes is further seen by the fact that data transfers are less than one-third of the transfer ratio for large caches.

From these figures it is apparent that improvements due to caching continue at least to 8Mb, although with diminishing returns beyond 1-2Mb. This size includes only data; it does not include directories, file descriptors, or executable program space. However, it provides a good starting point for sizing client caches.

Simulations were made using a variation of the protocol that invalidated all other clients immediately whenever a file was opened for write. This alternative made almost no improvement in miss ratio, while increasing the transfer ratio by a small 0.5%.

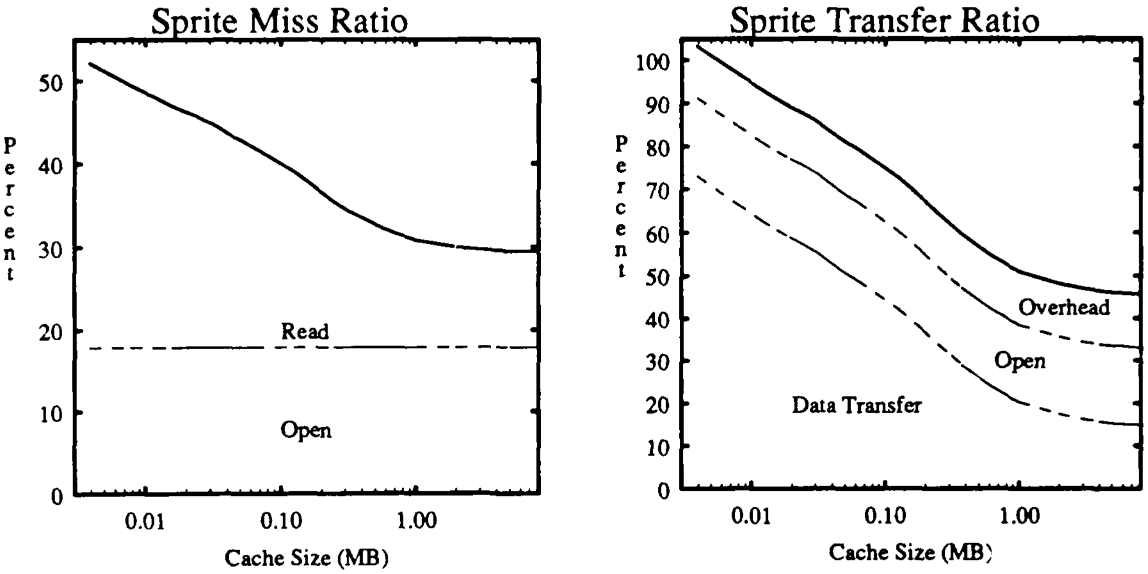


**Figure 7.1**: Sprite protocol. The upper curve in each case is the actual ratio. The lower curves divide the region under the curve into the components making up the metric. For example, the miss ratio for an 8Mb cache is about 30%. Of this, just over half (18%) is due to opens and under half (12%) to reads. The transfer ratio is a similar curve, with three component parts due to data transfers (both reads and writes), opens, and overhead (closes, deletes, and call/write-back requests). Each region is weighted as shown in Table 7.9. Because of this the overhead region, which is almost entirely closes, is smaller than the open region, even though there are roughly the same number of open and close messages.

### 7.6.2. Exclusive Protocol

The Exclusive protocol, described in Section 6.7.2, saves opens and closes for the majority of files, which are not shared, at the cost of a call-back for those that are. Figure 7.2 shows that the savings in opens decreases the miss ratio and transfer ratios compared to the Sprite protocol.

There are several ways to view these improvements. In absolute terms the decrease in miss ratio is only 5%, but in relative terms it is 15% lower than the Sprite miss ratio, for large cache sizes. Neither of these is a dramatic improvement, but consider this: the 30% miss ratio obtained with a 4Mb cache using the Sprite protocol is achievable with 400Kb using the Exclusive protocol — a 90% decrease! Another consideration, however, is that the Exclusive protocol (as well as all the remaining ones) requires a name cache. If name caching makes an open 20-30% more expensive then the Sprite protocol may still be advantageous. Name caching is discussed further as a topic for future research in Chapter 8.

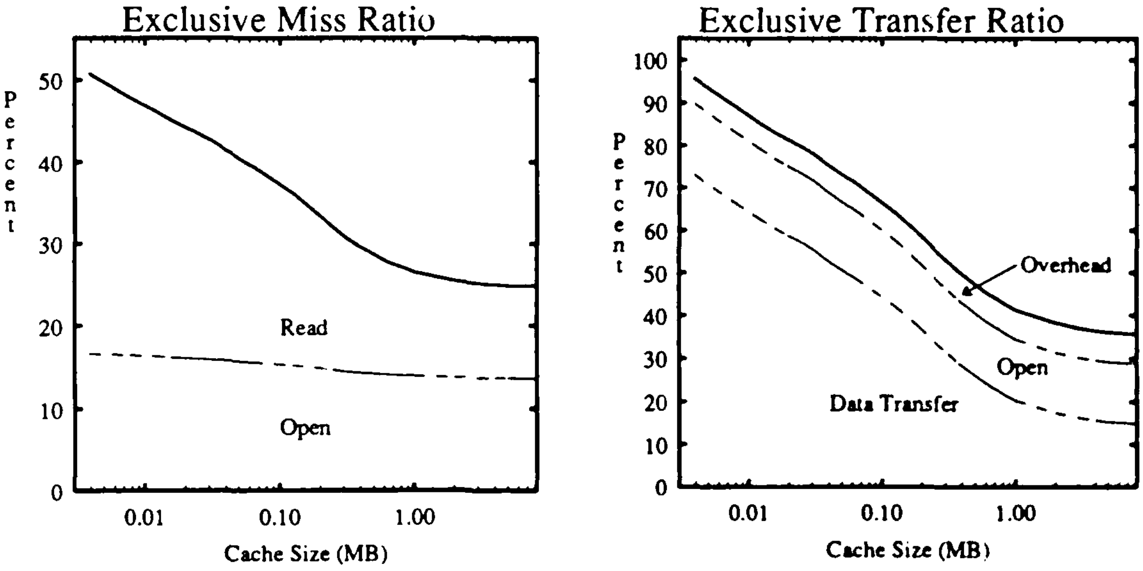


**Figure 7.2**: Exclusive protocol. Both the miss and transfer rations are reduced by the avoidance of opens and closes. Overhead increases by about 2% due to write-back requests when files become shared.

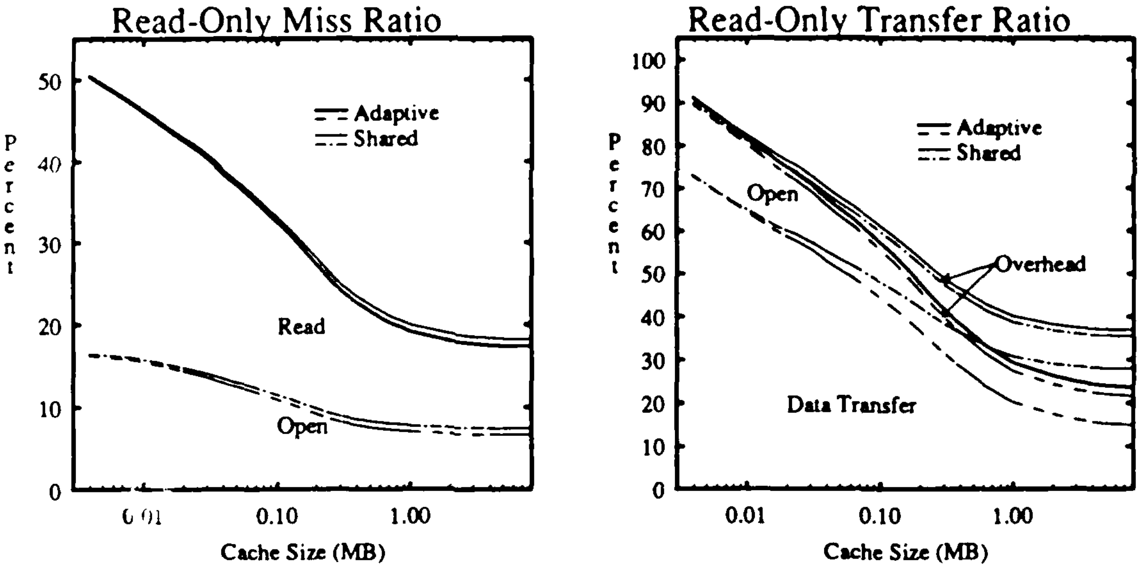
The Exclusive transfer ratio is nearly 25% less than Sprite due to the reduction in opens and overhead (primarily closes). Again, name caching could negate some of this improvement. However, it is clear that the preponderance of single-user files is easily exploitable.

Simulations were also made using the variation which did not notify the server when a dirty file was flushed. This caused a small (0.5%) increase in the transfer ratio for small cache sizes, but no measurable difference for caches above 50K.

### 7.6.3. Read-Only Protocol

The Read-Only protocol described in Section 6.7.3 favors files that are read-only, whether or not they are shared. It avoids all closes and subsequent opens as long as the file is not written. The two options after a write are to retain exclusive control, or write-back the file and assume it is again read-only. The former option is actually an adaptive protocol; with possibly one additional open, all single-user files perform the same as with the Exclusive protocol. (The additional open occurs if the single-user file is first opened for read and then for write; the Read-Only protocol must notify the server when it is opened for write, whereas the Exclusive protocol already has exclusive access to the file.)

Figure 7.3 shows the miss and transfer rations for both Read-Only options. There is almost no difference in the miss ratios for the two options. Both are 25% below the Exclusive miss ratio. The right-hand graph shows the transfer ratios and the portion due to data transfers, opens, and overhead for both options. Both options have reduced the portion attributable to opens compared to the Exclusive protocol, and overhead has almost vanished. The difference between the two options is almost entirely due to the data transfers required to write-back the file for the shared option. It is obviously better to delay the write in hopes of a re-write or deletion.


Read-Only Miss Ratio

Read-Only Transfer Ratio

**Figure 7.3**: Read-Only protocol. The Adaptive option retains exclusive access to modified files. The Shared option writes the file at close so that other clients can read it. The higher transfer ratio for the Shared option is almost entirely due to increased data transfers caused by these writes. The small overhead region (indicated by the arrows) is entirely write-back and call-back messages, reflecting the small amount of write sharing.

It is not surprising that the adaptive option is preferred since the simulation applied the same protocol to all files, and most are either exclusive or read-only. However, it is possible that the other alternative is preferable for some types of files, those which are heavily shared and written. Surprisingly, this is not the case, as seen in the following analysis.

The analysis assumed that each close after a write was a decision point — the client could either write the file immediately or delay the write — and computed the payoff in messages saved for delaying the write. If the file was subsequently rewritten or deleted before it was read by another user then the payoff was equal to the number of dirty blocks. The write of these blocks was (presumably) avoided by delaying the write. If the file was read by another user first, then the payoff was -1, since a write-back message was required, and would have been avoided if the file had been written on close. The payoff for all other cases was zero. At the end of the trace, an average payoff was computed for each type of file. If the average payoff was positive, then the write should be delayed; if negative then the file should be written immediately.

Table 7.10: Payoff By Delaying Write

| File Type | Payoff | Weighted Payoff |
|---|---|---|
| ACCOUNT | 0.04651 | 1.395 |
| ADMIN | -0.08758 | 1.65 |
| LIB-MAIL | -0.1881 | 1.232 |
| PASS | 1.75 | 5 |
| RWHO | -0.2424 | 0.9866 |
| SPOOL | -0.07143 | 1.714 |
| SPOOL-MSGS | -1 | -2 |
| TEMP | 49.38 | 125.3 |
| TEMP-EDIT | 0.4636 | 3.109 |
| TEMP-MAIL | 0.3635 | 2.893 |
| TEMP-SOFT | 6.499 | 18.1 |
| USER | 4.188 | 12.33 |
| USER-BINARY | 28.14 | 72.23 |
| USER-CC | 1.947 | 6.764 |
| USER-INCLUDE | 0.6029 | 3.416 |
| USER-MAIL | 1.538 | 5.769 |
| USER-OBJECT | 3.079 | 9.556 |
| USER-PASCAL | 4.136 | 12.2 |
| USER-TEXT | 2.406 | 7.661 |

**Table 7.10:** Payoff by delaying write-back. This table shows the expected payoff by delaying write-back, compared go writing the file on close. The latter saves a call-back message when the file is read by another client (a negative payoff). The payoff in the second column is in terms of messages saved; in the third column the messages are weighted according to Table 7.10. A positive payoff indicates that delayed write is preferred.

The analysis as presented assumed all messages are equally costly. If this is the case then several file types should be written on close, as seen in Table 7.10, although none is strongly negative. However, the computation of the transfer ratio assumed that some messages, particularly data transfers, were more expensive. When the weighting factors from Table 7.9 were applied, only one file type should be written on close, Spool-Msgs. These are public message files in */usr/msgs* which are written only once, never deleted over the short term, and always shared. The conclusion is that the adaptive option is nearly always the best.

### 7.6.4. Write Broadcast Protocol

Our final protocol, the Write Broadcast protocol described in Section 6.7.4, is designed to favor files that are shared and written, by using write broadcast to keep all copies current. However, it contains elements of the Read-Only protocol, such as not performing read closes or opens, and not broadcasting when a file is exclusively held. Therefore it should perform well for most files.

Figure 7.4 shows this to be the case. The miss ratio is one-third lower than the miss ratio for the Read-Only protocol, due almost entirely to a reduction in reads. This reduction is a result of the fact that client caches are not invalidated on write, and therefore do not have to re-fetch cached files. The transfer ratio, on the other hand, is slightly greater than the Read-Only transfer ratio. Although there are fewer data transfers, there is more overhead compared to Read-Only, arising from two sources. The first are the write-backs when an exclusive file first becomes shared, since the Exclusive protocol is used while only one cache has the file. The second are the closes after a write. The latter could be avoided with some adjustment to the protocol, but only if all caches use the same protocol (a relatively minor concession).

Although this protocol does very well, keep in mind that it assumes a "cheap" reliable broadcast protocol exists. Specifically, the simulation does not count acknowledgement messages for this or any other protocol, even though acknowledgements are probably more expensive with reliable broadcast. The goal here was to demonstrate the potential for improvement, which has been done.

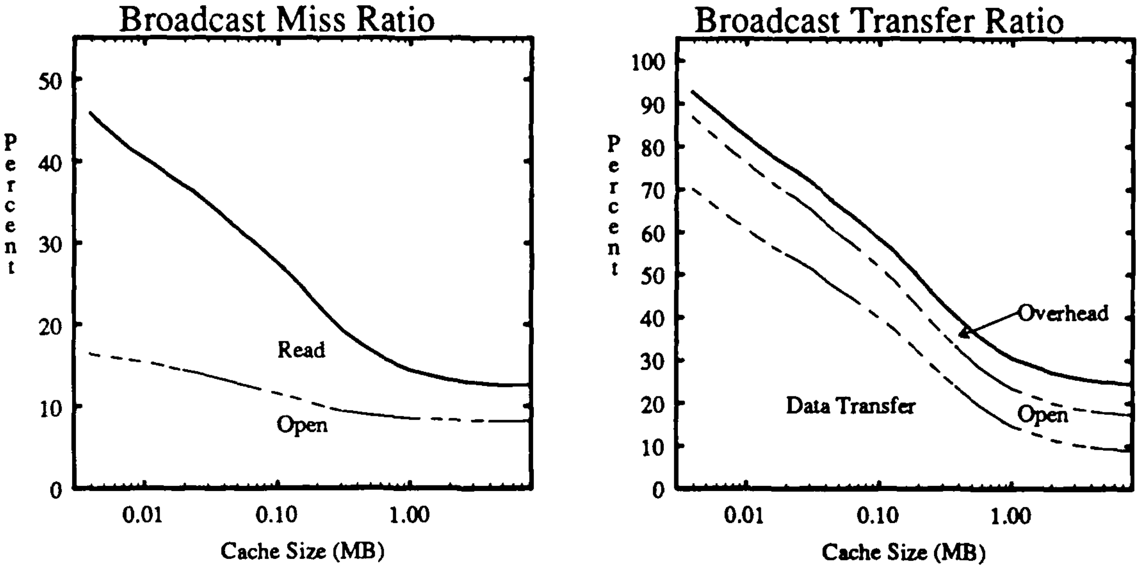


**Figure 7.4:** Write broadcast protocol. The miss ratio is the lowest of any protocol, since clients do not have to invalidate and re-read modified files. The overhead includes closes of written files. The transfer ratio does not include the costly acknowledgements required by a reliable broadcast media.

### 7.6.5. Optimal Protocol

Each of the previous protocols improved on its predecessor by some amount. It is interesting to know how long this can continue — what is the best that any protocol can do? This section describes a protocol that is in some sense optimal.

We argue that the protocol is optimal by showing that it minimizes each of the actions which form the miss and transfer ratios — opens, closes, reads, writes, and other consistency actions. First, the protocol requires the first open for each file from each cache to notify the server in order to perform name resolution. No other opens or closes are required as long as the file is cached. No protocol could cause fewer opens or closes unless a client had built-in knowledge of the file system. All reads are satisfied from cache if possible, otherwise they are satisfied from the server, as usual. Reads are minimized because the protocol does not do invalidation, so any block only has to be fetched once (until it is replaced). Writes are minimized by modifying the block in cache and delaying the write-back as long as possible, until *just before* another cache reads the file. At that time all and only the dirty bytes are broadcast to all clients, updating all cached copies. (It is possible to do fewer read fetches if a client is allowed to anticipate the need for a block and read the data off the network when another cache fetches it. Again, this requires built-in knowledge that we do not allow.) The protocol assumes that a cheap reliable broadcast medium is used; each broadcast ''costs'' a single data transfer, regardless of the number of clients involved. There is no overhead in the form of write-back or call-back messages; the cache simply ''knows'' when to broadcast the changes. This is therefore almost certainly an unrealizable protocol. Simulation of the protocol is possible since the simulator sees all requests and can broadcast when appropriate. As a simplification, the simulation uses LRU replacement for this as well as all other protocols. Use of a MIN replacement algorithm could have reduced the miss ratio somewhat [Matt70, Smit76], but would have introduced an extraneous variable.

Figure 7.5 compares all of the consistency protocols, including the optimal. Two things are apparent. First, the choice of consistency protocol can have a marked affect on performance, whether measured in terms of delay by the miss ratio, or server load by the transfer ratio. For example, the Read-Only protocol, which does not require any optimistic assumptions about network overhead, has a miss and transfer ratio which is half that of the Sprite protocol for large caches. The second observation is that there is a definite potential for further improvement. The Optimal protocol has a miss and transfer ratio which is about half that of the best of the other protocols for large caches. Additional research is needed, particularly into reasonable broadcast protocols. Since sharing is, if anything, less in the environment we measured, and broadcast performs well, it has potential for greater improvement in other environments.

## 7.7. Write Policy

Next we consider different write policies, varying from writing the data immediately to the server (write-through) to delaying the write until the block is replaced (write-back). As discussed in Chapter 3, write-through severly limits the improvement possible with caching; using this

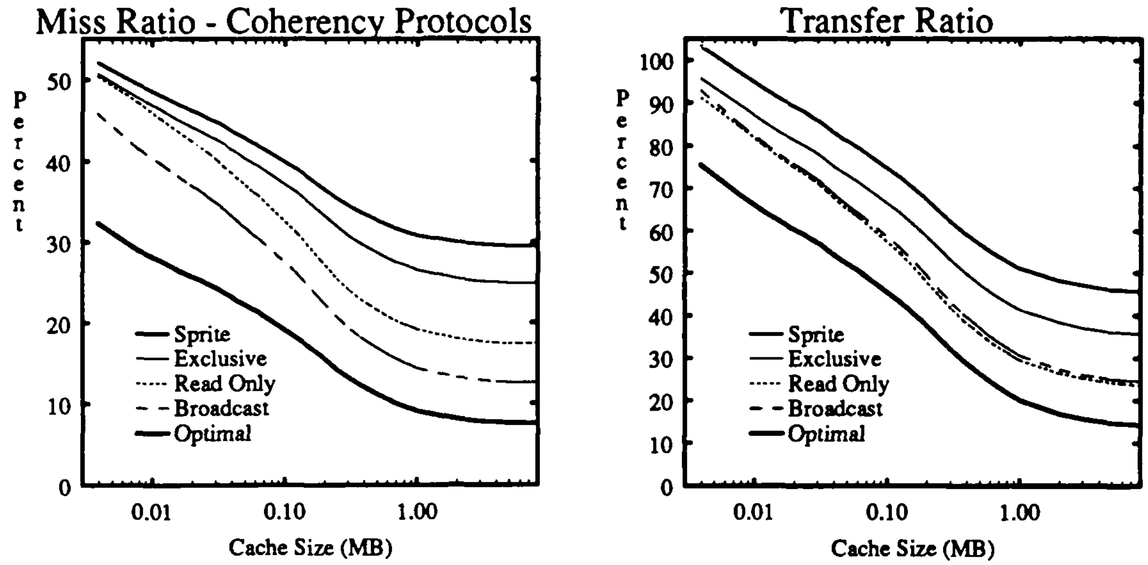


**Figure 7.5**: Comparison of consistency protocols. The graphs show the miss and transfer ratios for each cache consistency protocol. The consistency protocol has a larger affect on performance than either the write or fetch policy, with a 20-30% absolute difference between the highest and lowest protocols. The gap between the optimal protocol and the others shows the potential for improvement in future protocols.

policy Figure 7.6 shows that data transfers account for nearly two-thirds of the transfer ratio. The problem with the other extreme (write-back) is that a significant amount of data may be at risk if the client crashes. Ousterhout, et al. found that 20% of all dirty blocks remained in cache for more than 20 minutes with a cache size of 4Mb. We noted in our simulations that nearly half of the cache was dirty an any time. This section considers several intermediate write policies to address this situation. All use the Exclusive consistency protocol from section 7.6.

The first intermediate policy simulated was a *buffered write* policy which delayed the write to the server until the last byte of the block was written, or the file was closed. This is actually the policy which is used in most ''write-through'' caches; no reasonable file system would transfer partial blocks to the file server, unless it performed no buffering. The purpose of showing both this policy and the true write-through is to demonstrate the effect of this buffering. If user requests are in block-size units (i.e. if they use *stdio* exclusively), then it will make little

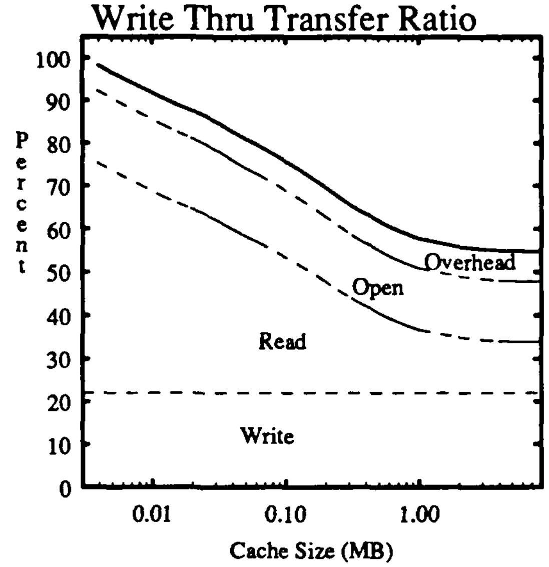


**Figure 7.6**: Write-through transfer ratio. Data transfers dominate the server costs when all writes go directly to the server. This simulation used the Exclusive consistency protocol.

difference. However, the policy will decrease the transfer ratio when the user programs perform file system I/O in smaller than block-size units, while providing high reliability in most cases. In Figure 7.7 we see that this simple policy cuts the transfer ratio nearly 10% compared to write-through. This policy is essentially equivalent to the "write-through" policy analyzed by Ousterhout et al [Oust85], since they lacked precise reads and writes, and therefore assumed that all I/O was done in blocksize units.

The next two policies simulate the UNIX policy of periodically using the *sync* command to write all dirty blocks. We simulated writes every 30 seconds and 5 minutes. Ousterhout et al. simulated these same policies and found that they reduced disk I/O by 25% and 50% respectively for a centralized server cache. In Figure 7.7 we see that they also save 25% and 50% of the possible improvement over buffered write (that is, the distance between buffered write and write-back). The reduction in the transfer ratio is actually only 3-5%.

The final two write experiment simulated a mixture of write policies. Not all files require the same degree of protection from loss: most temporary files, for example, are nearly useless if

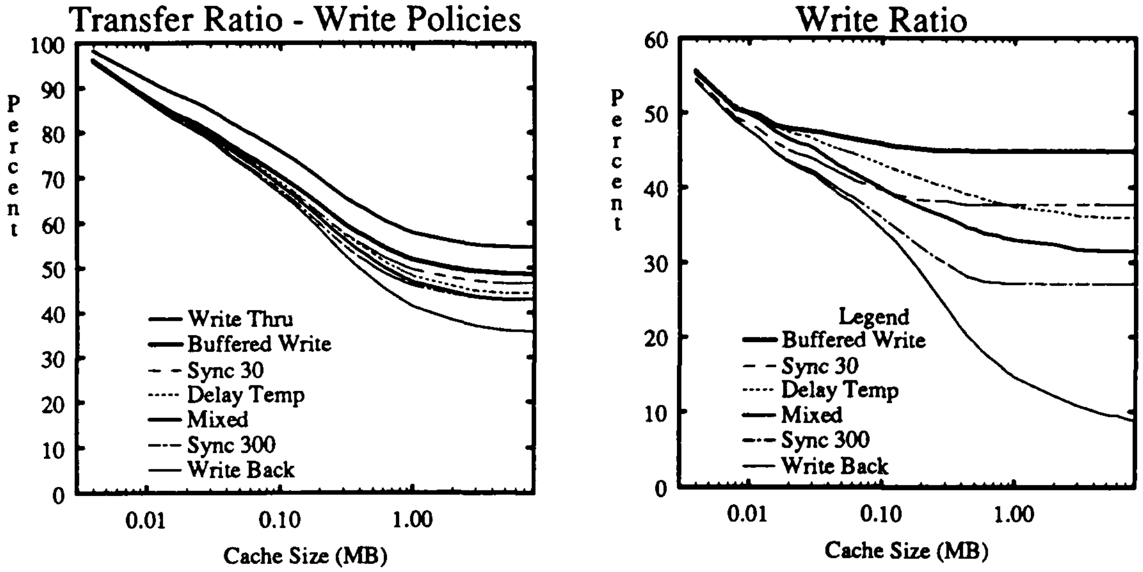


**Figure 7.7:** Comparison of write policies. The left graph shows the transfer ratio for each policy. The right graph shows the ratio of physical writes to logical writes, in percent (thus true write-through is 100%). The left end of these curves shows that nearly 45% of all writes are saved by caching a single block. Delaying all writes by 30 seconds saves 25% of the remaining writes; delaying 5 minutes saves another 25%. The Mixed policy shown in Table 7.11 performs little better than a policy that only delays Temporary files.

the client fails; derived files (such as object files) are easily recoverable; editor temporary files are intended to preserve keystrokes across a failure and therefore should be quickly written. The NEC disk cache provides this ability to mix write policies [Toku80]. We selected the combination of policies shown in Table 7.11 to provide protection appropriate to the file type. The file types that require the least reliability also account for over 40% of all bytes written. Figure 7.7 shows that the transfer ratio for this mixture is nearly indistinguishable from performing a sync every 5 minutes, but with much better reliability.

Figure 7.7 also shows the result of applying buffered-write to all except temporary files, which used write-back. This option provides high reliability to all user data, while Figure 7.7 shows that it still saves nearly 30% of all writes. The transfer ratio for this policy is nearly as good as the mixture of policies discussed above.

Table 7.11: Combination of Write Policies

| File Type | Policy |
|---|---|
| Edit Temp | Sync 1 sec |
| Other Temp | Write-back |
| Object/Executable | Sync 5 min |
| Other Files | Sync 30 sec |

### 7.8. Fetch Policy

Because of sequentiality it is often possible to anticipate references to a block and fetch it in advance [Smit78b]. There are two potential advantages to this *prefetch*. The first is in reducing access time to the user program by overlapping the fetch with processing of other blocks. Second, if the prefetched block is requested with the preceding block then a server request is saved. If neither block is cached in the server then this is also advantageous to the server, since it is nearly always less expensive to read two blocks at a time from disk (assuming a reasonable placement policy on disk [McKu84]). For example, using figures from Lazowska et al., a request to read two 1K blocks requires 2*17ms = 34ms of server CPU time; to read both blocks at once requires 29ms. There is also a possibility that the second request will be further delayed by seek or latency. Even when the blocks are cached there is a savings of the time required for the server to receive, recognize, and acknowledge a request. Based on the weightings used for this study, requesting two 4K blocks saves 12% compared to requesting each block separately (a cost of 3+8*2.5=23 compared to 2*(3+4*2.5)=26).

We simulated several prefetch policies, beginning with the UNIX lookahead policy of prefetching if sequential access has been detected. The policy is as follows: if block $i-1$ was the last to be accessed, and the current access is to block $i$, then fetch both $i$ and $i+1$ (if it exists). This policy avoids unnecessary prefetches but also loses opportunities to prefetch for any file smaller than three blocks. Ousterhout et al. [Oust85] showed that nearly 90% of all sequential runs are two blocks (8K) or less, and 80% of files are less than two blocks in length. There are therefore a great many missed opportunities.

The second policy is one that *always* prefetches the next block if it exists, a form of *one-*

*block lookahead* (OBL) [Smit78b]. The policy also fetches the first block of the file in conjunction with the open — an obvious optimization for one-block files and usually advantageous for other files.

The third policy is the complement of the UNIX prefetch; it prefetches the first block on open and continues prefetching until *non-sequentiality* is detected. It resumes prefetching if sequentiality is detected again. We refer to this as an *optimistic* prefetch, since it assumes that all files are accessed sequentially. The major advantage of this policy is the ability to perform prefetch even for small (1-2 block) files.

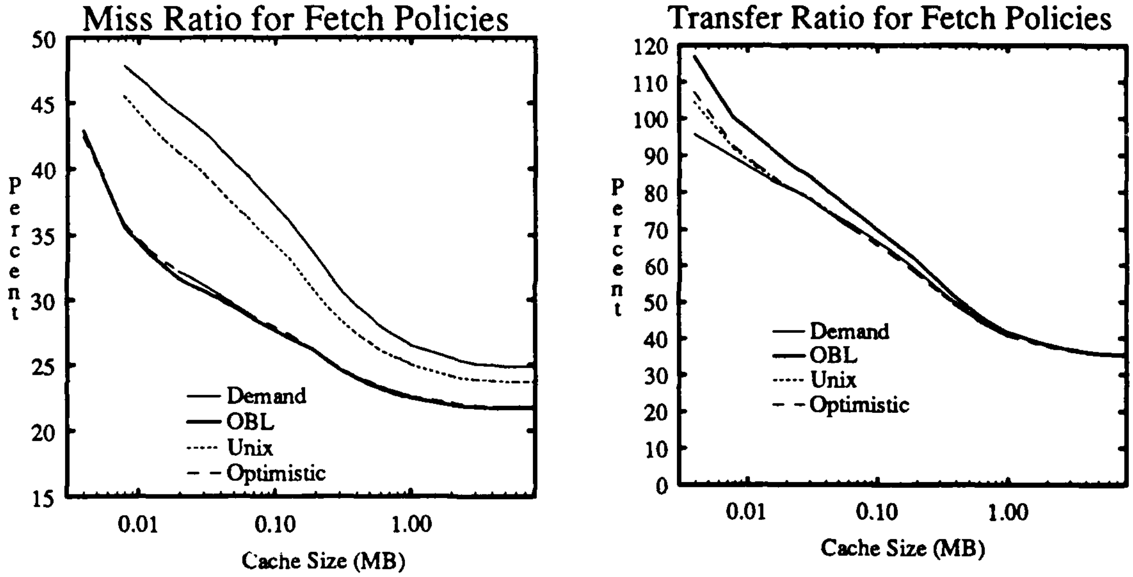


**Figure 7.8**: One-block prefetch policies. The UNIX prefetch misses many opportunities to prefetch small files, resulting in a higher miss ratio. The One-block Lookahead prefetches even for random files, producing extra data transfers.

Figure 7.8 compares these policies to demand fetch, again using the exclusive consistency protocol. The transfer ratios for UNIX prefetch and demand prefetch are virtually indistinguishable for caches larger than 100K, while optimistic is slightly lower, based on its prefetch-on-open. The OBL transfer ratio is higher because of the large number of unnecessary prefetches. The missed opportunities of UNIX prefetch are apparent in the miss ratios, which although lower than demand fetch, is well above the miss ratios for the other two policies. Keep in mind, however, that some of the reduction in miss ratio is exaggerated by the simulation assumption that all requests are completed before the next request is processed, including prefetch. If the time required to process a block is shorter than the time to fetch a block, then prefetch will only reduce, not eliminate, the fetch delay. Lazowska et al. measured the average user processing time at 106ms per 4K block for a system of SUN 2 workstations — longer than the fetch delay of 70ms [Lazo86]. Faster processors will decrease this time, but the advantage of prefetching remains.

Kent also studied the UNIX prefetch along with prefetches of multiple blocks [Kent86]. He found that fetching more than one block actually increases the miss ratio and transfer ratio (which he terms the I/O ratio) when the block size is 4K. The I/O ratio improves when multiple 512-byte blocks are fetched. None of the ratios differ by more than 2%.

One way to further reduce the number of server requests is to fetch the entire file in a single request on open. Several file systems do this [Morr86,Schr85], but each of these systems caches on local disk. A full file prefetch would appear to also reduce the miss ratio, since all blocks will be in memory when referenced. However, the fetch of a single large file could easily flush many potentially-useful blocks from cache. Also, the simulation assumption that all I/O completes at once exaggerates the decrease. A real system would probably not delay completion of the open until the entire file was fetched. Instead it would begin the processing of the first blocks while the rest is fetched.

We simulated three full-file prefetch policies. The first always prefetched the whole file, while the second prefetched only if the file was "small" (defined initially to be any file less than 100K bytes). The second policy was motivated by an observation that several large files were not read in their entirety (notably the password and network address files). The third policy prefetched the full file only if it was being read in its entirety, and is discussed below. As with all of the policies, the file was only prefetched if the file was opened for read only. (As an aside, in each case the file was fetched "backwards" so that the resulting stack had block one at the top, followed by block two, and so forth. It would be foolish for a real cache to prefetch all of a file which exceeds the size of the cache; it would surely stop when the cache was full. Our policies have this effect. In addition it ensures that the most likely blocks to be referenced are nearest the top, and the ones to be pushed if another file is referenced are the ones least likely to be read soon.)

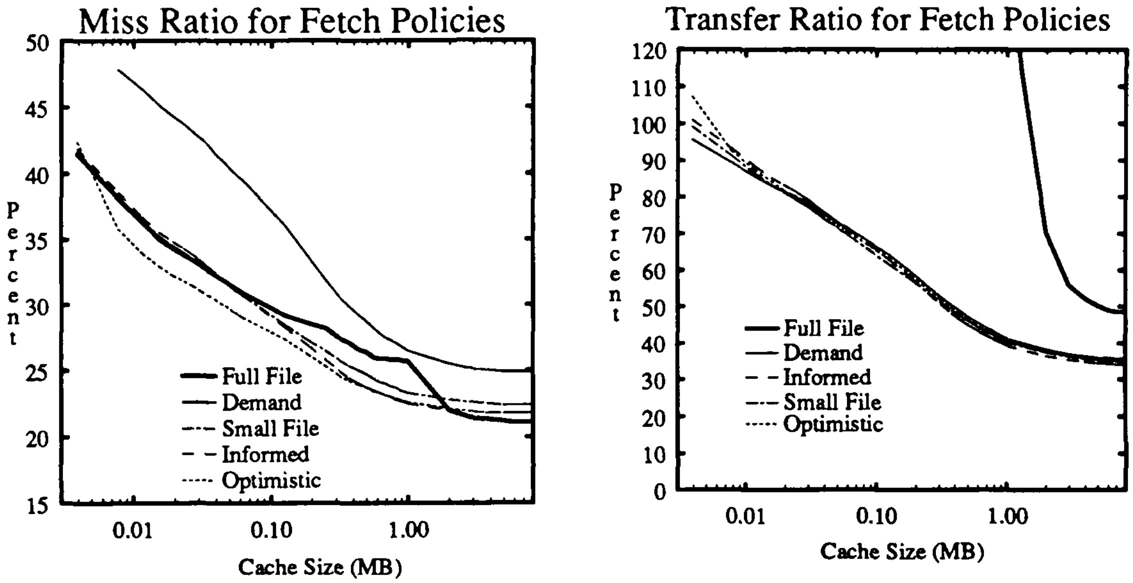


**Figure 7.9**: Full-file prefetch policies. Always prefetching the full file produces excessive server traffic, even though it has the lowest miss ratio for large caches. Both of the other policies effectively reduces the miss ratio compared to Demand fetch. The transfer ratio is decreased very little because the cost of requesting a block is relatively small compared to the cost of transferring the data.

Instead of second-guessing the user, an additional parameter could be supplied with the open to inform the file system of the intended use (e.g. "Open for random access"). To test the utility of this parameter we simulated an *informed* prefetch policy which prefetches the full file if the entire file is going to be used. The open trace records show the number of bytes accessed; we assumed that the full file was accessed if this number was larger than the file size.

Figure 7.9 shows that all three policies effectively reduce the miss ratio, although none does as well as the one-block policies for caches smaller than 2MB. This is because the one-block policies properly handle cases where two large files are being concurrently accessed, whereas the full-file fetches will displace the first file from small caches as soon as the second is opened. Although the full-file policies have a cost advantage by fetching the whole file at once, this is offset by the fact that there is a high cost for fetching blocks which are never used. The transfer ratio for the policy which always fetches the full file is so poor that it is off the graph for most sizes. On the other hand, the small file fetch does well for all cache sizes. The informed prefetch also does well, but only for large cache sizes.

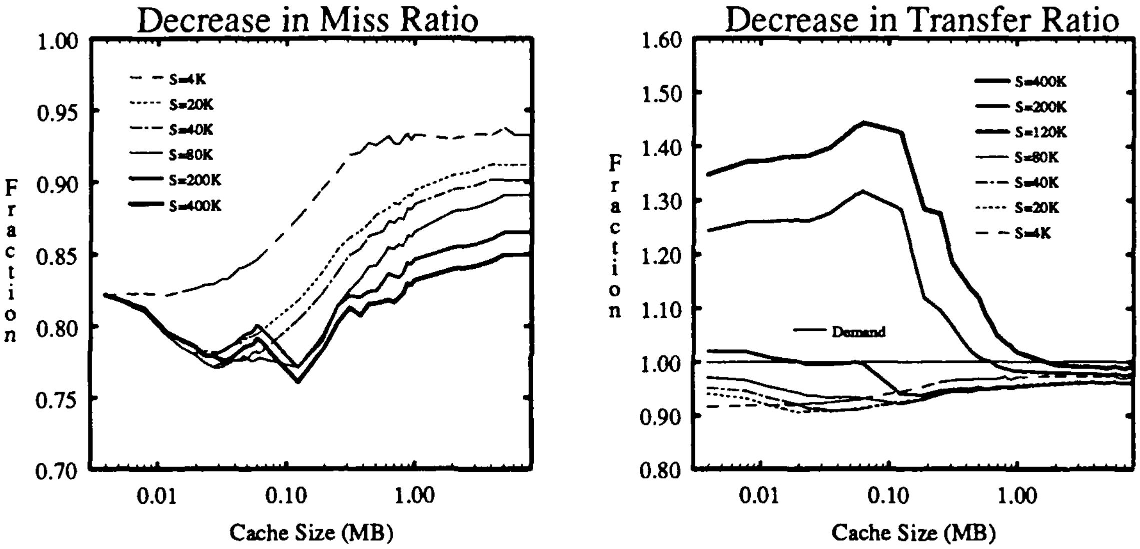


**Figure 7.10**: Effect of varying the definition of a "small" file. Each curve is normalized by dividing by the Demand miss or transfer ratio to accentuate the differences. The miss ratio varies little between values, while the transfer ratio increases dramatically when files larger than 120K are prefetched. These files tend to be accessed randomly, while the cost of prefetching them is high.

The small file simulations were repeated varying the definition of "small" from one block (4K) to 100 blocks (400K). The results are shown in Figure 7.10. Each experiment is displayed as a fraction of the demand miss or transfer ratios. We see for cache sizes less than 1MB that prefetching files larger than 120K dramatically increases the transfer ratio. The miss ratio continues to drop as the definition increases, indicating that some large files are being accessed sequentially. (Note that the prefetch of an unneeded block increases the miss ratio only if the block it displaces is needed, whereas it always increases the transfer ratio substantially, due to the high cost of data transfers.) Based solely on this data, our initial definition of small as 100K and prefetch of small files is reasonable. If extended to the server, the success of the small-file prefetch for large caches implies that a careful placement of mid-sized files (10K-100K, say) so that they can be easily fetched with one or a few I/O's would be beneficial. This falls in the category of server caching, where many similar studies are possible.

## 7.9. Conclusions

This chapter has demonstrated the wide variety of designs which can be efficiently simulated using one-pass techniques. In the process it has shown that the choice of consistency protocol can be a key factor in file system performance. The relative efficiency of the Read-Only protocol shows that the protocol should be tuned to perform well for files which are shared but not written. We also conclude that an optimistic approach to prefetch is best in most cases.

# Chapter 8

# Conclusions and Future Directions

## 8.1. One-Pass Analysis

In this dissertation we have described innovative techniques for efficiently analyzing a wide variety of cache designs, and used these techniques to study caching in a network file system. If there is a key result to this research it is this: It is possible using one-pass techniques to analyze all cache sizes in at most twice the time required to analyze a single size. Furthermore, the techniques are completely general and can be applied equally well to studies of processor caches, virtual memory, or file system caches.

The key to one-pass analysis, as we have said repeatedly, is the inclusion property -- if something is true for cache size $k$ then it is true for all sizes larger than $k$. Inclusion was first noticed by Mattson, et al. [Matt70] who used the inclusion property of block validity as the basis for their stack analysis technique. In Chapter 3 we showed that inclusion applies to dirty blocks, and used this to describe an algorithm to analyze the effects of writes in a write-back cache. We further showed that periodic write-back and block deletions could be similarly analyzed. The discovery of these techniques is particularly timely, since bus and network limitations have currently increased the impetus to use write-back caches.

Chapter 3 also showed that sub-block caches can be analyzed in one pass, by observing that sub-block validity obeys inclusion. It also proved that load-forward prefetching is a one-pass algorithm.

This dissertation has also extended one-pass analysis into a new dimension -- multiple caches which must be kept consistent. Chapter 5 described algorithms for a class of multiprocessor consistency protocols for snooping caches. The key observation was that the states used to maintain consistency are composed of three characteristics -- validity, ownership, and sharing -- each of which obeys inclusion. Several restrictions, necessary to allow one-pass analysis, were described. The main restriction was that all cache sizes must be related to one another, the most practical special case being that all caches are the same size. Given these restrictions, it is possible to compute miss ratios and bus traffic for all cache sizes in one pass.

Chapter 6 completed the discussion of one-pass algorithms by extending the work of Chapter 5 to network file system caches. It described the differences between multi-processor and network file system caches, including the higher-level open/close operations and the unreliable nature of the network. It then defined the state space and a class of consistency protocols containing several options, including both invalidation and write-broadcast protocols. The tradeoffs among options were discussed in the description of four subset protocols: a simple protocol; a protocol optimized for private files; one optimized for read-only files; and a write-broadcast protocol. The chapter concluded by showing that these are one-pass algorithms.

## 8.2. Simulation Techniques

In addition to the discovery of new one-pass algorithms, this dissertation also discussed techniques for implementing a one-pass simulator. The most time consuming operation in any one-pass or stack simulation is finding the referenced block and computing its depth in the memory stack, or stack distance. Chapter 4 described several implementations of the memory stack proposed by others, including linked-list, linked-list with hash table, static binary tree (called a B&K tree after its proponents, Bennett and Kruskal), and balanced AVL tree. We also proposed a hybrid structure combining a linked-list for common accesses near the top of the stack, with a B&K tree for accesses with high stack distance. In tests against various traces, including program address traces, disk address traces, and logical file system traces, we found that our hybrid approach performed well against all types. However the improvement was only marginal. For program address traces which have low mean stack distances, the simple linked-list implementation is completely adequate, while a more complicated B&K tree is required for disk or file system traces which have high mean stack distances.

### 8.3. Results

This dissertation was not limited to a discussion of techniques; it presented numerous results from the application of the techniques. The following summarize these results, not necessarily in the order presented in the text.

#### 8.3.1. Network File System Consistency Protocols

Perhaps the most significant result was the comparison of file system consistency protocols. Applying all of the techniques discussed in earlier chapters, Chapter 7 showed that the proper selection of protocol can cut the miss and transfer ratio by more than half. The study is the first to consider all the costs and potential benefits of caching, including savings in Open/Close. The best overall protocol was the Read-Only protocol, which is an adaptive protocol favoring read-only files and private files, in that order. The Write-Broadcast protocol also performed very well. Significantly, both the Read-Only and Write-Broadcast protocols had almost double the miss and transfer ratios of an optimal protocol. This indicates a potential for improved protocols through future research (See below).

#### 8.3.2. Other File System Simulation Results

Chapter 7 also presented a comparison of write policies for client file system caches. It concluded that delaying write-back by 30 seconds could reduce the number of writes to the server by 25%, although the resulting decrease in transfer ratio was less than 10%. More significant was the result that delaying just temporary files produced nearly the same savings, with little risk since temporary files are generally useless after a system crash. Chapter 7 also discussed prefetch policies. It concluded that the UNIX prefetch (prefetch the next block after noticing sequentiality) misses a lot of opportunities for effective prefetch, and that an optimistic prefetch (prefetch until noticing non-sequentiality) performs much better. An analysis of full-file prefetch showed that always fetching the full file on open is a poor policy if not restricted; the prefetch of large random files unnecessarily increases network traffic and displaces other needed files. If restricted to small files (less than 100K), fetching the full file on open reduces the miss ratio by 20% compared to demand fetch, with little increase in transfer ratio.

### 8.3.3. File Sharing

Chapter 7 also contained a study of file sharing, one of the important characteristics affecting the performance of the consistency protocols. It showed that actual concurrent sharing of files is very rare. Sequential sharing, which also requires consistency controls, is more prevalent; only 5% of files are shared but they account for two-thirds of all opens. Over half of the shared opens are to read-only files, partly explaining the success of the Read-Only protocol. Also, over 80% of all writes were to single-user files.

### 8.3.4. Probability of Write-Back

The write-back analysis technique presented in Chapter 3 provides a method to verify Smith's conjecture about the probability that a replaced block is dirty as a function of cache size. This probability can be an important parameter of a queuing network model of a cache. Chapter 3 generally confirmed the suggestion that the probability increases with cache size, with some exceptions. Two LISP programs, containing many read-only blocks, showed a generally declining probability. An interesting future topic might be to explain and try to exploit this characteristic.

## 8.4. Future Work

### 8.4.1. Adaptive MOESI Protocols

In Chapter 5 we restricted the MOESI protocols to consistently make the same choice for a given state and action, because of a few cases where use of different rules violated inclusion. However, in Chapter 7 we showed that an adaptive protocol performed the best. It would be interesting to explore adaptive subsets of the MOESI protocol. For example, Sweazey and Smith [Swea86] suggest that a cache invalidate itself when it receives a write-broadcast if the block priority is less than some value (e.g. if the block has not been used in $X$ seconds). Intuitively, this would still be a one-pass algorithm since the priority applies to all sizes and therefore the block would be invalidated in all sizes.

Another adaptive protocol has been suggested by Karlin, et al. [Karl86]. It attempts to balance the costs of invalidation and refetch against the cost of write broadcast by counting the number of reads between writes and writes between reads and switching protocol whenever one dominates. Several writes between reads favors invalidation, while several reads per write favors write-broadcast. Again, we believe intuitively that this would be a one-pass algorithm, provided the choice is the same for all cache sizes. However, since the counts used to make the choice are reset when a block is replaced in cache, the choice probably *is* a function of cache size. Future work should formulate this adaptive policy in terms of the MOESI protocols, attempting to structure the statistics so that it is a one-pass algorithm, then evaluate it against real traces.

### 8.4.2. Broadcast File System Protocols

In Chapter 7 we showed that the Write-Broadcast protocol performed very well compared to other protocols. However, the simulation made very optimistic assumptions about the over*head of the underlying reliable broadcast protocol.* Write broadcast shows great promise because it reduces the both the miss ratio and the server involvement, and the underlying networks are broadcast-capable. There should be continued study of reliable broadcast for use by a network file system along the following lines. First is a need to quantify the need for write broadcast; how many files and accesses are most efficiently managed using broadcast. Second, design protocols both at the file system and underlying network level suited to this need. Here again, an adaptive protocol may be useful, since many files do not require write broadcast. Finally, the simulations should be repeated using the new protocols and more realistic assumptions about the network overhead.

### 8.4.3. Name Caching

Most of the improvements in the miss and transfer ratio reported in Chapter 7 were due to the avoidance of opens, which can only be done if clients cache file names. There is a need for research into the techniques and cost of efficient name caching. The costs of name caching include the cost of obtaining, storing, and searching the names, as well as the cost of keeping the cached names consistent. The latter must account for a name change or deletion by another client, as well as changes to higher-level directories along the path.

There are two obvious alternatives to implementing name caching: caching the names themselves, and caching the directories containing the names. Caching directories is conceptually the easiest; just treat the directories as files and use the existing file consistency mechanism. This is essentially the method used by the SUN NFS [Sand85]. Directory caching can increase the server involvement since each directory in the file-name path may require a server access. Floyd reported that each path averaged 2.5 components [Floy86b]. On the other hand, once the directory working set has been loaded, the name cache misses will be lower than if just the names are cached, since users tend to work in one directory for a time. Floyd also reported that a global cache of ten directories achieved an 85% hit ratio, and that thirty entries achieved a 95% hit ratio [Floy86b]. Caching of directories also increases the probability that changes by one user will interfere with another user, particularly in directories such as */tmp*.

Caching just the file names reduces the space requirements, and eliminates the effort required to load the directories to the client. It also has lower interference between users. On the other hand, it may require more server overhead, since the server has more objects to manage and search. It is possible to recover some of the advantages of directory caching by managing ranges of names, as done in the Quicksilver file system [Cabr87]. All of these options need to be analyzed, including the default option of handling all opens at the server.

### 8.4.4. Server Caching

There have been several studies involving centralized file system caches, and we studied client caching, but there have been no comprehensive studies of server caching when combined with client caching. The basic question is whether there is sufficient locality remaining in the client "misses" to justify a large cache. Certainly the client caches will absorb much of the locality of reference to user files, but we showed in Chapter 7 that there are many references to shared system files which may still benefit from caching. If client caches are invalidated when a new version is written, caching of the new version in the server is advantageous to other clients refetching a file; this advantage is not present with write-broadcast.

In addition to looking at cache size, it is useful to explore other parameters of server caches when combined with client caching. There is no reason, for example, that the clients and server must use the same write or fetch policy. If the client delays writes then the server certainly shouldn't. Also, if the file is properly placed on disk then there is little cost for the server to fetch all of a small file on open, even if it is not advantageous for the client to do so. Placement policies also need to be considered; if the client delays writes then the entire file may be available to the server so that it can be placed together on disk for quicker writing and fetching.

Either client or server caches of varying size can be simulated in one pass, but we know of no way to do both at once, since the server request stream clearly depends on the client cache sizes. The technique which minimizes simulations is to run the client simulation for a particular set of cache sizes and collect the sequence of misses and writes. This sequence then becomes the trace input to a one-pass simulation of the server.

### 8.4.5. More Data

Every dissertation such as this concludes that there should be more study with different data; we concur. One of the values of this dissertation is that it presents efficient ways to analyze the data; unfortunately it doesn't make gathering the data any easier. Two types of additional data are needed: data from other environments and data from other systems. There is a real need to gather data from environments other than a university setting, particularly ones where there are stronger group interactions that might reveal different types of sharing. There is also a need to gather data from a network file system, particularly one using high-performance workstations, in order to affirm that file access patterns are the same, or if there are unique file types (font files, for example) with very different access patterns and requirements.

### 8.5. Summary

This dissertation has explored one-pass techniques for performing trace-driven analyses of cache performance, and presented a few interesting results. The techniques greatly expand the range of cache designs which can be efficiently simulated to include write-back caches, sub-block caches, and multi-processor caches using various cache consistency protocols. The possible simulation results have been expanded to include the effects of writes, deletions, and bus traffic required to maintain cache consistency. Using these techniques, a researcher can analyze the performance of all cache sizes in little more time than it takes to analyze a single size. The techniques are completely general and can be applied to all types of caches, from processor caches to disk and file system caches.

The dissertation concluded with the application of the techniques to a study of cache consistency algorithms, write policies, and fetch policies for a distributed file system. The study showed that the cache consistency protocol can have a major effect on system performance, and suggests that further study is needed to refine the protocols offered here. Again, the significance of this dissertation is that it offers the means for others to efficiently explore these new protocols and a variety of other cache parameters.

# Chapter 9

## References

[Adle66] Adel'son-Vel'skii, G. M. and Y. M. Landis, "An algorithm for the orgnization of information", *Dokl. Aked. Nauk. USSR 146* (1966). 263-266.

[Arch86] Archibald, James and Jean-Loup Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model", *ACM Transactions on Computer Systems 4*, 4 (November 1986), 273-298.

[Bela66] Belady, L. A., "A Study of Replacement Algorithms for Virtual Storage Computers", *IBM Systems Journal 5* , 2 (1966), 78-101.

[Bela69] Belady, L. A., R. A. Nelson and G. S. Shedler, "An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine", *Communications of the ACM 12*, 6 (June 1969), 349-353.

[Benn75] Bennett, B. T. and V. J. Kruskal, "LRU stack processing", *IBM Journal of Research and Development*, July 1975, 353-357.

[Birr84] Birrell, A. D. and B. J. Nelson, "Implementation of Remote Procedure Call", *ACM Transactions on Computer Systems 2*, 1 (February 1984), 39-59.

[Cabr87] Cabrera, Luis F. and Jim Wyllie, "QuickSilver Distributed File Services: An Architecture for Horizontal Growth", IBM Res. Rep RJ 5578, IBM Almaden Research Center, April 1987.

[Chan84] Chang, J. M. and N.F. Maxemchuk, "Reliable Broadcast Protocols", *ACM Transactions on Computer Systems 2*, 3 (1984), 251-273.

[Cher83] Cheriton, David and Willy Zwaenapoel, "The distributed V kernel and its performance for diskless workstations", *Proc 9th Symposium on Operating System Principles*, 1983, 128-140.

[Cher87] Cheriton, David and Carey Williamson, "Network Measurements of the VMTP Request-Response Protocol in the V Distributed System", *Proc. 1987 Sigmetrics*, Banff, Alberta Canada, May 1987, 216-225.

[Cho86] Cho, James, Howard Sachs and Alan Jay Smith, "The Memory Architecture and the Cache and Memory Management Unit for the Fairchild CLIPPER Processor", *University of California, Berkeley/Computer Science Department 86/289*, March, 1986. Submitted for publication..

[Coff71] Coffman, E. G. and B. Randell, "Performance Prediction for Extended Paged Memories", *Acta Informatica 1*, 1 (1971), 1-13.

[Denn72] Denning, Peter J., "On modeling program behavior", *Proc. Spring Joint Computer Conference*, 1972, 937-944.

[East78] Easton, E J., "Computation of Cold Start Miss Ratios", *IEEE Transactions on Computers C-27*, 5 (May 1978), 404-408.

[Eswa76] Eswaran, K. P. and et al., "Consistency and Predicate Locks in Database Systems", *Communications of the ACM 19*, 11 (November 1976), 624-633.

[Ferr78] Ferrari, Domenico, *Computer Systems Performance Evaluation*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Floy86a] Floyd, Rick, "Short-term file reference patterns in a UNIX environment", Technical Report 177, Univ. of Rochester, Mar 1986.

[Floy86b] Floyd, Rick, "Directory reference patterns in a UNIX environment", Technical Report 179, Univ. of Rochester, Aug 1986.

[Fost73] Foster, C. C., "A Generalization of AVL Trees", *Communications of the ACM 16* (1973), 513-517.

[Fran84] Frank, Steven J., "Tightly Coupled Multiprocessor System Speeds Memory Access Times", *Electronics*, January 12, 1984, 164-169.

[Fran77] Franta, W. R. and Kurt Maly, "An Efficient Data Structure for the Simulation Event Set", *Communications of the ACM 20*, 8 (Aug. 1977), 596-602.

[Gecs74] Gecsei, J., "Determining Hit Ratios in Multilevel Hierarchies", *IBM Journal of Research and Development 18*, 4 (July 1974), 316-327.

[Gibs86] Gibson, G., Personal communication.

[Good83] Goodman, J. R., "Using Cache Memory to Reduce Processor-Memory Traffic", *Proc. Tenth Int'l. Symp. on Computer Architecture*, Stockholm, Sweden, June 1983, 124-131.

[Gree74] Greenberg, Bernard S., "An Experimental Analysis of Program Reference Patterns in the Multics Virtual Memory", MAC Technical Report-127, MIT, Project MAC, Cambridge, MA, Jan. 1974.

[Gros85] Grossman, C. P., "Cache-DASD storage design for improving system performance", *IBM Systems Journal 24*, 3/4 (1985), 316-334.

[Hill84] Hill, Mark D. and A. J. Smith, "Experimental evaluation of on-chip microprocessor cache memories", *Proc. 11'th Int'l. Symp. on Computer Architecture*, Ann Arbor, Michigan, June 1984, 158-166.

[Hors83] Horspool, R. N. and R. M. Huberman, "Demand Prepaging Algorithms with the Memory Inclusion Property", *Proc. 16'th Ann. Hawaii Int'l. Conf. on System Sciences*, 1983, 138-145.

[Karl86] Karlin, Anna R., Mark S. Manasse, Larry Rudolph and Daniel D. Sleator, "Competitive Snoopy Caches", *IEEE Annual Symposium on Foundations of Computer Science*, 1986.

[Katz85] Katz, R., S. Eggers, D. A. Wood, C. Perkins and R. G. Sheldon, "Implementing a Cache Consistency Protocol", *Proc. 12'th Int'l. Symp. on Computer Architecture*, Boston, Mass., June 1985, 276-283.

[Kent86] Kent, Chris, *Cache Coherence in Distributed Systems*, PhD Dissertation, Purdue University, August 1986.

[Knut68] Knuth, D. E., *Fundemental Algorithms*, Addison-Wesley Publishing Co., Reading, MA, 1968.

[Knut73] Knuth, D. E., *Sorting and Searching*, Addison-Wesley Publishing Co., Reading, MA, 1973.

[Kubo75] Kubo, Hidehito and Makoto Kobayashi, ''A Cost-Oriented Approach to Optimal Page Size'', *Proc. Second USA-Japan Computer Conference*, 1975, 258-263.

[Lazo86] Lazowska, E. D., J. Zahorjan, David Cheriton and W. Zwaenpoel, ''File Access Performance of Diskless Workstations'', *ACM Transactions on Computer Systems 4* , 3 (Aug 1986), 238-268.

[Lipt68] Liptay, J. S., ''Structural Aspects of the System/360 Model 85, Part II: The Cache'', *IBM Systems Journal 7 (1)* (1968), 15-21.

[Lucc76] Luccio, F. and L. Pagli, ''Rebalancing in Height Balanced Trees'', *IEEE Transactions on Computers TC-25* (1976), 87-90.

[Matt70] Mattson, R. L., J. Gecsei, D. Slutz and I. L. Traiger, ''Evaluation Techniques for Storage Hierarchies'', *IBM Systems Journal 9*, 2 (1970), 78-117.

[McCr84] McCreight, Edward M., ''The Dragon Computer System: An Early Overview'', Technical Report, Xerox PARC, June, 1984.

[McKu84] McKusick, Marshall K. and et al., ''A Fast File System for UNIX'', *ACM Transactions on Computer Systems 2*, 3 (August 1984), 181-197.

[Morr86] Morris, J. H. and et al., ''Andrew: A Distributed Personal Computing Environment'', *Communications of the ACM 29* , 3 (March 1986), 184-201.

[Nels87] Nelson, Michael, Brent Welch and John Ousterhout, ''Caching in the Sprite Network File System'', University of California, Berkeley/Computer Science Department 87/345, March 1987. To appear in ACM Transactions on Computer Systems.

[Olke81] Olken, F., *Efficient Methods for Calculating the Success Function of Fixed Space Replacement Policies*, Master's Report, University of Cal., Berkeley, Cal., May 1981.

[Oust85] Ousterhout, John K., Herve' DaCosta, David Harrison, John A. Kunze, Mike Kupfer and James G. Thompson, "A Trace-Driven Analysis of the UNIX 4.2BSD File System", *Proc. Tenth Symposium on Operating System Principles*, Dec. 1985, 15-24.

[P896] , IEEE P896.1 Draft Standard, Backplane Bus, (Futurebus), November, 1986.

[Papa84] Papamarcos, Mark and Janak Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories", *Proc. 11'th Int'l. Symp. on Computer Architecture*, Ann Arbor, Mich., June, 1984, 348-354.

[Patt83] Patterson, D.A., P. Garrison, M.D. Hill, D. Lioupis, C. Nyberg, T.N. Sippel and K.S. Van Dyke, "Architecture of a VLSI Instruction Cache for a RISC", *Proc. Tenth Int'l Symp. on Computer Architecture*, Stockholm, Sweden, June 1983, 108-116.

[Porc82] Porcar, Juan M., *File Migration in Distributed Computer Systems*, PhD Dissertation, University of California, Berkeley, 1982.

[Rama86] Ramakrishnan, K. K. and Joel S. Emer, "A Model of File Server Performance for a Heterogeneous Distributed System", *Proc. Communications Architectures and Protocols 16 3* (Aug. 1986), 338-347.

[Ruan87] Ruan, Zuwang and Walter F. Tichy, "Performance Analysis of File Replication Schemes", *Proc. 1987 Sigmetrics*, Banff, Alberta Canada, May 1987, 205-215.

[Rudo84] Rudolpf, Larry and Zary Segall, "Dynamic Decentralized Cache Schemes for MIMD Parallel Architectures", *Proc. 11'th Int'l. Symp. on Computer Architecture*, Ann Arbor, Mich., June, 1984, 340-347.

[Sand85] Sandberg, R. and et al., "Design and Implementation of the Sun Network Filesystem", *Proc. of the USENIX 1985 Summer Conf.*, June 1985, 119-130.

[Saty81] Satyanarayanan, M., ''A Study of File Sizes and Functional Lifetimes'', *Proc. 8th Symposium on Operating System Principles*, Dec. 1981.

[Saty85] Satyanarayanan, M. and et al., ''The ITC Distributed File System: Principles and Design'', *Proc. Tenth Symposium on Operating System Principles*, Dec. 1985, 35-50.

[Schr85] Schroeder, M. D., D. K. Gifford and R. M. Needham, ''A Caching File System for a Programmer's Workstation'', *Proc. Tenth Symposium on Operating System Principles*, Dec. 1985, 25-34.

[Shed76] Shedler, G. S. and D. R. Slutz, ''Derivation of miss ratios for merged access streams'', *IBM Journal of Research and Development 20*, 5 (Sept. 1976), 505-517.

[Silb83] Silberman, G. M., ''Stack Processing Techniques in Delayed-Staging Storage Hierarchies'', *Communications of the ACM 26*, 11 (Nov. 1983), 999-1007.

[Slea85] Sleator, D. O and R. E. Tarjan, ''Self-Adjusting Binary Search Trees'', *Journal of the ACM 32*, 3 (July 1985), 652-686.

[Slut72] Slutz, D. R. and I. L. Traiger, ''Determining Hit Ratios for a Class of Staging Hierarchies'', IBM Res. Rep RJ 1044, May 1972.

[Smit76] Smith, A. J., ''Analysis of the Optimal, Look Ahead, Demand Paging Algorithms'', *Siam Journal on Computing 5*, 4 (December 1976), 743-757.

[Smit77] Smith, A. J., ''Two Methods for the Efficient Analysis of Memory Trace Data'', *IEEE Transactions on Software Engineering SE-3*, 1 (January 1977), 94-101.

[Smit78b] Smith, A. J., ''Sequential Program Prefetching in Memory Hierarchies'', *IEEE Computer 11*, 2 (Dec. 1978), 7-21.

[Smit78a] Smith, A. J., ''A comparitive study of set associative memory mapping algorithms and their use for cache and main memory'', *IEEE Transactions on Software Engineering SE-4*, 2 (March 1978), 121-130.

[Smit79] Smith, A. J., ''Characterizing the storage process and its effect on the update of main memory by write-through'', *Journal of the ACM 26*, 1 (Jan 1979), 6-27.

[Smit82] Smith, A. J., ''Cache memories'', *ACM Computing Surveys 14*, 3 (Sept. 1982), 473-530.

[Smit84] Smith, Alan Jay, ''Trends and Prospects in Computer System Design'', University of California, Berkeley/Computer Science Department 84/219, June 1984.

[Smit85a] Smith, A. J., ''Cache Evaluation and the Impact of Workload Choice'', *Proc. 12'th Int'l. Symp. on Computer Architecture*, Boston, Mass., June 1985, 64-73.

[Smit85b] Smith, A. J., ''Disk cache - miss ratio analysis and design considerations'', *ACM Transactions on Computer Systems*, August 1985.

[Svob84] Svodobova, Liba, ''File Servers for Network-Based Distributed Systems'', *ACM Computing Surveys 16*, 4 (December 1984), 353-398.

[Swea86] Sweazey, Paul and Alan Jay Smith, ''A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus'', *Proc. 13th Int'l. Symp. on Computer Architecture*, Tokyo, Japan, June 3-5, 1986.

[Thom78] Thompson, Ken, ''UNIX Time Sharing System: UNIX Implementation'', *Bell System Technical Journal 57*, 6 (July-Aug. 1978), 1931-1946.

[Thom85] Thompson, J., ''File Deletion in UNIX Systems: Its Impact on File System Design and Analysis'', CS 266 term project, Computer Science Division, EECS, University of California, Berkeley, CA, Apr. 1985.

[Toku80] Tokunaga, T., Y. Hirai and S. Yamamoto, ''Integrated disk cache systems with file adaptive control'', *Proc. IEEE Computer Society Conf.*, Washington, DC, Sept. 1980, 412-416.

[Trai71] Traiger, I. L. and D. R. Slutz, ''One Pass Techniques for the Evaluation of Memory Hierarchies'', *IBM Research Report RJ 892*, Yorktown Heights, NY, July 1971.

[Welc87] Welch, Brent, Personal communication.

[Yu76] Yu, F. S., *Modeling the Write Behavior of Computer Programs*, PhD Thesis, Stanford University, Palo Alto, CA, May 1976.

[Zhou85] Zhou, S., H. Da Costa and A. J. Smith, "A File System Tracing Package for Berkeley UNIX", University of California, Berkeley/Computer Science Department 85/235, May 1985.