Compressed Caching and Modern Virtual Memory Simulation

by

Scott Frederick Kaplan, B.A.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 1999

Compressed Caching and Modern Virtual Memory Simulation

Approved by
Dissertation Committee:

For Jeni

Acknowledgments

Paul Wilson has served as my advisor for all but my first semester. He has infused me with enthusiasm for this material, and taught me a great deal about how to think, analyze, and create. He has treated me as a peer throughout my time as a graduate student, and I thank him for helping me to begin my career so well.

There have also been a number of other graduate students with whom I've worked and conversed. All of them have made my graduate experience and my background richer, have provided perspective, and have helped me towards my goals. Yannis Smaragdakis has played a significant role in much of the research presented here, and his effort, intelligence, and energy have been invaluable. It is important to note that Chapters 3, 4, and 6 were each derived from papers published with Paul Wilson and Yannis Smaragdakis. Both of them have had a significant hand in developing the work presented here.

Mark Johnstone, Sheetal Kakkad, Doug van Wieren, and Artie Gold have been involved in related research, and my interaction with them kept my interest high during a long process.

Emery Berger has been supportive of my work, included me in his life, and showed an overall enthusiasm for anything that I've managed to accomplish. He implicitly reminded me the value of interaction in this kind of work. Mike Stoff has seen me from one end of the graduate process to the other, and has seen me grow as a person during critical years. I was lucky to have him as a friend.

There are some without whom the Department of Computer Sciences would come to a cataclysmic halt. Of course, I am referring to the administrative staff. In particular, I thank Gloria Ramirez and Katherine Pecore, without whose help I could not have finished this dissertation from afar. I also want to thank Boyd Merworth, Patti Spencer, and the other system administrators who maintain a daunting network.

I want to extend a special thanks to Bill Pugh and the Department of Computer Science at the University of Maryland. When I moved from Austin, I desperately needed a place to continue my work, surrounded by others who would be interested in it. They were kind enough to provide for me access to their resources.

I cannot imagine being in this position without the constant, unwavering support and love of my parents and family. They taught me the value of education, ensured that I had every opportunity, and encouraged me to make the most of each. They supported me when I was unsure of myself, and showed their pride in everything that I have done. They are my friends, I have enjoyed sharing this experience with them.

Most of all, and no surprise, I thank my fiancée, Jeni. She has encouraged me to do what I love. She has supported me throughout the writing of this dissertation, and through other significant transitions. She has been my best friend and my most vociferous advocate. I enjoy where I am and what I do because of her.

SCOTT FREDERICK KAPLAN

The University of Texas at Austin
December 1999

Compressed Caching and Modern Virtual Memory Simulation

Publication No. ____________________

Scott Frederick Kaplan, Ph.D.
The University of Texas at Austin, 1999

Supervisors: Paul R. Wilson and Donald S. Fussell

CPU speeds have increased so quickly that few programs suffer from a lack of CPU capacity. However, many programs in all domains of computing *are* hindered by the slowness of memory at some level of the virtual memory hierarchy. One of the greatest pitfalls in system performance is that of virtual memory paging.

Improving the performance of this part of the virtual memory hierarchy is essential. However, the understanding of program reference behavior and its relation to virtual memory design and performance has not been significantly advanced in the past thirty years. Recent ideas that could improve virtual memories have been evaluated through direct implementation as part of an operating system kernel. Unfortunately, this approach does not allow for controlled, reproducible experiments that help a researcher to understand how programs use their memory.

Trace-driven simulation allows for these kinds of controlled, scientific experiments. However, the difficulties in gathering, storing, and processing large reference

traces have kept researchers from adopting this approach. This dissertation begins by outlining an approach to reducing reference traces for use in simulations of virtual memory. We have used reference traces to explore compressed caching—the insertion of a new, compressed level of RAM into the memory hierarchy. We will present the trace-driven investigations into the compression of in-memory data. We will also present a new, principled approach to making page replacement and compressed caching agilely adapt to a program's reference behavior, made possible by controlled simulations.

Contents

Acknowledgments **v**

Abstract **vii**

List of Tables **xiv**

List of Figures **xvi**

Chapter 1 Introduction **1**

1.1 Motivation . . . 1

1.1.1 Why Study Virtual Memory? . . . 1

1.1.2 Why Study Compressed Caching? . . . 2

1.1.3 Why Use Simulation? . . . 3

1.2 Virtual Memory Background . . . 4

1.2.1 A Brief History . . . 4

1.2.2 Current Use . . . 6

1.3 Trace Driven Simulation . . . 7

1.3.1 A Problem for Virtual Memory Simulation . . . 7

1.3.2 Why Simulation Should Be Used . . . 8

1.3.3 New Ideas Presented Here . . . 9

1.4 Reference Behavior and Page Replacement . . . 9

1.4.1 A Brief History 9
1.4.2 New Ideas Presented Here 11
1.5 Compressed Caching 12
1.5.1 Revisiting a Good Idea 12
1.5.2 New Ideas Presented Here 13
1.6 Road Map 14

Chapter 2 Concepts **17**
2.1 Caching, Page Replacement, and Timescale Relativity 17
2.1.1 The On-line Caching Prediction Problem 19
2.1.2 Temporal and Spatial Locality 20
2.1.3 Three Well-Known Replacement Policies 21
2.1.4 Timescale Relativity I: Virtual Memory Time and "Soonness" 23
2.1.5 Timescale Relativity II: Aggregate Behavior 36
2.1.6 Timescale Relativity III: Adaptivity and Decay 39
2.1.7 LRU Approximations for Real Systems 42
2.2 Compression 46
2.2.1 Background 46
2.2.2 Modeling 49
2.2.3 Encoding 59
2.3 Compressibility vs. Caching 65

Chapter 3 Trace Reduction **67**
3.1 Introduction 67
3.2 Background and Motivation 70
3.2.1 Overview of Related Work 70
3.2.2 The Value of Our Techniques 72
3.3 The Algorithms 75

3.3.1 Safely Allowed Drop (SAD) 75
3.3.2 Optimal LRU Reduction (OLR) 79
3.3.3 Trace Manipulation Issues 83
3.4 Experimental Results 85
3.4.1 Reduction Results 85
3.4.2 CLOCK and SEGQ Simulations 94
3.5 Comparison to the Glass and Cao Technique 111
3.6 Conclusions 115

Chapter 4 Adaptive Caching **117**

4.1 Introduction and Overview 117
4.2 Motivation and Related Work 121
4.3 The EELRU Algorithm 124
4.3.1 General Idea 124
4.3.2 A Concrete Algorithm 130
4.3.3 EELRU vs. LRU 134
4.4 Experimental Assessment 135
4.4.1 Settings and Methodology 135
4.4.2 Locality Analysis 138
4.4.3 First Experiment: Memory-intensive Applications 146
4.4.4 Second Experiment: Small-scale Patterns 158
4.5 Thoughts on Program Behavior and Replacement Algorithms 165
4.6 Conclusions and Future Work 167

Chapter 5 In-memory Data Compression **169**

5.1 Motivation and Background 169
5.1.1 Compression for Compressed Caching 171
5.2 The Wilson-Kaplan (WK) Compressors 173

5.2.1 The Modeling/Encoding Phase 174
5.2.2 The Packing Phase 183
5.3 Experimental Evaluation 187
5.3.1 Experimental Design 187
5.3.2 Results 190
5.4 Related Work 231
5.4.1 The Original WK and The Newton 232
5.4.2 X-match 232
5.4.3 Zero Removal 233
5.4.4 Parallel, Cooperative Dictionary Compression 234

Chapter 6 Compressed Caching **236**

6.1 Introduction 236
6.2 Calculating the Costs and Benefits 239
6.2.1 An Example 239
6.2.2 Generalization and Derivation 241
6.2.3 Calculations for Real Programs 245
6.3 Fixed-Size Compressed Cache Simulations 251
6.3.1 Experimental Design 252
6.3.2 Results 257
6.4 Adaptively Adjusting the Compressed Cache Size 266
6.4.1 Online Cost-Benefit Analysis 267
6.4.2 Adapting to Recent Behavior 268
6.5 Size-Adaptive Compressed Cache Simulations 269
6.5.1 Results of Detailed Simulations 272
6.5.2 Technology Trends 293
6.6 Related Work 301
6.7 Conclusions 303

Chapter 7 Conclusions **305**

7.1 Timescale Relativity and Adaptive Caching 306

7.2 Trace Reduction and Simulation 309

7.3 Compressed Caching 310

Bibliography **312**

Vita **320**

List of Tables

3.1 Even for small reduction memories, significant reduction factors can be achieved. 93

3.2 Reduced traces are often highly compressible. 94

3.3 Reduction of the traces from [GC97]. Ratios of the **number of events** in traces reduced by OLR and the reduction technique of Glass and Cao (GlCa) are shown. A ratio of x means that the GlCa trace is x times larger. 112

3.4 Reduction of the traces from [GC97]. Ratios of the **trace file size** from traces reduced by OLR and the reduction technique of Glass and Cao (GlCa) are shown. A ratio of x means that the GlCa trace is x times larger. 113

3.5 Reduction of the traces from [GC97]. Ratios of the **compressed trace file size** from traces reduced by OLR and the reduction technique of Glass and Cao (GlCa) are shown. A ratio of x means that the GlCa trace is x times larger. 114

4.1 Information on traces used in [GC97] 137

6.1 The processing times (in seconds) for each program in the test suite on each processor used in this study. If enough memory is available such that no paging occurs, these times will be the turnaround times. 270

List of Figures

1.1 In a traditional virtual memory hierarchy, when there are more pages than can fit in main memory, some pages are evicted to the backing store. In a compressed caching hierarchy, main memory is divided into an *uncompressed cache* and a *compressed cache*. Pages that do not fit in the uncompressed cache are evicted into the compressed cache, which in turn evicts pages to the backing store. 13

2.1 Sample LRU access histogram *(from artificial data)*. The x-axis is increasing LRU position, and the y-axis is the number of references to that LRU queue position. 29

2.2 Sample LRU miss histogram *(from artificial data)*. The x-axis is memory size, and the y-axis is the number of misses suffered at that memory size. The mostly-decreasing shape of the access histogram is common to many programs. **Note that the decrease is severe enough that a log scale is needed.** 30

2.3 LRU access histogram for a program that loops over 100 pages of memory. 32

2.4 LRU miss histogram for a program that loops over 100 pages of memory. LRU's prediction that recently used pages will be used again soon is exactly the wrong prediction. Note that the number of misses does not decrease as memory beyond about 20 pages is added until the entire loop can be held in memory. 33
2.5 How references to pages are managed in a segmented queue: Consider a segmented queue comprising FIFO and LRU segments, with the initial state shown in **(a)**. A reference to page `4` will cause that page to be taken from the LRU segment and inserted at the head of the FIFO segment. Page `3` will be extracted from the tail of the FIFO segment and inserted at the head of the LRU queue. The resulting state of the segmented queues is shown in **(b)**. Finally, a reference to any page within the FIFO queue will cause no change in either segment of the segmented queue. For example, a reference to page `5` will bring about the segmented queue state shown in **(c)**, which is identical to **(b)**. 44
2.6 In a 1^{st}-order Markov model, each symbol in the source alphabet corresponds to a node in the state machine. Each weighted edge from a given node represents the probability that a given symbol will be the next one in the input. By assigning shorter bit representations to the heavily weighted edges, a smaller representation can be found if the input corresponds to the model. 51
2.7 In a 2^{nd}-order Markov model, each node represents the two most recently seen input symbols. Although the higher order of the model means that it is responding to more context, it also means that the storage requirements are significantly larger. 53

2.8 A binary tree that represents a Huffman encoding for the alphabet a, b, c, d, e, f. A symbol's encoding is described by the path from the root to the leaf that contains that symbol. The traversal of an edge to the left child is encoded by a `0`, and to the right child by a `1`. For example, the path from the root to `c` is `left-left-right`, or `001`. . . 62

3.1 A_{second} can be eliminated because the LRU distance between A_{first} and A_{third} is less than the reduction memory size of five pages. . . . 76

3.2 OLR_CORE accepts the sequence of fetched and evicted pages for a k-page LRU memory, and produces the shortest reference trace that would cause the same behavior for that same memory. 80

3.3 SAD and OLR reduction factors over many reduction memory sizes for **cc1**. 87

3.4 SAD and OLR reduction factors over many reduction memory sizes for **GhostScript**. 88

3.5 SAD and OLR reduction factors over many reduction memory sizes for **lindsay**. 89

3.6 SAD and OLR reduction factors over many reduction memory sizes for **Acrobat Reader**. 90

3.7 SAD and OLR reduction factors over many reduction memory sizes for **Netscape Navigator**. 91

3.8 SAD and OLR reduction factors over many reduction memory sizes for **Microsoft Word**. 92

3.9 The absolute percent error (by number of faults) introduced into CLOCK simulations for **grobner** by reduced traces from SAD and SD. Notice that reduction memory sizes were chosen so that each reduced trace was approximately 100 times smaller than its original. 98

3.10 The absolute percent error (by number of faults) introduced into CLOCK simulations for **go** by reduced traces from SAD and SD. . . . 99
3.11 The absolute percent error (by number of faults) introduced into CLOCK simulations for **cc1** by reduced traces from SAD and SD. . . 100
3.12 The absolute percent error (by number of faults) introduced into CLOCK simulations for **GhostScript** by reduced traces from SAD and SD. 101
3.13 The absolute percent error (by number of faults) introduced into CLOCK simulations for **Acrobat Reader** by reduced traces from SAD and SD. 102
3.14 The absolute percent error (by number of faults) introduced into CLOCK simulations for **Netscape Navigator** by reduced traces from SAD and SD. 103
3.15 The absolute percent error (by number of faults) introduced into SEGQ simulations for **Acrobat Reader** by reduced traces from SAD and SD. For each plot, the FIFO segment size is fixed, and the x-axis shows the total memory size obtained by combining that FIFO with an LRU segment. 105
3.16 The absolute percent error (by number of faults) introduced into SEGQ simulations for **cc1** by reduced traces from SAD and SD. 106
3.17 The absolute percent error (by number of faults) introduced into SEGQ simulations for **go** by reduced traces from SAD and SD. 107
3.18 The absolute percent error (by number of faults) introduced into SEGQ simulations for **GhostScript** by reduced traces from SAD and SD. . 108
3.19 The absolute percent error (by number of faults) introduced into SEGQ simulations for **powerpoint** by reduced traces from SAD and SD. . 109

3.20 The absolute percent error (by number of faults) introduced into SEGQ simulations for **Microsoft Word** by reduced traces from SAD and SD. 110

4.1 General EELRU scheme: LRU axis and correspondence to memory locations. 125
4.2 Example recency distribution of page touches. **(a)** With LRU, many references are to pages not in memory. **(b)** After evicting early, some less recent pages stay in memory. Since references to less recent pages are common (in this example distribution), evicting early yields benefits. 127
4.3 EELRU with WFL fallback: LRU axis and correspondence to memory locations. 131
4.4 Probability of being in memory for a page with a given recency. 132
4.5 Recency-reference graph for wave5. 139
4.6 Recency-reference graphs for **gnuplot**. 141
4.7 Recency-reference graphs for **m88ksim**. 142
4.8 Recency-reference graphs for **perl**. 143
4.9 Recency-reference graphs for **gcc**. 144
4.10 Fault plot for **applu**. 147
4.11 Fault plot for **gnuplot**. Note that the SEQ curve almost overlaps the OPT curve. 148
4.12 Fault plot for **ijpeg**. 149
4.13 Fault plot for **m88ksim**. 150
4.14 Fault plot for **murphi**. Note that the EELRU curve almost overlaps the LRU curve. 151
4.15 Fault plot for **perl**. 152
4.16 Fault plot for **trygtsl**. 153
4.17 Fault plot for **wave5**. 154

4.18 A different fault plot for **gnuplot**. For this second simulation, an extra very early eviction point was available to EELRU. 156
4.19 Fault plot for **espresso**. 159
4.20 Fault plot for **gcc**. 160
4.21 Fault plot for **grobner**. 161
4.22 Fault plot for **GhostScript (gs3.33)**. Note that the EELRU line overlaps the LRU line. 162
4.23 Fault plot for **lindsay**. Note that the EELRU line overlaps the LRU line. 163
4.24 Fault plot for **p2c**. Note that the EELRU line overlaps the LRU line. 164

5.1 The WK compression decompression algorithm. The compressor reads the source page one word at a time, looking for zeros, exact matches to the dictionary, and partial matches to the dictionary. It stores the modeling results as a simple, ad-hoc encoding, centered around two bit tag values that indicate which kind of match was found for each word. 180
5.2 The decompressor reads those tags, plus other necessary bits, to direct its decompression. By maintaining a dictionary in the same fashion as the compressor, it can reconstruct the uncompressed page, one word at a time. 181
5.3 The sixteen consecutive tag entries, two bits apiece, are represented by pairs of alphabetic characters. The uppercase letter is the first bit in the tag, and the lowercase letter is the second bit. The zero bits are the "meaningless" ones that need to be eliminated by packing. Each group of four entries is handled as a machine word. 185
5.4 The means for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **espresso**. 191

5.5 The standard deviations for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **espresso**. . 192
5.6 The means for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **cc1**. 193
5.7 The standard deviations for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **cc1**. 194
5.8 The means for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **gnuplot**. 195
5.9 The standard deviations for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **gnuplot**. . 196
5.10 The means for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **grobner**. 197
5.11 The standard deviations for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **grobner**. . 198
5.12 The means for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **Ghostscript**. 199
5.13 The standard deviations for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **Ghostscript**.200
5.14 The means for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **lindsay**. 201
5.15 The standard deviations for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **lindsay**. . . 202
5.16 The means for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **p2c**. 203
5.17 The standard deviations for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **p2c**. 204

5.18 The means for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **RScheme**. 205
5.19 The standard deviations for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **RScheme**. 206
5.20 The means of the compression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **grobner**. 213
5.21 The standard deviations of the compression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **grobner**. . 214
5.22 The means of the decompression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **grobner**. 215
5.23 The standard deviations of the decompression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **grobner**. . 216
5.24 The means of the compression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **Ghostscript**. 217
5.25 The standard deviations of the compression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **Ghostscript**.218
5.26 The means of the decompression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **Ghostscript**. 219
5.27 The standard deviations of the decompression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **Ghostscript**.220
5.28 The means of the compression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **lindsay**. 221
5.29 The standard deviations of the compression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **lindsay**. . . 222
5.30 The means of the decompression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **lindsay**. 223

5.31 The standard deviations of the decompression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **lindsay**. . . 224
5.32 The means of the compression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **RScheme**. 225
5.33 The standard deviations of the compression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **RScheme**. 226
5.34 The means of the decompression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **RScheme**. 227
5.35 The standard deviations of the decompression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **RScheme**. 228

6.1 The LRU miss histogram provides needed information for the calculation of the cost and benefit of dividing some amount of RAM to hold both an uncompressed and a compressed cache. 240
6.2 An LRU miss histogram, labeled for the derivation of cost-benefit formulae. 242
6.3 The LRU miss histogram for p2c, a Pascal to C compiler. 247
6.4 The LRU cost-benefit performance curve for p2c, which shows the total runtime for p2c over a range of compressed cache sizes for a 110 page memory. 248
6.5 The LRU miss histogram for Netscape Navigator, a popular web browser. 249
6.6 The LRU cost-benefit performance curve for Netscape Navigator, which shows the total paging time for a given range of compressed cache sizes in a 116 page memory. 250
6.7 The LRU miss histogram for the Winstone benchmark. The actual data provided only part of the curve, and so the rest was extrapolated.251

6.8 The LRU cost-benefit performance results for the Winstone benchmark, which shows the total paging time for a range of compressed cache sizes given an 8 Mbyte memory. 252
6.9 For a given physical memory size, the paging time for **espresso** with a compressed cache that is some percentage of that physical memory. For comparison, the "Traditional VM" time shows the paging time required for no compressed cache. 258
6.10 For a given physical memory size, the paging time for **cc1** with a compressed cache that is some percentage of that physical memory. For comparison, the "Traditional VM" time shows the paging time required for no compressed cache. 259
6.11 For a given physical memory size, the paging time for **grobner** with a compressed cache that is some percentage of that physical memory. For comparison, the "Traditional VM" time shows the paging time required for no compressed cache. 260
6.12 For a given physical memory size, the paging time for **Ghostscript** with a compressed cache that is some percentage of that physical memory. For comparison, the "Traditional VM" time shows the paging time required for no compressed cache. 261
6.13 For a given physical memory size, the paging time for **lindsay** with a compressed cache that is some percentage of that physical memory. For comparison, the "Traditional VM" time shows the paging time required for no compressed cache. 262
6.14 For a given physical memory size, the paging time for **p2c** with a compressed cache that is some percentage of that physical memory. For comparison, the "Traditional VM" time shows the paging time required for no compressed cache. 263

6.15 Results from simulating a compressed cache whose size is adapted for **espresso** across a wide range of memory sizes. *(Note that the y-axis is on a log scale.)* 273
6.16 Results from simulating a compressed cache whose size is adapted for **cc1** across a wide range of memory sizes. *(Note that the y-axis is on a log scale.)* 274
6.17 Results from simulating a compressed cache whose size is adapted for **gnuplot** across a wide range of memory sizes. *(Note that the y-axis is on a log scale.)* 275
6.18 Results from simulating a compressed cache whose size is adapted for **Ghostscript** across a wide range of memory sizes. *(Note that the y-axis is on a log scale.)* 276
6.19 Results from simulating a compressed cache whose size is adapted for **p2c** across a wide range of memory sizes. *(Note that the y-axis is on a log scale.)* 277
6.20 Results from simulating a compressed cache whose size is adapted for **RScheme** across a wide range of memory sizes. *(Note that the y-axis is on a log scale.)* 278
6.21 Normalized paging times for different compression algorithms with **espresso**. Even though all algorithms yield benefits, some are significantly better than others. 280
6.22 Normalized paging times for different compression algorithms with **cc1**. Even though all algorithms yield benefits, some are significantly better than others. 281
6.23 Normalized paging times for different compression algorithms with **gnuplot**. Even though all algorithms yield benefits, some are significantly better than others. 282

6.24 Normalized paging times for different compression algorithms with **Ghostscript**. Even though all algorithms yield benefits, some are significantly better than others. 283
6.25 Normalized paging times for different compression algorithms with **p2c**. Even though all algorithms yield benefits, some are significantly better than others. 284
6.26 Normalized paging times for different compression algorithms with **RScheme**. Even though all algorithms yield benefits, some are significantly better than others. 285
6.27 Normalized paging times for compressed caching on different processors with **cc1** and **WKdm**. 287
6.28 Normalized paging times for compressed caching on different processors with **cc1** and **LZO**. 288
6.29 Normalized paging times for compressed caching on different processors with **Ghostscript** and **WKdm**. 289
6.30 Normalized paging times for compressed caching on different processors with **Ghostscript** and **LZO**. 290
6.31 Normalized paging times for compressed caching on different processors with **RScheme** and **WKdm**. 291
6.32 Normalized paging times for compressed caching on different processors with **RScheme** and **LZO**. 292
6.33 A sensitivity analysis of **espresso** with disks of various speeds. These plots can be interpreted not only for faster disks, but also for the cases of slower CPUs, perfect prefetching, and larger page sizes. 294
6.34 A sensitivity analysis of **cc1** with disks of various speeds. These plots can be interpreted not only for faster disks, but also for the cases of slower CPUs, perfect prefetching, and larger page sizes. 295

6.35 A sensitivity analysis of **gnuplot** with disks of various speeds. These plots can be interpreted not only for faster disks, but also for the cases of slower CPUs, perfect prefetching, and larger page sizes. . . . 296
6.36 A sensitivity analysis of **Ghostscript** with disks of various speeds. These plots can be interpreted not only for faster disks, but also for the cases of slower CPUs, perfect prefetching, and larger page sizes. 297
6.37 A sensitivity analysis of **p2c** with disks of various speeds. These plots can be interpreted not only for faster disks, but also for the cases of slower CPUs, perfect prefetching, and larger page sizes. 298
6.38 A sensitivity analysis of **RScheme** with disks of various speeds. These plots can be interpreted not only for faster disks, but also for the cases of slower CPUs, perfect prefetching, and larger page sizes. 299

Chapter 1

Introduction

1.1 Motivation

1.1.1 Why Study Virtual Memory?

CPU speeds have been increasing by about 60% every year for a number of years. Meanwhile, memory speeds have been improving at a much slower rate. RAM, once able to provide data at the processor's rate, is now slow enough that its bandwidth is sometimes a bottleneck on some systems. System designers have inserted smaller and faster memory hardware levels between RAM and the processor to cache the most-used main memory data. As this performance gap between processors and memory hardware widens, the ability to manage that memory becomes increasingly important.

No gap is wider, and no gap is growing faster, than the one between processor speed and disk speed. Virtual memory systems use disks as slow, very large storage for program data when main memory is full. For modern hardware, RAM can provide data five to six orders of magnitude faster than a disk. Since disk latency is improving by less than 10% each year, the use of disk to store in-memory program data will only become more relatively expensive.

It is critical that virtual memory systems manage program data well. The addition of new layers of memory hardware is not enough, as programs tend to be written to use the resources available. If virtual memory systems do not make intelligent decisions about what data to keep in RAM, programs will *page*—that is, use disk for storage because RAM is full. Such programs will spend an unbearable amount of time waiting for data to be fetched from disk.

What this dissertation offers. We seek to improve the understanding of how programs use memory, how that understanding can guide improvements in main memory caching, and how such improvements can be scientifically evaluated. We will investigate *compressed caching* as an approach to using RAM more effectively to avoid paging. We will evaluate the idea through *trace driven simulation*, so that our experiments can be controlled and repeated. Such experimentation will help us to develop a better understanding of *timescale relativity*, which dictates how virtual memory systems should adapt to program behavior. We will also use simulation to evaluate a new approach to *page replacement*, in which a virtual memory system adapts to how a program uses its memory. All of these concepts will be better defined and described throughout the rest of this introduction.

1.1.2 Why Study Compressed Caching?

Because disk accesses are so expensive, we are interested in ideas that improve the utilization of main memory. *Compressed caching* is the division of main memory into two pools: An *uncompressed cache* of pages in their "natural", uncompressed representation, and a *compressed cache* of pages in compressed form. By compressing some of main memory, the effective memory size is made larger, and disk accesses can be avoided. A more thorough description of a compressed cache and how it fits into the memory hierarchy is given in Section 1.5.

Compressed caching is an idea that had been prematurely discarded. Early evaluations, performed by implementation in real systems, yielded ambiguous results. The misconception then developed that compressed caching was useful only for machines without a disk, such as palmtop computers. We believe that compressed caching can provide a reduction in paging to a much broader range of computing hardware and workloads. Furthermore, we believe that compressed caching can smooth the performance degradation that occurs when programs need more memory than is physically available; no longer must performance become painfully poor when slightly more memory is needed than a machine has.

New developments in compressed caching. We will show that compressed caching should easily improve paging performance because of the discrepancy between processor and disk speeds. We will introduce new compression algorithms that are designed to work on in-memory data, and as part of a compressed caching system. We will also use what we have learned about timescale relativity and reference behavior to make a compressed cache adapt its size to a program's needs—a very important factor to compressed cache performance.

1.1.3 Why Use Simulation?

There have been a number of ideas proposed to improve virtual memory performance: compressed caching, prefetching, network caching, and better page replacement, to name a few. Many of these ideas have been evaluated through implementation in real systems. Because of the complexity of real systems, these kinds of experiments make it difficult to control variables and to reproduce the results.

As a result, good ideas have been prematurely discarded, and the understanding of these ideas has remained rudimentary. Compressed caching was one good idea that was not understood because early, real-system experiments could not reveal which component of the implementation led to poor performance.

Simulation of a virtual memory system makes possible controlled and reproducible experiments that allow researchers to develop understanding and correlations. However, simulations take as input a *reference trace*—a log of the memory locations used by a program during execution. These traces tend to be so unmanageably large, particularly if augmented with extra information, that simulation is rarely used. We have developed solutions for the problem of trace length that make simulation is a desirable approach for virtual memory experimentation.

New solutions to trace reduction. In order to make simulation practical, we have developed methods of trace reduction that are specific to virtual memory. These reduction methods have made it possible for us to examine more complex virtual memory concepts than can usually be handled by simulation.

We have used simulation to address two ideas that are designed to reduce paging: First, we have designed *EELRU*, a new page replacement policy, that adapts to program reference behavior. Second, we revive the topic of *compressed caching* by developing new compression algorithms and applying the same kind of adaptivity used for EELRU to make a system adjust its compressed cache size at runtime.

1.2 Virtual Memory Background

1.2.1 A Brief History

What is virtual memory? The concept of *virtual memory* has existed for over thirty years [Den70]. Although it has taken many forms, the central concept of virtual memory is to allow (or even to force) programs to refer to their data through some abstracted reference mechanism, rather than via explicit commands to hardware devices.

In its earliest forms, virtual memory was not a single, well defined idea. As it was first being devised, there were two central concepts that are now common to all virtual memory systems:

1. *Automatic paging:* Virtual memory implementations divide memory into fixed-size blocks known as *pages*. Previously, programmers explicitly controlled the movement of pages from main memory (core or RAM) to and from the backing store (drums or disk). System with automatic paging assume complete control of these decisions.

2. *Unsegmented address space:* The manner in which a program refers to its data defines that program's *address space*. Previously, memory was divided into *segments*, requiring programs to refer to their data via a two-dimensional address, where the first component was a segment name. In such a system, programmers would explicitly create and manage these segments. With an unsegmented address space, programs refer to data only through a one-dimensional, numerical address.

Some early virtual memory systems used only one of these two ideas. For example, the influential Multics operating system [CV65] employed automatic paging, but provided a segmented address space as its addressing abstraction. Today, however, virtual memory systems provide unsegmented address spaces in addition to automatic paging.

Early issues in virtual memory. Both automatic paging and unsegmented address spaces were met initially with some skepticism. Some system designers believed that a programmer's intelligence was required to ensure that the correct pieces of data were available in main memory at the right times. The danger of making this management automatic was that too much time might be wasted on disk transfers due to poor management.

In response to this skepticism, research was performed to determine how programs used their memory, that is, their *reference behavior*. This behavior was used to determine how virtual memory systems should provide a referencing abstraction, and how automatic management could be performed without introducing unacceptable overhead. The basic performance results of these studies indicated that virtual memory was an excellent idea. In most cases, allowing the system to manage a program's memory *increased* performance. It seems that programmers often make poor decisions with regard to memory management.

One of the most important topics in virtual memory management was (and still is) *page replacement policies.* When main memory is full, the system must determine which page is to be evicted to disk in order to make space for some other, needed page. The page replacement policy determines which page is evicted.

A good policy reduces the number of disk accesses by choosing to evict pages that will not be used for some time, thus keeping in memory pages that will be used soon. Notice that choosing well requires that the policy guess whether or not a page is likely to be needed soon. Such a choice demands certain assumptions about how a program is likely to reference its pages in the future, making page replacement a difficult problem. More on page replacement and our approach to improving it is provided in Section 1.4.

1.2.2 Current Use

The intention of the virtual memory abstraction was to provide a single, simple view for program storage at runtime. By providing automatic paging, programmers were encouraged to concentrate on their solutions and not on the details of memory management. If the program used more memory than could be stored in main memory, then the system would manage the situation. Moreover, while there would inevitably be a performance decrease, that decrease would be gradual.

Today, programmers again must be aware of memory management because of the limitations of paging. Since disk access is so slow, many programs are hand crafted to avoid exceeding the main memory capacity. If a program does page, its performance degrades drastically, leaving the user either to adjust their use of that program, to find another program, or to buy more RAM.

We believe that virtual memory systems can be made to more effectively manage data, and thus restore the intended power of its abstraction. By addressing the performance gap caused by increasingly slow disks, we can reduce the need for programmers to write code tailored to fit into the most common RAM size.

1.3 Trace Driven Simulation

Trace driven simulation is the use of a log of events, taken from real program execution, to allow a simulator to mimic the behavior of a system. Because the simulator need only implement the system components that are being studied, the implementation and control are simpler than with a real system. A virtual memory simulator can use a reference trace to track the movement of pages from one memory level to another, recording the costs that would be incurred to fetch and evict those pages.

1.3.1 A Problem for Virtual Memory Simulation

One of the greatest obstacles for virtual memory simulation is the size of reference traces. A program that executes for only a few seconds can perform enough memory references to require many gigabytes to store the whole sequence, even if that sequence comprises only the memory addresses used. If the trace needs also to store other information about the data being used, such as timing information or the values of the data, then the trace will be that much larger.

The length of these traces can be compressed with compression utilities like `gzip`, but even then the traces are tens or hundreds of megabytes. Worse, compression utilities do not help to reduce the simulation time, which is proportional to the length of the trace. If simulation is slow, then it becomes infeasible to examine a meaningful range of memory sizes, hardware speeds, and virtual memory configurations.

Where simulation has been used. Studies of *page replacement policies*, which determine how to select a page in RAM for eviction to disk when memory is full, have often utilized trace driven simulation. For these studies, the reference traces contain the sequence of pages used, augmented perhaps with additional information such as dirtiness (whether the page was modified by a write operation), or timestamps. A simulator needs only to implement the page replacement policy for a given memory size, and the performance can be estimated based on the number of fetch and eviction operations that the replacement policy would require.

Where simulation has not been used. More complex virtual memory concepts have *not* often been evaluated through trace-driven simulation. This approach has been avoided largely because the reference traces with the needed information have been so large. For example, our studies of compressed caching required traces that contained actual copies of each 4 Kbyte page as it existed when it was referenced. In order to be able to gather and use traces with so much information per reference, we have developed trace reduction methods that are specific to virtual memory studies.

1.3.2 Why Simulation Should Be Used

When positive improvements to a real virtual memory system are observed, they provide incontrovertible evidence of the efficacy of the idea. However, when there are negative or ambiguous results, little should be concluded. Real systems are complex, and implementation details may hinder the performance of the experimental system, obscuring evidence that the concept being tested may be valid.

Worse, these real-implementation experiments are difficult to control. Limiting the changes in a real system to isolate the effects of different variables can be a formidable task. Further, such simulations are not always exactly reproducible—small differences in the execution of a workload may change the outcome in significant ways. Also, the hardware needed to execute a real, experimental system may only be available during the time of the experiment, and not available to future investigators.

Simulated experiments of virtual memory systems can be controlled and repeated. As such, they can provide insight into how programs use memory, as well as the effect of changes to isolated components of a virtual memory system. To understand how an idea can change a virtual memory system, investigation should begin with simulation. If simulations provide position results, and if the experimental concept is well understood, *then* verification of those simulations should be performed via real implementation.

1.3.3 New Ideas Presented Here

We will present two new methods: *Safely Allowed Drop (SAD)* and *Optimal LRU Reduction (OLR)*. These reduction methods discard information that is not important to a virtual memory system's decisions on which pages to keep in main memory, and which to evict. Instead, these policies will discard information about references that have little or no effect on virtual memory decisions—references that account for the vast majority of trace length. Traces reduced by these methods are more easily stored, and greatly increase the speed with which simulations can be performed.

1.4 Reference Behavior and Page Replacement

1.4.1 A Brief History

Because virtual memory was initially greeted with such skepticism, researchers analyzed reference behavior and to evaluated virtual memory performance. These studies helped to develop important concepts, such as the *principle of locality* [Den70], which states that only a small fraction of a program's memory is used most of time. If locality did not exist for the vast majority of programs, virtual memory would not likely have worked. Later, we will see that locality also makes compressed caching an attractive idea.

Well known replacement policies. Early studies also concentrated on the development and comparison of page replacement policies. For many of these policies, the choice of page to evict is evident from the policy's name: There was *First In First Out (FIFO)*, *Least Recently Used (LRU)*, and *Least Frequently Used (LFU)*, to name a few. Over time, it became apparent that the *Least Recently Used (LRU)* policy performed best in practice. (This policy and its performance will be more fully described in Chapter 2.) Today, all real systems use replacement policies that are approximations of LRU.

Part of this early work on replacement policies also included the off-line policy *MIN* [Bel66] replacement policy (commonly known as OPT). This algorithm could "cheat" by examining future references, therefore avoiding poor decisions. It is often used to determine the smallest possible number of page faults for a reference sequence, and thus serves as the ideal against which other replacement policies can be measured.

Adapting to reference behavior. All of these replacement policies are, to some extent, *adaptive*—they all act in response to the order in which a program references its pages. However, there are many levels at which a page replacement policy, or even other parts of a virtual memory system, could react to information gathered about a program's behavior.

An early example of this kind of adaptivity is *loop detection*. When a program performs a loop, it will usually access some amount of memory in a cyclic pattern.

Unfortunately, this reference pattern does not exhibit good locality—a large portion of memory is used with equal frequency, thus making it difficult for a replacement policy to reduce the number of main memory misses by caching effectively.

In response, the developers of the *ICT ATLAS* machine [BFH68] implemented heuristics that attempted to determine if the program were performing a loop. Their heuristics did not perform very well, but the concept of adapting to specific, common types of program behavior is an important one. By understanding reference behavior, we can make a virtual memory system more successfully detect patterns such as loops, and adjust their assumptions accordingly.

1.4.2 New Ideas Presented Here

We have been able to develop the concept of *timescale relativity*—the idea that the view of a program's reference behavior appears, to a virtual memory system, to be very different depending on the memory size alloted to that program. That is, a program's reference behavior for a 512 Kbyte cache is quite different from that of a 128 Mbyte RAM. By understanding how memory size dictates the relevance of reference behavior, a virtual memory system can be made to ignore patterns that occur so quickly that they should be ignored, and notice patterns that occur slowly enough to be exploited. Timescale relativity is more thoroughly defined and discussed in Section 2.1.

Given a better understanding of these concepts, we have been able to develop *recency-based adaptivity*. Virtual memory policies that use this form of adaptivity take advantage of timescale relativity and the information provided by LRU page replacement—specifically, how many other pages are used between successive references to a page. By using this information, a virtual memory system can respond to features of a program's reference behavior that cause traditional virtual memories to suffer a performance breakdown. This information has allowed us to detect when a virtual memory system would evict a page, only to have to fetch it into main memory soon after. By detecting that situation, a system can be made to avoid it.

We applied this form of adaptivity to develop *Early Eviction LRU (EELRU)*, a family of page replacement policies that performs LRU replacement by default, but can perform non-LRU replacement for reference patterns that cause LRU to perform horribly. EELRU rarely performs worse than LRU, and can do so only by a small constant factor. Further, it can outperform LRU when non-LRU replacement is beneficial by a factor proportional to the memory size.

1.5 Compressed Caching

In a compressed caching system, RAM is divided into two levels, as shown in Figure 1.1. RAM is used to store an *uncompressed cache* of pages in the "normal" encoding, and a *compressed cache* of pages stored in some compressed format. If the uncompressed cache is full, pages are evicted into the compressed cache. Similarly, when the compressed cache becomes full, pages are evicted to disk.

1.5.1 Revisiting a Good Idea

This idea was initially evaluated through implementation in the Sprite operating system kernel [Dou93]. Ambiguous results were presented; some programs exhibited a reduction in paging costs, others an increase. While the author noted that the results would become increasingly positive as the gap between CPU and disk grew, the perception for many years was that compressed caching was a flawed concept.

Evaluation by simulation would have shown that the concept is valid, and that the performance trends would soon allow compressed caching to help real programs. Further, an understanding of how compressed caching interacts with program reference behavior could have been developed. That information would guide future developments and allow for a better real implementation.

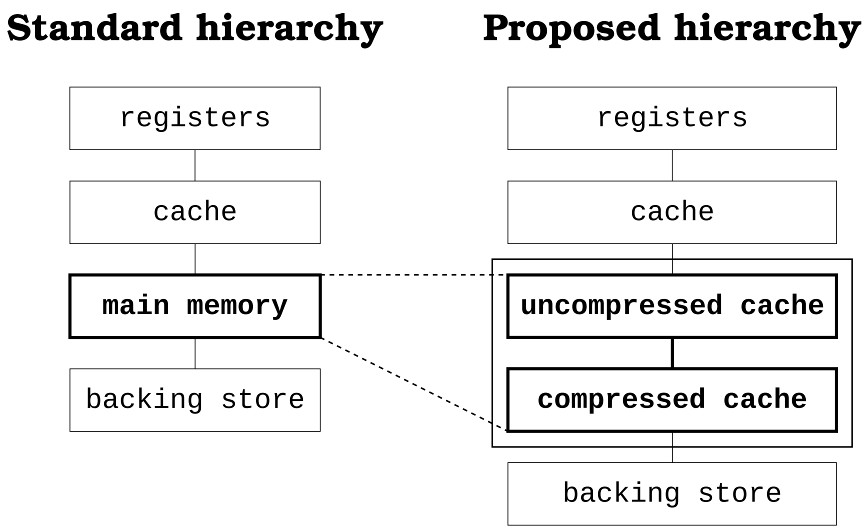

Figure 1.1: In a traditional virtual memory hierarchy, when there are more pages than can fit in main memory, some pages are evicted to the backing store. In a compressed caching hierarchy, main memory is divided into an *uncompressed cache* and a *compressed cache*. Pages that do not fit in the uncompressed cache are evicted into the compressed cache, which in turn evicts pages to the backing store.

1.5.2 New Ideas Presented Here

The poor performance of early compressed caching implementations was not clearly due to a flawed concept. By evaluating compressed caching in simulation, we have been able to study and address the design of individual components that contribute to compressed caching. The demands on a compression algorithm in this context are different from those of a compression scheme designed for file system data. We have been able to develop the *Wilson-Kaplan (WK)* family of algorithms, specifically designed for the compression of in-memory data held in a compressed cache.

Because we were also able to perform simulations of compressed caching, we were able to apply our adaptive caching concepts used in EELRU to the dynamic resizing of a compressed cache. Because RAM must be divided between uncompressed and compressed pages, it is critical that the size of those two caches be arbitrated carefully. Recency based adaptivity has made it possible to resize dynamically the compressed cache size.

Through this experimental infrastructure, and through the concepts we were able to develop, our results show that compressed caching is an idea that could benefit real systems. So long as a real implementation is designed to avoid excess overhead, compressed caching can provide up to an 80% reduction in paging costs, while rarely increasing paging costs, never by more than 10%. These reductions in paging cost will become more dramatic as the gap between CPU and disk speed increases. By studying virtual memory issues in a controlled and reproducible way, we have obtained positive results that show that this widening gap need not dominate the performance of machines.

1.6 Road Map

There are a number of concepts surrounding virtual memory, program reference behavior, and data compression that serve as background to the research presented here. Because each concept pervades many of the topics, Chapter 2 will be dedicated entirely to developing these ideas. In that chapter, we will describe timescale relativity and recency-based adaptivity in more detail, and show how they stem from the useful information that LRU provides about program behavior. These concepts will help us to understand what any good page replacement algorithm must do, at a minimum, to improve significantly on LRU itself. Chapter 2 will also describe issues of data compression. This area is, for some domains such as file systems and human text, quite mature and thoroughly studied. However, in both the trace reduction and in-memory data compression domains, less research has been done, and we will describe how these domains differ from others.

Chapter 3 will focus on the topic of trace reduction for virtual memory studies. We will present *Safely Allowed Drop (SAD)* and *Optimal LRU Reduction (OLR)*, which are two new methods that reduce reference trace size without affecting LRU simulation results. Furthermore, these methods do not introduce significant error into common, LRU-like policy simulations. These reduction techniques are timescale-relative in that they ignore references to very recently used pages that would matter only to small memories. Since RAM sizes are usually tens to hundred of Mbytes, it is possible to use these methods to remove information that is irrelevant to anything smaller than a common RAM size. We will see that with real reference sequences, these methods produce reduced traces that are smaller by factors of hundreds to tens of thousands.

We will then address, in Chapter 4, the methods of virtual memory adaptivity that we have developed. These methods keep the same kind of recency information that LRU does, but analyze that information in a timescale-relative way. We will discuss EELRU in greater detail. EELRU performs LRU replacement by default, because LRU is exactly the right policy for many important reference patterns. However, LRU does fail for some common patterns, and fails in a severe manner. EELRU detects these common patterns, and then modifies its replacement decisions. The information that EELRU needs is provided by LRU, and that information can be used to perform a simple cost/benefit function. This approach to adaptivity serves as a model of how to keep simple but relevant reference information to make virtual memory systems adaptive.

In Chapter 5, we will explore the compression of in-memory data. In particular, at the heart of any compressed caching scheme is the actual compression and decompression of virtual memory pages. We will address the regularities that commonly exist for in-memory data, as well as the desirable characteristics of a compression algorithm used for compressed caching. We will then present the *Wilson-Kaplan (WK)* compression algorithms, design for use within the compressed caching context. Finally, we will show the results of experimental evaluation of these new algorithms, as well as LZO and LZRW1, which are two fast Lempel-Ziv algorithms used in other compressed caching studies.

Compressed caching and its many facets are presented in Chapter 6. We will begin with an analytic approximation of how well a compressed cache would perform for a given reference trace, based only on average compression ratios and speeds. We will show, through simulations of fixed-size compressed caches, that the approximation is a good one. The accuracy of the approximation not only supports our understanding of the compressed caching problem, but will also be a valuable tool in developing a recency-based method of adaptivity. Simulations of our approach to dynamically adapting the size of the compressed cache will show that our understanding of reference behavior can make compressed caching yield significant reductions in paging time. Finally, we will examine the hardware trends that directly influence the increasing benefit that could be gained through compressed caching.

Chapter 2

Concepts

2.1 Caching, Page Replacement, and Timescale Relativity

Virtual memory is the abstraction provided by a system that each user program has a large, unsegmented address space to use. However, the memory hardware in a machine is not a single piece of uniformly addressable space. There are a number of different memory devices, from CPU registers, to on-chip and off-chip caches, to RAM, and finally to disk, that can store data for a machine. It is the task of the virtual memory system to map the single, simple, virtual address space onto those pieces of memory hardware.

The virtual memory hierarchy. Note that the various pieces of memory hardware listed above vary in size and speed. More specifically, the larger the memory device, the slower it is. This relationship between size and speed implies a *memory hierarchy*. At the top level of this hierarchy is the set of CPU registers. For the CPU to operate on any piece of data, that data must be moved to this top level. However, not all of a program's data is likely to fit in registers, so the data that cannot be held in registers is stored in the larger, slower on-chip cache. If not all of the data fits there, then some data is stored in the still larger and slower off-chip cache. This hierarchy continues down to RAM , and then to disk.

For a virtual memory system to be efficient, it must try to arrange the data within these storage levels such that little time is wasted accessing data stored in the larger, slower levels of memory. More specifically, as a fast and small level of memory becomes full, a *replacement policy* dictates which data is evicted from that memory level to the next one that is larger and slower. The smaller memory level is said to be a *cache* for the larger level. A good replacement policy will cache data needed by the program in the fastest memory levels, thus reducing the time spent waiting on the slower memory levels.

The weak link in the hierarchy. The last two levels of virtual memory—RAM and disk—are of particular interest. Between the on-chip and off-chip cache, or the off-chip cache and RAM, there is an increase in access cost of about one order of magnitude for the slower memory level. However, disk is approximately *six* orders of magnitude slower than RAM, and the gap between these levels is widening.

Applicability at many levels. This research centers on the virtual memory management of main memory and disk. Although many of the concepts presented here are applicable to other levels of the memory hierarchy, we will discuss "caching" in terms of the management of pages in main memory such that accesses to disk can be avoided. It is important to notice, however, that we believe that these same concepts apply to any level of the hierarchy. The very purpose of describing reference and management behavior in terms of timescale relativity is to evaluate memory use for all different levels of the hierarchy, irrespective of system-specific configuration.

Because of the poor performance that disks introduce into virtual memory, we will examine past work and discuss concepts surrounding page replacement policies.

In particular, we will begin with the off-line *OPT* policy that can determine, by "cheating" and examining future references, the smallest possible number of disk accesses that a virtual memory system could incur. Then, we will examine the *Least Recently Used (LRU)* policy, which is commonly used and approximated, but whose value is still not fully appreciated. Finally, we will attempt to reveal the causes of the poor performance of the *Least Frequently Used (LFU)* policy.

We will also develop the many faces of *timescale relativity*, and analyze the performance of the replacement policies above in terms of this concept. It will allow us to see how to take many billions of memory references, and from that information, observe patterns in program behavior. Replacement policies that are timescale-relative perform better because they are able to respond to these patterns.

Some of these concepts presented here will be background, while others will provide new perspective on old ideas and the new ones that will be presented in later chapters.

2.1.1 The On-line Caching Prediction Problem

Page replacement, in real systems, must be done *on-line*—the policy must decide which page to evict based only on information it has collected about previously used pages. The policy cannot assume that information about future references will be available, because in real systems, it will not be. As a result, page replacement policies must use available information from past references and attempt to choose a page that will not be used soon.

In other words, a page replacement policy must use past information to predict the future. If a policy successfully chooses to evict a page that will not be used soon, then its choice to evict that page will not cost the virtual memory system anything for some time. More importantly, some other page that may be needed sooner can be cached for a longer time, and therefore is available for smaller cost than if it had been chosen for eviction.

There are also *off-line* page replacement policies, which are allowed to examine future references before deciding which page to evict. These replacement policies cannot be used in real systems, as the information about future references cannot be obtained at runtime. However, they can provide us with useful information about the quality of the predictions made by on-line policies.

2.1.2 Temporal and Spatial Locality

In Chapter 1, we mentioned that the *locality of reference* exhibited by many programs made the automatic paging aspect of virtual memory practical. The basic idea of locality is that data which is referenced is close to other data that has been and will be referenced. In other words, data is naturally clustered together so that the caching of blocks (i.e. pages) of data is practical because any one page will likely be a unit of data that is close together.

Notice that the term "close" can be interpreted in at least two ways, most commonly in terms of both space and time. Therefore, both *spatial* and *temporal* locality are discussed when analyzing reference behavior and designing virtual memory systems. Two pieces of data are temporally local if they are accessed at nearly the same time during program execution.

Two pieces of data are spatially local if they are near to one another in the address space. Notice that this strict definition of spatial locality refers only to how data is distributed through the address space. However, spatial locality is often used to refer to pieces of data that are clustered together on a set of pages, and are referenced at nearly the same time during program execution. Consequentially, dividing memory into pages is a useful strategy because of *spatio-temporal locality.* Data that is temporally local also tends to be stored in a spatially local way. Fetching a page is useful because it is likely that more than one piece of data on that page will be used soon.

2.1.3 Three Well-Known Replacement Policies

Before we attempt to understand timescale relativity, as well as its relationship to reference behavior and replacement performance, we need a background in the mechanical operations of common page replacement policies. This section will outline OPT, LRU, and LFU, which are three of the best known, most studied page replacement policies. Most other replacement policies bear great resemblance to at least one of these three. The problem of page replacement can be understood by examining how each of these responds to common reference patterns.

OPT: The Optimal Off-line Replacement Policy

If a page replacement policy were given the entire sequence of page references over the whole execution of a program, how few page faults would that policy incur? That is the question that Belady's algorithm, MIN (a.k.a. OPT) [Bel66], answers. This page replacement policy must be used off-line. It provably yields the smallest number of page faults that any page replacement algorithm could. Therefore, it is used as the benchmark against which all other page replacement algorithms are measured. We will also use it later to aid in understanding on-line page replacement policies.

How OPT works. At any given page fault, the OPT policy chooses to evict the resident page that will next be referenced *furthest* in the future. In other words, OPT can examine the sequence of references that is going to occur. For each page that is in memory, it can determine which one will be used for the first time *after* all of the others. That page is selected for eviction.

There are a number of approaches to implementing OPT. The most obvious is a brute force search forward in the trace at each page fault. Alternatives include schemes that maintain a window into the future, as well as methods that require a pre-processing of the trace to augment it with timestamps. Since it is an off-line algorithm, the ability to implement it on-line is relatively unimportant.

LRU: Least Recently Used

The LRU replacement policy is an on-line policy, and it is aptly named—If a page must be evicted from one memory level to the next, then this policy chooses to evict the least recently used page. The implicit assertion that drives this policy is that a page that has not been used for a long time is not likely to be used soon. Such a page should therefore be reasonable choice for eviction. We will see, in Section 2.1.4, that this assumption is a very good one for many programs, but a disastrously poor one for others.

How LRU works. The algorithm used to implement this policy is also simple: The *Move To Front (MTF)* algorithm keeps a list of pages, known as an *LRU queue*[1]. When one of those pages is used, that page is moved to the front of the queue[2]. At any given time, the page at the back of the queue is the least recently used one.

Implementing LRU in a real system can be difficult. The MTF algorithm is often used in the implementation of LRU simulators, as the operations required to remove a referenced page from the middle of the list, and then insert it at the front, are inexpensive. However, those operations are much too expensive to perform in a real system at every memory reference. As a result, implementing an LRU policy in a real system cannot be done with MTF. In fact, real LRU implementation would require special hardware support that was not feasible for a long time, and is not now found in any processor.

[1] The data structure is commonly refered to as an *LRU stack*. We choose not to use the term *stack* here, as we believe that the data structure does not behave like a stack, and so the term is misleading.

[2] The use of the terms *front* and *back* to denote ends of a queue has often been applied inconsistently. Here, we insert items at the front of the queue, and remove them from the back.

However, the strong performance of this policy has driven the development of other policies that approximate LRU. We will address, in Section 2.1.4, both the CLOCK and SEGQ policies that are such approximations, and are used in real systems because they require less hardware support than LRU.

LFU: Least Frequently Used

The *Least Frequently Used (LFU)* policy is also an on-line policy. It operates on an assertion that is, at least superficially, similar to the one used by LRU: A page that is not used frequently is likely to be unused for some time, and so it is a good page to evict. While this assumption may initially seem reasonable, we will see that it can perform poorly for many programs. Moreover, we will see that frequency of use does not indicate much about future use.

How it works. The implementation of LFU is more daunting than for LRU. For every reference, the statistics of the referenced page must be updated. In simulation, a table of pages must be kept to store these statistics. How the least frequently used page is found in that table is not important in simulation.

In real implementation however, LFU is not practical. Keeping all of these statistics would make each reference too expensive. Approximation methods do not resemble LFU very well, as missing frequency information fundamentally changes the behavior of this replacement algorithm. We will see, later in this section, that LFU is not timescale-relative, and that fact is largely responsible for its poor performance.

2.1.4 Timescale Relativity I: Virtual Memory Time and "Soonness"

To a system, "*time*" can have many meanings, particularly when applied to computations and their execution. The choice of timescale for a study depends on the topic being studied. Time could be considered real time, as measured by a stopwatch of the number of CPU clock cycles that pass. However, a computation proceeds at the rate of the hardware resources it uses at any given moment, and so a timescale based on the speed of those resources may be more meaningful. For example, measuring computational time in terms of number of CPU instructions, where some instructions require more CPU clock cycles than others, may be more meaningful than real time.

Here, we are concerned with the fetching and evicting of pages from and to disk. Thus, memory references that have an effect on page replacement decisions are events with which the virtual memory system is concerned, and thus should dictate the timescale. That is, for a virtual memory system, *time* should advance when references are made that influence or require replacement decisions.

The size of main memory significantly effects the timescale to which a memory system should react. Consider a 512 Kbyte hardware cache, and a 128 Mbyte main memory. A program that loops over 600 Kbytes of data is going to cause a number of significant events to occur in the hardware cache, as many cache block replacements will be performed. The main memory, however, can easily store all of the data, and so no significant events occur. While this concept seems obvious, we will see that it has not been sufficiently considered in evaluating replacement policies.

Predicting what will happen "soon". When a page replacement policy chooses a page to be evicted from main memory to disk, it is performing a prediction. To reduce the number of *main memory misses*—references to pages that are not in main memory—the replacement policy must choose to evict a page that will not be referenced "soon".

In order to determine the quality of the prediction that a policy makes, we must develop the notion of "soonness". We have not established *how* soon a page must be used to imply that its eviction was a poor choice. A page that will be one of the next three pages used is clearly going to be used soon, and evicting it would be a mistake. A page that will never be used again is clearly not going to be used soon, and its eviction will not be a mistake.

But there is a middle ground—What if the page will be used after 1,000 other pages are used? Is that page going to be used soon enough that, by caching it, excess disk accesses will be needed? Or is that page not going to be used soon enough, and some other page should be cached so that it can be used sooner, without incurring a disk access?

Forward and backward page distance. Before we provide a useful characterization of "soonness", we must provide a definition: The *forward page distance* of a given page is the number of other, unique pages that will be referenced between the current time and the next reference to that page. For example, at some given moment of a program's execution, its next five references may be to pages $[B, C, B, A, C]$. The forward page distance for A at that point in the execution is 2, as pages B and C will be referenced before A is.

The *backward page distance* is the analog to the forward page distance: For a given page, it is the number of other, unique pages that have been reference between the last reference to that page and the current time. This metric has an interesting relationship to an LRU queue. The backward page distance to a given page is exactly the LRU queue position that it would occupy at that moment in the execution.

Given this definition, we can make two key observations about what it means for a page to be used soon:

- It is the **forward page distance** for a page that should determine how soon it will be used. It is tempting to think of soonness in terms of the raw number of references. However, consider the situation in which a few pages are referenced a great many times, and then some other page in which we are interested will be used. By considering the number of references, it may seem that the next reference to the page in question will not occur soon, even though that page should remain cached. If we view *soonness* in terms of number of references, we will be predicting how long a page will go unused with respect to *reference time*. However, we are concerned with *virtual memory time*; only events that influence replacement decisions should advance that time. Referencing just a few pages over and over again has little influence on page replacement, so virtual memory time does not advance, and the first reference to the page in question does occur *soon*, with respect to virtual memory time.

- While virtual memory time can help to determine which pages will be used sooner than others, we still must know whether or not a page is going to be used *soon enough*. That characterization must be made in terms of the **memory size** being used. For a 100 page memory, it is not likely wise to keep the page that has a forward page distance of 1,000. For a 100,000 page memory, caching that same page clearly is wise. Memory size directly influences whether or not a page will be used soon enough that it should remain cached.

In order to understand how well or poorly on-line page replacement algorithms predict how soon a page will be touched, we will explore how OPT respects both to the memory size and the number of other pages referenced between successive references to a page.

OPT and Soonness

Most descriptions of algorithms that implement the OPT policy use a number of references to measure how soon a page will be used. Because OPT is concerned only with choosing the resident page that will be used *least* soon, it needs only a relative ranking of the soonness with which each page is referenced. Even though the number of references is a metric at the wrong timescale, it is still a linear measure of time that will provide the needed ranking of soonness among pages.

It is more useful to use virtual memory time when analyzing the behavior of OPT. That is, OPT chooses the resident page that has largest forward page distance. In this way, OPT is simply choosing the resident page that will be used the least soon. Consequently, OPT is timescale-relative, in that it always selects a page that will not be used soon enough to make the decision a poor one.

While it is true that any timescale would be appropriate, given the manner in which OPT selects a page for eviction, it is important that OPT responds to events at the virtual memory timescale. Note that we are not discluding the possibility of responding to data collected at different timescales. It is possible to perform different kinds of pattern recognition at any timescale that may be relevant to virtual memory behavior. However, this kind of pattern recognition has not been persued. Furthermore, we are interested in detecting the simple regularities at the timescale of virtual memory systems, and we compare replacement policies from that perspective.

LRU and Soonness

LRU bases its prediction about the future use of pages by taking a kind of mirror image of the past: The longer that a page has gone unused, the longer it is going to be until it is used again. In other words, a page that has not been used lately is not going to be used soon. The least recently used page is likely to be the least soon used page, and should be evicted.

LRU performs very well for any access pattern over a number of pages that can fit into the available memory. It also performs very well for many access patterns over more pages than can fit into memory. However, there are a few access patterns, such a loop over slightly more pages than can fit into memory, for which LRU performs as poorly as possible. We will see why LRU performs so badly in those cases. Further, we will show that LRU is timescale-relative in those cases for which it performs well, but not timescale-relative for those patterns for which it makes poor decisions.

Recency and LRU distance. Before we analyze the behavior of LRU, we need to define the related concepts of *recency* and of *LRU distance*. Since LRU selects the least recently used page, it orders pages according to how recently each has been used in its LRU queue. So, the LRU queue provides information about the *recency* of each page at any given time in the execution.

LRU distance is a special case of backward page distance, used to measure the number of other pages used between successive references to the same page. Its name is taken from the distance that such a page would fall in the LRU queue ordering before the second of two successive references. That is, if $k-1$ other pages are used between two uses of some page A, then that page would reside in LRU queue position k. We say that when the second reference of A occurs, it has LRU distance k. This term provides us with specific recency information about a page when it is used.

LRU performs well in most cases. In order to understand why LRU performs well, we need to examine the manner in which real programs reference their memory. In particular, we want to view these reference patterns with respect to recency, as LRU does, to determine whether or not LRU's prediction of the future corresponds to reality.

In Figures 2.1 and 2.2, we see two views from the same access sequence to an LRU memory. The first is an *LRU access histogram*, which shows the number of references to pages at each LRU queue position over the complete execution of a program. The second, an *LRU miss histogram*, shows, for each memory size, the number of misses suffered. Note that the second histogram can be derived from the

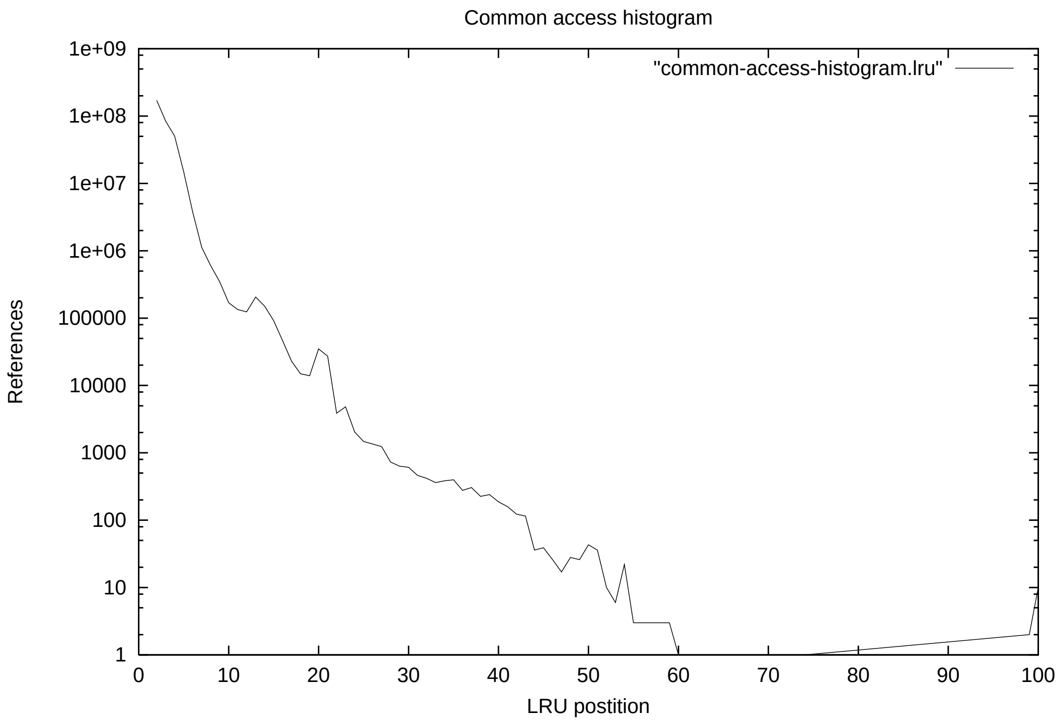

Figure 2.1: Sample LRU access histogram *(from artificial data)*. The x-axis is increasing LRU position, and the y-axis is the number of references to that LRU queue position.

first, as the number of misses for a k-page memory using LRU (heretofore an "LRU memory") is the sum of the accesses to all LRU positions beyond the k^{th} position.

Note that so many references are to recently used pages—those that are in the lowest LRU queue positions—that the plot must be shown with a log scale for the y-axis. Also notice that although the access histogram is not strictly non-increasing as LRU queue position increases, in general, less recently used pages are referenced less often. These plots demonstrate that many programs exhibit the strong temporal locality that makes automatic paging tractible.

Based on this plot, LRU's assumption that recently used pages are more

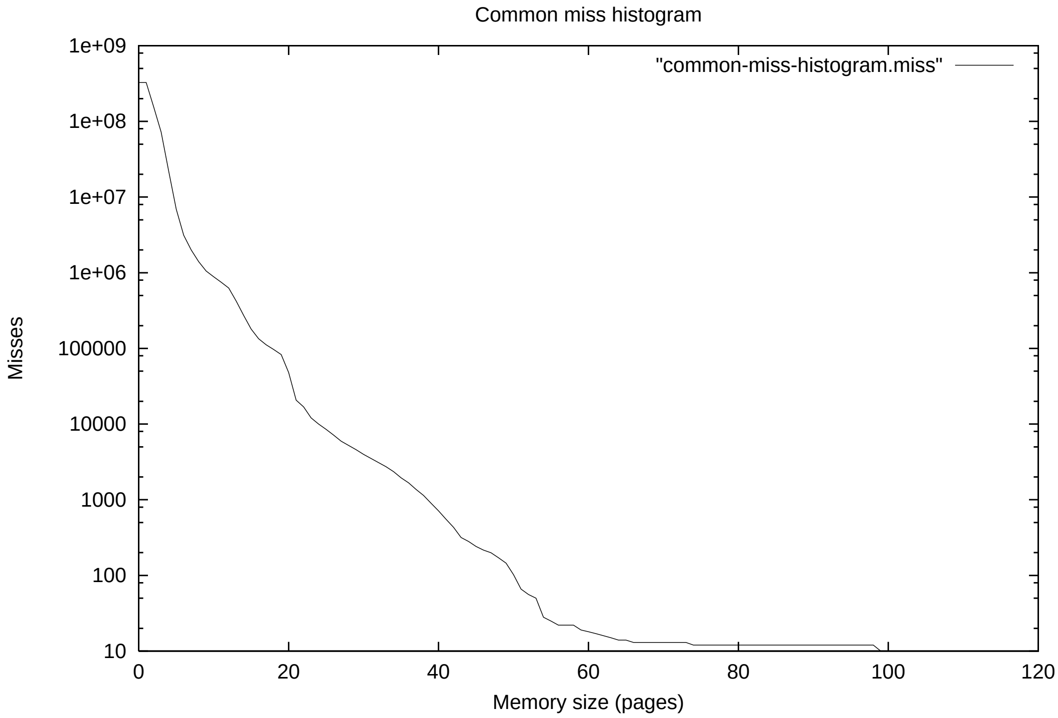

Figure 2.2: Sample LRU miss histogram *(from artificial data)*. The x-axis is memory size, and the y-axis is the number of misses suffered at that memory size. The mostly-decreasing shape of the access histogram is common to many programs. **Note that the decrease is severe enough that a log scale is needed.**

likely to be referenced soon than less recently used pages is valid. Moreover, the collection of this kind of recency information can reveal useful information about the access patterns of programs. Not only does LRU base its predictions on a strong regularity, but we can also see that *any* good replacement policy must avoid evicting recently used pages if it is to perform well for most programs.

Note that the vast majority of references are to the first few LRU queue positions. Since real memory sizes require LRU to evict pages from much higher LRU queue positions, all of these references to lower LRU queue positions have little effect. For example, many references to the most recently used 10 pages simply reorder the recency of those 10 pages. Those references don't advance virtual memory time, as they have little influence on replacement decisions. In other words, they have no effect on the prediction that page in some much higher LRU queue position will be the least soon used of those in memory—Those references don't affect LRU's notion of "soonness." So, for these kinds of access patterns, LRU is timescale-relative.

LRU performs terribly in some cases. While LRU's prediction—that the time since the last use of a page suggests the time to its next use—works quite well for most programs, we can easily imagine a case for which it fails. Are there real programs that reference memory in such a way that the longer a page has not been used, the *more* likely it will be used soon? Figures 2.3 and 2.4 show the LRU access and miss histograms for a program that loops over 100 pages. Each page is referenced, and then remains unused until the other pages used by the loop have been traversed. (Note that about 20 pages are referenced more often. These pages are referenced more than once per loop iteration, and may contain, for example, code or data used as part of a more frequent calculation.)

The assumption used by LRU is exactly wrong in this case. The longer a page is not referenced, the closer it is to being referenced again by the loop. When LRU must make an eviction choice, it will choose the worst possible page—the one,

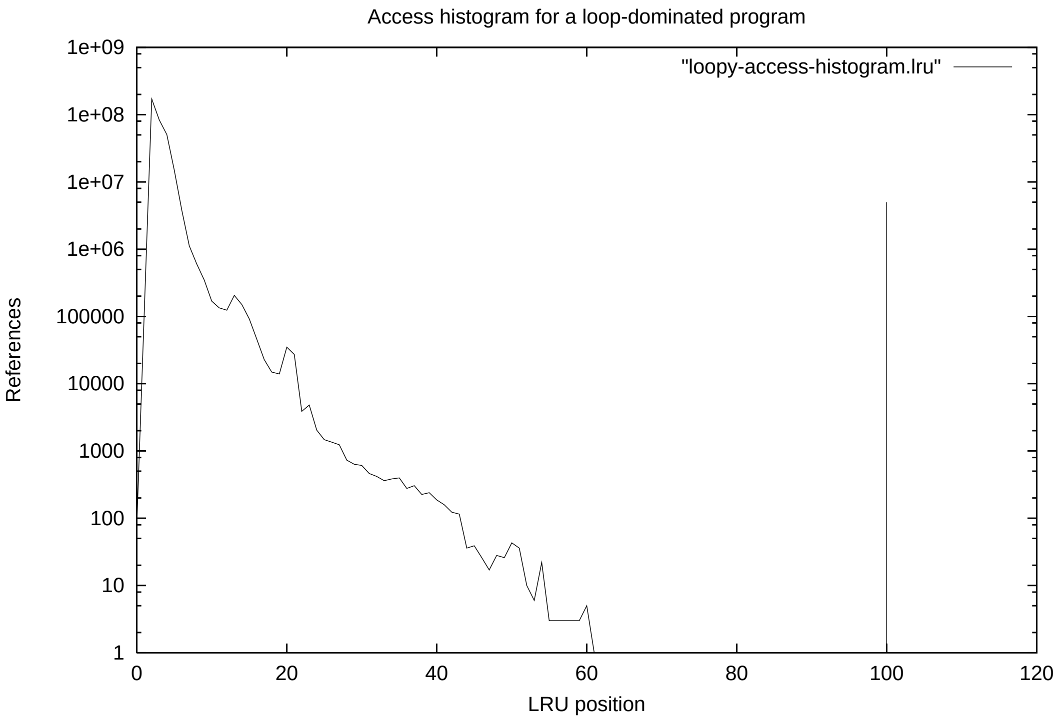

Figure 2.3: LRU access histogram for a program that loops over 100 pages of memory.

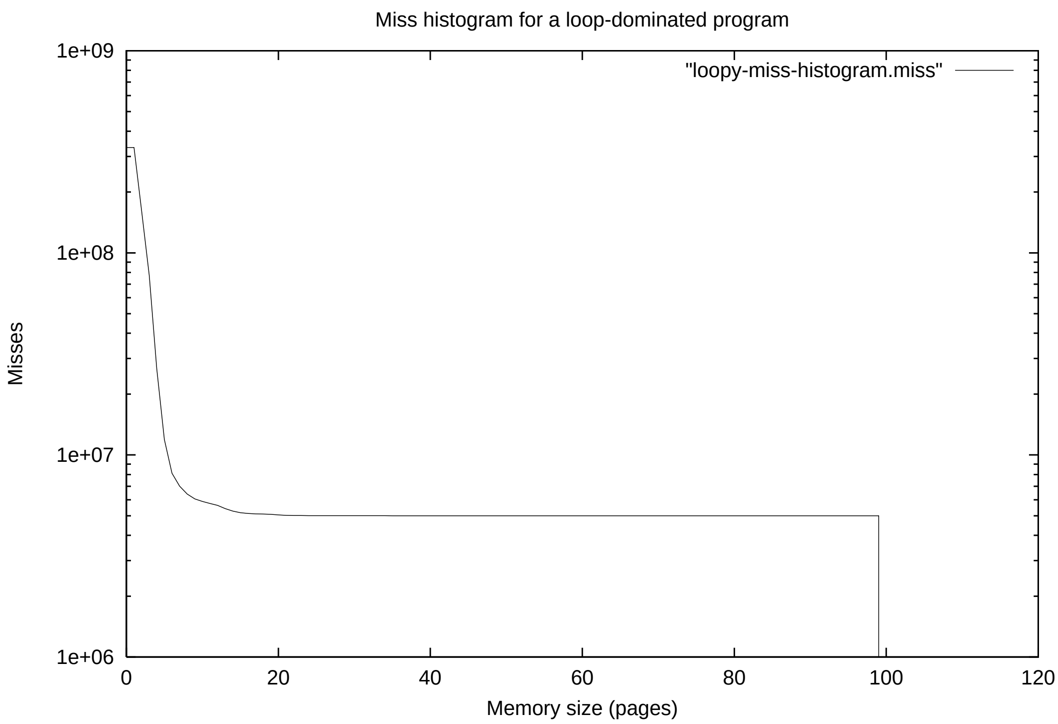

Figure 2.4: LRU miss histogram for a program that loops over 100 pages of memory. LRU's prediction that recently used pages will be used again soon is exactly the wrong prediction. Note that the number of misses does not decrease as memory beyond about 20 pages is added until the entire loop can be held in memory.

among those in memory, that will be used the soonest. In other words, LRU is not timescale-relative for this kind of access pattern.

If LRU and policies like it are based on exactly the wrong assumption for this common case, why is LRU still used? Although LRU is making the wrong prediction, its mistake will cause page faults only if the loop is larger than the available main memory. For the histogram shown, if main memory can hold 150 pages, all pages will be retained, and LRU's mistaken assumption will cost nothing. It is for this situation that we must remember the influence of memory size on timescale relativity. The absolute determination of how soon each page will used may not matter if all pages are used soon enough that a poor prediction doesn't matter.

A less obvious case in which LRU's poor prediction costs little is when the loop is significantly *larger* than main memory. If a program loops over 100 pages of data, but main memory can hold only 10 pages of data, then *no* replacement policy can yield good performance. Even OPT would require 91 pages to be evicted and then fetched again. LRU would require all 100 pages to be evicted and then fetched at each iteration, but that is less than 10% worse than optimal. Again, memory size directly influences the timescale. For this situation, any notion of soonness is going to perform nearly as well as possible, simply because the situation is hopeless.

LRU ignores useful recency information. LRU replacement damages performance significantly for a reference pattern like a loop over somewhat more pages than can be stored in memory. However, notice that the LRU access histograms in Figure 2.3 reveals this pattern: The high number of accesses around the 100^{th} LRU queue position indicate that for a memory somewhat smaller than 100 pages, the program will be referencing pages that had just been evicted.

We discuss the use of this information in more detail in Chapter 4. EELRU is designed to examine this recency information and detect when LRU would be evicting pages and then, soon after, fetching them again. When such a case is detected, EELRU begins to perform non-LRU page replacement to address the problem. We will examine in more detail how to detect these cases, as well as the problem of analyzing recency information on-line. We will also apply this kind of adaptivity to compressed caching in Chapter 6.

LFU and Soonness

When it must select a page for eviction, LFU predicts that the resident page that has been used the least often is the one that is least likely to be used soon. Notice that LFU does not base its prediction on time, but on frequency. LFU records its information with respect to reference time, at every reference, rather than virtual memory time.

For the kinds of access histograms shown above, in Figures 2.1 and 2.2, there is a strong correlation between recency and frequency. If a recently used page is likely to be used soon, then such a page will be used frequently.

Phase behavior reveals LFU's weakness. The correlation between recency and frequency does not always hold, even for such common access patterns. Programs often exhibit *phase behavior*: For some time (i.e. during one phase), a program will use one set of pages a great deal. Then, however, the program will need to do something different, and another set of pages will be favored during this new phase.

LFU does not adjust well to phase changes. Consider the execution of a program on a system with 100 pages of available main memory. In the first phase, let us say that the the program frequently uses a set of 60 pages. These pages are both the most recently and the most frequently used pages—LRU and LFU would both perform well. Let us then consider what happens if the program changes phases, and begins to use a different set of, say, 80 pages that have never been used before.

Because this new set of pages has never been used, both policies will suffer 80 *compulsory misses*. Note that these references influence replacement decisions, and so they advance virtual memory time. LRU responds by aging the old set of pages immediately, thus considering the new set more likely to be used soon. However, LFU will prefer to cache the pages of the old set, as it will require some execution time to pass before the new set has been used more frequently. Until the frequency statistics reflect the change, LFU will page significantly, even though the set of pages should easily fit into memory.

LFU gathers and responds to information from every reference—it does not respond to the timescale of the given memory size. Further, the information collected by LFU indicates how often a page will be used (if the future resembles the past), not how soon each page will be used. Therefore, LFU is not timescale-relative with respect to soonness.

Theoretical limitations. Despite its intuitive appeal—that pages used often should not be evicted—LFU has been shown both experimentally and theoretically to perform poorly. Analytically, Sleator and Tarjan showed in [ST85] that LRU was *competitive* with OPT. Given the same k page memory, LRU will never suffer more than k times as many faults as OPT. More generally, if LRU can use more memory than OPT, then the factor increase in faults for LRU is bounded by the ratio between the LRU and OPT memory sizes. In contrast, a proof was given that no on-line algorithm could improve on this competitive property. LFU was shown *not* to be competitive—it can incur an unbounded factor of faults more than OPT.

2.1.5 Timescale Relativity II: Aggregate Behavior

On-line replacement policies must predict the future. Specifically, they must choose pages that will not be used soon for eviction. Because those choices must be made one at a time—only one page is evicted as a result of a single reference to a non-resident page—there is a temptation to design policies that select *the* page that will be used least soon.

OPT Does Not Need Aggregate Information

This temptation is reinforced by the behavior of the off-line OPT policy, which does choose the page that will be used least soon. However, the nature of prediction suggests that this kind of specificity can be hazardous. An on-line policy will perform well for the vast majority of reference patterns if it reliably selects *any* page that will not be used soon. There is little need to predict exactly which page will be used least soon. When a replacement policy attempts to be so specific, its predictions become brittle. Small-scale variations in a reference pattern can confound the predictive heuristic, and damage performance.

It is important to remember that OPT does not need to predict how soon a page will be used, since it can calculate exact when each page will be used. Off-line algorithms do not need to balance robust prediction against specific information. On-line policies, however, should respond to coarse grained, aggregate reference behavior. The resulting decisions may not be the optimal ones, but they are less likely to be poor and costly choices.

LRU Uses Aggregate Information

The good performance of LRU is not because its predictions about the soonness with which pages will be used is more exact than the predictions of other policies. To the contrary, LRU performs well for a large number of access patterns because it does not rely on exact predictions of the future.

LRU's choice of pages for eviction is in two ways inexact:

1. Information is not kept for each page; LRU does not try to detect patterns

in how each page is used. Rather, each page is considered to be like the others, and only the information about its recency is taken into consideration. While there may be detectable patterns in how a page is used, recency is often a reliable predictor, whereas per-page pattern recognition can easily lead to poor eviction decisions.

2. The exact recency order of pages is not critical. For nearly all access patterns, evicting any of the least recently used pages in memory will yield approximately the same performance as LRU. We will see that in real systems, approximations of the LRU policy use this property to achieve good performance without maintaining exact recency information. This property also implies that small changes in access behavior is unlikely to degrade paging performance significantly—LRU is not sensitive to small reference behavior changes.

LRU responds only to aggregate behavior. It can maintain its timescale relativity, irrespective of minor differences in reference behavior and parameters such as memory size. A policy that is sensitive cannot adjust to different timescales as easily.

LFU Uses Per-Page Information

In order to determine which page is the least frequently used, the LFU policy must maintain statistical information about the reference history of each page. LFU therefore predicts how soon a page will be used based on that specific page's past use.

Keeping such per-page information contributes to LFU's difficulty with phase changes. LFU considers the past use of each page, rather than the past used of pages that are being used similarly to other pages at a different time in the execution. The statistics of a page may reflect patterns of use from previous phases, and mislead the policy with regards to the current use of that page.

LFU is not, however, sensitive to exact frequency ordering. As with LRU, the selection of any of the least frequently used pages is likely to yield approximately the same paging performance.

2.1.6 Timescale Relativity III: Adaptivity and Decay

Because programs exhibit phase behavior, we want the virtual memory system to be able to respond to those phase changes. Some such changes are to transient phases, while other changes are to persistent phases. If we are to adapt to phases, we need to be sure that our adaptivity is *agile*—that it does not attempt to respond to transient behavior from which no advantage can be gained, but it does respond quickly to persistent behavior that can be managed well.

In order to ignore such short-term behavior, but notice appropriate long-term behavior, again a virtual memory system must operate at the appropriate timescale. We will see again that OPT adapts immediately and perfectly, that LRU does adjust well to such phase changes most of the time, and we will again analyze LFU's failure to adapt to phase changes. We will also describe how the recency information provided by LRU histograms can itself be decayed, thus allowing that information to be used in a timescale relative way.

OPT and Decay

OPT does not keep information about past references. Because it is an off-line policy, it can rely on future reference information. It is interting to note, though, that OPT looks no further than the next reference to a page to make its determination. References that will occur in the near future provide all of the information needed, while other references in the more distant future need not be considered at all.

Since on-line policies must predict the future based on the past, it then seems that this characteristic of OPT supports the notion that the most relevant references are likely the most recent ones.

It is also useful to notice how quickly OPT adapts to new phases. While pages that had been used a great deal by the program were in no danger of eviction, the moment that a new phase begins, those pages become strong candidates for eviction. OPT adapts as quickly as possible—for every references that causes a fetch, OPT uses that opportunity to evict a page that no longer needs to be cached.

LRU And Decay

By decaying information collected about past reference behavior, more recent references are given greater weight than less recent ones. It is worthwhile to note that LRU performs a strict kind of decay on the virtual memory timescale.

LRU stores information only about the most recent reference to each page. Once a page is used, all knowledge about previous references to that page is discarded. LRU is therefore decaying, in a severe way, old information about past use of pages. The decay occurs only when references occur. More importantly to the behavior of LRU, only those references to pages that are in LRU queue positions that are near the corresponding memory size affect the decay. That is, as recently used pages are referenced, only those pages change in their LRU queue positions, and so the information about other pages is not decayed, as virtual memory time has not advanced.

Even this simple kind of decay is sufficient to allow LRU to adapt quickly to phase changes. Similarly to OPT, when a new phase is encountered, pages used in the old phase age very quickly and become candidates for eviction. That they had been used a great deal previously, and that they were once among the most recently used pages is not heeded. When phases change, virtual memory time moves quickly, and LRU responds.

LFU And Decay

One of the reasons that the statistics kept by LFU are not timescale relative is that all events in the past are given equal weight. As a consequence, when a phase change occurs, LFU gives too much weight to past references to pages, rather than more weight to the most recent used of each page. Pages that are not used by the current phase take too long to fall out of favor and be evicted.

The *Least Recently-Frequently Used (LRFU)* policy [LCK+99] uses a combination of recency and frequency information to select a page for eviction. This policy uses a single, tunable decay parameter. When no decay is performed, the policy performs LFU replacement. When the decay is at its maximum, and only information for the last reference to each page is kept, the policy performs LRU replacement.

Not surprisingly, low decay factors yield poor paging performance. However, that performance improves as the decay is performed more quickly, and the policy approaches LRU behavior. The experiments performed by the researchers that developed this policy indicate that fast decay, but not quite as fast as performed by LRU, provides the best performance. The improvement over LRU, though, was minimal in their experiments, usually less than 5%. The bulk of the improvement came from decaying LFU information quickly—thus making the replacement policy more timescale relative.

Adapting To Phases By LRU Histogram Decay

The recency information provided by LRU access and miss histograms, like those shown in Figures 2.1 through 2.4, are the result of information gathered over entire program executions. While these histograms provide an interesting picture of the overall reference behavior for the entire program, they do not weigh more recent referencing behavior more heavily.

If this kind of information is to be used on-line to detect reference patterns, such as the use of pages that have just been evicted (and could have been cached), then it needs to be made timescale relative. In particular, we would like to decay the histogram statistics as virtual time passes. If another timescale is used, then the statistics may decay too quickly or slowly. Overly rapid decay may lead adaptive heuristics to respond to transient behavior from which no advantage can be gained. It may also eliminate information needed to detect long-term, persistent behavior. Slow decay may prevent a heuristic from adapting to new phases at all.

In order to decay this histogram information at the correct rate, we advance virtual memory time when references are made to pages that are old enough, in recency terms, either to be near eviction or to have recently been evicted. References to very recently used pages, or to pages that have been evicted so long ago that no caching policy would hold them, have little or no effect on virtual memory time. By advancing time in this fashion, phase changes force time to move quickly, and so the adaptive heuristics will respond quickly. During phases, time will not advance much, and so that persistent behavior can be the target of adaptation.

This form of decay of virtual memory reference information is used both for EELRU, presented in Chapter 4, and for compressed caching, presented in Chapter 6. We will see that this form of adaptivity allows these management methods to perform well accross a wide range of memory sizes and for many different reference patterns.

2.1.7 LRU Approximations for Real Systems

LRU performs well in many situations, but as noted earlier, it would be difficult to implement in a real system. As a result, replacement policies that behavior very much like LRU have been developed. These policies are designed to work with little hardware support, so they can be and are used in real systems. They also perform well enough that they are the only kinds of replacement policies used in modern operating systems today.

There are two categories of policies that behave very much like LRU. We refer to the first as *LRU variants.* Such replacement policies keep some number of the most recently used pages in LRU order, but then use some other policy for the rest of the resident pages. Examples of these policies include FBR [RD90], GLRU [FLW78], SEQ [GC97], and our own EELRU [SKW99]. Although these policies differ from LRU, their general behavior is similar to it, as the majority of references are to pages that all of these policies keep in an LRU queue.

Policies like CLOCK [Car84] and *segmented queue* (SEGQ, also known as *two level replacement* or *hybrid FIFO-LRU*) [BF83], represent the category *LRU approximations.* These two policies are the most used replacement policies in real systems. They make their eviction selections based on recency, although their heuristics to record recency are not as exact as an LRU queue.

Because these policies are so similar to LRU, they tend to be just as timescale relative. Not surprisingly, they also tend to suffer from LRU's lack of timescale relativity for reference patterns such as a loop over slightly more than the available memory. The kind of solution we propose in Chapter 4 with EELRU should be equally applicable to these LRU variants and approximations.

The Segmented Queue (SEGQ) Policy

The **segq** policy uses two queues—a FIFO queue and an LRU queue. The FIFO queue is of some fixed size, while the LRU queue is unbounded. If a page that is *not* in the FIFO queue is referenced, then it is inserted into the front of the FIFO queue. So long as that page is in the FIFO queue, references to it have no effect on

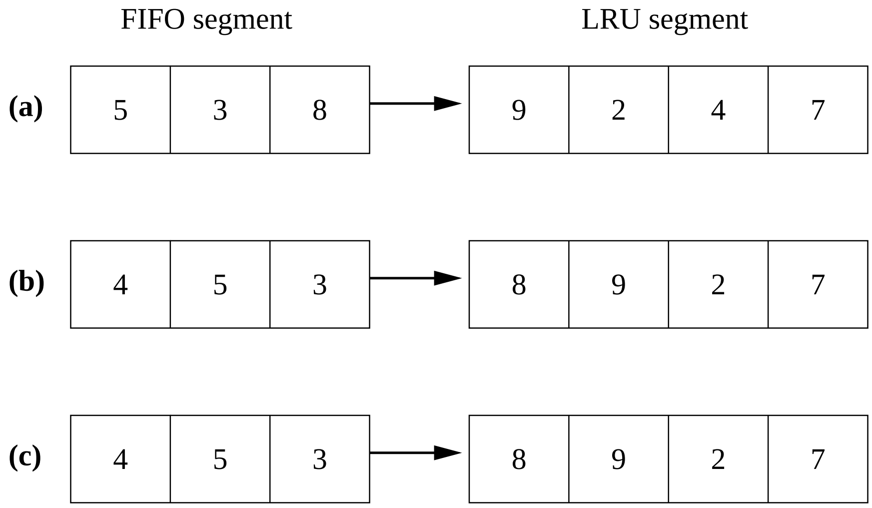

Figure 2.5: How references to pages are managed in a segmented queue: Consider a segmented queue comprising FIFO and LRU segments, with the initial state shown in **(a)**. A reference to page `4` will cause that page to be taken from the LRU segment and inserted at the head of the FIFO segment. Page `3` will be extracted from the tail of the FIFO segment and inserted at the head of the LRU queue. The resulting state of the segmented queues is shown in **(b)**. Finally, a reference to any page within the FIFO queue will cause no change in either segment of the segmented queue. For example, a reference to page `5` will bring about the segmented queue state shown in **(c)**, which is identical to **(b)**.

queue ordering—such references are free from the overhead of information collection. References to other pages that aren't in the FIFO queue will force this page towards the back of that queue. If enough of these other pages are inserted at the front of the FIFO queue, then this page must be evicted from it.

When a page is evicted from the FIFO queue, it is inserted at the front of the LRU queue. As other pages are evicted from the FIFO queue, this page will be pushed down in the LRU queue ordering. However, if that page is referenced, it is removed from the middle of the LRU queue queue, and then inserted at the head of the FIFO queue. Therefore, references to pages in the LRU queue *do* incur some re-ordering overhead. Figure 2.5 graphically depicts the movement of pages as they are referenced in either of the two queues, and the manner in which the FIFO queue "feeds into" the LRU queue.

The CLOCK Policy

The CLOCK policy views resident pages as being in a circular queue, much like the numbers on the face of a wall clock. At any given moment, there is a *clock hand* that points to one of these resident pages. When a page needs to be selected for eviction, the clock hand sweeps through the resident pages, looking for one that hasn't been used recently. As it searches for such a page, it updates the recency information of the pages that it passes over, aging each one.

Implementations of CLOCK take advantage of *hardware reference bits*—tags that are set by hardware with every reference to a page in memory. When the clock hand sweeps through the pages, aging recently used ones and searching for a less recently used one, it is these reference bits that are used to determine recency.

If a page's reference bit is set, then it has been touched "recently". It is not chosen for eviction, but it is aged by clearing that reference bit. If the page's reference bit is clear, then that page is selected for eviction, and the sweeping of the clock hand stops. The page being fetched is placed into the vacated location on the "clock face". Again, recently used pages are very likely to have their reference bit set. A page must remain unreferenced for some time if the clock hand is to find it with a clear reference bit, and select it for eviction.

Notice that multiple reference bits per page can be used. The hardware reference bit serves as the primary reference bit for each resident page, and the other, more minor bits can be stored in software. As the clock hand sweeps, it ages each page by shifting the reference bits down by one bit, thus clearing the primary bit. When the hand encounters a page for which none of the bits are set, then that page can be selected for eviction.

This most basic form of the **clock** policy suffers from a lack of timescale relativity. In particular, the hardware reference bits are set on the reference timescale, but the clock hand decays that information on the virtual memory timescale. More complex variants of CLOCK, ones that use interrupts and multiple clock hands to bring these two tasks nearer to the same timescale, have been developed and used in real systems.

2.2 Compression

The compression of data is a key component of two different ideas presented in this dissertation. The first application of data compression will be to the reduction of space required to store pages in a compressed cache, as presented in Chapter 5. The second, and less obvious form of data compression will be shown in Chapter 3, where two new methods of trace reduction will be presented.

We will begin by presenting some background compression issues and terminology. We will then go into details ahout *modeling*, which is the process of detecting patterns in data, and *encoding*, which is the generation of a compressed data representation based on the results of modeling.

2.2.1 Background

Data Compression Is Domain-Specific

Due to the pervasiveness and availability of common compression utilities, and because of theoretical results of limited scope, some believe that there are "general purpose" compression algorithms. The belief is that these algorithms can adapt to detect regularities in any kind of data. That done, such an algorithm can choose reduced representations for the regularities that occur most.

General purpose compression cannot exist, as data can exhibit regularities in an unbounded number of ways. The aforementioned theoretical results [BCW90] are based on assumptions about the data being compressed. That is, the type of data for which these compression algorithms yield reduced representations is in some sense fixed. For example, the very common, much studied *Lempel-Ziv (LZ)*[3] family of compression algorithms has been shown to be optimal *for data that can be generated by a Markov chain.* (See Section 2.2.2 for more detail on Markov models.)

These LZ algorithms are perceived by many to be general purpose, but their uses are limited. Furthermore, not only do some believe that these algorithms are universally applicable, but also that they are the best algorithms to use in nearly any domain.

The LZ algorithms, and others like them, do provide compression for different kinds of data, but not for all kinds. LZ algorithms are not the ideal algorithm in many domains; another algorithm designed for a specific kind of data may provide better compression at a lower computational cost. Such a domain-specific algorithm may not be applicable accross such a wide range of data types, but often compression is applied to a narrow domain. For such situations, compression algorithms designed to detect the regularities expected in that data are likely to perform best.

A compression algorithm *must* look for some set of regularities in the input data. If there are no expectations about the data, then no compression can be performed. Similarly, if the expectations are incorrect for a given input, again no compression will be gained. In fact, for all bit strings of a given length, the representation that will yield the smallest average is the "uncompressed" representation of enumerating all of those possible bit strings. Any other representation may be able to store some of those strings in a compressed encoding, but the average length will be at least as long, and usually longer. Therefore, a compression algorithm must

[3]By strict order of authorship, these algorithms should be known as *Ziv-Lempel*. However, because the *LZ* ordering is so commonly used, this dissertation will use it throughout.

operate on the kind of data for which it can shorten the representation most of the time.

Lossless vs. Lossy Compression

Compression can be broken into two general categories: *Lossless* and *lossy*. A *lossless* compression scheme is one for which all information is maintained when an input is compressed. In other words, there is a one-to-one mapping of compressed and decompressed representations, and the result of compressing and then decompressing a piece of data is an exact restoration of the original input.

A *lossy* compression scheme, however, discards some information from the input as it compresses; it is a many-to-one mapping of uncompressed to compressed representations. By discarding information that is presumed to be "superfluous", the reduction can be much greater. The key to these methods are the assumptions that a compression algorithm makes about the intended use of the data. For example, the popular *JPEG* compression format is a lossy method for graphic images. It makes assumptions about the visual acuity of humans and the need for accuracy in reproducting an image. These assumptions allow the JPEG method to discard some information from the input image. So long as the image is decompressed and viewed by people, as JPEG assumes it will be, then the image is of sufficient quality. However, if the decompressed image is used for other purposes—perhaps a bit-by-bit comparison with an original—then JPEG is an inappropriate compression scheme.

In Chapter 5, we will examine the use of compression algorithms on the in-memory data that would be paged into and out of a compressed cache. Because these compression algorithms are used to manage a program's data, they must be lossless. In contrast, our new methods of trace reduction, presented in Chapter 3, are lossy methods that discard reference information that is not relevant to LRU and LRU-like replacement policies. Like other lossy methods, if the reduced (that is, compressed) traces are used for some other purpose, then the information missing from the reduced trace may affect the results.

The inputs (in-memory data and reference traces) and the contexts (compressed caching and trace driven simulation) for these forms of compression are very different. Nonetheless, they are both compression schemes that can provide significant reduction by assuming regularities in their inputs.

Modeling vs. Encoding

In order to understand how a compression algorithm works, we must address the two interacting but separate tasks of *modeling* and *encoding*. The process of examining an input and detecting regularities in it is the process of *modeling*. A compression algorithm, by looking for these regularities, is building an internal representation of how the input relates to the algorithm's expectationed repetitions and predictable values. Once the input has been modeled, the compression algorithm must *encode* the output—that is, it must try to emit a representation of the data that is smaller than the original input. These two phases must interact, as the choice of encoding schemes is based on the kinds of regularities that are found while modeling. However, they are two distinct tasks that are sometimes blended into one.

2.2.2 Modeling

A compression algorithm must implement a model of what it expects to find in the data. For example, in order to model English text, a compression algorithm could hold a dictionary of common words. Each time one of those words is encountered, the algorithm would record the occurrence. A input that was English text would correspond with the model, and the more often those common English words are detected by the algorithm, the greater the potential for a compressed representation when the encoding is performed.

Many compression algorithms, though, do not restrict their domain of applicability so narrowly. Instead, weaker and lower-level expectations about the data are assumed. Many compression algorithms try to find various kinds of repetitions in the bytes of an input, and do not assume something so high-level as human language regularities. Although rarely implemented directly, *Markov models* are one of the more common constructs used to detect regularities in a wide variety of inputs.

Markov models

What is a Markov model? There are many approaches to modeling data. *Markov models* are a form of probabilistic modeling that have received a great deal of attention in compression research [BCW90]. For a given alphabet of symbols that may appear in an input string, a Markov model will indicate, based on the last k symbols seen, the probabilities that each symbol may be the next one. Such a model can be represented as a finite state machine, and is known as a k^{th}*-order Markov model.*

In Figure 2.6, we see an example of a 1^{st}-order Markov model for a subset of the English alphabet, specifically the letters `a`, `b`, and `c`. There is one node in the state machine for each character in this alphabet. When a symbol is taken from the input string, the state machine enters the node with that symbol. (Note that, strictly speaking, there should be *start* and *stop* states for entry into and exit from the state machine. We have not depicted these for this example.)

Directed edges from each node are weighted by the probability that the edge will be followed—that is, the probability that the node at the destination of the directed edge contains the next symbol. So, after reading an input symbol and following the appropriate edge into that symbol's corresponding node, this model can predict how likely it is that each symbol will follow. So, the input is being modeled as a set of probabilities.

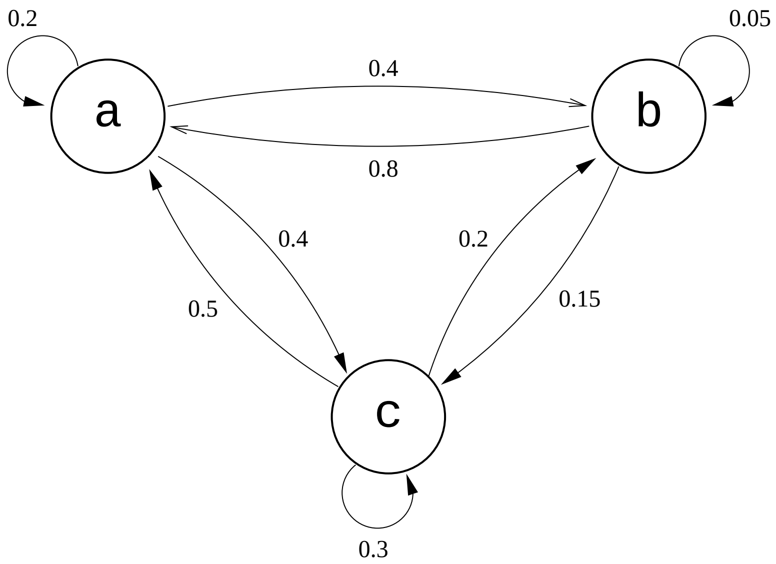

Figure 2.6: In a 1^{st}-order Markov model, each symbol in the source alphabet corresponds to a node in the state machine. Each weighted edge from a given node represents the probability that a given symbol will be the next one in the input. By assigning shorter bit representations to the heavily weighted edges, a smaller representation can be found if the input corresponds to the model.

We will see later that encodings can be chosen such that more probable transitions are represented with fewer bits than less probable transitions. If the Markov model is sufficiently representative of the input, such that many of the heavily weighted (that is, more probable) edges are traversed, then a compressed representation can be produced. If the input does not correspond to the Markov model's weightings, then the representation will expand.

With this kind of 1^{st}-order Markov model, the state machines only "remembers" the most recent symbol from the input. A model that attempts to make accurate predictions based on so little context is less likely to succeed than a model that considers more of the past input. To make more accurate predictions, and to predict accurately a greater variety of input strings, we want the state machine to respond to a larger context.

For example, we could use the 2^{nd}-order Markov model shown in Figure 2.7. Each node represents the last two symbols seen from the input, and edges again represent the probabilities for each symbol that may follow. Note that the number of nodes and edges required is larger. The state machine must contain nodes for every possible string of length 2, and from each of those nodes, one edge per symbol in the alphabet.

The Markov model storage problem. We notice here one of the significant weaknesses of Markov models for compression of real data: the *space requirements.* For any alphabet Λ, a k^{th}-order Markov model requires as many as $|\Lambda|^k$ nodes, and $|\Lambda|^{k+1}$ edges. This exponential growth in the storage of increasingly higher order Markov models quickly becomes prohibitive. To make predictions based on just the last three symbols from a string of ASCII characters (of which there are 128) requires over two million nodes and two hundred and fifty million edges. We will see later that the popular Lempel-Ziv compression algorithms address this storage problem by storing only some of the possible (and more probable) symbol combinations.

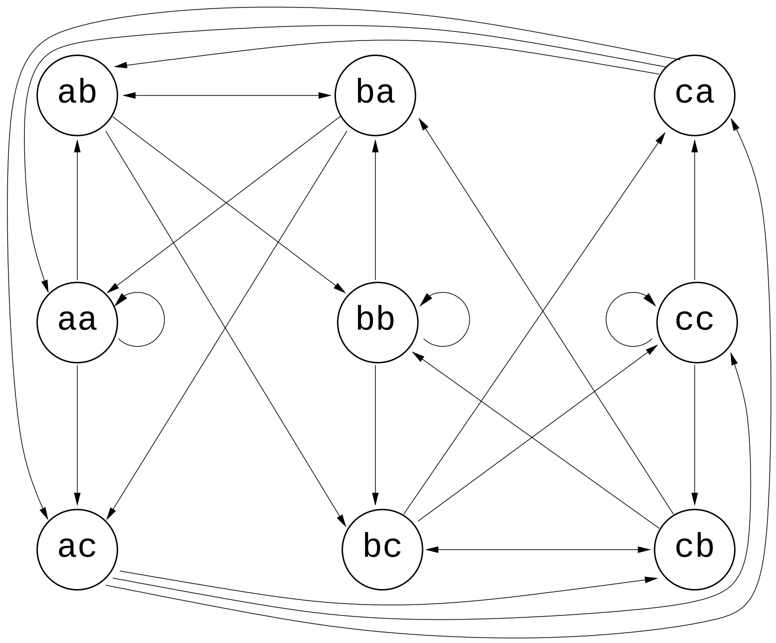

Figure 2.7: In a 2^{nd}-order Markov model, each node represents the two most recently seen input symbols. Although the higher order of the model means that it is responding to more context, it also means that the storage requirements are significantly larger.

Determining the transition probabilities. Whether or not these state machines can model an input well enough to provide compression is largely dependent on the probability weights assigned to the edges. Such weightings can be assigned *statically* or *dynamically*. If they are assigned statically, then an assumption is being made about the sequences in which symbols occur. If such a set of weightings were devised based on English text, then an application of that model to some other kind of input, from non-English human text to binary machine code, is not likely to yield as tight a compression as a different weighting would yield. However, the computational costs of a static weighting are low, as no updates needs to be performed.

Dynamic assignment of weights requires more computation. As a source string is read, the weights can be modified to reflect the actual frequency with which the edges are traversed. If those sequences are reasonably consistent throughout the input, then the Markov model will eventually be modified to provide accurate predictions, and thus compression.

While this dynamic approach to setting the probabilities can yield better compression on a larger set of inputs, it does not ameliorate the need for higher order models to capture more complex regularities. Adapting the probabilities cannot make up for the small context used by lower-order models. Further, the higher the order of the Markov model, the longer an input string must be for the probabilities to reach a useful equilibrium. Even if a high order Markov model can be applied in practice, smaller inputs will not be significantly compressed, if at all.

Markov models do not capture all regularities. These types of models are mathematically tractable in such a way that theorems about their compression properties can be proven. Such proofs have been mistaken as evidence that these models are capable, if the order of the model is high enough and the probabilities appropriately set, of capturing any kind of regularity. However, there are weaknesses to the Markov modeling approach that make it incapable of capturing some high-level regularities.

These state machines respond to literal sequences of symbols. As such, there is no modeling of the similarities among those sequences—each of the nodes and edges in the state machine is independent of the others. Markov models can respond only to exact, literal sequences of symbols, and not to general, repeating patterns.

As an example, consider again the modeling English text. A modeling method that breaks the input into English words (rather than individual characters) could be designed to predict that nouns are often followed by verbs. While such modeling is ad-hoc, it is also high level and able to provide compression on words never seen before in the input. A Markov model, however, will *never* capture this kind of regularity, irrespective of the order of the model and the size of the input. While literal occurrences of some noun-verb combinations will be captured, no generalization will occur.

The Lempel-Ziv Family of Compression Algorithms

Much of the research on modeling has centered on *statistical models* like Markov models, where symbols are assigned probabilities. Compression is achieved by correctly predicting which of those symbols is more like to occur. However, *dictionary based modeling* obtains its compression by encoding some portions of the input as entries into some dictionary known to both the compressor and decompressor. For example, one approach to modeling could use a table of common English words. If those words are found in the input, then the output would contain only the index into that table.

These two methods, *statistical* and *dictionary modeling*, are not mutually exclusive. However, statistical models are generally considered to be computationally expensive and only of theoretical value[4]. Dictionary methods are considered

[4]Some researchers do believe that, as hardware speeds increase and computation becomes inex-

to be very practical, particularly the methods of Ziv and Lempel, who proposed two popular and influential dictionary based compression algorithms on which most other popular compression algorithms are based.

The total set of Lempel-Ziv algorithms is non-trivial, and it is beyond the scope of this dissertation to discuss them all. However, many variants and improvements of their algorithms have been developed and studied. We will concentrate here on the original, influential algorithms, `LZ77` and `LZ78`.

The `LZ77` compression algorithm assumes that there will be literal repetitions of symbols in the input string. If it can find a portion of the input that is such a repetition, it can model and encode that portion merely by pointing to the previous occurrence. Since the decompression algorithm will recreate the input in serial, it too will have a copy of that previous occurrence, and so it can recreate the repetitive portion of the input string.

Note that although `LZ77` is considered a dictionary compression algorithm, the input is not matched against what is commonly considered a "dictionary". Instead, some number of previously encoded symbols are stored, serving as *sliding window* into the past input string. The algorithm then reads and tries to match as many symbols from the current input as it can against the contents of the sliding window. (Note that this "lookahead buffer" of current input symbols is limited in size.)

This simple approach to compression has been very successful. It can be implemented to be efficient in both space and time, and provides significant reductions. Some commonly used utilities, such as `gzip`, employ simple variants of this compression algorithm.

The `LZ78` algorithm is similar to its predecessor in that it attempts to match current input sequences against past ones. However, the approach to building the

pensive, statistical methods will become practical [BCW90]. In spite of incredible increases in CPU speed, that has not yet happened.

structure for matching is quite different. The past input is not stored as a simple sliding window of substrings. Instead, a dictionary of observed character sequences is built up from the past input. When a sequence of input symbols can be matched to a dictionary entry, that repetition is encoded by an index into the dictionary itself.

Furthermore, each time a sequence of input symbols can be matched against an entry in that dictionary, the entry itself is modified by adding one more symbol from the current input sequence. For example, if the next four characters in an input sequence are `acbdf`, and there is a dictionary entry for the past sequence `acb`, then the dictionary entry is updated to be `acbd`. Therefore, this algorithm can respond to increasingly long repetitions in the input. If the dictionary grows to be larger than a predetermined bound, then it is discarded, and the process of building one begins anew.

`LZ78` also provides significant compression for reasonable space and time efficiency. It has also been shown that, given an input generated by an ergodic source (such as a Markov model), this algorithm will achieve asymptotically optimal compression for an arbitrarily long string. Unfortunately, real inputs may not be so easily represented by a Markov model. It is even less likely that a real input is going to be arbitrarily long. Practically, `LZ78` achieves compression that is comparable to other LZ compression schemes.

Recency-based Modeling

The LZ compression algorithms maintain a limited dictionary of some of the recently seen substrings from the source input. Literal recurrences of these substrings in the input is only one regularity that may exist. Not only do symbols and substrings often repeat in an input, but commonly those same symbols and substrings are grouped together. In other words, recently seen patterns are more likely to recur than not recently seen ones.

As such, *recency-based* modeling commonly uses the *Move To Front (MTF)* algorithm to perform its compression. The dictionary is not just an arbitrarily ordered bag of observed substrings; rather, it is ordered from the most recently used entry to the least. If an entry provides a match with a substring from the input, then that entry is moved to the front of the dictionary. Later, we will see that encodings can be chosen to take advantage of the regularity modeled by LRU-ordered dictionaries—smaller bit representations can be assigned to the more recently used entries.

Not only does this form of modeling lend itself to a beneficial encoding, but it also helps maximize the effectiveness of the dictionary. Extensive dictionaries based on past input can be very large. A large dictionary not only increases the space requirements of an algorithm, but will also increase the time requirements, as more operations will be required to attempt to match the current input to each entry. By choosing to discard the least recently used entries in a dictionary of bounded size, the entries that are kept are more likely to yield matches.

The idea of using the MTF algorithm to perform compression modeling was proposed on at least four separate occasions [BCW90]. In [BSTW86], the *MTF compression algorithm* was implemented and experimentally evaluated. Although recency is a simple form of modeling, this study demonstrated that it represents a regularity that exists in many kinds of data, as the results of the study were promising.

Note that MTF is used both for recency based compression and to implement LRU page replacement. There are similarities between predicting the contents of data to be compressed and predicting future page references. (We will address in Section 2.3 the limitations of these similarities.) Both often exhibit the simple regularity that the more recently a particular pattern was observed, the more likely it is to occur soon. As with page replacement, MTF performs well for compression modeling because it forms its predictions based on a strong regularity in data that can easily be detected. It is also important to note that recency based modeling does not have to be used in isolation—it is a regularity that can be detected in combination with sequential repetitions in data, making it a robust form of modeling.

2.2.3 Encoding

Once the regularities have been found by modeling an input string, an output representation must be chosen. The process of selecting this representation and then emitting the output in this representation is known as *encoding*. While compression is not possible without effective modeling, the actual reduction in representation size depends just as much on the encoding. The modeling phase must detect regularities; the encoding phase must choose reduced bit representations for those detected regularities if the final output is to be smaller than the original input.

There are a number of approaches to encoding. While some of them have been shown to exhibit minimal space under certain assumptions, an encoding can be any choice of binary representation. Note that the most obvious encoding for any k-bit input is the "uncompressed" representation, in which all 2^k possibilities are enumerated by using k bits. While this uncompressed representation yields the smallest possible *mean* number of bits over *all* of those 2^k strings, we hope to devise an encoding that is smaller for the majority of input strings that we are likely to encounter.

Ad-hoc Encoding

A simple form of encoding can be one that comprises a set of bit combinations that reflect, in a space efficient manner, the data collected through modeling. The Wilson-Kaplan compression algorithms that we will present in Chapter 5 will use this kind of encoding. This kind of binary representation usually represents directly how the model found its regularities, and thus how a decompressor can reconstruct the original representation.

The disadvantage to this kind of encoding is that it is not likely to result in the smallest compressed output, nor necessarily be close to it. These encodings often exhibit regularities of their own that could be detected by another modeling scheme, and be further compressed. However, in exchange for suboptimal reduction in the representation, speed can be gained. As we will see in the context of compressed caching, a simple, ad-hoc encoding can be implemented to generate output quickly, yet yield sufficient compression for the context. The simplicity will also allow the decompressor to parse quickly the compressed representation.

One example of a specialized form of ad-hoc encoding, used in a number of contexts, is *difference encoding.* This approach is used for data in which many of the bits, from one symbol to the next, are the same. Rather than encode each symbol, difference encodings will represent a single symbol, and then others will be represented by their difference from that single symbol. For example, the string of integers `1001, 1000, 1003` could be encoded as `1001, -1, +3`, where the last two values are calculated in reference to the literal value `1001`.

While difference encoding can be performed quickly and simply, it should only be applied to specific kinds of data. If the data is known to comprise large values that differ by little, then difference encoding may be very applicable. Reference traces are such a context, as spatial locality implies that many references will be to pages in the same part of the address space. The *Mache* [Sam89] method of trace reduction is a lossless compression technique designed specifically for reference traces that makes significant use of difference encoding.

Huffman Encoding

While ad-hoc encoding can be simple to devise and implement, some contexts demand better compression ratios, even if increased time and space are required to achieve them. *Huffman encoding* has been shown to be an optimal *prefix coding*[5]—an encoding for which the bits used to represent any one symbol cannot be the prefix of the bits used to represent any other symbol.[6]

Note that for the common, uncompressed encoding of data, all symbols are represented by a fixed number of bits. A Huffman encoding, however, is a *variable length encoding*, in which different numbers of bits may be used to represent each symbol. Since the number of bits is not fixed, a valid variable length encoding must be a prefix code. In other words, to make it possible to decompress a variable length encoding, there can be no ambiguity as to whether a particular sequence of bits encodes one symbol, or is just the beginning of an encoding of some other symbol.

Huffman encoding requires that each symbol from the input alphabet be assigned a probability of occurrence. Once those probabilities have been assigned, a binary tree can be constructed as follows:

1. Insert all symbols as nodes, with their probabilities, into a set.

2. Select and remove from the set the two nodes the lowest probabilities.

3. Assign those two nodes as the left and right child of a new node.

4. Insert the new node into the set.

5. Repeat until only one element remains.

[5]For a proof of this property, see [CLR89]

[6]Intuitively, the term should be *prefix-free coding*, but we will use the standard terminology here.

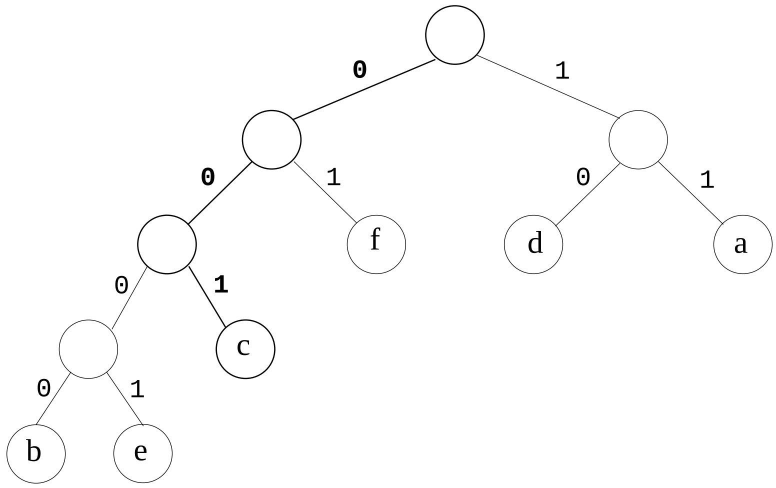

Figure 2.8: A binary tree that represents a Huffman encoding for the alphabet a, b, c, d, e, f. A symbol's encoding is described by the path from the root to the leaf that contains that symbol. The traversal of an edge to the left child is encoded by a `0`, and to the right child by a `1`. For example, the path from the root to `c` is **`left-left-right`**, or `001`.

This greedy algorithm will construct a binary tree like the one seen in Figure 2.8. Each edge in the tree is labeled with a bit, where left child edges are labeled with a 0 and right child edges are labeled with a 1. By following the path from the root to the desired symbol, the sequence of bits that encodes that symbol is revealed by sequence of binary values with which each traversed edge is labeled. Note that prefix code property is guaranteed because symbols reside only at leaf nodes.

Huffman encoding is not merely an encoding scheme—really, it is an assignment of bit representations based on a 0^{th}-order Markov model. The probabilities assigned to each symbol are analogous to the probabilities assigned to the edges of a Markov state machine that has only one vertex, and does not "remember" the last character observed. Also note that the assignment of these probabilities is performed before the input is seen. This particular kind of encoding is therefore known as *static* Huffman encoding.

Forms of *dynamic* Huffman encoding do exist. The probabilities, and thus the variable length binary representations of each symbol, are modified as the input is compressed. The most simple approach to updating the encoding would be to recalculate the probabilities and then reconstruct the encoding tree with every input symbol. However, such frequent and extensive computation is prohibitive, and so other schemes for incremental updates are have been used in practice.

Arithmetic Encoding

Note that for Huffman encoding, each symbol in the input is represented by some whole number of bits in the output. *Arithmetic coding* does not use a one-to-one mapping of input symbols to output bit strings. Rather, input symbols can be represented by partial numbers of bits. (Of course, the total output must be provided in an integral number of bits, so there may be some fraction of a bit that does not carry useful information.) Arithmetic encoding can therefore yield encodings that are even smaller than those created by Huffman encoding.

The basic idea behind an arithmetic encoding is to represent a message as a single, real valued number of arbitrary precision. As with Huffman encoding, a 0^{th}-order Markov model is assumed, although other kinds of models can be applied. The probabilities assigned to each symbol can be statically fixed, or dynamically modified as the input is processed. For the sake of simplicity, we will describe arithmetic encoding using static probabilities.

The value of the real number that is output can be computed, one input symbol at a time, as follows:

1. Initialize the range in which the output number will fall to $[0, 1)$.

2. Divide the current range according to the probabilities for each symbols. That is, if the range is currently $[0.25, 0.75)$, and the probability for the symbol `a` is 0.5, then earmark the range $[0, 25, 0.5)$ for that symbol.

3. Read a symbol from the input.

4. Restrict the range in which the output number will fall to the range earmarked for that symbol. For example, if `a` is the symbol read in the example given above, restrict the range to $[0.25, 0.5)$.

5. Repeat until no symbols remain in the input. Note that the input must end with a special *end of string* symbol.

6. Output any number within the final restricted range.

Note that the special *end-of-string* symbol is necessary to the decoding algorithm. Without it, the value 0.0 could encode *a, aa, aaa....* While this method of encoding can yield a very compressed representation, it can be computationally more expensive than ad-hoc methods.

2.3 Compressibility vs. Caching

The underlying concepts of on-line caching and of compression are similar. Both require predictions that are based on patterns in past behavior. Because of these similarities, some try to analyze one domain in terms of the other.

Measuring locality via compressibility. One popular approach to determining the locality for a program's references is to compress its reference trace. That is, given a reference trace for a program, use a compression utility in order to see how compressible that reference trace is. The assumption is that a program that exhibits good locality of reference will yield a compressible trace, while a program with poor locality will generate a poorly compressible trace.

This assumption is faulty. The cost incurred due to a poor prediction is quite different for these two domains. For compression, if the model does not predict the next input symbol well, then the cost will likely be only one or two bits of overhead in the compressed output. For caching, a poor prediction of the next page to be referenced can cause a reference slowdown of as much as six orders of magnitude.

High compressibility does not imply good locality. To illustrate how important the difference in poor-prediction cost is, consider the following example: A reference trace may contain a repeating pattern of 10,000 references to the same page (or some very small number of pages), and then a reference to a randomly chosen page.

This reference trace would be extremely compressible. The 10,000 repeated references to a very small number of pages could be reduced to less than one bit per reference. Each of the randomly chosen references to the other pages may require an expanded representation by one or two bits, but that will harm the total compressibility little.

In contrast, if the number of randomly chosen pages is much larger than memory can hold, then most of the references to those pages will require the page to be fetched from slower memory. Modern processors are fast enough to issue hundreds of millions of memory references in a second. If one in 10,000 references causes a page fault, then the processor could generate tens of thousands of page faults per second. However, a disk requires a few milliseconds to fetch a page. A disk can therefore only service hundreds of page faults per second.

Even though the few pages that are referenced so often would easily be cached, the randomly chosen references would likely be to uncached pages, and thus cause the system to thrash. Any program that causes such a hopeless level of page faults irrespective of replacement policy must be considered to have *poor* locality of reference. As such, high compressibility does not necessarily imply good locality of reference.

Low compressibility does not imply poor locality. While one may not be able to glean the quality of locality easily from compressibility, it may seem possible to rank the locality exhibited by programs through this metric. However, the example above can be augmented to demonstrate that the relative rank of trace compressibility does not indicate the relative rank of reference locality.

Consider a second reference trace that contains a uniform number of references to many thousands of pages. Such a trace is likely less compressible than the trace described above, as each reference will require more bits, on average, to represent in compressed form. However, since most systems can store a few thousand pages, the locality of this program would be excellent, as it would suffer few page faults. Compressibility provides little, if any, information about locality of reference.

Chapter 3

Trace Reduction

3.1 Introduction

Trace-driven simulation is a common approach to studying virtual memory systems. Given a *reference trace*—a sequence of the virtual memory addresses that are accessed by an executing program—a *simulator* can imitate the management of a virtual memory system. Thanks to reference traces, experiments on virtual memory management policies can be reproduced in a controlled environment. Unfortunately, these traces can be extremely large, easily exceeding the capacities of modern storage devices even for traced executions lasting only a few seconds. The size of traces impedes both their storage and processing. *Trace reduction* is the compression of reference traces (either lossless or lossy) so that they can be stored and processed efficiently.

There are many existing methods for trace reduction. However, these methods have undesirable characteristics for virtual memory simulation: Some discard so much reference information that the reduced trace introduces significant error into the simulation of common page replacement policies. Other methods make it difficult to control how much information is discarded, and thus what size memories can be simulated accurately. Some methods reduce the storage costs without reducing the number of references and thus the time required to process a trace.

We present two trace reduction methods—*Safely Allowed Drop* (SAD) and *Optimal LRU Reduction* (OLR)—that do not suffer from these deficiencies. Both allow a user to control the degree of reduction by the specification of a *reduction memory size*. SAD is the simpler of the two: it removes references that are guaranteed not to affect the LRU and OPT behavior of a trace, provided that the simulated memory sizes are no smaller than the reduction memory size. Under the same assumption, the OLR algorithm yields the shortest possible trace that can be used for exact LRU simulations in place of the original trace. OLR is useful both because it provides greater reduction than SAD and because its output gives a lower bound for the length of a reduced trace. Both algorithms are efficient in practice, and significantly reduce storage *and* processing costs.

Guaranteeing accurate simulation for LRU and OPT may not seem exciting at first. If the trace were used only with these policies, the simulation could be run only once and the results stored and re-used. Our approach is effective, however, for simulations of *many* virtual memory replacement policies. Nearly all replacement policies used or studied with real workloads are either variants or approximations of LRU in a weak sense that is sufficient for our trace reduction techniques. This similarity of common page replacement policies is hardly surprising—good replacement algorithms should not evict pages that are in current use. Such LRU variants and approximations are discussed more in Section 2.1.7.

LRU variants (e.g., GLRU [FLW78], SEQ [GC97], FBR [RD90], EELRU [SKW99]) keep the k most recently referenced pages in memory, even though not all m pages in memory ($m > k$) are the m most recently accessed (as they are in pure LRU). Our approach to trace reduction is applicable in all such cases for a reduction memory size of at most k. Even small values of k (10 to 100) are enough to allow

OLR and SAD to achieve reduction factors of up to several orders of magnitude, while guaranteeing exact simulations.

SAD and OLR are also useful when studying LRU approximations, such as CLOCK and SEGQ. These replacement policies ignore the same high-frequency referencing information that nearly any replacement policy will ignore, and that SAD and OLR discard from traces. This information is ignored not because these are LRU approximations, rather than pure LRU, but because references to recently used pages matter only to replacement on the timescale of much smaller memories than are managed by virtual memory systems. These high frequency references don't influence replacement decisions, nor does the hardware in real systems support the efficient collection of such information.

We show that the error introduced by SAD and OLR for both CLOCK and SEGQ replacement simulations is small—under 2% in number of faults for most cases. We also compare SAD and OLR to *Stack Deletion (SD)* [Smi77], which is a commonly known technique for removing high frequency reference information from virtual memory traces. Given reduced traces of comparable size created using all three methods, SAD and OLR introduce less error on average into CLOCK and SEGQ simulations. Further, SD introduces more error into simulations based on the reference traces used here, many times in excess of 10%, than the original research on SD would indicate.

Additionally, the ability of the SAD algorithm to produce reduced traces valid for exact OPT simulations is a pleasant side-effect: it means that a single trace can be used for all experiments in a virtual memory study. Such studies often compare a new algorithm to LRU and OPT.

3.2 Background and Motivation

Given the importance of trace reduction, it is not surprising that there has been a wealth of research work on reduction techniques. It is impossible to exhaustively address all the approaches here. Instead, Section 3.2.1 presents an overview, and Section 3.2.2 positions our method relative to the most closely related techniques. A good further reference is the recent survey of trace-driven simulation by Uhlig and Mudge [UM97].

3.2.1 Overview of Related Work

Lossless Reduction Techniques

A straightforward approach to lossless trace reduction is to apply standard data compression techniques on a trace. Simple Lempel-Ziv compression algorithms provide reduction factors of about 5 for typical traces [UM97]. Higher degrees of reduction can be achieved by combining compression algorithms with difference encoding techniques. The best known such instances are the Mache [Sam89] and PDATS [JH94] systems, which explore spatial locality in the reference trace to encode it differentially. Subsequently, standard text compression techniques are applied and result into further reduction of its size.

It is useful to note again the limitation of "general purpose" compression algorithms, as some believe LZ compressors to be. While these algorithms do achieve significant reduction by adapting to some of the regularities in reference traces, they are by no means the most effective compressors. Simple, efficient compression algorithms like PDATS and Mache are domain-specific compression algorithms that can achieve much tighter compression than LZ algorithms. If the regularities of the data for a domain are known, and compression is needed, a domain-specific compression is likely to perform better than an adaptive, general purpose compression algorithm.

Lossless techniques can be used to reconstruct a trace accurately for all purposes. Nevertheless, the compression ratios achieved are not as high as those possible with lossy trace reduction. More importantly, traces need to be uncompressed before simulation is performed. Thus, the reduction gains of lossless compression do not translate into simulation speedups.

Lossy Reduction Techniques

When performing trace reduction, one usually has some knowledge of the future uses of the trace. Lossy trace reduction techniques attempt to exploit such knowledge so that the trace size is reduced dramatically but enough information is maintained for the intended uses.

The simplest lossy reduction technique is *blocking* [AH90]. Blocking replaces references to individual addresses with references to memory pages. Subsequent references to addresses within the same page can then be reduced to a single reference. This reduction does not affect the simulation of *time-independent paging algorithms*—algorithms that do not consider the exact time of each reference in making replacement decisions. Such algorithms are LRU, OPT, etc., but not, for instance, Working Set [Den76]. Blocking is so widely applicable that it is practically assumed in most simulation work. For the remainder of this paper, when we refer to an *original* trace, we are referring to a blocked trace.

Blocking is also interesting in that it is exploiting a different kind of regularity than most reduction techniques. Whereas other lossy reducers concentrate on the *temporal* locality of a program trace, blocking exploits *spatial* locality and results in an extra significant factor of reduction.

Many trace reduction methods are kinds of *trace sampling* or *trace stripping* (see [Puz85]). Both are better suited for high-speed hardware cache simulations, as they introduce inaccuracy into fully-associative virtual memory policy simulations.

The remaining types of lossy trace reduction methods are oriented towards virtual memory simulations. These techniques address the same concerns as our algorithms and are directly comparable to them. The next section discusses such related reduction techniques in detail. Because SAD and OPT are also lossy techniques, other work in this area is the most relevant for comparison to our new methods.

3.2.2 The Value of Our Techniques

Our approach fills a prominent gap in the spectrum of trace reduction techniques. Most existing techniques either do not guarantee accurate simulations or do not achieve the same high reduction factors as our method. We isolate three approaches that stand out as particularly related to ours.

- Smith's *Stack Deletion (SD)* [Smi77] consists of only keeping references that cause pages to be fetched to a k-page LRU memory. SD is directly comparable to the SAD algorithm. Both techniques are very simple and have similar preconditions: for both, the reduction memory size chosen when the trace is reduced determines the minimum simulation memory size for which the results will be accurate. However, SAD guarantees that no error is introduced for LRU and OPT simulations, while SD does not guarantee exact results for any replacement policy. Smith experimentally demonstrated that the error introduced by SD is small. However, that error is small only if the reduction memory size is much smaller than the simulated memory (typically 20% to 50% of its size). Hence, SAD can use a much larger reduction memory, which will yield greater reduction, and still achieve exact results. Additionally, we will show that SD introduces larger error than both SAD and OLR for CLOCK and SEGQ simulations for reduced traces of the same size. In conclusion, SAD and OLR are both safer (i.e., introduce less error) and more effective (i.e., yield smaller traces useful for comparable purposes) than SD.

- The technique of Coffman and Randell [CR70] can be seen as an alternative to both SAD and OLR for LRU simulations. Their approach consists of using the *LRU behavior sequence*—the sequence of pages fetched and evicted—for a k-page LRU memory to perform exact simulations of LRU memories no smaller than k pages. The behavior sequence is typically very short, even for small values of k. The biggest drawback of the Coffman and Randell approach, however, is that the product of reduction is not itself a reference trace. In the best case, the simulator, as well as any other tools (e.g., trace browsers), will need to change to accept the new format. This is a practical burden to the simulator implementors and makes it hard to distribute traces in a compatible form. This is the main reason why this simple technique has not become more widespread. Our OLR algorithm is complementary to the approach of Coffman and Randell: it offers an efficient way to turn the behavior sequence format into the shortest possible trace exhibiting this LRU behavior. Other advantages of our algorithms exist. For instance, SAD is also applicable to OPT simulations, and we will show that both SAD and OLR introduce little error for simulations of CLOCK and SEGQ.

- Just like our techniques, the reduction method used by Glass and Cao [GC97] is applicable to exact virtual memory simulations of some policies. Like Coffman and Randell's method, the Glass and Cao technique suffers from the need to modify the simulator to accept the reduced trace format, which is not a reference trace. The modifications are far from trivial, and it can be hard to use the reduced trace information for simulations of policies other than those studied in [GC97] (LRU, OPT, and SEQ—an experimental replacement algorithm that could not itself be exactly simulated using their own reduction method). Another drawback of this technique is its lack of control over the interesting memory ranges. It is not possible to specify directly the memory sizes for which the simulation should be exact. Instead, the trace filter allows only indirect control over the minimum memory sizes for which the simulation is valid; worse, that minimum size cannot be determined until *after* the trace has been gathered. The method seems to be less efficient than our approach, at least for LRU simulations. We did not have access to the traces used by Glass and Cao in unreduced form, but were able to derive the OLR-reduced form of these traces (directly from the Glass and Cao reduced traces). These OLR-reduced traces were several times shorter than the reduced form used by Glass and Cao, both in terms of absolute size and in terms of significant events. The detailed results of this comparison will be presented in Section 3.5.

Other applications of our algorithms are possible. Because of its optimality properties, OLR is ideal for the purposes of trace analysis. It provides an estimate of the amount of reordering done inside an LRU memory. This is useful for evaluating whether a trace will behave similarly under LRU and under LRU approximations (e.g., CLOCK or SEGQ implementations). Another possible application of OLR is in trace synthesis. Given any exact sequence of fetched and evicted pages from an LRU memory, OLR can produce a minimum length trace that will cause the same fetches and evictions. This could provide an alternative to statistical trace synthesis techniques (e.g., [Bab81]).

Finally, we should mention that our techniques are complementary to reduction algorithms that exploit different principles. Since the output of either of our algorithms is itself a trace, other trace reduction techniques can be applied (e.g., [JH94, AH90]). As we will see, simple file compression of our reduced traces with the `gzip` utility yields much smaller files, further decreasing storage requirements.

3.3 The Algorithms

3.3.1 Safely Allowed Drop (SAD)

Full traces commonly contain a large number of references that are ignored by virtual memory replacement policies. These references account for the majority of space required to store a trace, and consume the majority of time required to perform a virtual memory simulation. Safely Allowed Drop (SAD) removes references from a trace that do not affect the order of fetches into and evictions from an LRU memory of some user-specified size.

We will show that SAD allows for exact simulations not only of LRU, but also of OPT. We will also show, in Section 3.4, that it introduces very little error into the simulation of LRU approximations such as CLOCK and SEGQ.

Finding References to Drop

Recall that in Chapter 2, we defined *LRU distance* as the number of *other* pages referenced between the two references to the same page. The idea behind SAD is simple: *For any three references to the same page in a trace, if the LRU distance between the first and third references is less than k, then removing the middle reference does not affect the outcome of LRU and OPT simulations on memories of size no greater than k.* Section 3.3.1 describes why the elimination of these middle references has no effect on LRU and OPT.

SAD is an application of this observation. The user specifies a reduction memory size k. SAD searches the trace from left to right in search of triplets of the above form—references to the same page, such that the LRU distance between the first and third reference is less than k. All middle references of such triplets are eliminated.

Figure 3.1 shows three references to page `A`. The LRU distance between the

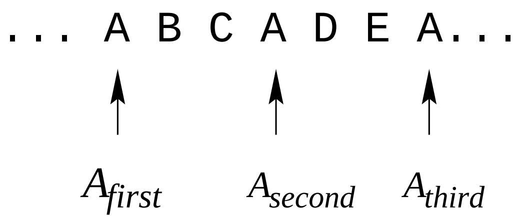

Figure 3.1: A_{second} can be eliminated because the LRU distance between A_{first} and A_{third} is less than the reduction memory size of five pages.

first reference A_{first} and the third reference A_{third} is 4, as there are four distinct pages (`B`, `C`, `D`, and `E`) that are referenced between A_{first} and A_{third}. If the memory size chosen for reduction is at least `5`, then we can safely drop A_{second} without affecting the results of an LRU or OPT simulation.

Nearly all programs frequently reference pages that were recently used. Due to this temporal locality, references eliminated by SAD constitute the vast majority of references in usual program traces, even for small reduction memories.

SAD Algorithm Implementation

SAD needs only to determine LRU distances between pairs of references to the same page in order to find middle references that can be eliminated. The search proceeds from left to right, allowing reduction to be performed in a single forward traversal of the original trace.

As the trace is processed, the algorithm maintains an LRU queue of the requested size. It also stores some of the most recently input references from the original trace. By keeping both the LRU queue and a recent history of references, the algorithm can find groups of three references to the same page where the LRU distance between the first and third references is less than the reduction memory size. Therefore, this information is enough to find middle references that can be eliminated.

Although it is necessary to store recent references to find these triplets, the number of references can be bounded. It is only necessary to store at most $2k+1$ of the most recent references in order to find the LRU distance between first and third most recent references to a page. Specifically, the implementation needs to store at most the two most recent references for those pages in k-page LRU queue, plus a third reference to one of those pages as a triplet is found. If a triplet is found, the middle reference is eliminated, and again only two recent references for that page are stored. Triplets could not possibly be found for other pages, so their recent references need not be stored.

Given something like a hash table to help find recent references to pages, performing this reduction is little more than an augmented LRU queue simulation; it can be executed efficiently.

Exact Simulation of LRU and OPT

If SAD reduces a trace using a k page reduction memory, then that reduced trace can be used for the exact simulation of both LRU and OPT memories that are at least k pages.

Exact LRU simulation. Consider an LRU queue of unbounded length and its contents for both the unreduced and the reduced trace. By dropping references, SAD allows pages to drift further away from the front of the LRU queue, as each page is referenced less often. These pages, however, are guaranteed to be in the first k positions of the queue; each eliminated reference is following by another reference to the same page that is an LRU distance less than k from the previous reference.

Other pages are not adversely affected by removing a reference. Their position in the LRU queue can only be closer to the top for the reduced trace than it would have been for the original one. The only positions in the queue that may have different contents for reduced traces are the ones from 1 to k. Therefore, the results of LRU simulations for memories of size k or larger will be identical for the reduced and the unreduced trace.

We illustrate this argument by examining Figure 3.1. For a memory of size 5 or larger, `A` will remain in memory between A_{first} and A_{third}. The middle reference A_{second} has no effect on LRU replacement and if it is dropped, the reference A_{third} will ensure that `A` is not incorrectly evicted.

Exact OPT simulation. Consider again the three references in Figure 3.1. When OPT must choose a page for eviction, it selects the resident page first referenced furthest in the future. We can show, case by case, how the removal of A_{second} does not affect the replacement decisions made by OPT:

- If OPT is processing references before A_{first}, then the removal of A_{second} will not affect its eviction choices, as A_{first} is the reference that OPT will use to determine whether `A` is evicted.

- If OPT is processing references between A_{first} and A_{third}, then we already know that fewer than k distinct pages are referenced between those two references to `A`. Note also that the page currently being referenced is not already in memory (since it caused a replacement) and cannot be a candidate for eviction, making the number of other distinct referenced pages preceding A_{third} less than $k-1$. Therefore, if the memory size is at least k, page `A` cannot be the one first referenced furthest into the future (because of reference A_{third}). The absence of A_{second} does not affect the replacement decision.

- If OPT is processing references that follow A_{third}, then none of these three references to page `A` will affect decisions. OPT examines future references to make its decisions, so the missing reference A_{second} will have no effect.

3.3.2 Optimal LRU Reduction (OLR)

The SAD algorithm obtains significant reduction factors for actual traces. Nevertheless, SAD-reduced traces are not necessarily the smallest for which either LRU or OPT simulations are exact. For instance, consider the reference sequence:

`A B C B A C D A B D`

Applying SAD with a reduction memory of 3 pages to this trace yields no reduction. Nevertheless, the shorter trace

`A B C A D B`

has exactly the same LRU behavior as the original for a memory of size 3 or larger. Recall that the LRU behavior of a trace for a memory of size k is the sequence of pairs of pages fetched into and evicted from memory when the given trace is applied. The LRU behavior of the two above traces for a memory of size 3 is:

$\langle \mathtt{A}, NF \rangle, \langle \mathtt{B}, NF \rangle, \langle \mathtt{C}, NF \rangle, \langle \mathtt{D}, \mathtt{B} \rangle, \langle \mathtt{B}, \mathtt{C} \rangle$

where the special value NF denotes that the memory is not full and, hence, the insertion of one element does not cause the eviction of another.

The importance of LRU for virtual memory systems has motivated the design of the OLR algorithm for computing such optimally short sequences. OLR takes a reference trace as input and outputs the smallest trace that has the same LRU behavior as the input for a memory of size k or larger. The output of OLR is only a function of the behavior—two different input traces exhibiting the same behavior for an LRU memory of size k will produce the same output. Hence, the first step of OLR is to simulate the input trace on an LRU memory of size k and derive its behavior sequence. That sequence is the input to the OLR_CORE algorithm shown in Figure 3.2. Following common terminology, we will often use the term *block* as a synonym for *page* and *touch* as a synonym for *reference.*

Some explanation of the conventions followed in the algorithm description is necessary: The input sequence, *behavior*, is a Coffman and Randell LRU behavior

```
OLR_CORE(behavior, k)
  lkahead ← 0, current ← 0
  future_fetch ← ∅, prev_evict ← LowerLimit
  q ← ∅(k)
  ▷ ∅(k) denotes an empty LRU q of size k
  while behavior[current] ≠ Last
    do must_touch ← q.BLOCKS_AFTER(behavior[current].evict)
      lkahead_done ← False
      while behavior[lkahead] ≠ Last and ¬lkahead_done
        do if behavior[lkahead].evict ∈ future_fetch
            then lkahead_done ← True
            else
              if q.MORE_RECENT(prev_evict, behavior[lkahead].evict)
                then PRODUCE_REF(behavior[lkahead].evict)
                  must_touch ← must_touch \ {behavior[lkahead].evict}
              prev_evict ← behavior[lkahead].evict
              future_fetch ← future_fetch ∪ {behavior[lkahead].fetch}
              lkahead ← lkahead + 1
      for x ∈ must_touch
        do PRODUCE_REF(x)
      PRODUCE_REF(behavior[current].fetch)
      future_fetch ← future_fetch \ {behavior[current].fetch}
      current ← current + 1

PRODUCE_REF(block)
  q.TOUCH(block)
  OUTPUT(block)
```

Figure 3.2: OLR_CORE accepts the sequence of fetched and evicted pages for a k-page LRU memory, and produces the shortest reference trace that would cause the same behavior for that same memory.

sequence represented as an array for simplicity. The special value LAST signals the end of the sequence. OLR_CORE uses a data structure q, which is an LRU queue augmented with two operations:

- *blocks_after(block)*: returns the set of blocks touched less recently than *block*, but still in the data structure (i.e., within the last k distinct blocks touched). If *block* has the special value NF, the returned set is empty (this is useful for uniform treatment of the boundary case where the structure is being filled up).

- *more_recent*$(block_1, block_2)$: returns a boolean value indicating whether $block_1$ was touched more recently than $block_2$. If $block_1$ has the special value LOWERLIMIT, or $block_2$ has the special value NF, FALSE is returned.

Here we will discuss the algorithm at an intuitive level which we hope will convey some of the insights behind its development. A proof of correctness of the OLR_CORE algorithm can be found in [Sma98].

Recall that OLR_CORE takes as input the sequence representing the behavior of a trace for an LRU memory of size k. This behavior sequence consists of pairs of fetched and evicted pages for the LRU memory. The sequence of references to fetched pages has to be a subsequence of the output of OLR_CORE, as every reference that causes a fetch operation must remain in the reduced trace so that the sequence of fetches is preserved. The purpose of OLR_CORE is to find the minimum set of *extra* references that need to be added so that the sequence of evictions is also preserved. That is, each reference in the reduced trace either causes the fetch and eviction specified by the behavior sequence, *or* it causes a reordering of the simulated LRU queue such that a later fetch will cause the *correct* eviction.

At every point during the algorithm's execution, the q data structure reflects the contents of an LRU queue of size k, to which the output (up to the present point) has been applied. At every iteration of the outer loop of the algorithm in Figure 3.2 (lines 4-20) the *current* index points to successive elements of the input behavior sequence.

For each iteration of the outer loop, before a reference that causes a fetch is output (by line 18), all pages in the LRU queue that were less recently referenced than the corresponding expected evicted page are touched. The set *must_touch* contains exactly these pages (line 5), and lines 16-17 ensure that they get touched. This way, for a reference causing a "fetch", the least recently touched page in the LRU queue is the corresponding evicted page, as indicated by the input.

Before we describe the essence of OLR_CORE it is useful to briefly review a simple algorithmic problem:

> Given an LRU queue and a desired recency ordering for some of the pages currently in the queue, what are the fewest page references required to reorganize the queue so that the desired ordering holds?

One can show that there is a simple algorithm to produce such a minimal sequence of references:

> Examine the desired ordering in inverse order (i.e., least recent element first). Find the first page that is out-of-order in the queue (i.e., it is less recent than its previous page in the desired ordering). Produce references for this page and all subsequent pages in the desired ordering.

This is exactly the algorithm implemented by the inner loop of the OLR_CORE algorithm (lines 7-15). The LRU queue that is modified is q. The desired ordering is described by the evicted pages in the smallest subsequence of *behavior* that begins at position $current+1$ and contains events where a page is fetched in the LRU queue and the same page is evicted. The latter condition is detected using the *future_fetch* set in the algorithm of Figure 3.2, which contains the pages fetched after position *current* in the behavior sequence. The *lkahead* variable is used as a lookahead index for elements of *behavior* beyond position *current*. When the evicted part of such an element is found in *future_fetch* (line 8), the inner loop ends.

In short, the essence of the OLR_CORE algorithm is that for each reference to a page not in the LRU queue, as few as possible extra references to pages already in the LRU queue are produced, so that the recency order in the queue matches the expected eviction order *up to the point where a page is evicted that has not yet been last fetched*. Intuitively, the reason the latter condition is necessary is that the entire LRU queue will need to be reorganized (i.e., every page will need to be touched) between the points that a page is fetched and the same page is evicted. Hence, no benefit can be achieved by "looking further" than the eviction of pages that have not yet been last fetched. No reordering should be done, since all pages will be reordered later, when they are last referenced before one of them is evicted.

Finally, we should note that OLR_CORE is very efficient, so that the main component of the running time of OLR is the LRU simulation performed on the input trace to derive its behavior sequence [Sma98]. That is, OLR execution is about as fast as a simple LRU simulation on the input trace for a memory of size k. The algorithm performs just a single forward pass with bounded look-ahead (at most k elements) and, thus, is ideal for online applications.

3.3.3 Trace Manipulation Issues

In our discussion of SAD and OLR we used a simplified form of reference traces (only containing address information for the page being referenced). Real trace formats may need to contain other information, such as the operation that required a reference (i.e. an instruction, read, or write operation), the exact instruction causing that operation, the program counter (or any timer information), etc. Additionally, a trace may need to be re-blocked so that experiments can be conducted for different page sizes. Such standard trace manipulation is perfectly compatible with both SAD and OLR. For instance:

- **Re-blocking**: a reduced trace for a reduction memory of size k can be re-blocked for any larger page size and simulations will continue to be accurate for memories of size k or larger (note that the size refers to the number of pages—the actual minimum memory size in Kbytes for which simulations are exact is larger after the re-blocking). This is a consequence of the stack algorithm [MGST70] properties of LRU and OPT.

- **Maintaining Dirtiness Information**: many virtual memory studies measure the cost of writing dirty pages to a backing store upon eviction. Such studies require traces in which each reference is marked as a *read* or *write* operation. Both SAD and OLR can be augmented to tag references with the appropriate operation.

  In order to maintain the dirtiness information about each page in reduced traces, the reduction methods must notice which pages would be modified by a *write* operation while in a k page LRU memory. Since both methods maintain such a memory during reduction, an implementation can record whether a page is dirtied while in that memory. If a page is dirtied while in the reduction memory, then the last reference to that page before it is evicted is marked as a *write* operation. A simulation based on the reduced trace will mark the page as dirty before it is evicted from a k page or larger memory.

- **Maintaining Timing Information**: timing information is trivial to maintain for SAD, since the algorithm only removes references from the original trace. For OLR, where reference reordering may occur, it makes sense to keep time information for references causing a page to be fetched into memory. These are guaranteed to be exactly the same (and, hence, in the same order) as in the original trace.

3.4 Experimental Results

We applied our trace reduction methods to traces collected both on Windows NT and UNIX platforms. The nine Windows NT traces include the full set of the commercially distributed traces gathered using the utility `Etch` [LCB+98]. These include well-known Windows NT applications (Acrobat Reader, Netscape, Photoshop, Powerpoint, Word) as well as various other programs (CC, Compress, Go, Vortex). The six UNIX traces (Espresso, GCC, Grobner, Ghostscript, Lindsay, P2C) were gathered using `VMTrace`, our portable tracing tool based on user level page protection; these traces are freely available on our web site. The Windows NT traces were blocked for 4 Kbyte pages so that they would be appropriate for virtual memory simulations. The UNIX traces were generated as references to 4 Kbyte pages.

In this section, we show the reduction factors achieved over a range of reduction memory sizes. We also used reduced traces to simulate both the CLOCK and SEGQ replacement policies. These two policies cannot be simulated exactly using reduced traces, but we show that the error introduced into their simulation is small in practice. We also show that the error introduced is significantly less than with stack deletion [Smi77], a well known reduction method. We chose to simulate CLOCK and SEGQ because they are the two replacement policies most used in real systems. As approximations of LRU, they are similar to many replacement policies that discard information about references to the most recently used pages.

3.4.1 Reduction Results

Each of the traces was reduced using both SAD and OLR over a range of reduction memory sizes. Recall that the "original" traces are blocked on 4 Kbyte pages, and yet are hundreds of Mbytes to a few Gbytes each. We measured the number of bytes required to store the original trace and each of the reduced traces. Because each reference in these traces is a text representation of the virtual memory page number in hexadecimal, each record comprises at most (and usually exactly) five bytes. Thus, there is a direct correspondence between number of bytes and number of records in a trace.

The plots in Figures 3.3 through 3.8 show the reductions achieved by SAD and OLR on six of the fifteen original traces. The curves shown plot the reduction ratio achieved as a function of increasing reduction memory size. We chose to show the reduction results from three of the original traces per platform because remaining programs show similar increase in reduction with memory size, as well as equally high reduction factors.

Note that the reduction factors increase quickly as the memory size grows. The reduction achieved for a particular reduction memory size is a direct result of the locality exhibited by the traced program. Since the vast majority of references are to pages that have been recently used, a small reduction memory can yield large benefits. Note that the size of the OLR-reduced trace is a good measure of program locality: it is the smallest trace that has the same LRU behavior as the original for a memory at least as large as the reduction memory.

Since many virtual memory systems simulate hundreds or even thousands of pages, traces can be made hundreds of times smaller while still being appropriate for experimental studies. Using a reduced trace can allow a researcher to perform simulations that much more quickly, as the simulation time is usually proportional to the length of the input trace.

Also note that SAD achieves reduction factors close to those of OLR. Although SAD is a much simpler algorithm, it provides nearly optimal reduction, while still allowing for exact OPT simulation as well as exact LRU simulation.

It is hard to tell from our plots if high reduction ratios can be achieved

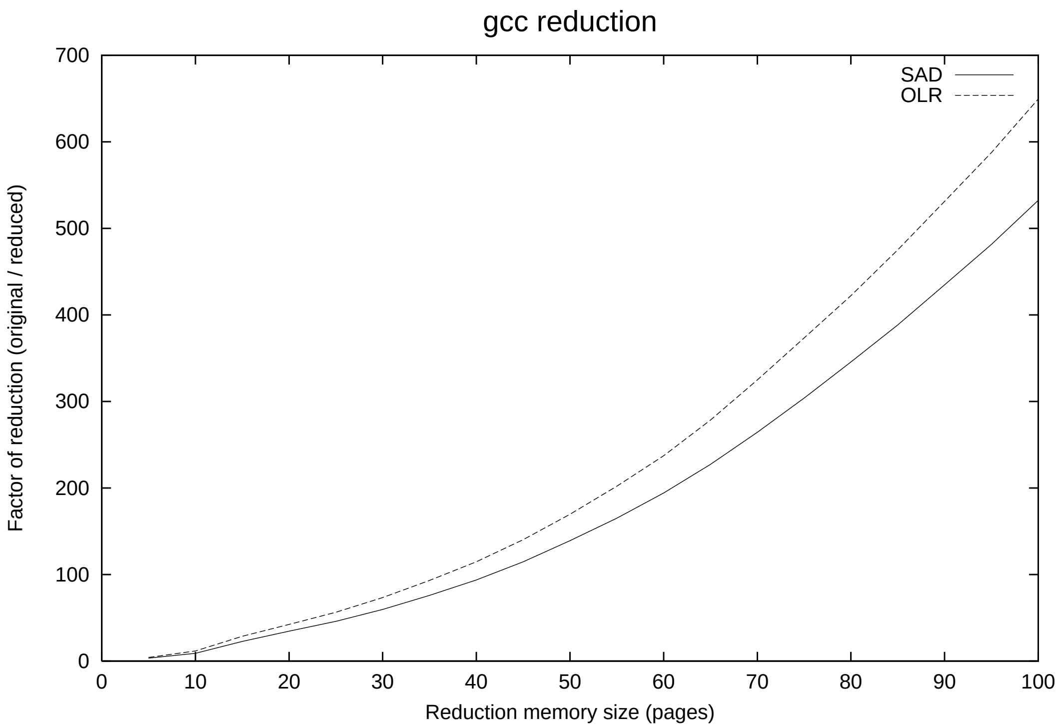

Figure 3.3: SAD and OLR reduction factors over many reduction memory sizes for **cc1**.

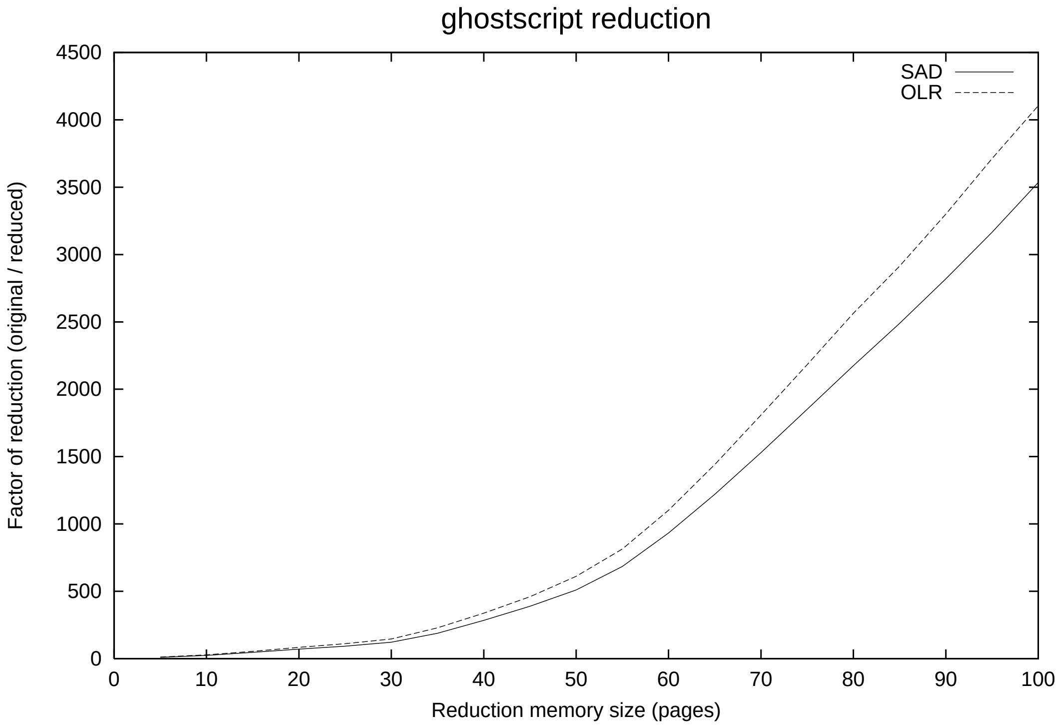

Figure 3.4: SAD and OLR reduction factors over many reduction memory sizes for **GhostScript**.

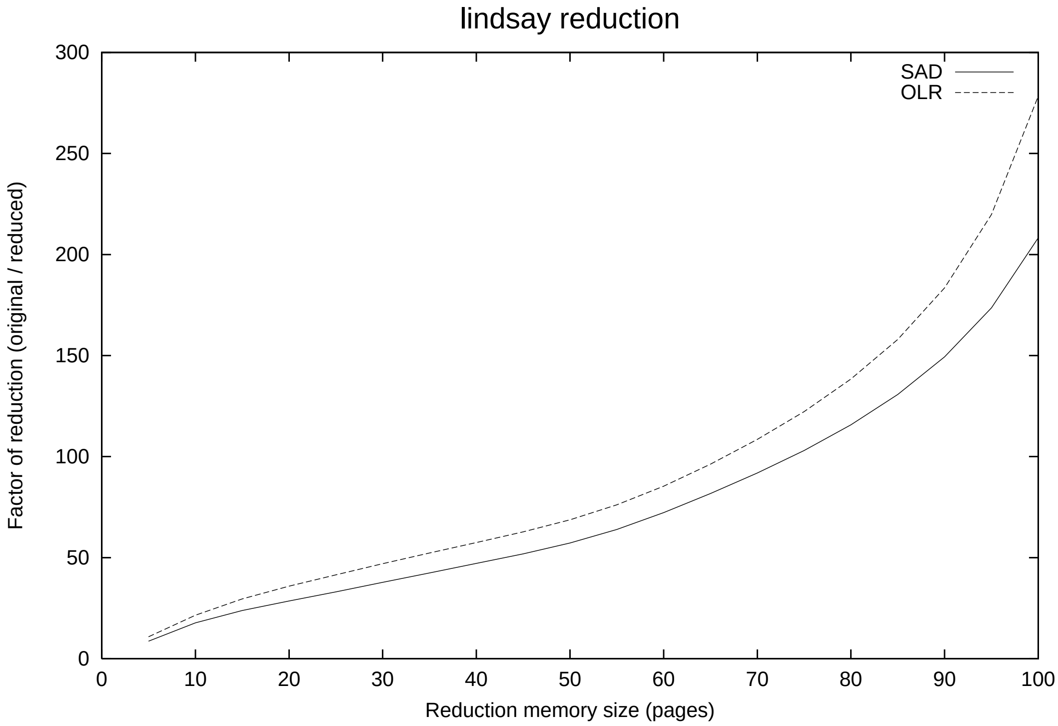

Figure 3.5: SAD and OLR reduction factors over many reduction memory sizes for **lindsay**.

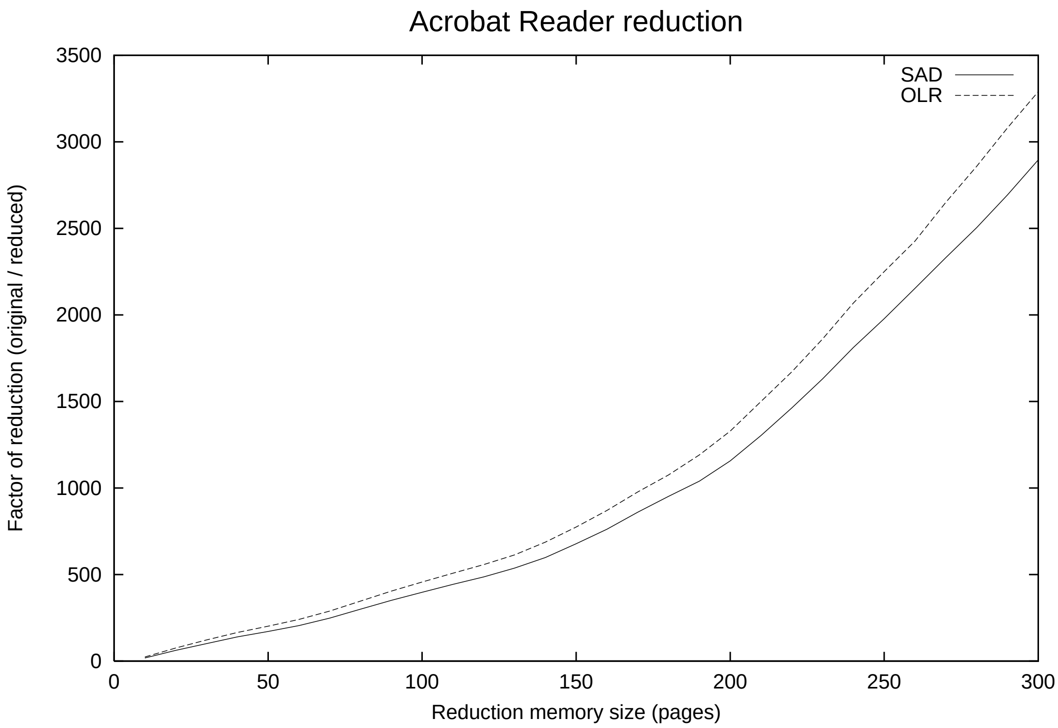

Figure 3.6: SAD and OLR reduction factors over many reduction memory sizes for **Acrobat Reader**.

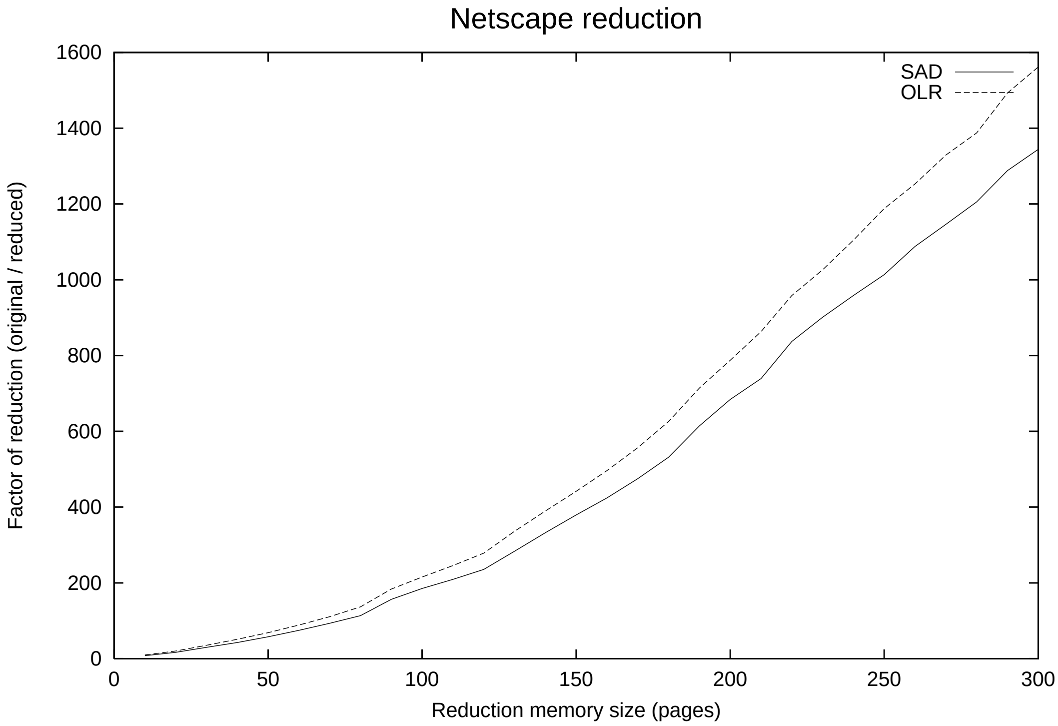

Figure 3.7: SAD and OLR reduction factors over many reduction memory sizes for **Netscape Navigator**.

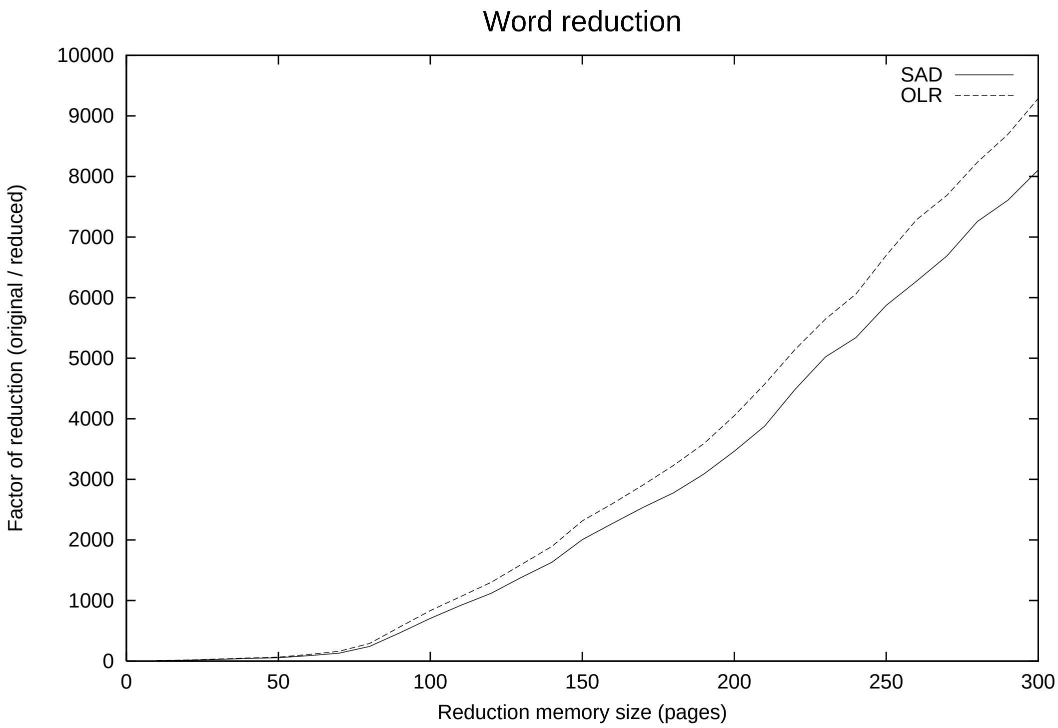

Figure 3.8: SAD and OLR reduction factors over many reduction memory sizes for **Microsoft Word**.

for small reduction memory sizes. As we show in Table 3.1, both SAD and OLR perform very well even for very small reduction memories (20 pages for the Windows NT plots and 5 pages for the Unix plots, as the Windows NT programs have much larger footprints). These sizes were chosen to show that significant reduction can be achieved even with a reduction memory that is likely much smaller than the smallest desired simulation memory.

Trace	**Reduction memory size**	**Reduction ratio**	
		(SAD)	**(OLR)**
acroread	20	62.01	75.72
cc1	20	16.12	19.52
compress	20	7.32	8.11
go	20	5.16	6.34
netscape	20	16.76	20.24
photoshop	20	61.06	72.76
powerpoint	20	10.81	12.66
vortex	20	7.04	8.68
winword	20	14.62	18.01
espresso	5	29.03	43.44
gcc	5	3.39	4.31
grobner	5	8.17	10.78
ghostscript	5	9.97	12.26
lindsay	5	8.66	10.82
p2c	5	5.39	6.91

Table 3.1: Even for small reduction memories, significant reduction factors can be achieved.

It is worth noting that our reduced traces can be further compressed by applying lossless trace reduction techniques (for instance, [JH94, Sam89]). Even though we did not experiment with any such methods, we found that SAD and OLR reduced traces are highly compressible using standard text compression tools. Table 3.2 shows the compression factors achieved by the Unix `gzip` utility on our reduced traces (the ratios shown are "reduced trace size" divided by "compressed reduced trace size").

The results below are not representative for all reduction memory sizes. As reduction memories become larger (and reduced traces become dramatically smaller), compression factors shrink. Eventually, compression ratios approach a ratio of 3:1, which is largely an artifact of representing each reference as text. These traces, however, are thousands of times smaller than the originals, and their storage requirements are negligible.

Trace	**Reduction memory size**	**gzip compression ratio (SAD)**	**gzip compression ratio (OLR)**
acroread	20	31.21	25.48
cc1	20	19.3	18.59
compress	20	17.55	14.46
go	20	13.77	12.25
netscape	20	26.48	21.52
photoshop	20	74.76	64.11
powerpoint	20	30.73	25.08
vortex	20	40.52	42.12
winword	20	38.5	32.74
espresso	5	13.74	14.03
gcc	5	13.6	11.67
grobner	5	11.58	10.36
ghostscript	5	22.2	20.55
lindsay	5	6.8	5.36
p2c	5	9.71	8.69

Table 3.2: Reduced traces are often highly compressible.

3.4.2 CLOCK and SEGQ Simulations

We simulated both the CLOCK and SEGQ replacement policies using traces reduced by SAD, OLR, and Smith's SD method. The results of these simulations were compared with simulations based on the original, unreduced traces.

We chose these two policies not only because they are so common, but also because they are similar to any page replacement policy likely to be used in practice. They are approximations of LRU because recency information tends to be an excellent predictor of future reference patterns. They also discard information about recently referenced pages because of the hardware available in all machines. If hardware reference bits are supported, CLOCK can be used. SEGQ was designed for machines that did not have such hardware support, yet allowed efficient recency based page replacement.

Our CLOCK simulator simulated a single-hand, two-reference-bit implementation. Interestingly, the reduced traces have removed the high-frequency, small-memory timescale information that causes this kind of basic CLOCK implementation to suffer from the timescale relativity problems mentioned in Section 2.1.7.

We also simulated the SEGQ replacement policy with the original and reduced traces. These simulations used a fixed-size FIFO queue, and we then examined the results for LRU queues of all possible sizes.

Error Introduced by SAD and OLR

While SAD and OLR cannot be used to perform exact simulations of CLOCK and SEGQ, we found that little error is introduced into simulation if the ratio of simulation memory to reduction memory is sufficiently large. (The error is defined as the absolute value of the difference in the number of page faults incurred using the unreduced and reduced traces.) In practice, a ratio of 5:1 yields uniformly low error. A ratio of 2:1 also yields small error for the majority of memory sizes, but sometimes introduced unacceptable error in excess of 10%—large enough to lead to erroneous conclusions due to inaccurate results. Programs that occupy a small footprint yielded our largest errors, while programs with a large footprint suffered the smallest errors. Overall, the larger this ratio, the better the results will be.

We can summarize the results for all of the simulations over all of the source traces in a few observations. (Graphical depictions of the error introduced for selected reduction memory sizes can also be found below, in Figures 3.9 to 3.20, where our methods are compared with SD.) These observations are valid for simulations in which the simulated memory to reduction memory ratio is at least 5:1, and at least 1,000 paging events occur (a virtual memory study with traces causing fewer faults is unlikely).

- For the vast majority of memory sizes, less than 2% error was observed for both reduction methods.
- Under CLOCK, SAD never introduced more than 3% error. OLR performed slightly worse than SAD on average, and in isolated cases exhibited nearly 10% error.
- Under SEGQ, neither OLR nor SAD exhibited more than 6% error. OLR performed as well as SAD on average.
- The reduced traces sometimes caused too many misses, and sometimes too few. However, there was no pattern nor bias for reduction or increase in miss numbers.
- The smaller the footprint of the program, the larger the observed error. For programs with larger footprints (at least hundreds of pages), error was often near zero for all memory sizes.

It is crucial to note that the ratio between the simulation memory size and the reduction memory size has a large effect on how much error is introduced. Most virtual memory studies are of simulated memories with sizes in the hundreds or thousands of pages. We have shown that large reduction factors can be achieved with reduction memories whose sizes are in the tens of pages. It should therefore be possible to produce significantly reduced traces with a ratio of at least 10:1 to allow for acceptably accurate simulations.

Comparison to Stack Deletion

An often referenced form of trace reduction is Smith's *stack deletion* (SD) [Smi77]. It is interesting to compare SD to our reduction techniques because its value has been demonstrated exclusively through experimental arguments. SD does not guarantee exact simulations, but has been shown to introduce small error into the simulation of replacement policies (namely, LRU, OPT, and CLOCK). We compared SAD and OLR to SD with our suite of fifteen traces and found that our techniques, particularly SAD, consistently yield smaller error.

For each reduction method, we chose a reduction memory size that would yield a reduced trace that was 100 times smaller than the original (that is, the traces for all three methods were approximately the same size). An alternative would be to use the same reduction memory size for each method. This would be unfair for SD since it keeps less information than both OLR and SAD.

CLOCK results. We performed CLOCK and SEGQ simulations using each reduced trace. A subset of the CLOCK results are shown in Figures 3.9 through 3.14. These plots show the percent error (i.e., difference in number of page faults) introduced by SAD, OLR, SD on CLOCK simulations. The traces from `go` and `grobner` were chosen to represent programs that use a small footprint, with 61 and 228 pages respectively. The programs `gcc` and `ghostscript` occupy medium sized footprints of 450 and 551 pages each. `acrobat reader` and `netscape` are larger footprint programs that use 1914 and 1022 pages, respectively.

For each of the programs, SAD and OLR match or exceed the accuracy provided by SD. On average OLR introduces more error than SAD but less than SD. SAD provides smaller error than SD at almost every memory size.

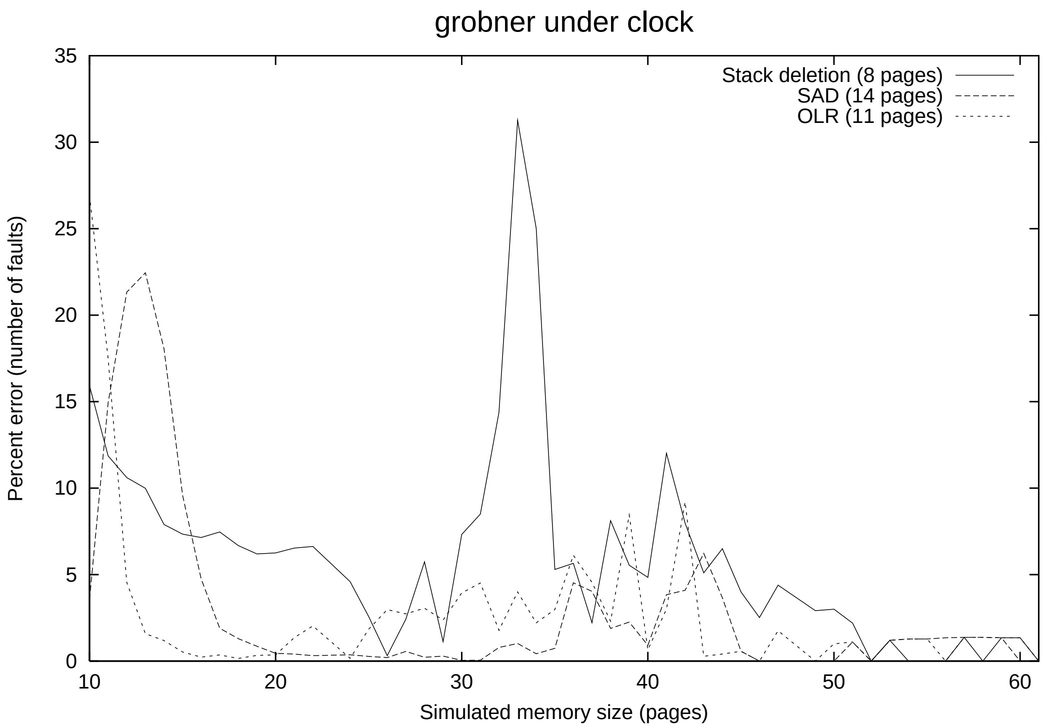

Figure 3.9: The absolute percent error (by number of faults) introduced into CLOCK simulations for **grobner** by reduced traces from SAD and SD. Notice that reduction memory sizes were chosen so that each reduced trace was approximately 100 times smaller than its original.

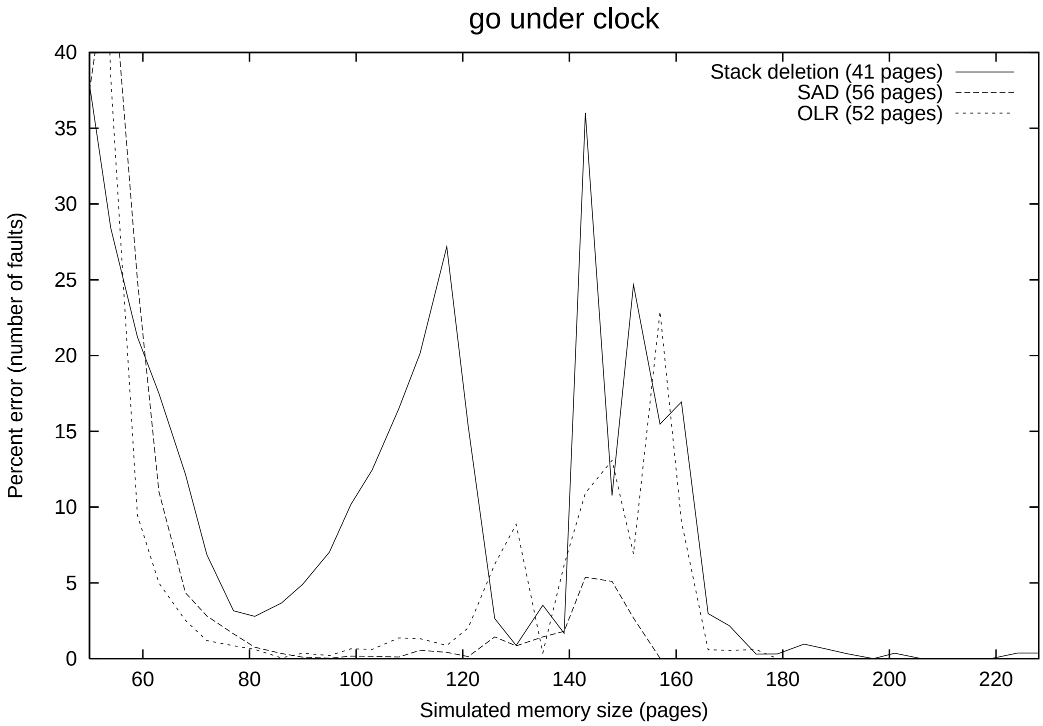

Figure 3.10: The absolute percent error (by number of faults) introduced into CLOCK simulations for **go** by reduced traces from SAD and SD.

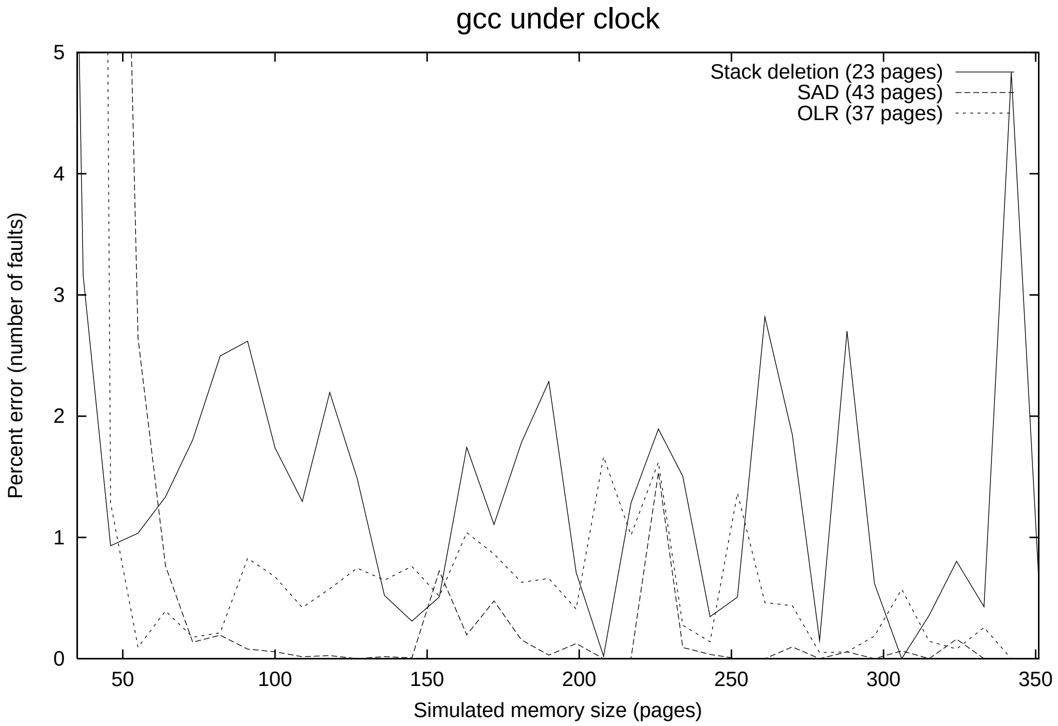

Figure 3.11: The absolute percent error (by number of faults) introduced into CLOCK simulations for **cc1** by reduced traces from SAD and SD.

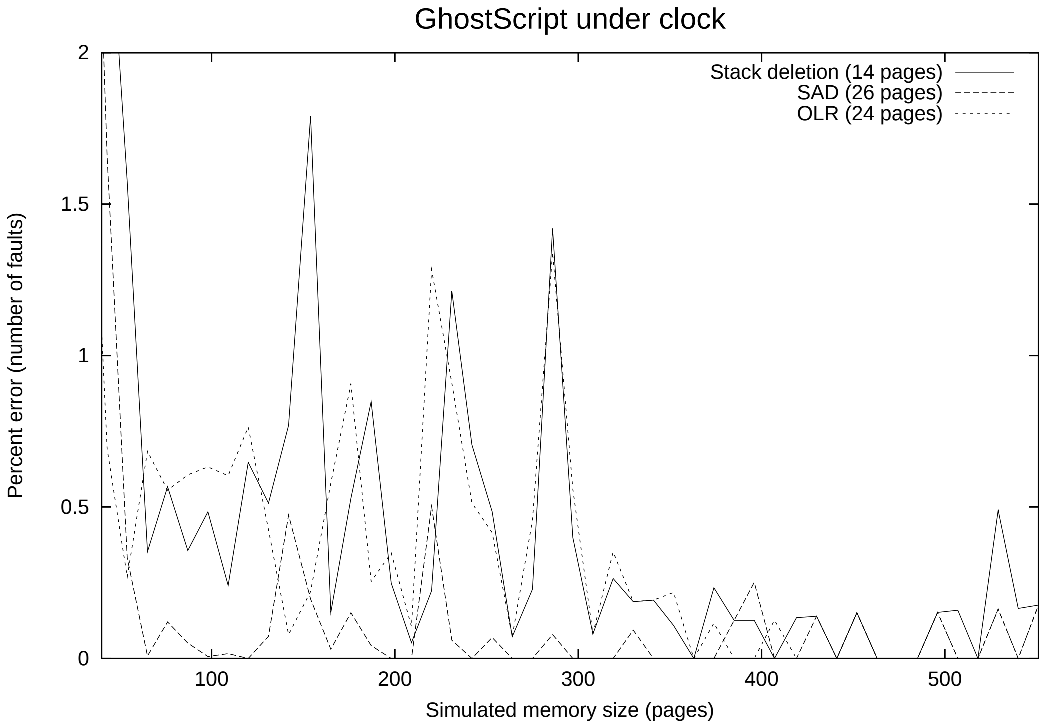

Figure 3.12: The absolute percent error (by number of faults) introduced into CLOCK simulations for **GhostScript** by reduced traces from SAD and SD.

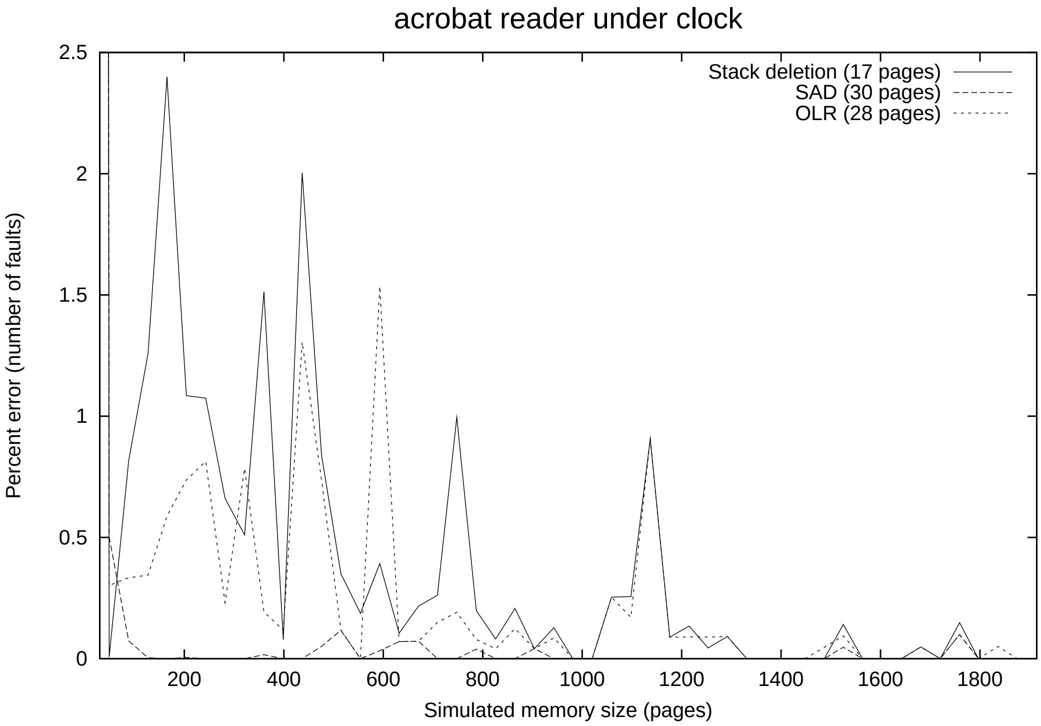

Figure 3.13: The absolute percent error (by number of faults) introduced into CLOCK simulations for **Acrobat Reader** by reduced traces from SAD and SD.

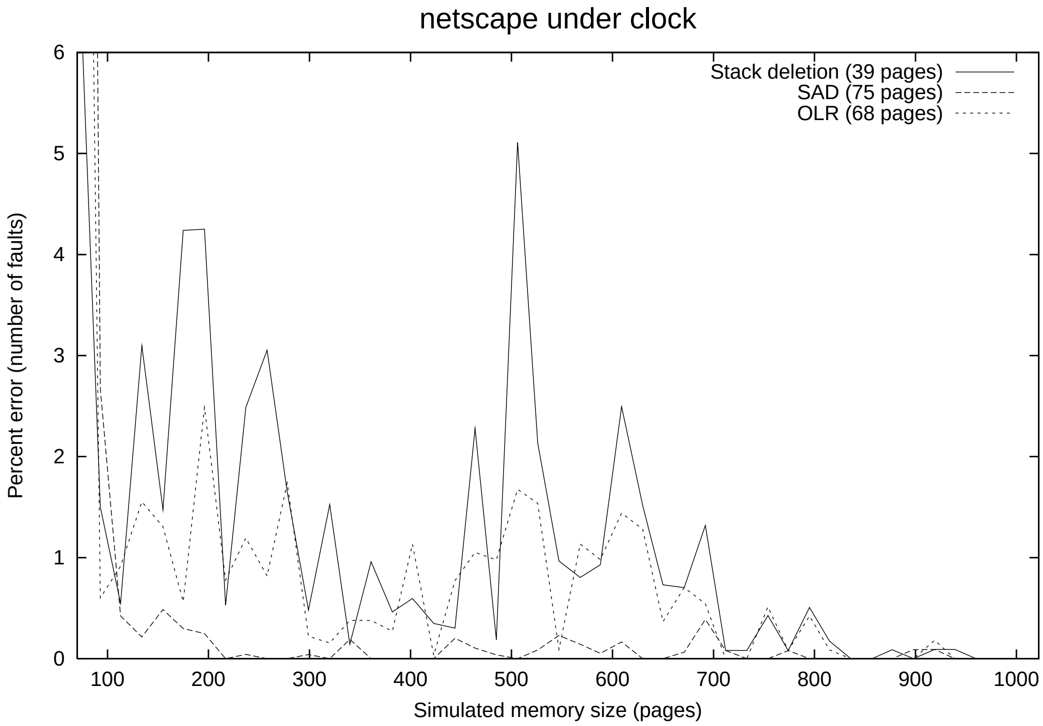

Figure 3.14: The absolute percent error (by number of faults) introduced into CLOCK simulations for **Netscape Navigator** by reduced traces from SAD and SD.

Note that the leftmost portion of each plot contains large error, as simulations of memories comparable to the reduction memory size are inaccurate. For all of the plots, the error drops significantly around a ratio of 2:1 of simulated memory size to reduction memory size. In most cases, SD reaches a reasonable level of error at a slightly smaller memory size than SAD and OLR, as it used a smaller reduction memory size.

Smith claimed that a ratio of 2:1 would be sufficient for experimentation with SD. We found that, with the 2:1 ratio, there is significant error in specific cases. For example, SD introduces more than 30% error into the simulation of `grobner` at a ratio of 4:1. It also introduces more than 35% error into `go` at a ratio of about 3.5:1. SAD and OLR also suffer unacceptably large error at these ratios in isolated cases. Consequently, we recommend a ratio of at least 5:1 for uniformly negligible error.

SEGQ results. For SEGQ simulations the results were similar, although less error was introduced on average for all reduction methods. We again selected six traces from the original fifteen, as the results of these are similar to the rest. Note that the error introduced is irregular; the behavior of SEGQ is dominated by FIFO, which is not a stack algorithm and can produce unpredictably different results with slightly different memory sizes.

In the plots shown in Figure 3.15 through 3.20, the size of the FIFO segment is fixed, and the percent error is shown for increasing LRU segment sizes. Note that the plots do not show the results for OLR, as it performed similarly to SAD, and because its inclusion makes these noisy plots difficult to read. The FIFO segment size chosen for these plots is approximately twice the reduction size for SAD. Thus, the total simulated segmented queue memory is at least twice as large as the reduction memory for both reduction methods.

For both programs, SAD and SD introduce sufficiently small error as to be acceptable for most studies. SAD introduces less error for many memory sizes,

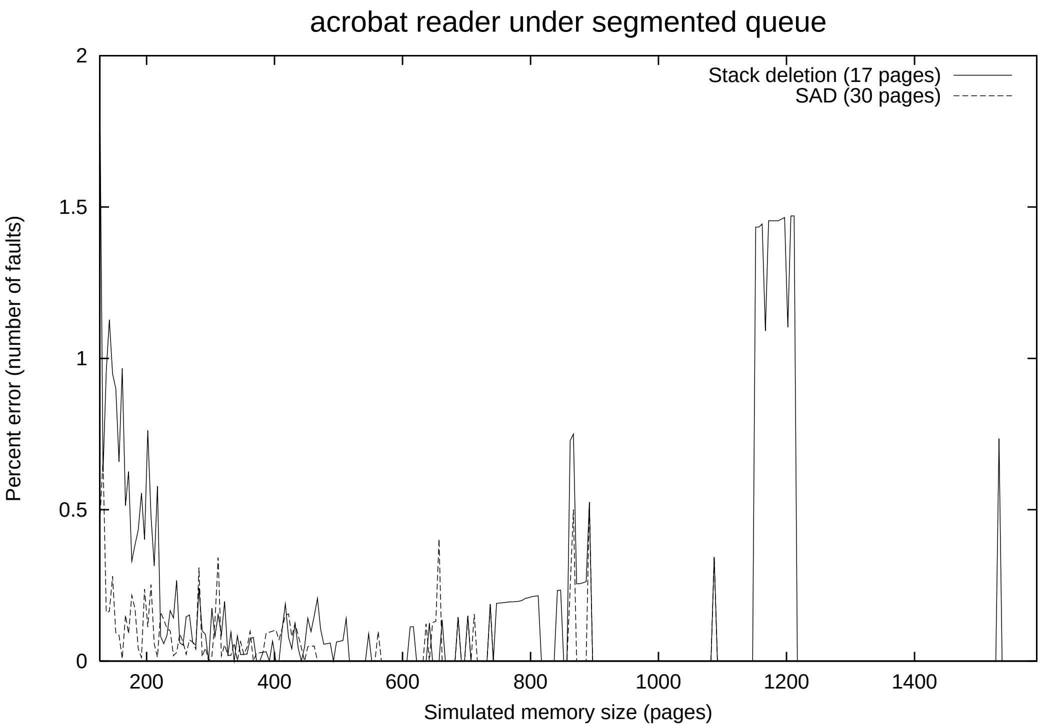

Figure 3.15: The absolute percent error (by number of faults) introduced into SEGQ simulations for **Acrobat Reader** by reduced traces from SAD and SD. For each plot, the FIFO segment size is fixed, and the x-axis shows the total memory size obtained by combining that FIFO with an LRU segment.

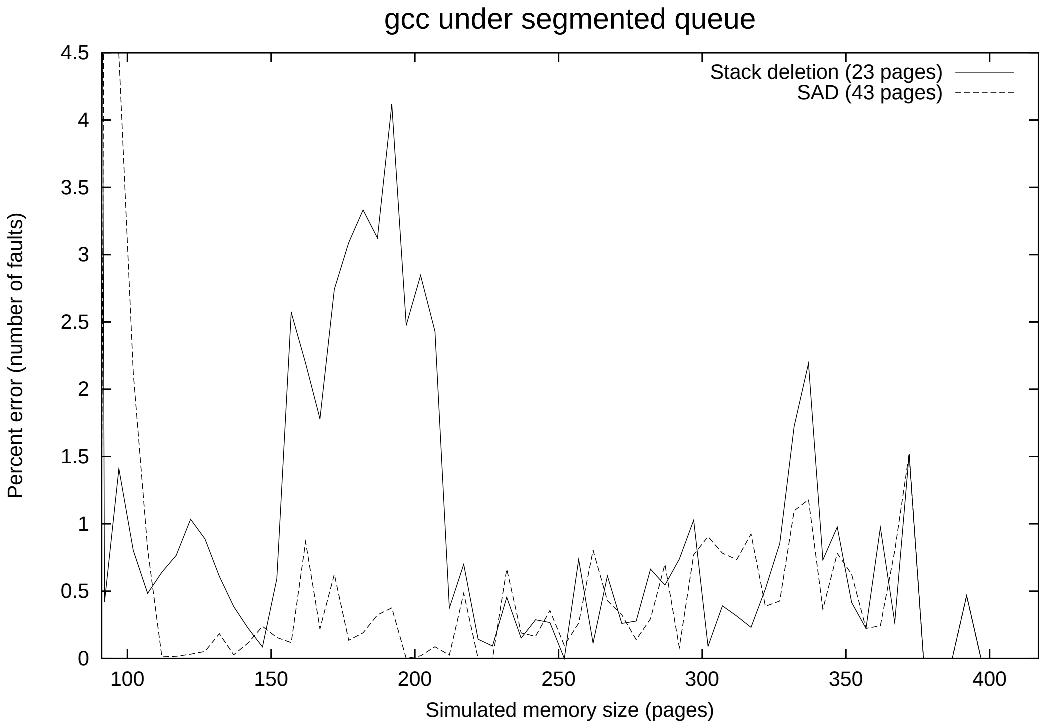

Figure 3.16: The absolute percent error (by number of faults) introduced into SEGQ simulations for **cc1** by reduced traces from SAD and SD.

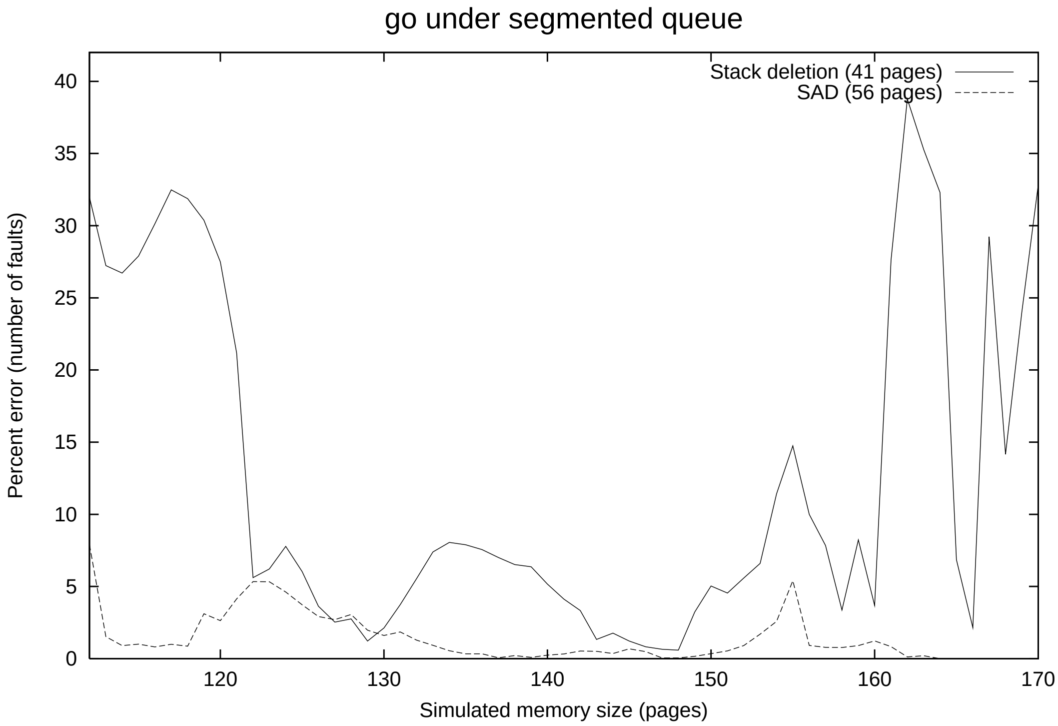

Figure 3.17: The absolute percent error (by number of faults) introduced into SEGQ simulations for **go** by reduced traces from SAD and SD.

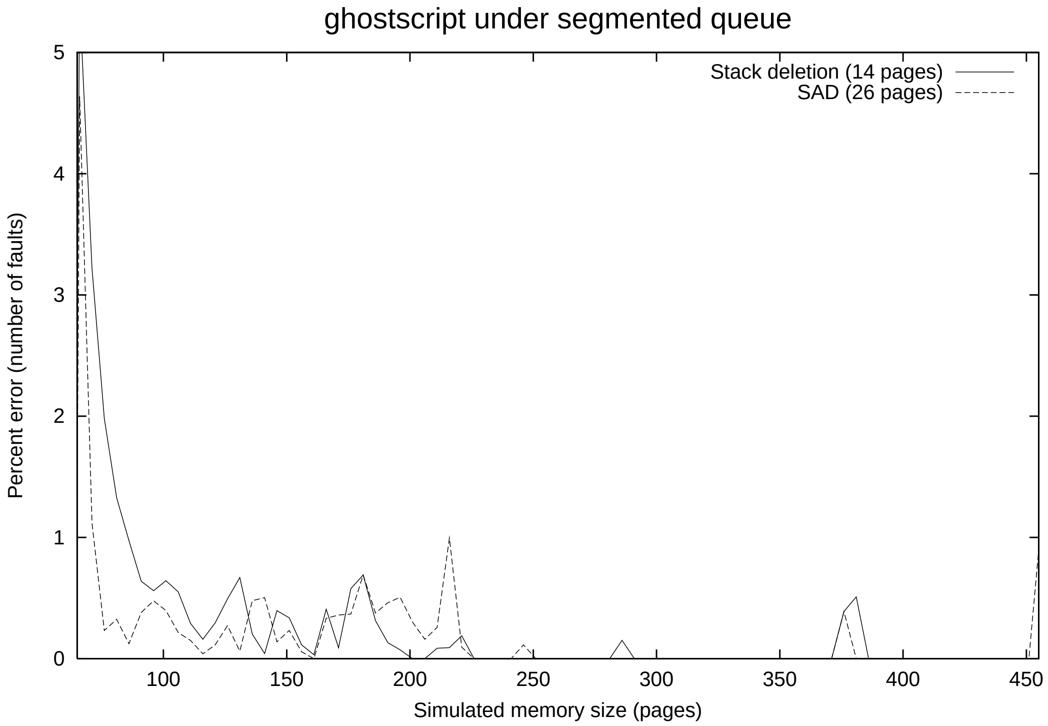

Figure 3.18: The absolute percent error (by number of faults) introduced into SEGQ simulations for **GhostScript** by reduced traces from SAD and SD.

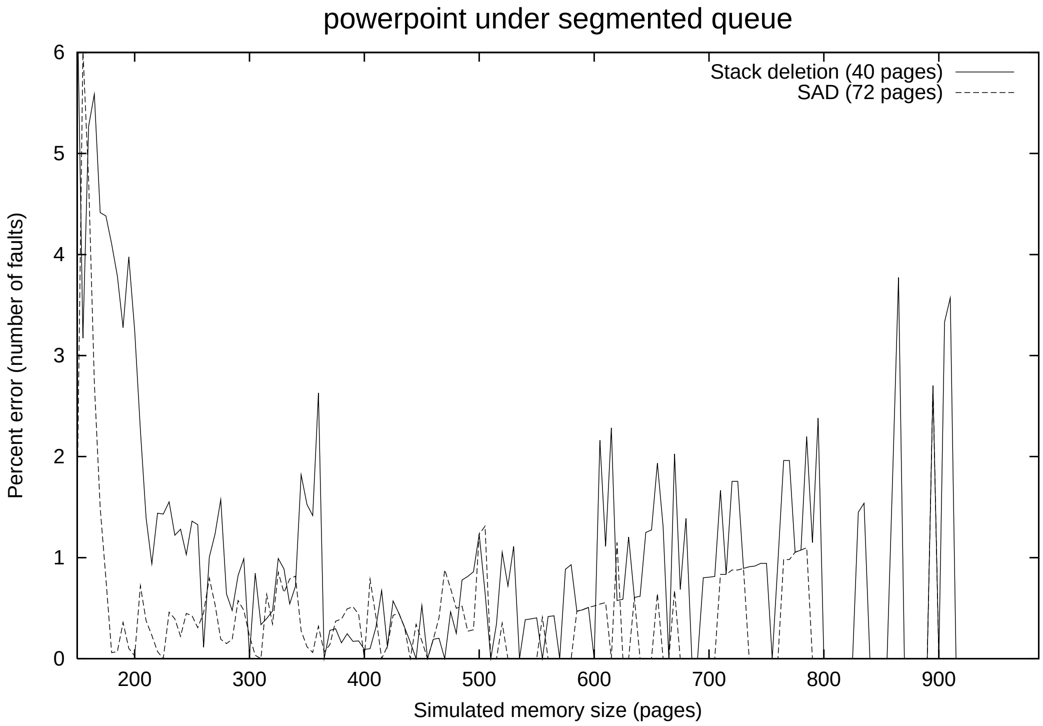

Figure 3.19: The absolute percent error (by number of faults) introduced into SEGQ simulations for **powerpoint** by reduced traces from SAD and SD.

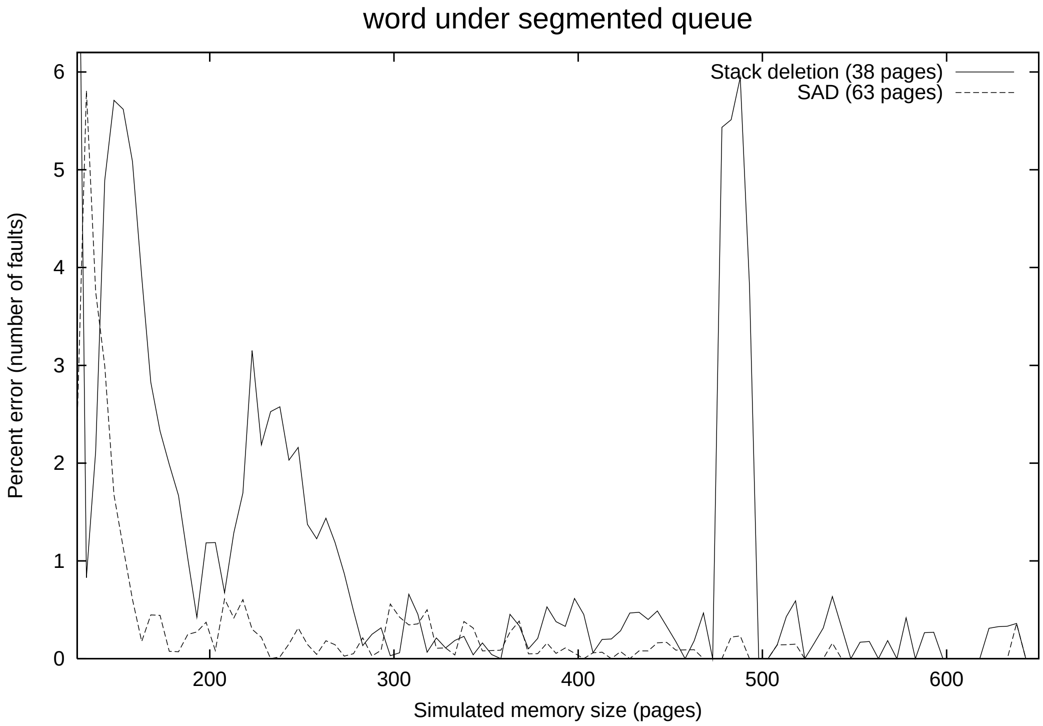

Figure 3.20: The absolute percent error (by number of faults) introduced into SEGQ simulations for **Microsoft Word** by reduced traces from SAD and SD.

although at a few memory sizes SD is the better method. Although it is not shown, OLR performed even better for SEGQ than it did for CLOCK, and its performance was comparable to SAD's. For many of these traces, the error introduced by either method is not significant.

Overall, SAD and OLR would be preferable to SD in practice. While all three introduce small error into simulations, SD introduces slightly more on average, and SAD consistently introduces the least. Further, SAD and OLR both allow for the exact simulation of LRU (and LRU variants like GLRU [FLW78], SEQ [GC97], FBR [RD90], EELRU [SKW99]), and SAD allows for the exact simulation of OPT.

3.5 Comparison to the Glass and Cao Technique

A limited comparison of our reduction technique to the method of Glass and Cao is presented here. As we had no access to the unreduced form of the traces used in [GC97] it was not possible to reduce them using SAD. We did, however, manage to obtain OLR-reduced traces for memory sizes above the threshold for accurate LRU simulation.

As mentioned in Section 3.2.2, the reduction technique of Glass and Cao does not allow to select the memory range for which simulations should be exact. Instead, the "sampling" granularity of reduction is used to arbitrarily determine a minimum memory size for which LRU and OPT simulations are possible (see any of Tables 3.3 to 3.5). To obtain OLR-reduced traces for the minimum LRU simulatable size (or any size greater than that) we made a simple (2 source-line) change to the LRU simulator of Glass and Cao that caused the behavior sequence of the LRU memory to be output. OLR was then used to construct the smallest trace exhibiting this behavior for the given memory size.

Tables 3.3, 3.4, and 3.5 compare the size of the OLR-reduced trace (for a reduction memory size equal to the minimum simulatable LRU size) with the

Program	Min. simulatable mem. size (KB)		Trace size (events)		Ratio (GlCa:OLR)
	LRU	OPT	GlCa	OLR	
applu	2432	972	267863	149803	1.79
blizzard	5332	4772	989866	79775	12.41
coral	7084	6780	2126367	575291	3.7
gcc	1900	1052	410761	37844	10.85
gnuplot	1552	476	111488	52860	2.11
ijpeg	1112	748	947219	409392	2.31
m88ksim	1964	328	246135	124568	1.98
murphi	2132	1472	231993	156036	1.49
perl	9636	8428	3239741	705598	4.59
swim	6932	6216	180782	101881	1.77
trygtsl	2444	1400	118262	69384	1.7
turb3d	7720	6360	1964495	417783	4.7
vortex	3024	2028	374378	59473	6.29
wave5	3652	1708	159454	59830	2.67

Table 3.3: Reduction of the traces from [GC97]. Ratios of the **number of events** in traces reduced by OLR and the reduction technique of Glass and Cao (GlCa) are shown. A ratio of x means that the GlCa trace is x times larger.

Program	**Min. simulatable mem. size (KB)**		**Trace size (KB)**		**Ratio (GlCa:OLR)**
	LRU	**OPT**	**GlCa**	**OLR**	
applu	2432	972	4443	1074	4.14
blizzard	5332	4772	16417	543	30.21
coral	7084	6780	35268	3996	8.83
gcc	1900	1052	6766	251	26.96
gnuplot	1552	476	1832	398	4.6
ijpeg	1112	748	15558	3148	4.94
m88ksim	1964	328	4048	843	4.8
murphi	2132	1472	3847	1053	3.65
perl	9636	8428	53711	5270	10.19
swim	6932	6216	3000	691	4.34
trygtsl	2444	1400	1962	526	3.73
turb3d	7720	6360	32543	3105	10.48
vortex	3024	2028	6205	400	15.53
wave5	3652	1708	2632	442	5.95

Table 3.4: Reduction of the traces from [GC97]. Ratios of the **trace file size** from traces reduced by OLR and the reduction technique of Glass and Cao (GlCa) are shown. A ratio of x means that the GlCa trace is x times larger.

Program	**Min. simulatable mem. size (KB)**		**Trace size gzipped (KB)**		**Ratio (GlCa:OLR)**
	LRU	**OPT**	**GlCa**	**OLR**	
applu	2432	972	1964	259	7.59
blizzard	5332	4772	7107	88	80.84
coral	7084	6780	15455	999	15.47
gcc	1900	1052	2963	38	77.92
gnuplot	1552	476	732	108	6.78
ijpeg	1112	748	6838	494	13.83
m88ksim	1964	328	1904	249	7.66
murphi	2132	1472	1673	234	7.15
perl	9636	8428	23066	1396	16.53
swim	6932	6216	1176	181	6.49
trygtsl	2444	1400	642	142	4.51
turb3d	7720	6360	11527	837	13.77
vortex	3024	2028	2733	78	35.05
wave5	3652	1708	1154	127	9.06

Table 3.5: Reduction of the traces from [GC97]. Ratios of the **compressed trace file size** from traces reduced by OLR and the reduction technique of Glass and Cao (GlCa) are shown. A ratio of x means that the GlCa trace is x times larger.

reduced trace of Glass and Cao. Since the traces are in a different format (our implementation of OLR uses a simple text format, while the traces of Glass and Cao are memory images) there are many possible ways to compare them.

The first metric, shown in Table 3.3, presented is events-per-trace. This is as close as possible to an "apples-to-apples" comparison. Any possible error would be in favor of the Glass and Cao technique, as OLR events are strictly smaller than events in the Glass and Cao representation, which contains at least the reference information of OLR as well as extra ordering information.

The second metric, used for Table 3.4, shows the size of each reduced trace file Kbytes. While it is useful to see that traces reduced by OLR are smaller in a raw, uncompressed representation, this metric is probably the least informative. The choice of representation can have a significant effect on file size, and these two reduced trace formats differ significantly in that respect.

In order to reduce the effect of differing representations, we compressed each of the reduced trace files with the `gzip` utility. The results of these compressions are shown in Table 3.5. Given a compressed encoding for which redundant information has been largely removed, OLR has removed more information from the trace while still providing for the same exacting simulations.

3.6 Conclusions

Storing and processing long memory reference traces is costly. We have proposed SAD and OLR: two new methods for drastically reducing traces to alleviate *both* storage *and* processing requirements. These reduction methods are designed to eliminate information about references to the most recently used pages. Both allow for the exact simulation of LRU memories of a minimum size chosen explicitly by the user. SAD also allows for the exact simulation of OPT memories.

SAD and OLR are invaluable for realistic virtual memory studies. Most studied virtual memory policies are either variants or approximations of LRU. Traces reduced with SAD or OLR provide for accurate simulations with LRU variants (for memories larger than a user-defined threshold). Additionally, we have shown that our reduced traces introduce very little error into the two most commonly used LRU approximations, CLOCK and SEGQ.

Chapter 4

Adaptive Caching

4.1 Introduction and Overview

For several decades, LRU has been the dominant replacement policy, either used or approximated in a variety of contexts. LRU has been shown empirically to often be very good—typically within a modest constant factor of the optimal number of misses for a fixed memory size. Nevertheless, LRU exhibits the well known failure mode shown in Section 2.1.4. In particular, the performance of LRU suffers for regular access patterns larger than the size of main memory.

LRU performs badly for some loop-like access patterns. These kinds of patterns are common in real programs. For instance, consider any roughly cyclic (loop-like) pattern of accesses over modestly more pages than will fit in memory. Such cyclic patterns could be induced either by a loop in the program execution or by a runtime system, such as a garbage collector that reuses memory. When pages are touched cyclically, and do not all fit in main memory, LRU will always evict the ones that have not been touched for the longest time, which are exactly the ones that will be touched again *soonest*.

One way to look at this phenomenon is that LRU keeps each page in memory for a long time, but cannot keep all of them for long enough. LRU keeps these pages in memory for so long because its prediction about future use is incorrect. LRU assumes that the longer a page has been unused, the longer it will be before it is used again. In this case, the longer a page has been unused, the sooner it will be used again. LRU has a notion of soonness that is incorrect.

In contrast, an optimal algorithm will evict some pages shortly after they are touched in a given iteration of the loop. Evicting these pages *early* allows other pages to remain in memory until they are looped-over again. Thus, an optimal algorithm for large cyclic reference patterns is "unfair"—there is nothing particularly noteworthy about the pages chosen for "early" eviction relative to LRU. The pages are simply sacrificed because fairness is a disaster in such a situation.

EELRU deviates from LRU when LRU performs badly. The above observations form the intuition behind *Early Eviction LRU (EELRU).* EELRU performs LRU replacement by default but diverges from LRU by evicting pages early when it notices that too many pages are being touched in a roughly cyclic pattern that is larger than main memory. Such patterns can be reliably detected using recency information. This is the same kind of information maintained by LRU, but EELRU maintains it for resident *and* non-resident pages. In particular, EELRU detects that LRU under-performs when *many of the fetched pages are among those evicted lately.*

Bounds on EELRU vs. LRU. The behavior of EELRU provides an interesting guarantee on its performance relative to LRU. The number of misses incurred with EELRU can never be more than a constant factor higher than with LRU, where the constant factor depends on the parameters picked for EELRU. The reason for this bound is that EELRU deviates from LRU only when the latter incurs many faults, and it reverts back to LRU as soon as EELRU starts incurring more faults.

In other words, if EELRU is performing poorly, it will quickly return to LRU-like behavior. LRU, in contrast, can perform worse than EELRU by a factor proportional to the size of memory in the worst case—a factor is usually in the thousands. This guaranteed property of EELRU is interesting both because of the ubiquity of LRU (and its approximations, e.g., segmented queue [TL81, BF83]) and because of the commonality of the LRU worst-case pattern (a simple, large loop) in practice.

EELRU is designed to be timescale relative. Additionally, EELRU is firmly based on a distinct principle of program locality studies, that of *timescale relativity* (see Section 2.1 and [WKM94]). Timescale relativity comes into play because real programs exhibit strong phase behavior. EELRU tries to adapt to phase changes by assigning more weight to events of the recent past with respect to virtual memory time. EELRU is designed to track virtual memory time only when "relevant" events occur. Relevant events are only references to pages that are *not* among the most recently touched. Intuitively, EELRU ignores all high-frequency references as these do not influence replacement decisions and may dilute virtual memory time so much that important regularities are impossible to distinguish.

Also according to timescale relativity, time in EELRU advances at a rate inversely proportional to the memory size. For larger memories, a proportionally larger number of relevant events have to occur for the algorithm to decide to adapt its behavior. The reason behind this choice is that the timescale most relevant to caching decisions is the one at which eviction decisions turn out to be good or bad, i.e., roughly the time a page is likely to spend in memory, and either be touched or not before being evicted. At any given eviction, the replacement policy should attempt to choose a page which will not be touched soon, specifically after "many" other pages are touched—more pages than will fit in memory. This timescale is also crucial for adaptation because an online adaptive replacement policy must detect program behaviors that last long enough to matter to its choice of caching strategies.

We argue that timescale relativity represents a sound principle upon which to design a replacement policy. Also, we propose that a special kind of plot, termed a *recency-reference* graph, is appropriate for studying program locality behavior (Section 4.4).

Experimental evaluation. To validate EELRU experimentally, we applied it to fourteen program traces and studied its performance. Most of the traces (eight) are of memory-intensive applications and come from the recent experiments of Glass and Cao [GC97]. Glass and Cao used these traces to evaluate SEQ, an adaptive replacement algorithm that attempts to detect linear (not in recency but in *address* terms) *faulting* patterns. This set of traces contains representatives from three trace categories identified in [GC97]: traces with large memory requirements but no clear memory access patterns, with small access patterns, and with large access patterns. An extra six traces were collected as representatives of applications that are not memory-intensive but may have small-scale reference patterns.

The results of our evaluation are encouraging. EELRU performed at least as well as LRU in almost all situations and significantly better in most. Results of more than 30% fewer faults compared to LRU were *common* for a wide range of memory sizes and for applications with large-scale reference patterns.

A comparison with the SEQ algorithm [GC97] was also instructive: SEQ is based on detecting patterns in the address space, while EELRU detects patterns in the recency distribution. Although our simulation was quite conservative (see Section 4.4), EELRU managed to obtain significant benefit even for traces for which SEQ did not. On the other hand, SEQ is an aggressive algorithm and performed better for programs with very clear linear access patterns in the address space. Even in these cases, however, EELRU captured a large part of the available benefit.

Overall, EELRU is a simple, soundly motivated, effective replacement algorithm. As a representative of an approach to studying program behavior based on recency and timescale relativity, it proves quite promising for the future.

4.2 Motivation and Related Work

The main purpose of this section is to compare and contrast the approach taken by EELRU to other replacement policies. This will help illustrate the rationale behind some of the design choices in EELRU.

EELRU uses recency information to distinguish between pages. A recency-based standpoint (see also [Spi76, FLW78, WFL83]) dictates that the only way to differentiate between pages is by examining their past history of references, without regard to other information about the pages (e.g., proximity in the address space). This view ensures that looping patterns of several different kinds are treated as the same. Note that access patterns that cause LRU to page excessively do not necessarily correspond to linear patterns in the memory *address* space. For instance, a loop may be accessing records connected in a linked list or a binary tree. In this case, accesses are regular and repeated, but the addresses of pages touched may not follow a linear pattern. That is, interesting regularities do not necessarily appear in memory arrangements but in how recently pages were touched in the past.

The SEQ replacement algorithm [GC97] is one that bases its decisions on address information (detecting sequential address reference patterns). Consequently, it is lacking in generality (e.g., cannot detect loops over linked lists connecting interspersed pages). Section 4.4 compares EELRU and SEQ extensively.

EELRU is based on the principle of timescale relativity, which helps it detect and adapt to phase changes. The first application of timescale relativity in EELRU is in determining that time advances at a slower rate for larger memories, or, equivalently, that the length of what constitutes a "phase" in program behavior is proportional to the memory size examined. This idea is by no means new. In fact, it is commonplace in many pieces of theoretical work on paging (e.g., [ST85, Tor98]), where an execution is decomposed into phases with working sets of size equal to that of memory.

The second application of timescale relativity in EELRU dictates that only events that matter for replacement decisions should count to advance time. In the past, several replacement policies based on good ideas have yielded rather underwhelming results because they were affected by events at the wrong timescale.

An example is the *Inter-Reference Gap (IRG)* model developed by Phalke [Pha95]. IRG attempts to predict how soon pages will be referenced in the future by looking at the time between successive past references. A simpler version of the same idea is the well-known Atlas loop detector [BFH68] that examines only the last successive references. This loop detector fails because time is measured as the number of memory references performed—the prediction is made based on reference time, not virtual memory time.

If a loop repeats with significant variation per iteration, then the time between successive references may vary. Such variance is not unusual: loops may perform different numbers of operations per step during different iterations—as is, for instance, the case with many nested loop patterns. The Atlas loop detector would fail to recognize the regularity. More complex, higher order IRG models (such as those studied by Phalke) can detect significantly more regularities in the presence of variation. This complexity, however, makes them prohibitive for actual implementations.

In contrast, consider one possible timescale-relative approach to loop detection, for which the time between page use for each iteration is measured by the number of other pages used. Using this metric, the detection of loops would more likely be a simple matter—all pages are likely to be used when they've aged about the same amount in recency terms. Combined with a response to aggregate behavior, rather than specific reference distances for each page, this kind of approach can easily tolerate variation within a loop.

A view based on recency and timescale relativity can be applied to other work in the literature. Most work on replacement policies deals with specific formal models of program behavior. Indeed EELRU itself is inspired by the LRU stack model (LRUSM) [Spi76], as we will discuss in Section 4.3.2. LRUSM is an independent events model, where events are references to pages identified by their *recency* (i.e., the number of other pages touched after the last touch to a page). An optimal replacement algorithm for LRUSM is that of Wood, Fernandez, and Lang [WFL83]. Unfortunately, programs cannot be modeled accurately as independent recency events. On the other hand, short program phases can be modeled very closely using LRUSM. Hence, a good on-line recency algorithm needs to be adaptive to detect phase changes. Timescale relativity (as in EELRU) is crucial for providing such adaptivity reliably.

Other well-known models are those in the class of Markov models (e.g., [CV76, FG74]). The straightforward case of a 0^{th}-order Markov model corresponds to the well known *Independent Reference Model (IRM)* [ADU71]. An optimal replacement algorithm for Markov models can be found in [KPR92].

We believe that replacement algorithms based on Markov models fail in practice because they try to solve a far harder problem than that at hand. A replacement algorithm is only trying to select a page that will not be used soon, as discussed in Section 2.1. Markov models express the probability of occurrence for specific sequences of references. As such, they are not just trying to predict which page will not be used soon—Markov models are trying to predict the order in which *all* pages are likely to be used. In our view, Markov models attempt to predict program behavior at the reference timescale, not the virtual memory timescale. This makes Markov model-based replacement too brittle for actual use—realistic models cannot offer any accuracy at a large enough timescale (such as that of memory replacement decisions).

Finally, EELRU can be viewed as a way to negate partially an often-stated assertion about the limits of actual eviction algorithms. Quoting from [Tor98]:

> Stated another way, the guaranteed performance of any deterministic on-line algorithm degrades sharply as the intrinsic working set size of an access sequence increases beyond [the memory size] whereas the performance of the optimal off-line algorithm degrades gracefully as the intrinsic working set size of an access sequence increases beyond [the memory size].

Even though this assertion holds under theoretical worst-case analysis, in practice program reference sequences exhibit strong and simple regularities that an on-line algorithm can exploit. EELRU is an example of this approach and adds to LRU the ability to gracefully degrade its performance for large working sets when reference patterns are roughly cyclic.

4.3 The EELRU Algorithm

4.3.1 General Idea

EELRU algorithm structure. The *early-eviction LRU* (EELRU) algorithm has a reasonably simple structure:

1. Perform LRU replacement unless, in the recent past, **many** pages fetched had just been evicted.

2. If, in the recent past, **many** pages fetched had just been evicted, apply a *fallback algorithm*: either evict the least recently used page or evict the e-th most recently used page, where e is a pre-determined recency position.

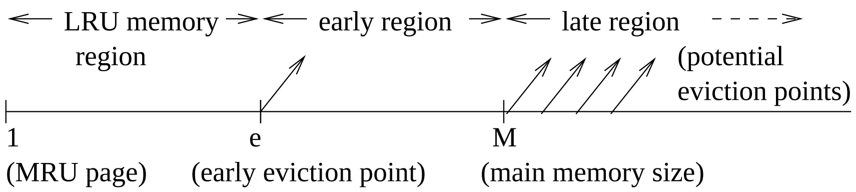

Figure 4.1: General EELRU scheme: LRU axis and correspondence to memory locations.

To turn this idea into a concrete algorithm, we need to define the notions of "many", "recently", etc., (highlighted above), as well as an exact fallback algorithm. By changing these aspects we obtain a family of EELRU algorithms, each with different characteristics. We will only discuss a single fallback algorithm (one that is particularly simple and is based on sound theoretical motivation). The algorithm is described in Section 4.3.2. In this section we describe the basic flavor of the EELRU approach, which remains the same regardless of the actual fallback algorithm used.

Using recency information for adaptivity. Figure 4.1 presents the main elements of EELRU schematically, by showing the *reference recency axis* (also called the *LRU axis*) and the potential eviction points. The reference recency axis is a discrete axis where point i represents the i-th most recently accessed page (written $r(i)$). We see in Figure 4.1 that EELRU distinguishes three regions on the recency axis. The "LRU memory region" consists of the first e blocks, which are always in main memory. (Note that the name may be slightly misleading: the "LRU region" holds the *most recently used* blocks. The name comes from the fact that this part of the buffer is handled as a regular LRU queue.)

Position e on the LRU axis is called the *early eviction point*. The region beginning after the early eviction point and until the memory size, M, is called the "early region". The "late region" begins after point M and its extent is determined by the fallback algorithm used (e.g., see Section 4.3.2).

Recall that, at page fault time, EELRU will either evict the least recently used page or the page at point e on the recency axis (i.e., the e-th most recently used page). The latter is called an *early eviction* and its purpose is to keep not-recently-touched pages in memory for a little longer, with the hope that they will soon be referenced again. The challenge is for EELRU to adapt to changes in program behavior and decide reliably which of the two approaches is best in every occasion.

EELRU maintains a queue of referenced pages ordered by recency, in much the same way as plain LRU. The only difference is that the EELRU queue also contains records for pages that are *not* in main memory but were recently evicted. EELRU also keeps the total number of page references to both the early and the late regions of its recency queue. This information enables a cost-benefit analysis, based on the expected number of misses that a fallback algorithm would incur or avoid. In essence, the algorithm makes the assumption that the program recency behavior will remain the same for the near future and compares the page faults that it would incur if it performed LRU replacement with those that it would incur if it evicted pages early by using the fallback algorithm.

We will sketch here how this cost-benefit analysis is performed by means of an example. Consider Figure 4.2a, which shows the recency distribution for a phase of program behavior. That is, it shows for each position on the recency axis how many hits to pages on the position have occurred *lately.* The distribution changes in time, but remains fairly constant during separate phases of program behavior. The EELRU adaptivity mechanism is meant to detect exactly these phase changes.

If the distribution is monotonically decreasing, LRU is the best choice for replacement. Nevertheless, large loops could cause a distribution like that in Figure 4.2a, with many more hits in the late region than in the early region. This kind of distribution encourages EELRU to sacrifice some pages in order to allow others to

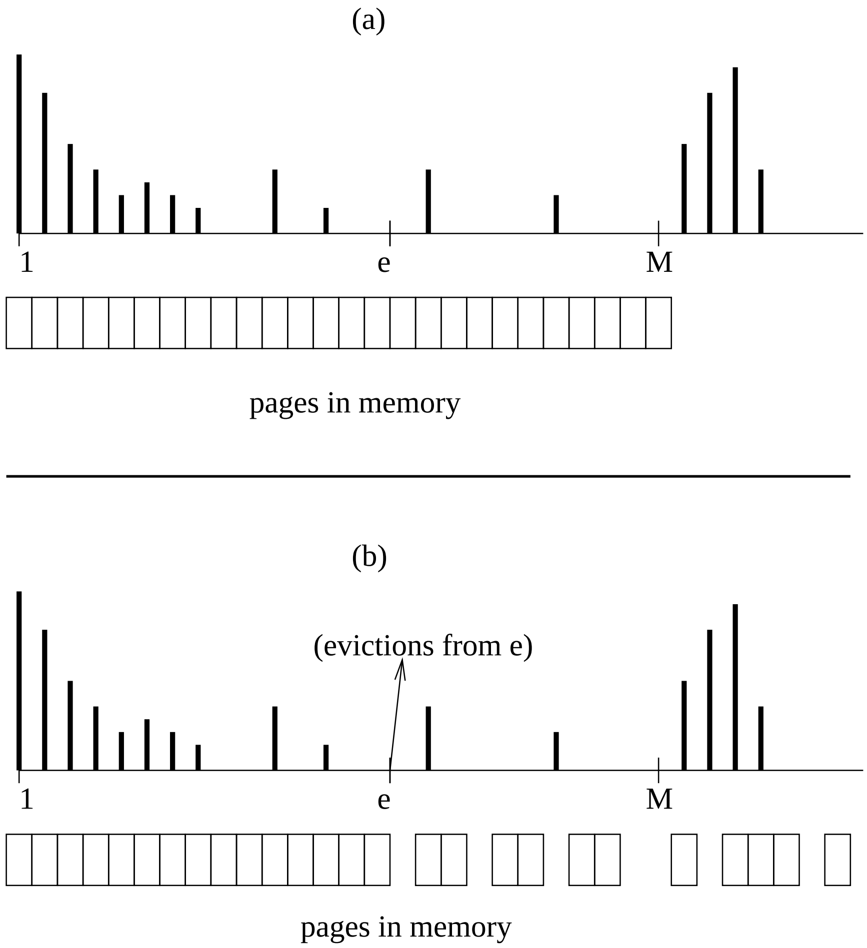

Figure 4.2: Example recency distribution of page touches. **(a)** With LRU, many references are to pages not in memory. **(b)** After evicting early, some less recent pages stay in memory. Since references to less recent pages are common (in this example distribution), evicting early yields benefits.

stay in memory longer. Thus, EELRU starts evicting pages early so that eventually more hits in the late region will be on pages that have been cached (Figure 4.2b).

Timescale relative adaptivity to phase behavior. EELRU is not the first algorithm to attempt to exploit such recency information for eviction decisions (e.g., see [FLW78]). Its strength, however, is that it does so adaptively and succeeds in detecting changes in program phase behavior.

The key to detecting phase behavior with this cost-benefit analysis used by EELRU is that more weight is assigned to events of the recent past. As discussed in Section 2.1.6, decaying the information provided by past reference events according to virtual memory time is a critical part of adapting in a timescale relative manner. EELRU advances virtual memory time by detecting references that influence replacement decisions—those in the early and late regions. Notice that by using these regions, the timescale is appropriately determined by the memory size. High-frequency events (i.e., hits to the e most recently referenced pages) are totally ignored in the EELRU analysis. The reason is that allowing high-frequency references to affect our notion of time dilutes our information to the extent that no reliable analysis can be performed. The same number of memory references may contain very different numbers of relevant events during different phases of program execution.

To illustrate the need to adapt at the correct timescale, consider what would happen if we allowed hits to the e most recently referenced pages to affect the statistics. Actual programs reference their most recently referenced pages much more often than other pages in the LRU distribution. Thus our virtual memory time would become diluted to the extent that a cost-benefit analysis is completely unreliable. Within phases, the frequent references to the most recently used pages would cause time to advance quickly. As a result, statistics on the early and late region references, which occur much less often, will be decayed away. If the adaptive mechanism attempts to react to early and late region information quickly, before it decays, then the system may try to react to transient behavior that doesn't persist in the early and late regions. Time would be advanced at an incorrect rate, and no single constant bias will bring this reference timescale into agreement with the virtual memory timescale for all reference patterns.

Using multiple early-eviction points. The basic EELRU idea can be straightforwardly generalized by allowing more than one instance of the scheme of Figure 4.1 in the same replacement policy. In other words, one system can use several EELRU eviction policies at once, and choose the best on for each phase of program behavior. For instance, multiple early eviction points may exist and only the events relevant to a point would affect its cost-benefit analysis. The point that yields the highest expected benefit will determine the page to be replaced.

Efficient implementation. The simplicity of the general EELRU scheme allows for efficient implementations. Even though we have not provided an in-kernel version of EELRU, we speculate that it is quite feasible. In particular, EELRU can be approximated using techniques identical to standard in-kernel LRU approximations, such as **clock** and **segq** (segmented queue) [TL81, BF83]. These policies introduce no overhead for references to the most recently used pages, and only the remaining, less recently used pages incur traps into the kernel. Maintaining recency information for the less recently used pages would require only a few hundred additional processor cycles per trap.

EELRU needs only the recency information for less recently used pages, as pages that are used recently are likely to be used again soon, and should not be evicted early. If too many traps are occurring so that recency information can be maintained, then the number of less recently used pages for which information is being gathered can be reduced. That is, if it is too expensive to keep recency information for some more recently used pages, then it it would be a poor choice to evict them early anyway, and so that recency information need not be gathered.

Most architectures also contain hardware to support the CLOCK policy, such that some basic recency information is recorded for the most recently used pages. By invoking the CLOCK algorithm periodically with interrupts, this extra recency information could be used for our adaptive mechanisms. It should not matter that this recency information is not exact, as our EELRU simulator did not keep exact recency information either; recency information for accesses to the various early and late regions were stored, thus providing only coarse-grained recency.

4.3.2 A Concrete Algorithm

The first step in producing a concrete instance of EELRU is choosing a reasonable fallback algorithm. This will in turn guide our cost-benefit analysis, as well as the exact distribution information that needs to be maintained. An obvious candidate algorithm would be one that always evicts the e-th most recently used page. This is equivalent to applying Most Recently Used (MRU) replacement to the early region and clearly captures the intention of caching less recent pages. However, real programs exhibit strong phase behavior [Den80] which causes MRU to become unstable, as pages which may never be touched again will be cached indefinitely.

The algorithm of Wood, Fernandez, and Lang [FLW78, WFL83] (henceforth called *WFL*[1]) is a simple modification of MRU that eliminates this problem. The WFL replacement algorithm specifies two parameters representing an *early* and a *late* eviction point on the LRU axis. Evictions are performed from the early point, unless doing so means that a page beyond the late eviction point will be cached. Thus the algorithm can be written simply as:

[1] The WFL algorithm is called GLRU (for "generalized LRU") by its creators. To avoid confusion, we will not use this acronym, since it has been subsequently overloaded also to mean "global" LRU.

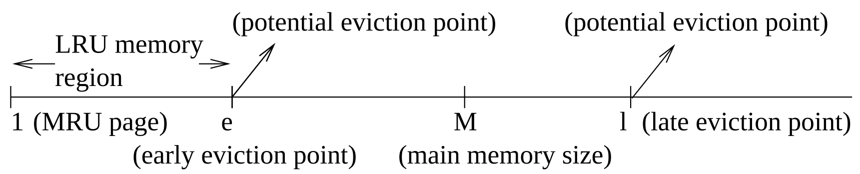

Figure 4.3: EELRU with WFL fallback: LRU axis and correspondence to memory locations.

if $r(l)$ is in memory

and the fault is on a less recently accessed page

then evict page $r(l)$

else evict page $r(e)$

(where e is the early and l the late eviction points). Figure 4.3 shows some elements of the WFL algorithm schematically.

Making WFL adaptive. It has been shown that there exist values for e and l such that the WFL algorithm is optimal for the LRU stack model of program behavior [WFL83] [Spi76]. Again, however, program phase behavior, even for well-defined, long lasting phases, can cause the algorithm to under-perform. This is not surprising: WFL is not an adaptive algorithm. Instead it presumes that the optimal early and late points are chosen based on advanced knowledge of the recency distribution. Thus, the adaptivity provided by EELRU is crucial, as it transforms WFL into a good on-line replacement algorithm. This is particularly true when multiple pairs of early and late eviction points exist and EELRU chooses the one yielding the most benefit.

Even though entire programs cannot be modeled accurately using the LRU stack model, shorter phases of program behavior can be closely approximated. Under the assumptions of the model, the WFL algorithm has the additional advantage

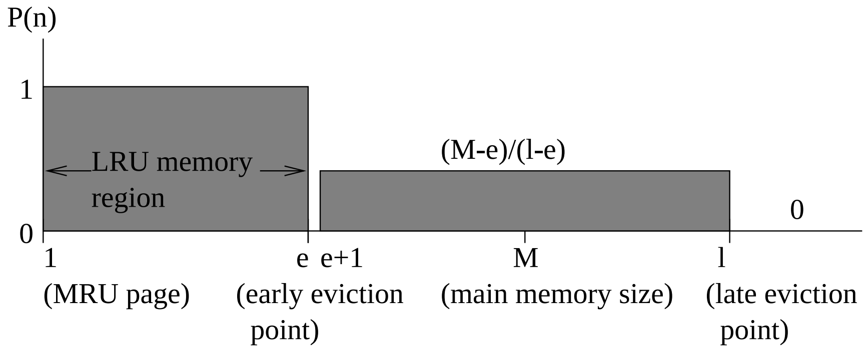

Figure 4.4: Probability of being in memory for a page with a given recency.

of simplifying the cost-benefit analysis. One of the properties of WFL is that when the algorithm reaches a steady state, the probability $P(n)$ that the n-th most recently accessed page (i.e., page $r(n)$) is in memory is:

$$P(n) = \begin{cases} 1 & \text{if } n \leq e \\ (M-e)/(l-e) & \text{if } e < n <= l \\ 0 & \text{otherwise} \end{cases}$$

The probability distribution is shown in Figure 4.4.

Applying the cost-benefit calculation. Given this simplified approach to calculating the cost and benefit, we can estimate the number of faults that WFL would incur (at steady state) and compare that to the number of faults that LRU would incur. We will call $total$ the number of recent hits on pages between e and l (in reference recency order). Similarly, we will call $early$ the number of recent hits on pages between e and M. The eviction algorithm then becomes:

if $total \cdot (M - e)/(l - e) \leq early$

or ($r(l)$ is in memory

and the fault is on a less recently accessed page)

then evict the least recently accessed page

else evict page $r(e)$

We can now consider the obvious generalization of the algorithm where several instances of WFL, each with different values of e and l, are active in parallel. By e_i, l_i, $total_i$, and $early_i$ we will denote the e, l, *total* and *early* values for the i-th instance of the algorithm. Then, the instance of WFL that will actually decide which page is to be evicted is the one that maximizes the expected benefit value $total_i \cdot (M - e)/(l - e) - early_i$. If all such values are negative, plain LRU eviction is performed.

Note that in the case of multiple early and late eviction points, EELRU adaptivity performs a dual role. On one hand, it produces on-line estimates of the values of e and l for which the algorithm performs optimally. (Plain LRU is no more than another case for these values). On the other hand, the adaptivity detects phase transitions and changing the values accordingly.

In the case of multiple early and late eviction points, one more modification to the basic WFL algorithm makes sense. Since not all late eviction points are equal, it is possible that when the i-th instance of WFL is called to evict a page, there is a page $r(n)$ in memory, with $n > l_i$. In that case, the algorithm should first evict all such pages (to guarantee that, in its steady state, all pages less recently referenced than l_i will not be in memory). Note that this modification of the basic WFL algorithm does not affect its steady state behavior (and, consequently, its proof of optimality for the LRU stack model, as presented in [WFL83]). Taking the change into account, our final eviction algorithm becomes:

let *benefit* be the maximum of the values

$total_i \cdot (M - e_i)/(l_i - e_i) - early_i$

and j be the index for which this value occurs

if benefit ≤ 0

or a page $r(n)$, $n > l_j$ is in memory

or ($r(l_j)$ is in memory

and the fault is on a less recently accessed page)

then evict the least recently accessed page

else evict page $r(e_j)$

This form of EELRU is the one used in all experiments described here.

4.3.3 EELRU vs. LRU

An interesting property of EELRU is that it is *robust* with respect to LRU under worst-case analysis. In particular, EELRU will never perform more than a constant factor worse than LRU, while LRU can perform worse than EELRU by a factor proportional to the number of memory pages. The exact values of these factors depend on the parameters chosen for the EELRU algorithm—e.g., the number and positions of early and late eviction points, and the speed of adaptation.

Although we will not offer a rigorous proof of this claim, the argument is straightforward. EELRU diverges from LRU only when the latter has incurred many faults in the recent past. EELRU reverts back to LRU when it detects that LRU would have performed better. Thus, over a complete reference sequence, LRU would only better than EELRU if LRU incurred faults such that EELRU would choose to use the fallback algorithm just as those faults would cease to occur under LRU. In other words, LRU would have to fault on recently evicted pages in bursts, at just the rate required to mislead EELRU. In such cases the ratio of miss rates of the two algorithms is never worse than a constant.

Conversely, a steady loop slightly larger than memory but within one of the late regions of EELRU will cause LRU to suffer misses for every page, while EELRU will suffer a constant number of misses per iteration.

For an illustrative example of this worst-case analysis, consider the extreme case where EELRU has a single early eviction point at position 1 of the recency axis (i.e., it may evict early the most recently used page) and a corresponding late eviction point at position $M+1$ (with M being the memory size). In this case, a steady loop over $M+1$ pages will cause LRU to suffer $M+1$ faults, while EELRU will only suffer a single fault per iteration. At the same time, EELRU will never be outperformed by LRU by a factor of more than 2 under *any* reference pattern. If it decides to diverge from LRU, it may incur at most 2 faults for each LRU fault that causes the divergence.

4.4 Experimental Assessment

4.4.1 Settings and Methodology

The test suite. To assess the performance of EELRU, we used fourteen program traces, covering a wide range of memory access characteristics. Eight of the traces are of memory-intensive applications and were used in the recent experiments by Glass and Cao [GC97]. Another six traces were collected individually from programs that use smaller footprints, and are described more fully in Section 6.3.1. This second set of traces was chosen to show that our approach to adaptivity is applicable to different memory sizes, and thus timescales, without any tuning of parameters.

Limitations of the Glass and Cao reduced traces. The eight traces from [GC97] are only half of the traces used in that study. The rest of the experiments could not be reproduced because the reduced trace format used by Glass and Cao sometimes omitted information that was necessary for accurate EELRU simulation. Specifically, consider the behavior of EELRU: at any given point, early evictions can be performed, making the algorithm replace the page at point e on the LRU axis. Thus, the trace should have enough information to determine the e-th most recently accessed page. As such, the trace must be sufficiently accurate for an LRU simulation with a memory of size e. The reduced traces of Glass and Cao impose limitations on the minimum memory size for which LRU simulation can be performed. Thus, the minimum simulatable memory size for EELRU, given one of these reduced traces, is dictated by the earliest early point that EELRU will consider.

For some traces of these eight trace, this limitation prevented us from simulating EELRU within the memory ranges tested in [GC97]. For example, the **cc1** trace used in the Glass an Cao study was could only be used for the LRU simulation of memories larger than 475 pages (see [GC97]). Given the early eviction points used by our simulator, the minimum EELRU simulatable memory size would be 1188 pages, well outside the memory range for this experiment. (The trace causes no faults for memories larger than 900 pages).

To reproduce as many experiments as possible, we picked early eviction points such that at least 40% of the memory was managed using strict LRU. This limitation makes our simulations quite conservative: it means that EELRU cannot perform very early non-LRU evictions. As mentioned earlier, simulations for eight of the traces are meaningful for this choice of points [2] (i.e., our simulated memory ranges overlap significantly with those of the experiments performed by Glass and Cao). Table 4.1 contains information on these traces.

It is worth noting that the above set of traces contains representatives from all three program categories identified by Glass and Cao. These are *programs with no clear patterns* (murphi, m88ksim), *programs with small-scale patterns* (perl),

[2] Two more traces from [GC97], "es" and "fgm", satisfy the restrictions outlined above but were not made available to us.

Program	Description	Min. simulatable LRU memory (4KB pages)
`applu`	Solve 5 coupled parabolic/elliptic PDEs	608
`gnuplot`	Postscript graph generation	388
`ijpeg`	Image conversion into JPEG format	278
`m88ksim`	Microprocessor cycle-level simulator	491
`murphi`	Protocol verifier	533
`perl`	Interpreted scripting language	2409
`trygtsl`	Tridiagonal matrix calculation	611
`wave5`	Plasma simulation	913

Table 4.1: Information on traces used in [GC97]

and *programs with large-scale reference patterns* (applu, gnuplot, ijpeg, trygtsl, and wave5).

The other six traces that we gathered were used to supply more data points. These are traces of executions that do not consume as much memory. The applications traced are espresso (a circuit simulator), gcc (a C compiler), ghostscript (a PostScript engine), grobner (a formula-rewrite program), lindsay (a communications simulator for a hypercube computer), and p2c (a Pascal to C translator).

Simulation parameters. All of the simulations were performed using twelve pairs of early and late eviction points. Three early points were used at 40%, 60%, and 80% of the memory size—That is, 40%, 60%, and 80% of the memory constituted the LRU region, respectively, leaving 60%, 40%, and 20% for the early region. The late points were chosen so that the probability $P(n)$ for $e < n <= l$ took the values 2/3, 1/2, 1/3, and 1/4.

One more parameter affects simulation results significantly. Recall that replacement decisions should be guided by program reference behavior of the recent past. To give more weight to recent virtual memory events, recency information needs to be "decayed". The rate of decay is tied directly to the memory size: the values for all our statistics are multiplied by a weight factor progressively so that the M-th most recent reference (M being the memory size) has 0.3 times the weight of the most recent one. The algorithm is not very sensitive with respect to this value. Values between 0.1 and 0.5 yield almost identical results. By decaying information exponentially on the correct timescale, we obtain the desired adaptivity, even with a large range of constant decay factors.

4.4.2 Locality Analysis

To show the memory reference characteristics of our traces, we plotted *recency-reference* graphs. Such graphs are scatter plots that map each page reference to the corresponding position on the recency axis. References to the most recently used pages are ignored, thus resulting in graphs that maintain the right amount of information at the most relevant timescale for a clear picture of program locality.

For instance, consider the recency-reference graph for the wave5 trace, plotted in Figure 4.5. The graph is produced by *ignoring* references to the 1000 most recently accessed pages. If such references were taken into account, the patterns that we see in Figure 4.5 would have been "squeezed" to just a small portion of the resulting plot; the high-frequency reference information would have dominated, and the patterns of interest would have been difficult or impossible to see.

Based on this graph, we can make several observations. First, wave5 exhibits strong, large-scale looping behavior. There seem to be loops accessing over 6000 pages. The horizontal lines represent regular loops (i.e., pages are accessed in the same order as their last access after touching the same number of other pages). Note also the steep "upwards" diagonals which commonly represent pages being

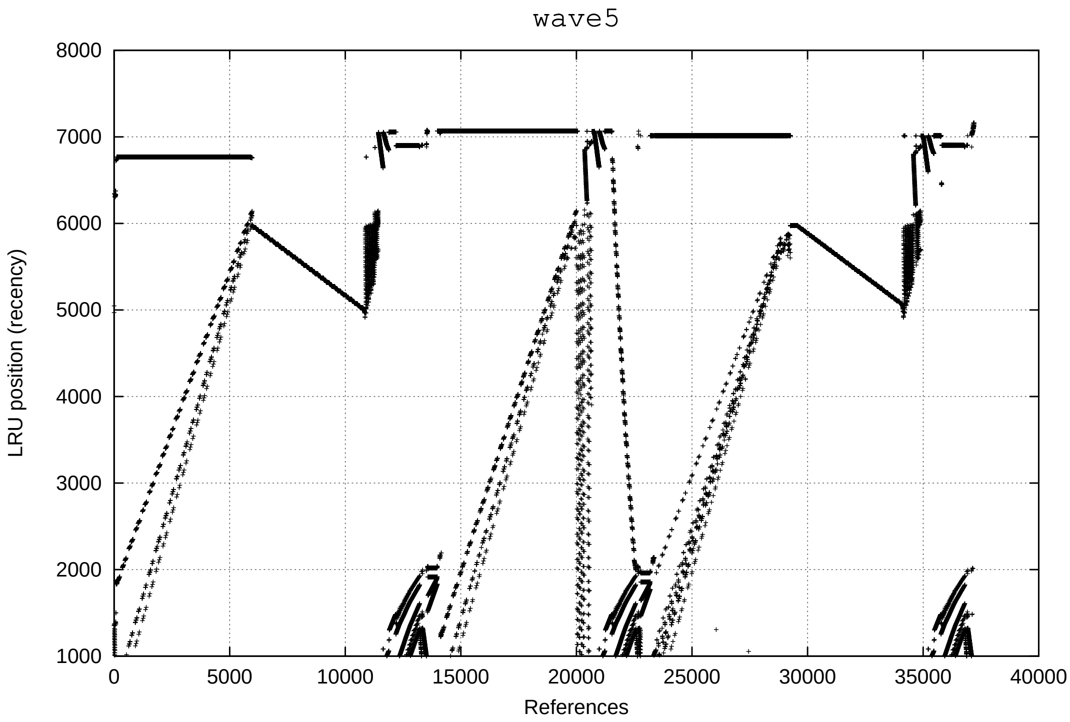

Figure 4.5: Recency-reference graph for wave5.

re-accessed in the *opposite* order from their last access.

Advantages of recency-reference graphs. A lot of information of this kind can be found in recency-reference graphs, giving such graphs several advantages over the usual space-time plots (e.g., see the plots in [Pha95, GC97]) for program locality analysis. To name a few:

- The information is more relevant. Instead of depicting which page gets accessed, recency-reference graphs show how recently a page was accessed before being touched again.
- High frequency information (e.g., hits to the few most recently accessed pages) dilutes time in space-time graphs. It is common that all interesting behavior (with respect to faulting) occurs only in a small region of a space-time graph. Such information does not affect recency-reference graphs, and can easily be filtered out depending on the memory size (and thus timescale) of interest.
- First-time reference information may dominate a space-time graph (e.g., allocations of large structures). Such information is irrelevant for paging analysis and does not appear in a recency-reference graph.

Seeing reference behavior in recency-reference graphs. Figures 4.6 through 4.9 present recency-reference graphs for representative traces. These plots lead to several observations we can make about the reference behavior of these programs:

- All programs exhibit strong phase behavior in recency terms. That is, their access patterns exhibit some clearly identified features that commonly persist. Comparing the values of the horizontal and vertical axes gives a good estimate of how long features persist. For all plots, features most commonly last for at least as many references as their "size" in pages.

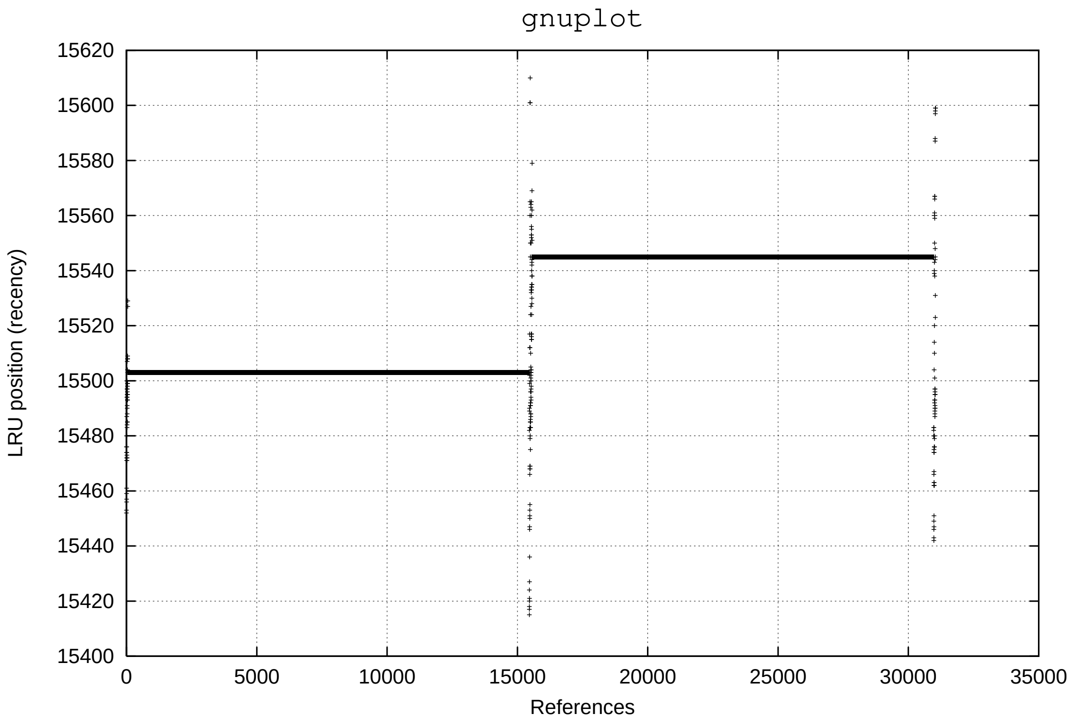

Figure 4.6: Recency-reference graphs for **gnuplot**.

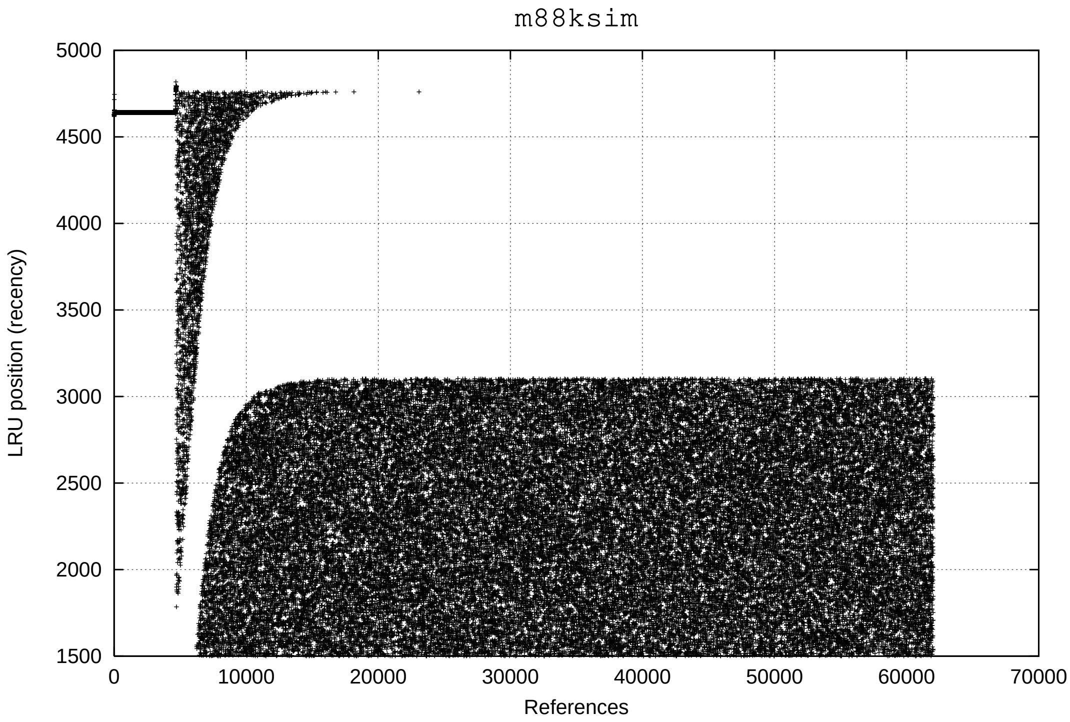

Figure 4.7: Recency-reference graphs for **m88ksim**.

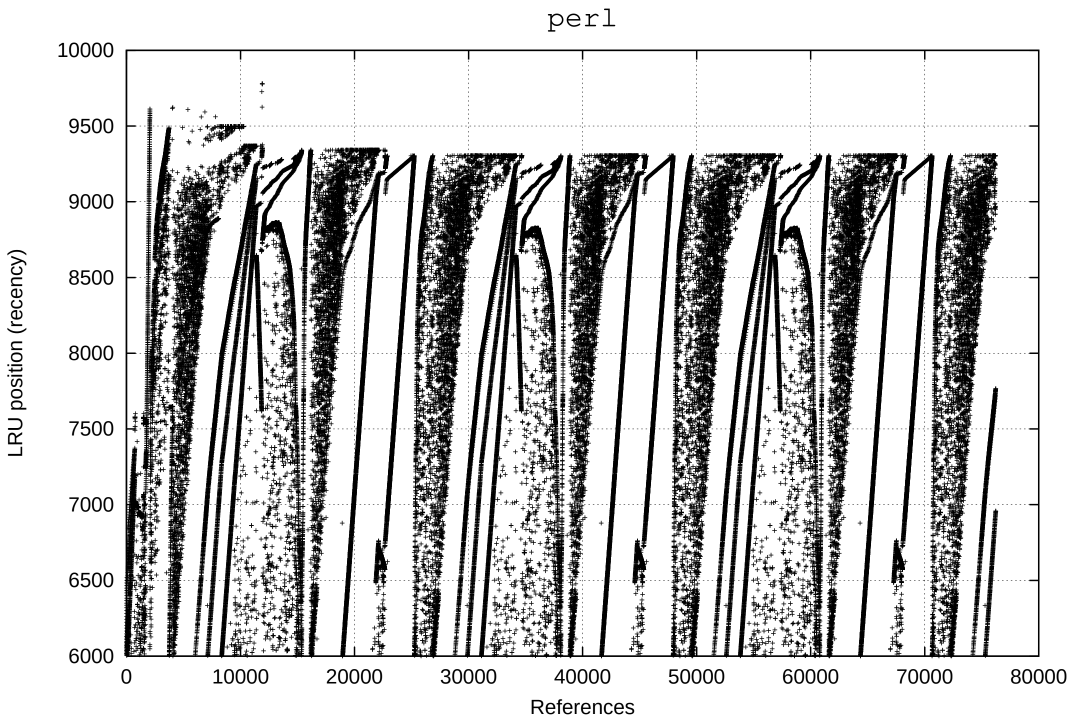

Figure 4.8: Recency-reference graphs for **perl**.

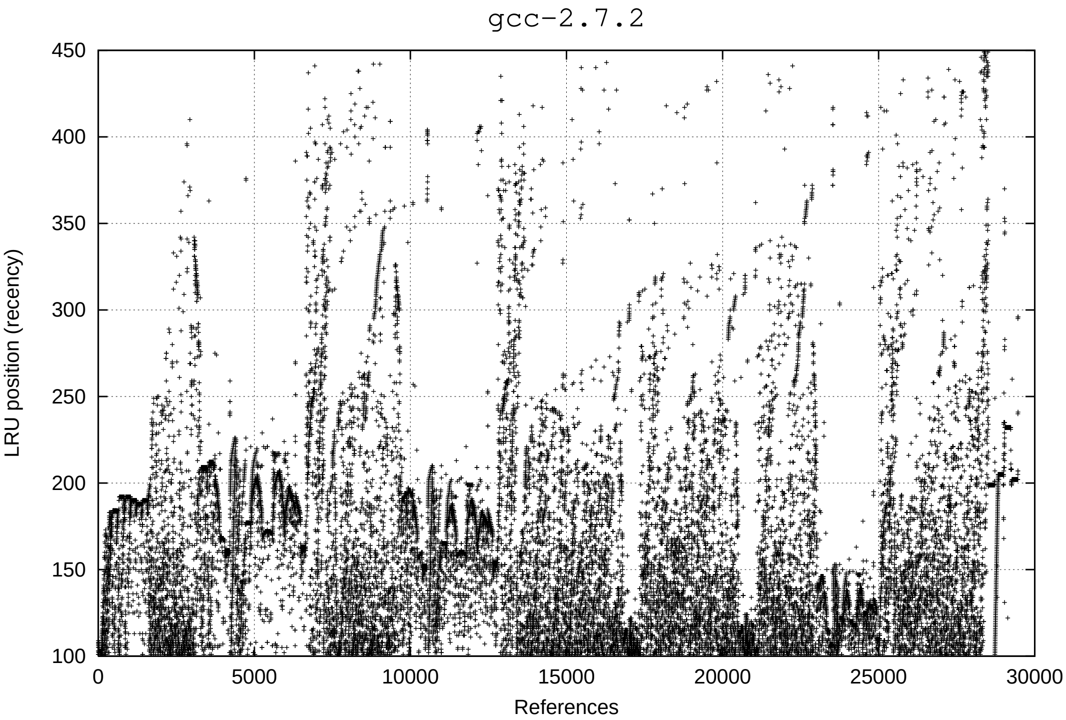

Figure 4.9: Recency-reference graphs for **gcc**.

- The gnuplot graph exhibits strong and large loops restricted to a very narrow recency region. All references in the gnuplot trace were either to the 200 most recently accessed pages (these are filtered out in the plot) or to pages above the 15,000 mark on the recency axis. This kind of access pattern is one for which LRU can perform terribly if less than 15,000 pages of memory are available.

- The m88ksim graph initially displays a large loop (note the short horizontal line), followed by a "puff-of-smoke" feature, and an area without distinctive patterns. The initial loop is a simple linear loop over a little more than 4500 pages. Note that it lasts for approximately the same number of references, indicating that it is just a loop with two iterations. The "puff-of-smoke" pattern is characteristic of *sets of pages* that are accessed *all together* but *in "random" (uniform) order*. When some pages in the set get touched, pages before them on the recency axis become less-recently-touched (i.e., move "upward"). Gradually, all accesses concentrate to higher and higher points on the recency axis (with the size of the set being the limit). The feature-less part of the m88ksim graph represents "random" accesses to a large structure. Given the nature of the application, we speculate that this could be a heavily used hash-table.

- The perl graph also exhibits "puff-of-smoke" features, together with steep diagonals (recall that these represent pages being re-accessed in the opposite order).

Other recency-reference graphs are similar to the ones shown. All six "small-scale" traces (espresso, gcc, grobner, ghostscript, lindsay, and p2c) behave like gcc, having no distinctive patterns. Murphi displays random references, much like m88ksim. The rest of the traces have regular patterns. In the case of trygtsl, these patterns are very clear, linear patterns (like those of gnuplot). In the cases of ijpeg and applu, the patterns look more like those of perl and wave5.

The reference patterns for which EELRU provides benefit. Based on the recency-reference graphs we can identify areas for which EELRU should exhibit a clear benefit. Thus, if a graph displays *high-intensity (dark) areas right above low-intensity (light) areas*, EELRU should be able to evict some pages early and keep in memory those that will be needed soon. Comparing these graphs with the results of our simulations (in the following sections) shows that this is indeed the case: the memory sizes for which EELRU is particularly successful are near such intensity boundaries.

4.4.3 First Experiment: Memory-intensive Applications

The eight plots in Figures 4.10 through 4.17 show the page faults incurred by EELRU, LRU, OPT, and SEQ for each of the eight memory-intensive traces. (SEQ is the algorithm of Glass and Cao [GC97], with which these traces were originally tested.) A detailed analysis of the behavior of LRU relative to OPT on these traces can be found in [GC97]. Here we will restrict our attention to EELRU.

EELRU consistently performs better than LRU for seven out of eight traces (for murphi, EELRU essentially performs LRU replacement). A large part of the available benefit, delineated by the OPT curve, is captured for all applications that exhibit clear reference patterns. A comparison with the SEQ algorithm is also quite instructive.[3]

How SEQ compares to EELRU. The idea behind SEQ is to detect access patterns by observing *linear faulting sequences*, where the linearity is in the address space. Thus, SEQ is based on detecting address-space patterns, while EELRU is based on detecting (coarse-grained) recency-space patterns. Each approach seems to offer distinct benefits:

[3]The results for SEQ were obtained by running the simulator of Glass and Cao on the traces. Testing SEQ on other traces would require significant re-engineering of the simulator, as its replacement logic is tied to their trace reduction format.

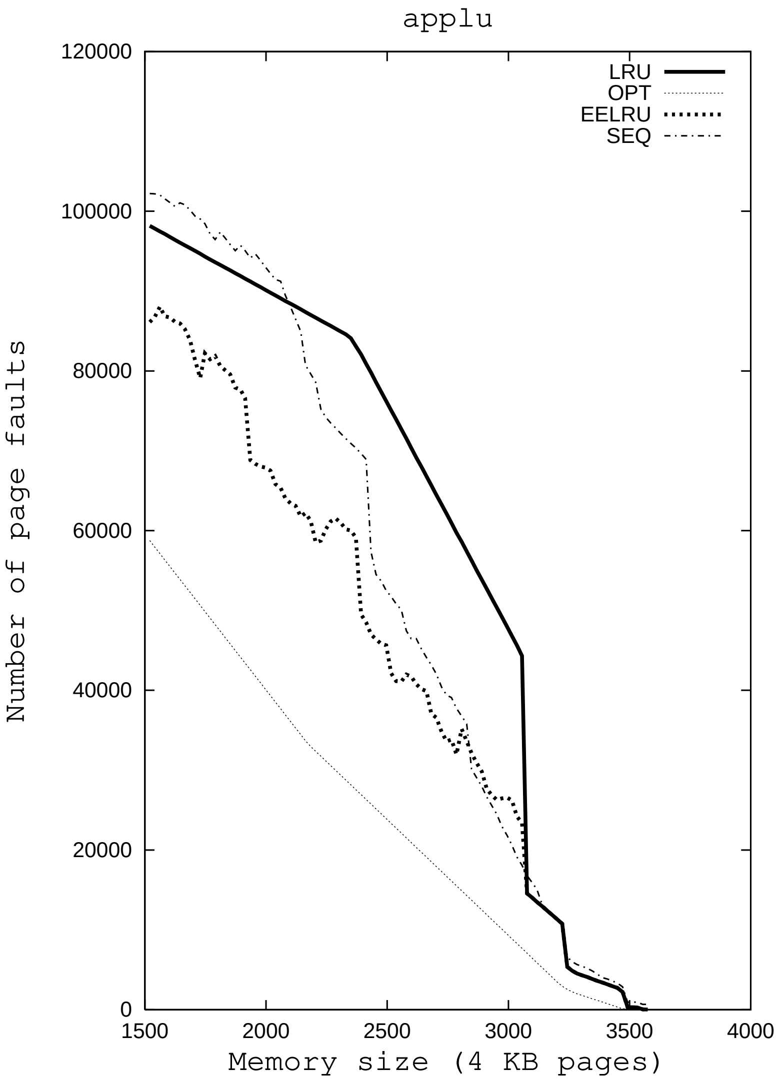

Figure 4.10: Fault plot for **applu**.

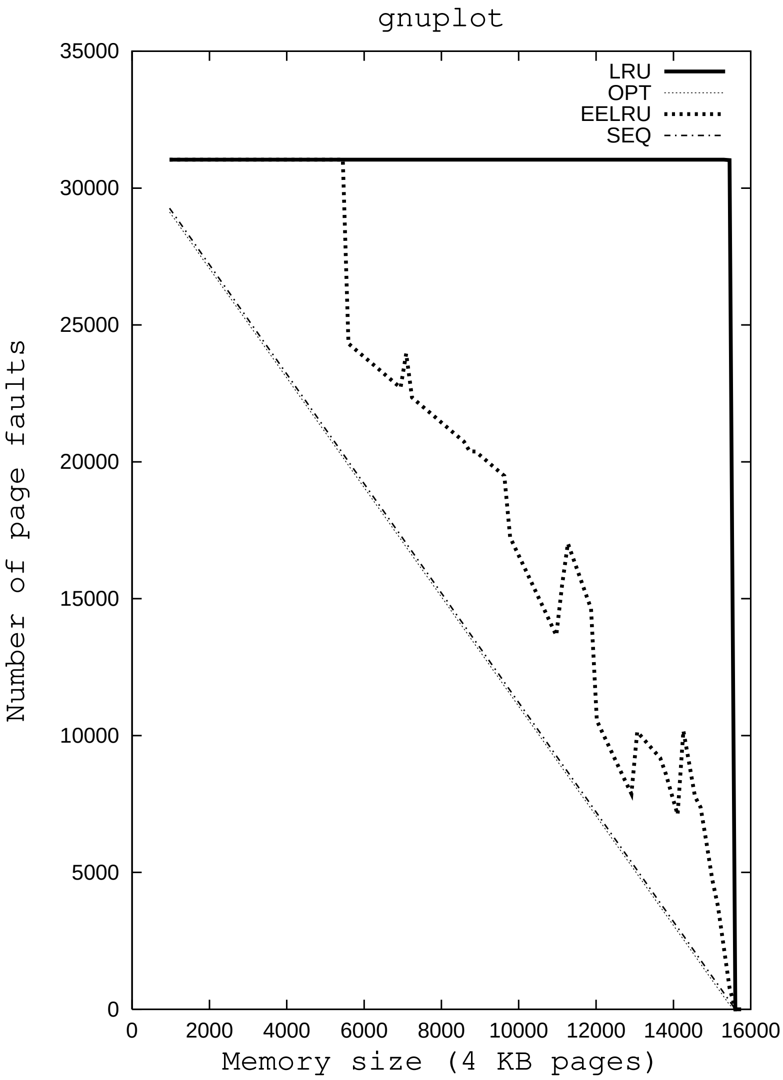

Figure 4.11: Fault plot for **gnuplot**. Note that the SEQ curve almost overlaps the OPT curve.

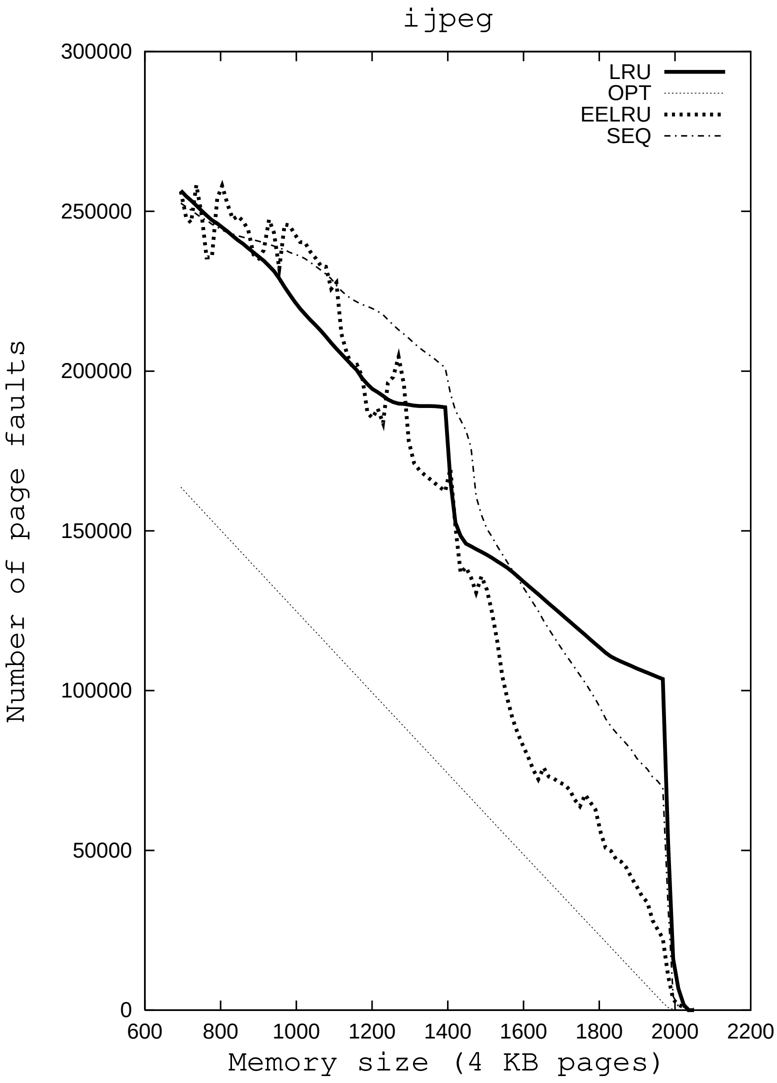

Figure 4.12: Fault plot for **ijpeg**.

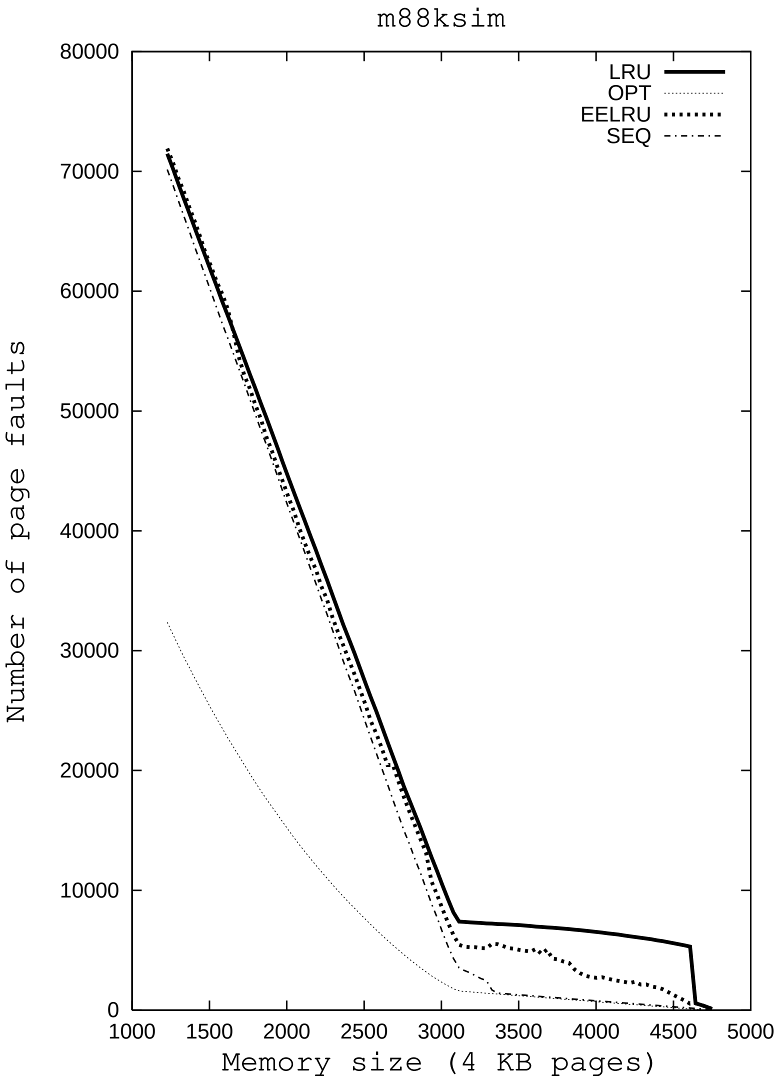

Figure 4.13: Fault plot for **m88ksim**.

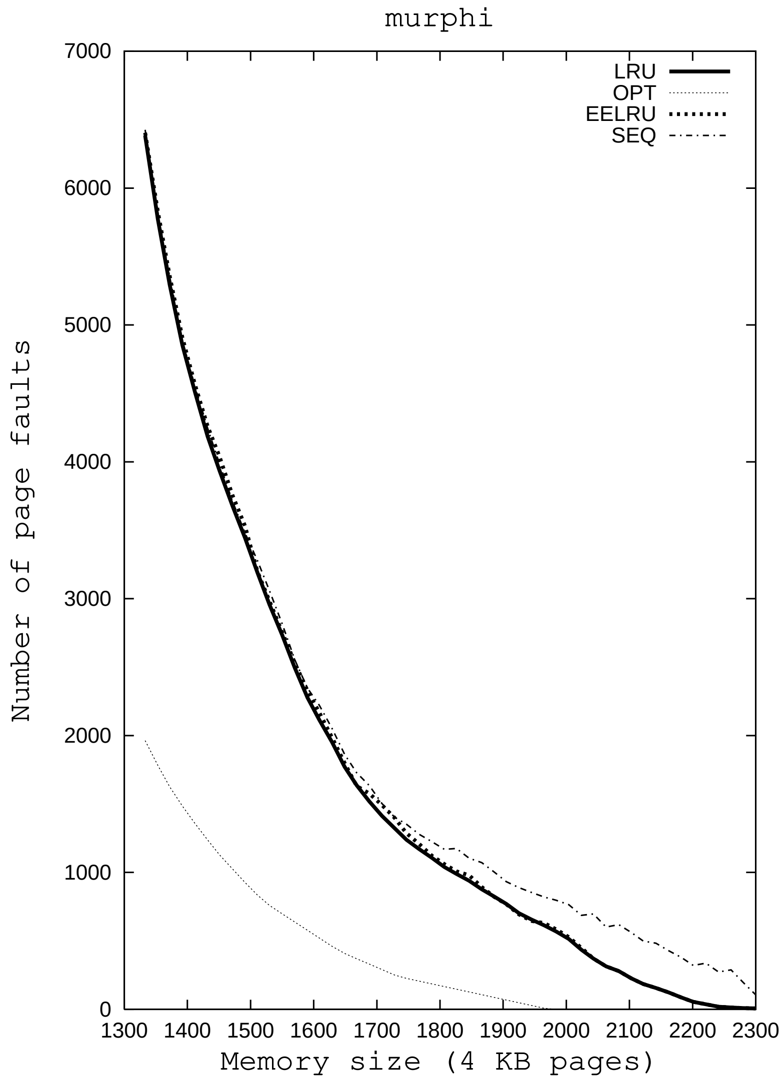

Figure 4.14: Fault plot for **murphi**. Note that the EELRU curve almost overlaps the LRU curve.

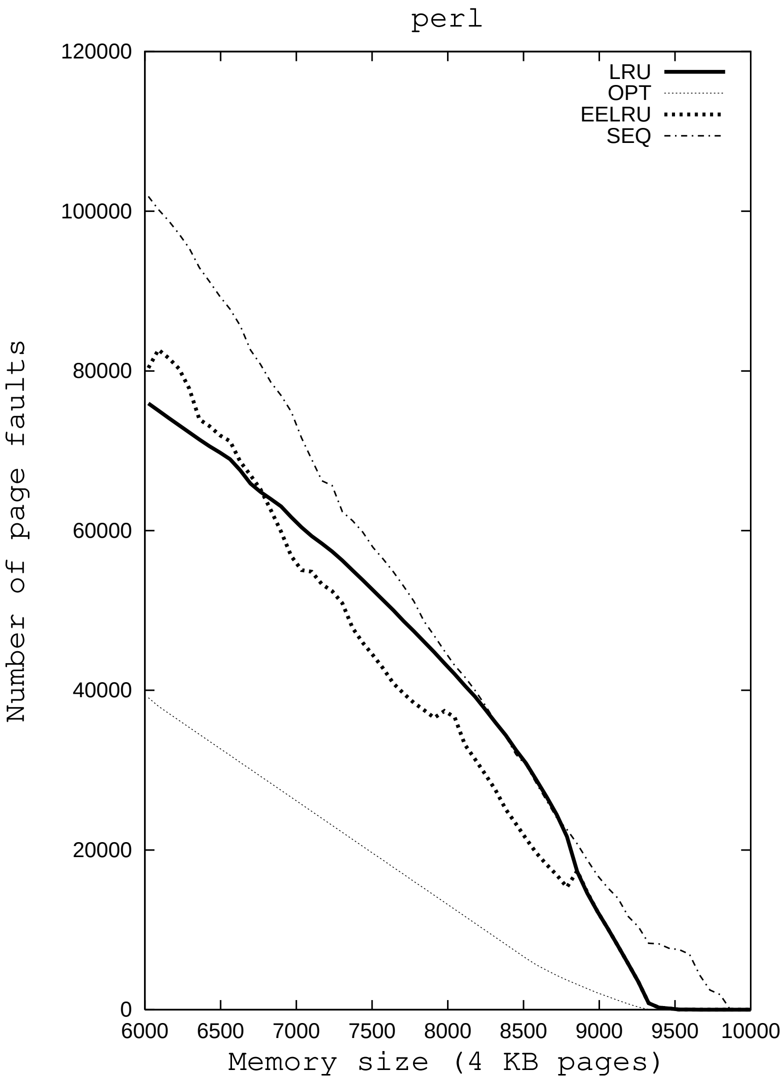

Figure 4.15: Fault plot for **perl**.

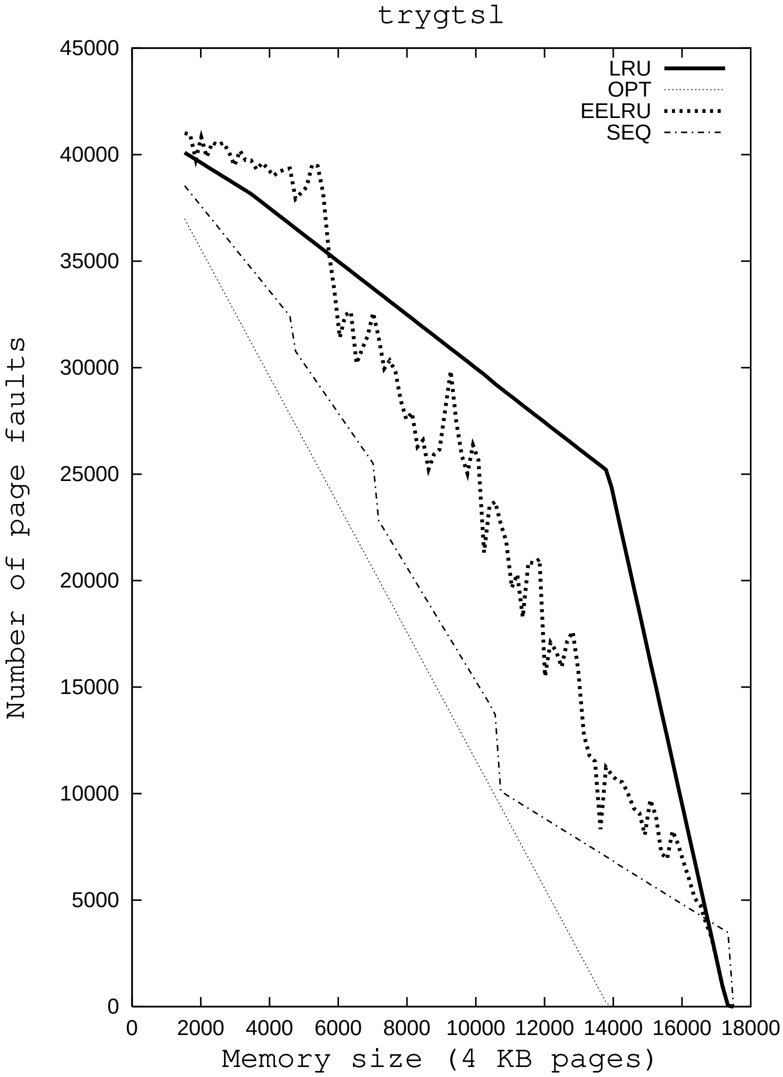

Figure 4.16: Fault plot for **trygtsl**.

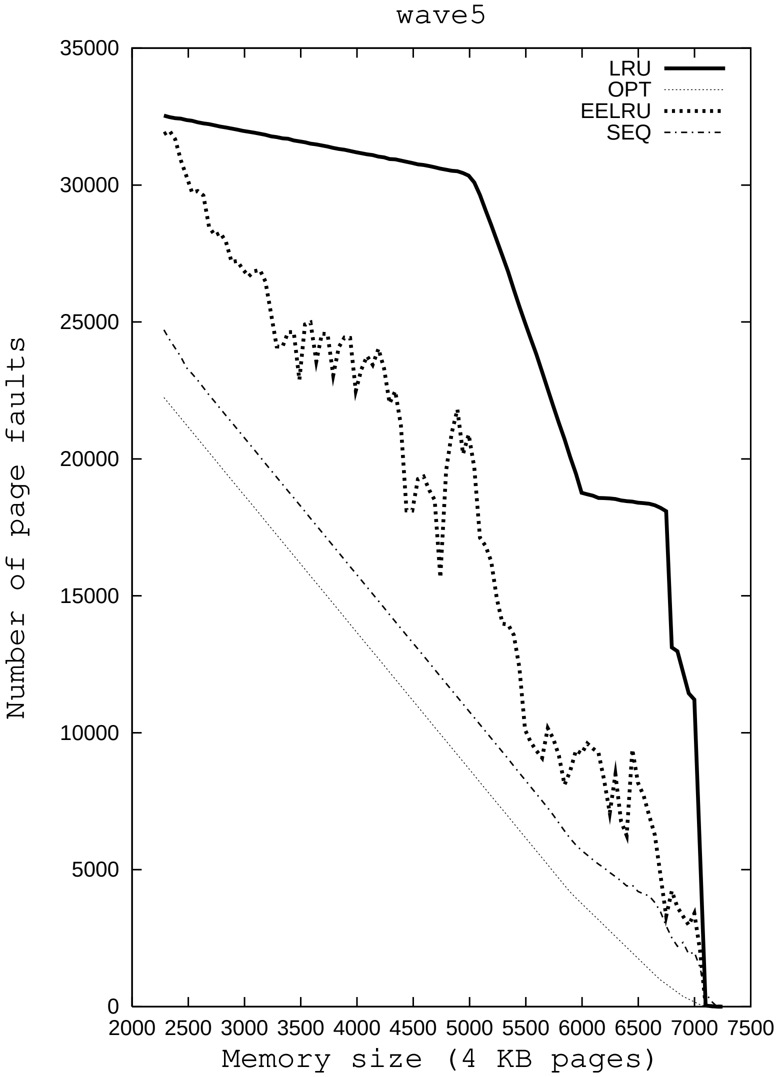

Figure 4.17: Fault plot for **wave5**.

- EELRU is capable of detecting regularities that SEQ cannot capture. For instance, a linked list traversal may not necessarily access pages in address order, even though it could clearly exhibit strong looping behavior. Such traversals are easily revealed by the recency information maintained by EELRU.

- SEQ can detect linear patterns more quickly than EELRU, and thus get more of the possible benefit in such cases. The reason is that recency information does not become available until a page is re-accessed (i.e., during the second iteration of the loop), while address information is immediately available.

The latter observation has consequences on the robustness of the two algorithms: EELRU is fairly conservative and only diverges from LRU in the case of persistent reference patterns. SEQ, on the other hand, is more risky and guesses about traversals which it has not encountered before (e.g., presumes that all sequential scans of more than 20 pages will be larger than main memory). Therefore, we expect that EELRU is much less likely to cause harm for programs with no large-scale looping patterns.

The results of our experiments agree with the above analysis. EELRU outperformed SEQ for three of the traces (applu, ijpeg, perl), SEQ outperformed EELRU for another three (gnuplot, trygtsl, and wave5), and for m88ksim and murphi the difference was small (EELRU was slightly better in one case and slightly worse in the other). Note that SEQ performed better for programs with very clear linear access patterns, as it could perform early loop detection. Even for these programs, however, EELRU captured most of the available benefit. For all traces with recency patterns that could be exploited, EELRU consistently yielded improvements to LRU, even when the SEQ results seemed to indicate that few opportunities existed (e.g., ijpeg, perl).

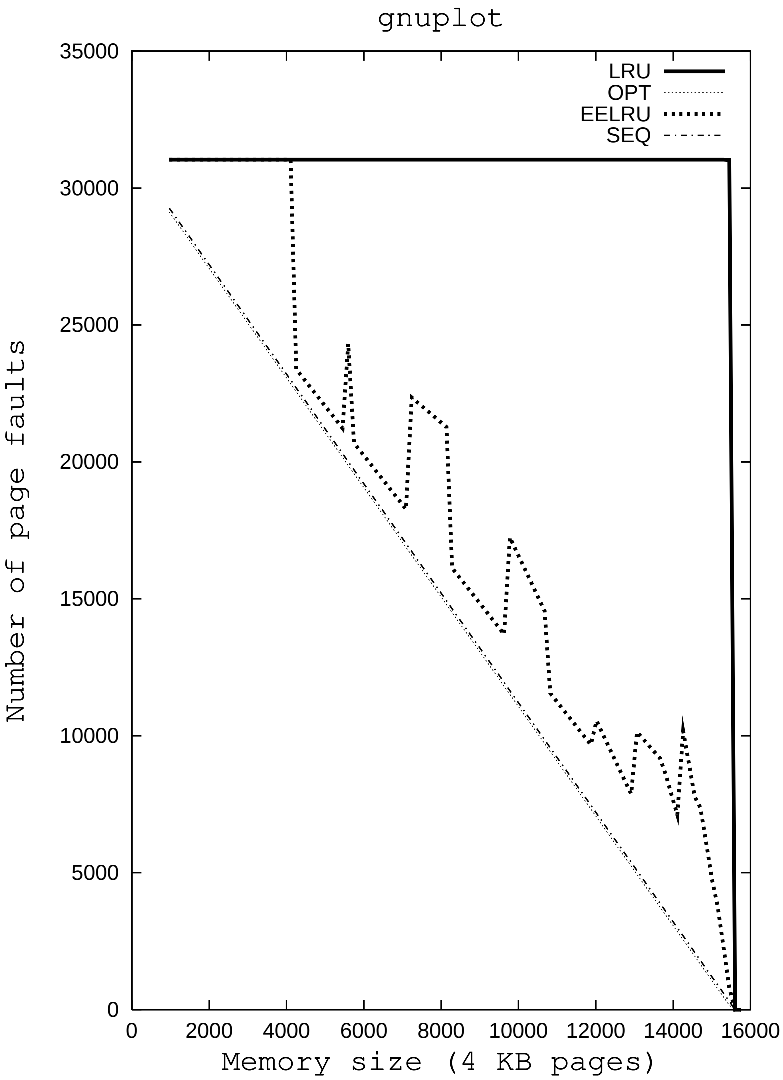

Figure 4.18: A different fault plot for **gnuplot**. For this second simulation, an extra very early eviction point was available to EELRU.

A less conservative EELRU simulation. Recall that, as mentioned in Section 4.4.1, the simulation of EELRU on the traces used for comparison with SEQ was conservative—at least 40% of the memory buffer had to be dedicated to the LRU region. This limitation caused some loss of effectiveness, especially for traces with clear access patterns. We can confirm the effect of this limitation for gnuplot, as the trace has enough information to enable meaningful simulations for LRU regions that are only 10% of the memory size. (That is, 90% of the memory is dedicated to the early region, and so the early eviction point is very early in recency terms.)

We added such an extra early eviction point and the result is shown in Figure 4.18. We can see that the performance of EELRU is much improved when compared to the conservative simulations shown in Figure 4.11. EELRU approaches the performance of SEQ and OPT, as expected. The gnuplot trace exhibits very regular behavior (in fact, it is dominated by a big loop). Hence, for memory sizes too small to contain the data needed for the loop, EELRU can evict many pages early so that more other pages can remain cached long enough to be used again.

EELRU is a more desirable approach to graceful performance degradation. Overall, we believe that the recency-based approach of EELRU is simpler, intuitively more appealing, and of more general applicability than address-based approaches like SEQ. The generality conjecture cannot, of course, be proven without extensive experiments and widely accepted "representative workloads" but the preliminary results of our experiments seem to confirm it.

Belady's anomaly. EELRU does exhibit what is known as "Belady's anomaly": increasing the size of physical memory may increase the number of page faults. EELRU suffers from this anomaly as a result of its adaptive behavior. It takes risks and tries to predict future program behavior based on past patterns.

For example, the decision to diverge from LRU depends on the memory size.

For a larger memory size, EELRU may decide that it is worth evicting some pages early in order to capture a loop which is significantly larger than the memory size. If the loop does not persist for long enough, EELRU will incur more faults for this larger size than it would for a smaller memory size, for which the adaptive heuristic does not attempt to evict pages early.

As another example of the risks taken by EELRU, recency patterns are often not steady (recall the diagonal lines in the plots of Section 4.4.2). Thus, the right early eviction point may not be immediately evident when EELRU starts diverging from LRU. This error in the EELRU estimate will have different results for different memory sizes, depending on the positions of late eviction points relative to the shape of recency patterns.

4.4.4 Second Experiment: Small-scale Patterns

Our second experiment applied EELRU to traces with smaller memory requirements. Even though these traces are not of programs that might make a real virtual memory system page, they help demonstrate that our approach is stable and handles a wide range of available memories. Additionally, these traces confirm that the patterns that EELRU recognizes are not unique to programs written with paging in mind. Also, the ability to adapt both to large-scale and to small-scale patterns is useful for any algorithm to be employed as a replacement algorithm in a multi-process environment. A short discussion on the potential of EELRU as a replacement algorithm in time-sharing systems can be found in Section 4.6.

Because these traces are of programs with smaller footprints, and the programs exhibit have no distinctive patterns (see the gcc recency-reference plot in Figure 4.9), we would expect EELRU to behave similarly to LRU. Such performance confirms the robustness of the algorithm—EELRU is unlikely to perform worse than LRU if no regular patterns exist. As can be seen in the fault plots of

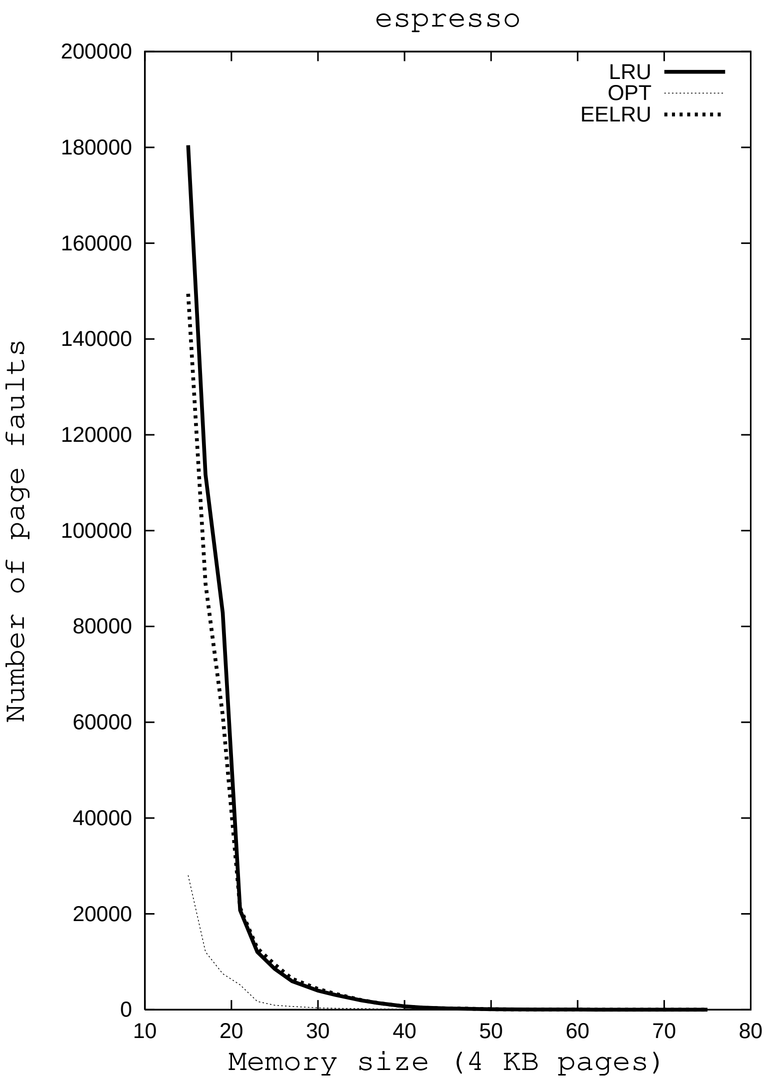

Figure 4.19: Fault plot for **espresso**.

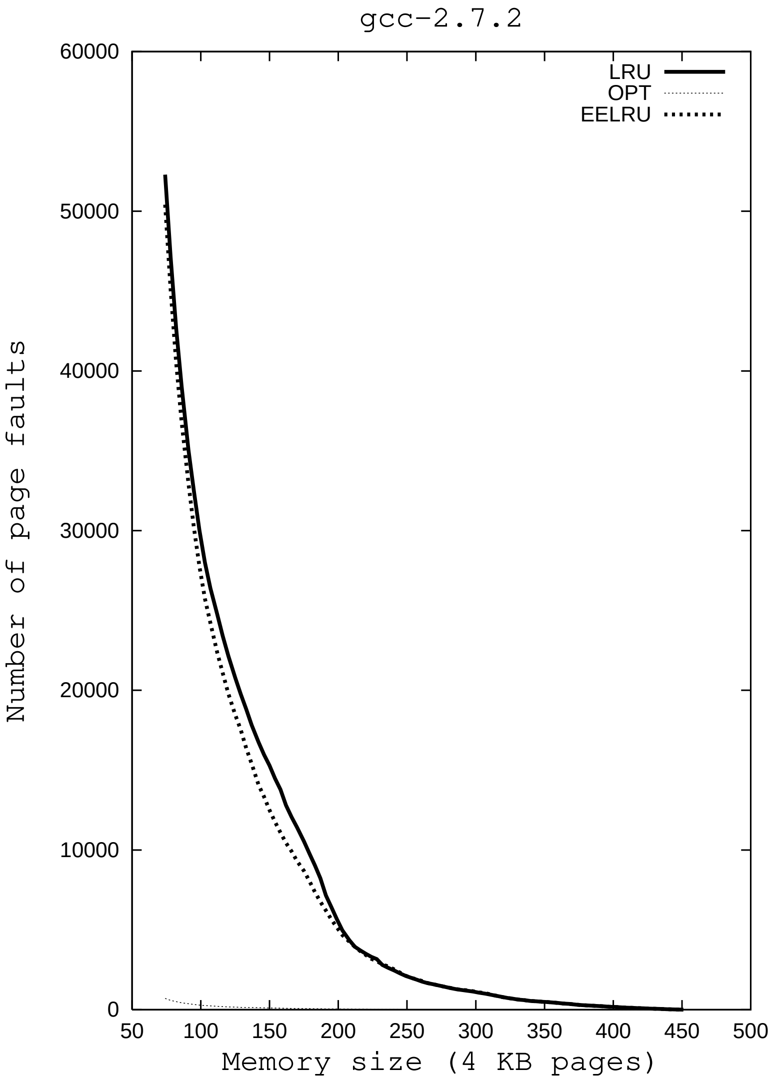

Figure 4.20: Fault plot for **gcc**.

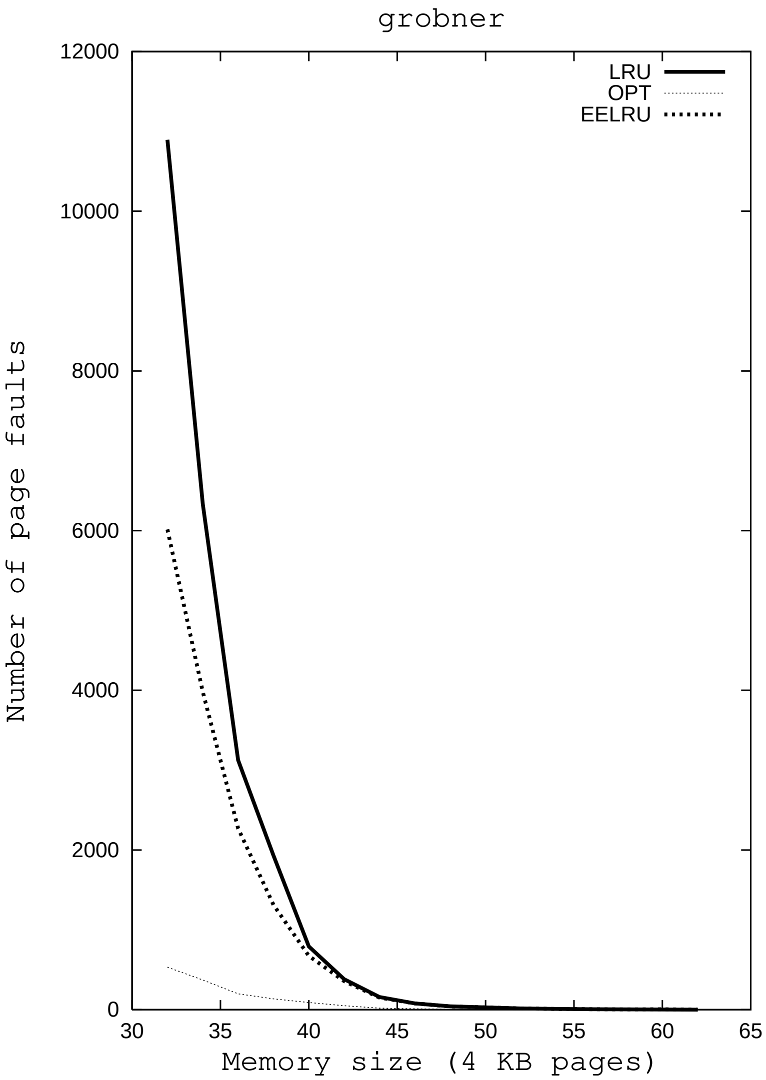

Figure 4.21: Fault plot for **grobner**.

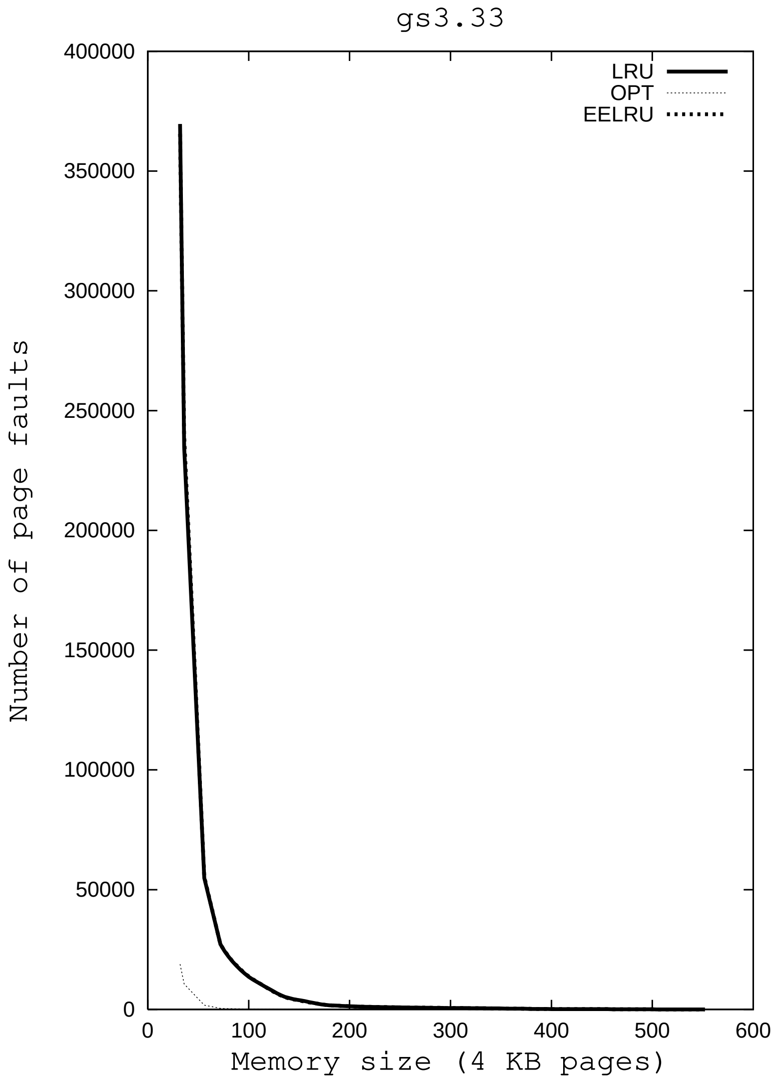

Figure 4.22: Fault plot for **GhostScript (gs3.33)**. Note that the EELRU line overlaps the LRU line.

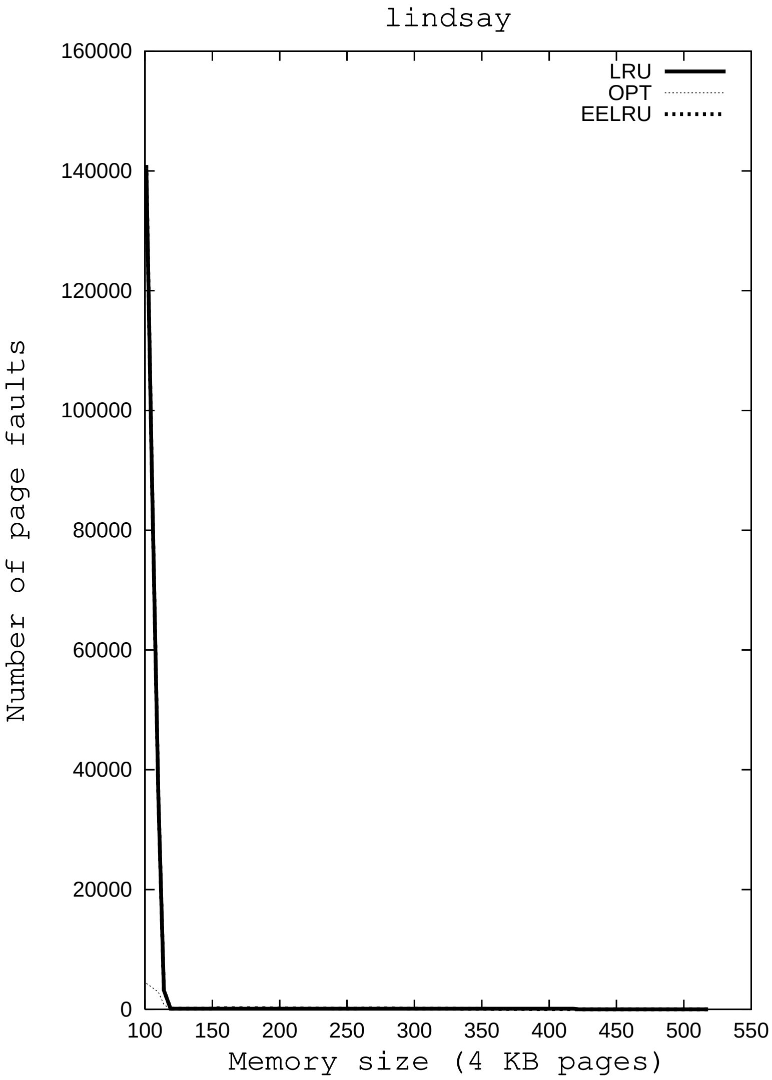

Figure 4.23: Fault plot for **lindsay**. Note that the EELRU line overlaps the LRU line.

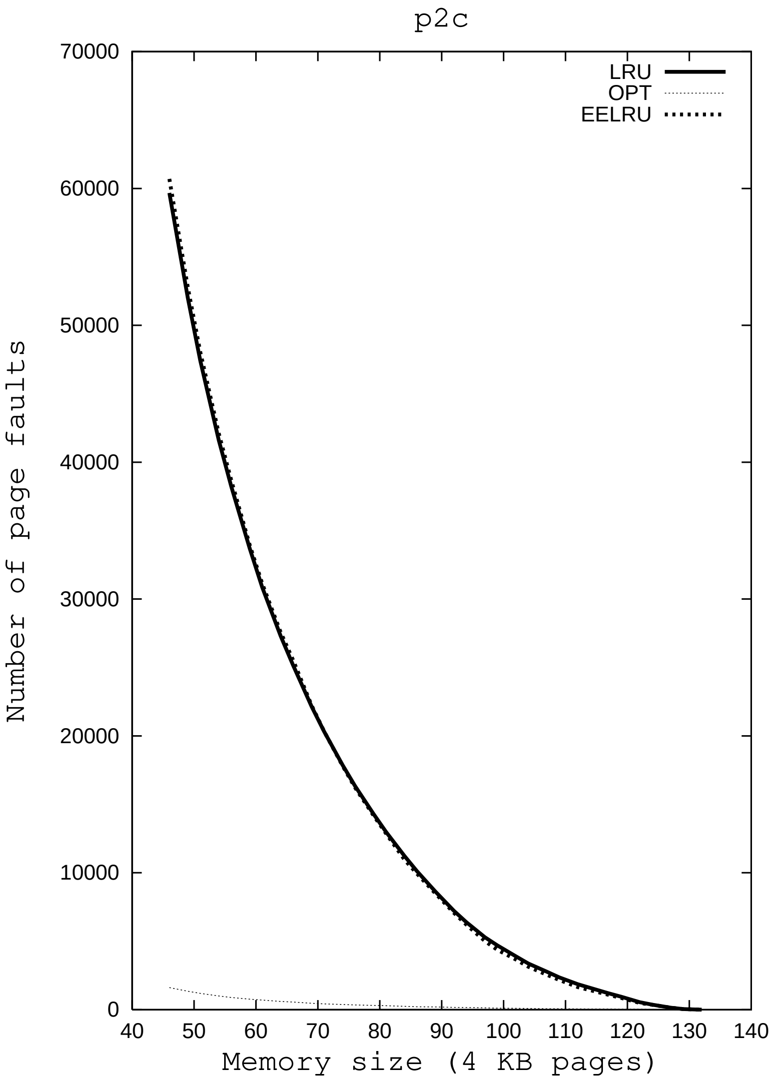

Figure 4.24: Fault plot for **p2c**. Note that the EELRU line overlaps the LRU line.

Figures 4.19 through 4.24, the performance of EELRU and LRU are nearly identical.

Note, however, that even for some of these traces EELRU manages to get a small benefit compared to LRU (around 10% less faulting on average). In particular, EELRU seems to be capable of detecting and exploiting even very small-scale patterns. An examination of the gcc recency-reference plot (Figure 4.9) is quite interesting. We see that there are two small regions where high-intensity (dark) areas are directly above low-intensity (light) areas. EELRU is exploiting exactly these small-scale patterns, and exhibits most of its benefit for a memory size around 150 pages—the boundary of the dark and light areas in the plot.

4.5 Thoughts on Program Behavior and Replacement Algorithms

LRU serves as a good reference point for understanding the performance of other replacement algorithms. We can roughly divide program behavior into four classes:

1. **LRU-friendly**: In very good cases for LRU, any excellent replacement policy must resemble LRU. Specifically, one must prefer to cache recently-touched pages the vast majority of the time, as LRU does. Programs that seldom touch more pages than will fit in memory, over fairly long periods of time, are LRU-friendly programs. If this regularity is sufficiently strong, LRU is nearly optimal, and many replacement policies (e.g., Random or FIFO) will also do well. This common case exhibits LRU's greatest strength.

2. **LRU-unfriendly**: If LRU chronically evicts pages shortly before they are touched again, any excellent replacement policy must evict some pages early in an attempt to cache other pages until they are used again, as MRU does. If a program's reference behavior consistently exhibits this kind of regularity, MRU-like behavior will be nearly optimal. This relatively common case

exhibits LRU's greatest weakness; addressing it properly is the main contribution of this research, and we demonstrate that this simple deviation from LRU-like replacement is sufficient to substantially improve on LRU.

3. **Unfriendly**: Programs which touch many pages that have not been touched for a very long time—i.e., pages that would have been evicted long ago by LRU—are unfriendly to *any* replacement policy, even OPT. LRU does "well" for them by default, because it is comparable to OPT in this case; our algorithm does the same.

4. **Mixed**: This catch-all category covers programs that often touch pages that would soon be evicted by LRU *and* pages that LRU would have recently evicted. That is, the aggregate recency distribution is not a particularly good indicator, by itself, of what a replacement policy should do. This is an area for future work.

Notice that for mixed behavior, a nearly optimal replacement policy must evict some pages in preference to others. It would not be sufficient to be uniformly fair like LRU, nor to be uniformly unfair like MRU. It must make finer distinctions, recognizing more subtle aspects of program behavior than we address here.

Conversely, notice that for the other three cases, such fine distinctions are unnecessary for good performance. Further, fine distinctions may even be harmful if not made very well; if any of the three regularities is strong enough, there is little benefit to be gained by attempting to make fine distinctions, because the crucial information is primarily in the *aggregate* behavior of large numbers of pages, *not* in the details of per-page reference histories.

4.6 Conclusions and Future Work

Replacement algorithms are important components of operating system design and can affect system performance significantly. We have presented EELRU: an adaptive variant of LRU that uses recency information for pages *not in memory* to make replacement decisions. We believe that EELRU is a valuable replacement algorithm. It is simple, soundly motivated, intuitively appealing, and general. EELRU addresses the most common LRU failure modes for insufficiently large memories, while remaining robust. Its performance can never be worse than that of LRU by more than a constant factor. Simulation results seem to confirm our belief in the value of the algorithm. The main axes of our future experimentation will examine EELRU as a replacement algorithm for time-sharing systems and the cost and performance of an in-kernel approximation of EELRU.

More generally, we have demonstrated that recency information can reveal reference behavior to which a virtual memory system can adapt. This information can be made timescale relative so that adaptation can respond to recent patterns that are likely to persist. We will show, in Chapter 6, that the same type of cost-benefit calculation based on recency information can be used to adapt the size of a compressed cache. In other words, the collection and timescale relative decay of recency information provides a system with useful information that can be used for a number of purposes.

Interestingly enough, there are several possibilities for a recency-based approach to a replacement algorithm in multi-processing systems. For one thing, EELRU itself could be useful as a "global" algorithm (i.e., managing all pages the same regardless of the process they belong to). For another, recency information of the kind maintained by EELRU could also help memory partitioning. That is, if a process incurs a lot of faults for recently evicted pages, a replacement algorithm could allocate more memory to that process, at the expense of a process for which a smaller memory space would not cause many faults. These ideas open up possibilities for quite sophisticated recency-based replacement algorithms. EELRU has already served to demonstrate some sound principles, like timescale-relative adaptivity, on which these algorithms should be based.

Chapter 5

In-memory Data Compression

5.1 Motivation and Background

The compression of data relies on the ability to find regularities in that data—a process known as *modeling* [BCW90, Nel95]. Once the modeling is performed, an *encoding* can be chosen that uses fewer bits to represent the regularities that have been detected. Although both the modeling and encoding phases contribute to compression, the latter problem is a simpler one. There are well understood approaches to the encoding problem that will provide a reduced representation of the input data.

Modeling an input to find its regularities is the more significant challenge for any data set. A modeling method may correspond well to the data on one domain, while failing to find any regularities in another domain. Moreover, the computational expense of a modeling method may be quite acceptable in some domains, and not in others.

The popular Lempel-Ziv (LZ) compression algorithms are designed to find regularities in byte streams, specifically human text. However, for compressed caching, we need to be able to compress small virtual memory pages, not large byte streams. While there are LZ variants, specifically LZO [Obe97] and LZRW1 [Wil91a], that can be applied to compressed caching, it is possible to devise compression algorithms that are better suited to this domain.

In-memory data exhibits different regularities than byte streams. The regularities are strong enough that we can describe a simple form of modeling, fundamentally different from LZ modeling, that is very effective on virtual memory pages. Compressed caching places unique demands on the performance characteristics of compression algorithms that can be met, but that LZ algorithms do not address.

In-memory data vs. text file regularities. Data stored in memory during a program's execution can be very different from the byte streams stored by file systems on which most lossless compression research has been performed. Most of these compression algorithms find repetitions of literal byte sequences, and then encode those in a small number of bits. The assumption made by these models is that sequences of bytes are likely to be exact repetitions of previously seen sequences of bytes.

In-memory data may not often comprise byte sequences. In order to model (and thus compress) in-memory data effectively, we must characterize what a virtual memory page is likely to hold. The following is a list of properties common to data held by virtual memory pages. These properties do not always hold, but we will find that they hold often enough to suggest a simple and effective model for in-memory data compression. Most in-memory data:

- is word or double-word aligned for faster handling by the processor, as unaligned loads and stores are slower.

- comprises a large portion of integers and pointers of word sized.

- contains integers that cover a range much smaller than the integer type can store, so that the majority of information is in the low bits.

- contains pointers to the same region of memory, so that the majority of information is in the low bits.

- contains many repeating, common values, particularly *zero*.

These regularities are not the only ones that exist for in-memory data. If the program stores complex structures that are stored in a consistent format, like a C `struct` or a C++ `class`, then there will also be sequential patterns to data. Sometimes in-memory data *is* a sequence of bytes; for example, the program may use significant input/output buffers. One program may or may not exhibit certain regularities if it is given different inputs.

The set of regularities listed above applies to most programs for most inputs. We will see later that these simple regularities are enough to provide significant compression, reducing virtual memory pages to 50% of their original size on average. Further, the quality of compression also achieved by LZ compressors suggests that there are other regularities that may be just as easy to capture with a domain-specific compressor.

5.1.1 Compression for Compressed Caching

The design of a compression algorithm depends not only on the regularities of the data, but also on the context in which that compression algorithm will be used. Again, most existing work in lossless compression has centered around the compression of byte streams from data on a file system.

In file systems, read operations are performed more often then write operations. As a result, a more computationally expensive compression algorithm is acceptable, as such operations will be performed only when data needs to be written to the file system. However, decompression is required for the many read operations, and so efficiency of decompression is very important. This asymmetry of compression and decompression efficiencies is common to the vast majority of LZ algorithms.

In a virtual memory system's compressed cache, read and write operations are likely to occur with nearly the same frequency. In order to service a fault on a page that is compressed, the incoming page must be decompressed *and* the page to the outgoing page must be compressed. Below is a list of desirable properties for a compression algorithm to be used for compressed caching:

- **Reduction:** If less space is needed to store a page, then more pages can be stored in a given compressed cache space. Not only does tighter compression more greatly expand the effective memory size, but it also requires that less RAM be taken from the uncompressed cache, thus reducing the number of uncompressed cache misses.

- **Compression and decompression speed:** If less time is required to compress one page and decompress another, then the overhead introduced by keeping some resident pages compressed is smaller. Therefore, it becomes less costly to take more RAM for the compressed cache in order to expand the effective main memory size. Note that both compression *and* decompression speed matter, as both must be performed with nearly the same frequency.

- **Small space overhead:** The purpose of a compressed cache is to make more effective use of RAM. Many compression algorithms require at least 16 Kbytes, commonly 64 Kbytes, and as much as 1 Mbyte to perform compression. The less space the compression algorithm needs, the more RAM that can be used to store both the uncompressed and compressed caches.

Note that for a page being evicted to disk, if it is clean, then it can be discarded, as the disk would contain a valid copy of it; disks are large enough to hold redundant copies of pages that are being cached in RAM. However, if a compressed cache were to hold redundant copies of pages also being held in the uncompressed cache, then valuable RAM is being wasted.

To avoid this redundancy, only one copy of a page should be held in RAM. Consequently, page evicted from the uncompressed cache into the compressed cache must be compressed, irrespective of its dirtiness. In Douglis' study [Dou93], redundant copies of pages were stored in both the uncompressed and compressed caches because the LZRW1 algorithm used in that study compressed slowly. We believe that the redundancy significantly reduced the effectiveness of the compressed cache, and was partially responsible for the ambiguous results.

5.2 The Wilson-Kaplan (WK) Compressors

Based on the simple regularities of in-memory data provided above, combined with the desirable properties of a compression algorithm for compressed caching, we have designed the family of *Wilson-Kaplan (WK)* compressors. These algorithms are designed to detect the regularities that exist in many virtual memory pages. They provide a reasonable trade off between effective reduction and computational time, all with a very small space overhead.

Note that we do not believe that our current WK variants represent an ideal approach to compression in this domain. We believe that there are further improvements in the modeling and encoding of in-memory data that should be investigated. However, the current WK compressors are of interest because, in spite of their unsophisticated modeling and encoding techniques, they provided the best performance in our compressed caching results (see Chapter 6).

The WK compression algorithms are recency based, dictionary compressors. A small number of recently seen symbols is stored in a dictionary. Each new input symbol is compared to those in the dictionary, in the hopes that recently used symbols will match or resemble the current input symbol. If the comparison is successful, then the algorithm has captured a regularity in the data that can be encoded in a small number of bits.

Because compression and decompression speeds matter, the modeling and encoding components of the WK compressors are interleaved. More specifically, the internal representation used to store the results of modeling is exactly the set of bits used for encoding. Once the entire input is modeled and encoded, there follows a *packing* phase that gathers the encoding bits so that they can be emitted in a fast and space efficient manner.

We will describe these two phases and their implementation. We will also describe the three WK variants, whose names are derived from the manner in which each organizes its dictionary of recently seen words. Specifically, `WKfa` uses a fully associative dictionary, `WK4x4` uses a 4-way set associative dictionary with 4 elements per set, and `WKdm` uses a direct-mapped dictionary. We will find that the associativity of the dictionary has a moderate effect on compression tightness, and a significant effect on compression time.

5.2.1 The Modeling/Encoding Phase

The WK compressors use a dictionary to match input symbols against recently seen symbols. By doing so, the compressor models the similarity of values that are spatially local. For each input symbol compared to those in the dictionary, the compressor must record the results of its modeling. Because speed of execution is very important for the WK compressors, the result of modeling a symbol is immediately recorded as the set of bits used for the compressed encoding. As such, the two tasks of modeling and encoding are interleaved.

Reading the input. Since in-memory data often comprises words, and pieces of data are aligned on word or double-word boundaries, the WK compressors take input one word at a time. By taking input in larger chunks than LZ implementations, which read bytes, the WK compressors reduce the number of input symbols it must read, and thus reduce the computational time. However, by reading words, it becomes more difficult for the WK compressors to find byte-based regularities.

Comparing the input to expected values. Based on the regularities of in-memory data, the WK compressor must compare each word to some expected values. The sequence of comparisons is based on both how likely the comparison will succeed, and how expensive that comparison is to perform.

The first expected value is a special case—*zero*. This special case exists because it is so common. By making it a special case, a dictionary entry need not be wasted to store zero as a recently seen value.

If the value is not zero, then the word is compared to every entry in the dictionary in search of an *exact match*. Such a match is desirable because it allows for a small encoding, as an index into the dictionary is all that is needed. These matches occur because, for many programs, a small set of identical values are used often, and clustered together onto pages.

We asserted that pointer and integer values of in-memory data are similar to one another, with most of the information residing in the low bits. In order to capture the redundant similarity in the high bits, we compare the input word to the same dictionary entries, this time in search of a *partial match*.

A partial match between a word read from the input data and a word in the dictionary is achieved when the upper 22 bits match, and the lower 10 bits differ.[1] The numbers chosen were based on empirical evidence, but that evidence also showed that compressibility would not change radically with a different split of high and low bits. So long as the information in the low bits can be separated from the redundancy of the high bits, a good deal of integer and pointer regularity can

[1]The choice of using 22 low bits and 10 high bits is based on a 32 bit machine word size. These numbers would likely change for a 64 bit architecture.

be captured.

How the dictionary is searched. Note that we have not specified the order in which the current input word is compared to the dictionary entries. The three different variants of the WK algorithms that we've implemented arrange the 16 word dictionary in difference ways. Here, we will describe how each of these three variants arranges the dictionary and performs updates to it.

- **WKfa:** The 16 word dictionary is *fully associative*. All 16 entries are kept in LRU order—the more recently seen a word is, the earlier it appears in the dictionary. When an input word is compared to the dictionary entries, the comparisons are ordered from the most recently seen word to the least. Every entry may be used for comparison, which is why this variant is fully associative.

- **WK4x4:** The dictionary is used as a 4-way, set associative cache of the most recently used words. Each set comprises 4 entries, and the entries within a set are maintained in LRU order. Before the current input word is compared to any dictionary entries, that word is mapped to one of the sets by a hash function. As with the fully associative version, comparisons are ordered from the most recently to least recently seen words. Unlike the fully associative version, only those words in the appropriate set are used for comparison.

- **WKdm:** For this variant, a hashing function maps the current input word to exactly one of the 16 dictionary entries. First exact and then partial matches are attempted against this entry, and no other entry is considered for a match. That entry is then updated with the most recently seen input word hashed into it.

The `WKfa` algorithm was the first to be implemented, as it was the most straightforward. It achieved reasonable compression, but was slower than desired. Much of the execution time was spent during attempts to match input words that were destined not to match any entry against every single word in the dictionary. Additionally, the cost of implementing the *Move To Front (MTF)* algorithm that maintained LRU order required a handful of operations.

In order to reduce the time spent searching the dictionary for matches, we used the idea of set associativity so common to hardware caches, and created `WK4x4`. Not surprisingly, the execution time was significantly reduced as a result of fewer futile comparison attempts. What was not expected was that this set associative dictionary yielded greater reductions in page size.

By instrumenting `WKfa` to accumulate the number of matches found at each dictionary entry, we found that the vast majority of matches were to the first four most recently seen words. The other twelve entries rarely yielded a match. As such, few input words failed to have dictionary matches with `WK4x4` because comparisons were made to only the four most recently seen words that were hashed into the same set. Further, the 12 rarely useful entries stored by `WKfa` served only to pollute the cache. By making it set associative, that pollution was reduced, and a greater number of matches were made possible.

Although `WK4x4` was faster and compressed more tightly than `WKfa`, it still wasn't faster than the best LZ implementations that we found. In order to eliminate even more unsuccessful matches, and to remove the MTF algorithm from the dictionary update process, `WKdm` was created. Because each input word can match only one dictionary entry, the size reduction provided by this new algorithm was not as good as with `WK4x4`. However, the tightness of the compression was still quite good, and the speed increase was significant. Not surprisingly, we will see in Chapter 6 that, when applied to compressed cache simulations, `WKdm` provides the best caching performance.

Storing modeling results as the encoding representation. The results of modeling each word need to be stored by the compression algorithm. For the sake of execution speed, the internal representation of this data is exactly the set of bits that will be used as the output encoding. While these two steps could be separated by an intermediate representation, directly recording these bits will make the actual emission of the output very fast during the packing phase.

Tag values. For every word compressed, there is a *two bit tag* that indicates what kind of match, if any, occurred when the word was compared against the expected values. Based on those tag values, the decompressor can determine how to reconstruct the decompressed word.

Depending on the kind of match that occurs during modeling, other bits may need to be output so that the decompressor has all of the needed information. Below, we list the tags that correspond to each kind of match found by the modeling phase, and then the bits that are output along with each tag.

1. **Zero:** Because zeros are so common, words that are all zero-valued bits are a special case for these algorithms. If the input word is zero, this tag value is used and no dictionary comparisons are performed.

   Other bits: *none*

2. **Exact match:** The input word is compared against entries in the dictionary. If the input word is equal to a word in the dictionary, then an exact match has been found.

   Other bits:

   - `4 bit dictionary index:` An index to the dictionary entry contains exactly this input word.

3. **Partial match:** If the input word matches a dictionary entry in the top 22 high bits, but differs in the low 10 bits, then a partial match has been found.

   Other bits:

   - `4 bit dictionary index:` An index to the dictionary entry contains the word whose 22 upper bits match the 22 upper bits of the input word.
   - `10 bit difference pattern:` The 10 low bits of the input word that differed from the 10 low bits of the dictionary entry.

4. **No match:** If this input word has not zero, and matches no dictionary entry, neither exactly nor partially, then it is not an expected value. This case is the only one for which the representation of the input word will expand.

   Other bits:

   - `32 bit literal pattern:` The entire 32 input word, literally represented.

This simple representation allows the results of modeling to be recorded quickly. It is also simple enough that it can easily and quickly be parsed by the decompressor.

The complete, interleaved modeling and encoding. Now that we know that the WK algorithms read the input as words, and compare each input word with recently seen words in a dictionary, we can provide a more concrete description of the algorithm. Figure 5.1 shows the algorithm employed by each WK compressor, and Figure 5.2 the algorithm for each WK decompressor. Note that the details of dictionary lookup and update aren't shown, as those are different for each WK variant.

```
WK_COMPRESS(uncompressed_page)
dictionary.INITIALIZE()
for each current_word in uncompressed_page
 do
  if current_word = 0
   then
     RECORD(zero_tag)
     continue
  entry ← dictionary.EXACT_MATCH(current_word)
  if entry ≠ NULL
   then
     RECORD(exact_match_tag)
     RECORD(entry)
     continue
  entry ← dictionary.PARTIAL_MATCH(current_word)
  if entry ≠ NULL
   then
     RECORD(partial_match_tag)
     RECORD(entry)
     low_bits ← EXTRACT_LOW_BITS(current_word)
     RECORD(low_bits)
     continue
  RECORD(miss_tag)
  RECORD(current_word)
```

Figure 5.1: The WK compression decompression algorithm. The compressor reads the source page one word at a time, looking for zeros, exact matches to the dictionary, and partial matches to the dictionary. It stores the modeling results as a simple, ad-hoc encoding, centered around two bit tag values that indicate which kind of match was found for each word.

```
WK_DECOMPRESS(compressed_page)
dictionary.INITIALIZE()
for each tag in compressed_page
  do
    if tag = zero_tag
      then
        OUTPUT(0)
        continue
    if tag = exact_match_tag
      then
        entry ← READ_INDEX()
        word ← dictionary[entry]
        OUTPUT(word)
        continue
    if tag = partial_match_tag
      then
        entry ← READ_INDEX()
        low_bits ← READ_LOW_BITS()
        word ← ASSIGN_LOW_BITS(dictionary[entry], low_bits)
        OUTPUT(word)
        continue
    ASSERT(tag = miss_tag)
    word ← READ_WORD()
    OUTPUT(word)
```

Figure 5.2: The decompressor reads those tags, plus other necessary bits, to direct its decompression. By maintaining a dictionary in the same fashion as the compressor, it can reconstruct the uncompressed page, one word at a time.

Choosing the order in which matches are found. In the compression algorithm, the search for regularities proceeds from the more specific to the more general. The special case of *zero* is used for comparison first. If that comparison fails, the input word is compared against dictionary entries for exact matching, and then for partial matching. If all of these comparisons fail, then the modeling predictions failed, and the input word is considered a "miss".

There is no reason that the input word must first be matched against the most specific case, and then later be compared to more general cases. Proceeding from specific to general is a common approach, but a different ordering may be more efficient. For example, if partial matches are common but exact matches rare, execution will be faster if partial matches are attempted before exact matches. If misses are more frequent than either partial or exact matches, then attempting a partial match first allows the algorithm to skip the attempt at an exact match.

It is also useful to note that changing the order of comparisons does not affect the decompression algorithm. Because the decompressor is guided by the sequence of tags, the decompressor will reconstruct the original words in exactly the same sequence as the compressor modeled them. Although none of our WK implementations took advantage of this property, it would allow the compressor to adapt its order of comparisons depending on the most commonly used cases.

This simple algorithm effectively captures the regularities that we described earlier. Word sized and aligned values with identical upper bits but differing lower bits are easily detected by a simple, recency based dictionary. The small dictionary is enough to provide significant compression, while still allowing for low space overhead and fast dictionary updates.

This simple algorithm could also be significantly improved. A larger dictionary, perhaps one like the sliding window used by LZ77, could provide more matches. A fast Huffman encoding implementation could yield significantly greater reduction. The model could also detect sequences of words, rather than recent repetition of words, to improve compression. However, this simple algorithm performs well, demonstrating the power of domain-specific compression.

5.2.2 The Packing Phase

In our earliest implementations, the bit values used to record the results of modeling were output in a stream—for each word of the input, the corresponding bits created by the modeling/encoding were emitted together. This approach required that the bit stream be accumulated, one word of emitted bits at a time, by performing a number of conditional checks, bit shifts, and bit-wise **or** operations. These are expensive instructions that must be performed for every tag, dictionary entry index, low bit string, and full word in the compressed representation.

This word-by-word approach to emitting the encoding was too expensive. Worse, the parsing of such a bit stream by the decompressor was slower than its creation: The compressor could group together some of the bits to be added to the output stream before performing the shifts and bit-wise **or** operations, but the decompressor would have to read individually each item of the bit string through a sequence of these bit manipulations.

A fast approach to emitting the encoding. Our initial decompressors were hopelessly slow compared to efficient LZ algorithms. Therefore, we developed a more time-efficient method of outputting the same encoding items. This new method reduced the time required to construct the encoded bits for output by an order of magnitude, down to only a few thousand operations.

Because there are four different types of items that comprise a WK compressed encoding—two bit tags, four bit dictionary entries, ten bit partial match patterns, and 32 bit "no match" patterns—four arrays are allocated during compression, one for each kind of encoding item. As modeling is performed for each word, each encoding item is recorded into its respective array. For efficiency, however, only one encoding item is written into each array entry. As such, the recording of modeling results is fast, but each entry these arrays contains a number of bits that store no information.

The encoded bits in the arrays must be packed together, leaving behind the meaningless, unused bits in each array entry. There are two key details to understanding the packing process and its efficiency:

1. By shifting the meaningful bits of one entry to correspond with meaningless bits in another entry, two entries can be combined by the bitwise XOR operation. The result can then be combined with more entries until no meaningless bits are left.

2. Each entry in an array may be only a byte, or a half word. Instead of combining just pairs of entries per XOR operation, whole machine *words* from the array can be combined at once. As a result, many entries can be combined in parallel, thus reducing the time required to pack an array.

As an example, consider Figure 5.3, in which sixteen consecutive entries in the array of tags are packed into a single, 32 bit word. This array is allocated as an array of bytes. Each byte from the array is used to store a single, two bit tag value. Therefore, each array entry contains two meaningful low bits, and six meaningless high bits.

A contiguous group of sixteen entries can be considered four words, with four tag entries per word. With each combining operation, four new entries are added to the final, accumulated word. By combining the four words, the tags are interleaved in a deterministic manner that can easily and quickly be unpacked by the decoder.

The number of entries combined, as well as the number of bits shifted, depends entirely on the size of the entry. Consider the analogous packing algorithm

The initial tag entries stored in their array:

```
Word #   Word value
--------------------------------------------
  1      000000Aa 000000Bb 000000Cc 000000Dd
  2      000000Ee 000000Ff 000000Gg 000000Hh
  3      000000Ii 000000Jj 000000Kk 000000Ll
  4      000000Mm 000000Nn 000000Pp 000000Qq
```

Shift word 2 left by 2 bits, and combine in with word 1:

```
  1      000000Aa 000000Bb 000000Cc 000000Dd
  2      0000Ee00 0000Ff00 0000Gg00 0000Hh00
--------------------------------------------
accum    0000EeAa 0000FfBb 0000GgCc 0000HhDd
```

Shift word 3 left by 4 bits, and combine it with the accumulated word:

```
accum    0000EeAa 0000FfBb 0000GgCc 0000HhDd
  3      00Ii0000 00Jj0000 00Kk0000 00Ll0000
--------------------------------------------
accum    00IiEeAa 00JjFfBb 00KkGgCc 00LlHhDd
```

Shift word 4 left by 6 bits, and combine it with the accumulated word:

```
accum    00IiEeAa 00JjFfBb 00KkGgCc 00LlHhDd
  4      Mm000000 Nn000000 Pp000000 Qq000000
--------------------------------------------
accum    MmIiEeAa NnJjFfBb PpKkGgCc QqLlHhDd
```

Figure 5.3: The sixteen consecutive tag entries, two bits apiece, are represented by pairs of alphabetic characters. The uppercase letter is the first bit in the tag, and the lowercase letter is the second bit. The zero bits are the "meaningless" ones that need to be eliminated by packing. Each group of four entries is handled as a machine word.

for the dictionary index entries. The indices are four bits apiece, each stored in one entry of an array of bytes. Only two new entries are accumulated with each XOR operation, as compared to the four new entries per XOR operation achieved for tags.

Because the low-bit patterns generated by a partial match are ten bits apiece, no real parallel packing can occur. These entries are combined in the straightforward manner. For efficiency, two bits per packed word of low-bit patterns are left as meaningless bits. Also note that full, 32-bit pattern generated by misses can be written directly to the output during the modeling phase, as no packing can be performed on those item.

Space requirements for fast packing. While this approach to packing requires extra space for the four arrays, the space required by these arrays is approximately 8 kilobytes—the equivalent of two virtual memory pages. The speed increase provided by this approach is significant, as the packing and unpacking become less than 10% of the total compression and decompression costs. The total space required for WK algorithms is still significantly smaller than for LZ compressors.

An oddity in Pentium execution times. In comparing the execution times for our different WK implementations and packing schemes, we used the Pentium cycle counter as one of our timing mechanisms. For what seem to us inscrutable reasons, the execution time could more than double based on the placement and order of lines within the source code of the compressors. Specifically, the modeling/encoding phase was a distinct block of code, and the packing phase was another. If we executed only the modeling/encoding phase, that execution would require approximately 25,000 cycles. If we executed only the packing phase, only about 5,000 cycles were needed. If we executed both in sequence, 150,000 cycles were required. We found that by rewriting these two block of codes, this odd effect disappeared.

5.3 Experimental Evaluation

The first and most simple kind of compressed cache would use a single compression algorithm on all pages. It is not *a priori* obvious which compression algorithm would be the best one to use on average. It is also not clear how the compressibility of pages varies with the program, the input, and the amount of memory allocated for a program's execution.

In order to answer some of these questions, we have collected statistics about the performance of four algorithms, all designed for speed. We will see that the mean compressibility of pages can vary significantly from one program to another. We will also see that the compressibility of pages does not vary drastically if we examine the paging traffic of different memory sizes. Further, we find that the standard deviation for the compressibility of pages can be hundreds of bytes; that is, for the same program execution, there is significant variance in how well pages compresses.

5.3.1 Experimental Design

The compression algorithms evaluated. We selected four compression algorithms. All four are designed for speed, and seemed suitable candidates for compressed caching.

1. **WKdm**: A recency based compressor that operates on machine words and uses a direct-mapped, 16 word dictionary and a fast packing implementation.

2. **WK4x4**: Also a recency based compressor that operates on machine words. Its 16 word dictionary is arranged as a four-way, set associative cache of the most recently seen words, where each set stores four elements. This implementation did not use the fast packing method.

3. **LZO**: Specifically, **LZO1F**, is a carefully coded Lempel-Ziv implementation designed to be fast, particularly on decompression tasks. It is well suited to

compressing small blocks of data, using small codes when the dictionary is small. While all compressors we study are written in C, this one also has a speed-optimized implementation for the Intel x86 instruction set.

4. **LZRW1**: Another fast Lempel-Ziv implementation. This algorithm was used by Douglis in [Dou93], as it was the fastest LZ at that time. While it does not perform as well as LZO, we wanted to demonstrate that even this algorithm provides good compression and low execution time. We will later see that with current hardware, this algorithm could be used to support a beneficial compressed caching system.

The input corpus. In order to obtain meaningful statistics, the experimental input data must be representative of the data that would need to be managed by a compressed cache. In order to obtain representative input, we collected reference traces—specifically, *page image traces*—that contained not only the sequence of pages numbers used, but also copies of the page images themselves. It is important to note that only data pages were traced, not code (a.k.a. text) pages. Code compression has been addressed by others, and an efficient 50% reduction should not be difficult to achieve with a similar, domain-specific compressor. This topic is addressed further in Section 6.6.

By using these page image traces as input to an LRU simulator, we were able to capture the paging traffic, including the contents of the pages, for an LRU memory of any size. We describe the process of trace gathering more specifically in Section 6.3.1.

The paging traffic of some LRU memory is a very good approximation of the input that a compression algorithm would see for a compressed caching system with an uncompressed cache of the the LRU memory's size. That is, by measuring the performance of our compression algorithms by using actual paging traffic, gathered from real program executions, we obtain a good approximation of the compression and decompression costs of a compressed cache.

The importance of representative input. Other studies of compression algorithms for in-memory data have not used paging traffic as input. Other work in this area, discussed in more detail below in Section 5.4, has been based on sampling the contents of memory during program execution. In [KGJ98], snapshots of the "working set"[2] are taken for a number of applications. In another study [Riz97], the swap space of different machines is sampled as input to compression algorithms for evaluations.

Neither of these forms of input represents the workload that would be presented to a compression algorithm used for compressed caching. Any sample of the address space will fail to capture information about the frequency with which some pages will be moved into and out of an uncompressed cache. Some pages may never be evicted from any reasonably sized uncompressed cache, while other may be paged in and out many times. The techniques used in these other studies would give equal weight to both kinds of pages.

Our earliest evaluations of our own WKfa implementation were based on "core" files—the whole of a program's executing memory, deposited into a file. Based on this input, we believed that WKfa would often reduce pages to approximately 33% of their original size. Only by using paging traffic did we discover that a reduction to 50% of the original size was more realistic. Choosing inappropriate input can greatly influence the results of this kind of study.

[2] It is not clear, from the description provided in the paper, whether the working set is the set of resident pages, the set of allocated pages, or some other subset of the program's pages.

5.3.2 Results

We measured the how much each compression algorithm could reduce each page. We also measured how quickly the compression and decompression algorithms executed per page. The measurements were taken for the paging traffic of a wide range of memory sizes—from a memory so small that the amount of paging would be overwhelming, to a memory so large that almost no paging occurs.

We will see that, in general, the compressibility of traffic at different memory sizes does not vary greatly. However, there are a number of interesting differences across programs, memory sizes, and compression algorithms. These differences can be used to guide the development of better compression algorithms.

From the results shown here, we cannot see the variance of compressibility at different points in a program's execution. We do expect that multiphasic programs may reference different kinds of data in each phase, and so the compressibility may vary with time. However, these results indicate that for real programs, data is never incompressible on average. The ability to adapt to changes in compressibility will be addressed in Section 6.4.

Reduction Performance

In Figures 5.4 to 5.19, we see the means and standard deviations for the sizes of compressed pages for each program in the test suite. Along the x-axis varies the memory size. On the y-axis is the mean or standard deviation of the compressed size for the paging traffic for an LRU memory of the given size. On each plot there are four curves—one for each compression algorithm.

Common features. The most gross feature of these curves, as a group, is that there is generally little change in mean or standard deviation as memory size changes. While there are patterns for the manner in which these values change for each

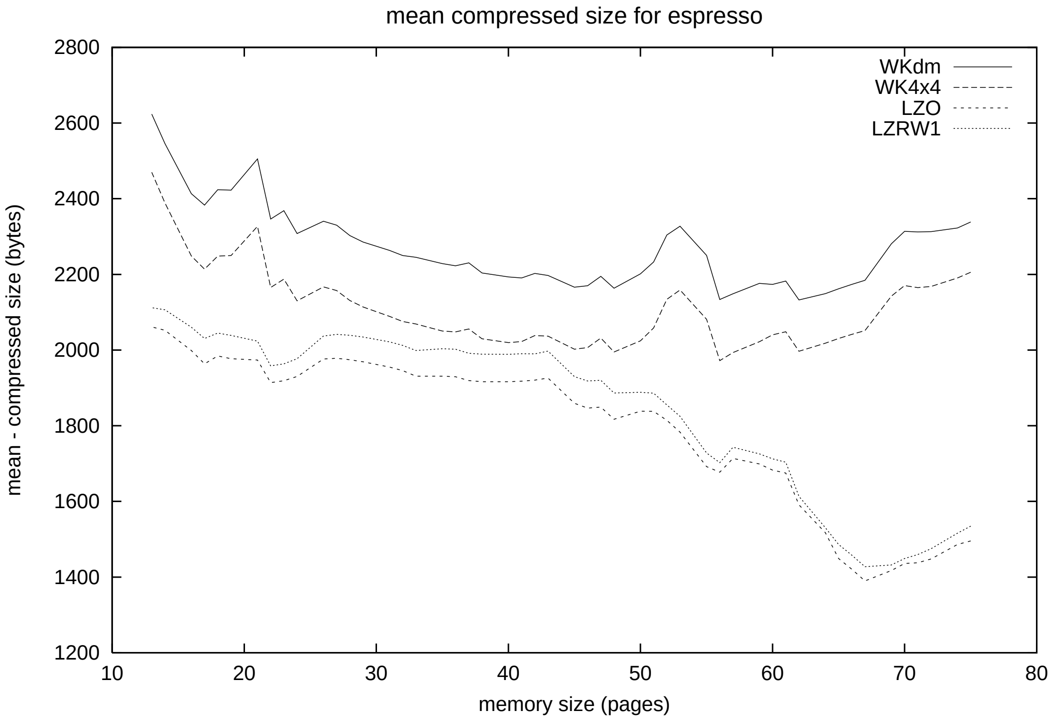

Figure 5.4: The means for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **espresso**.

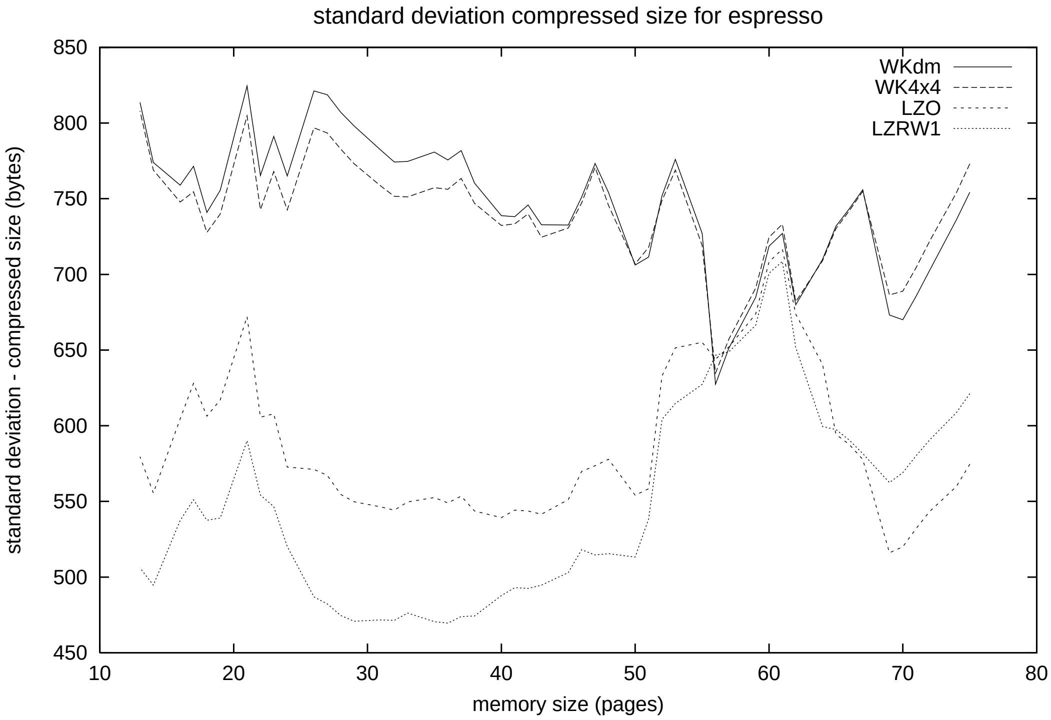

Figure 5.5: The standard deviations for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **espresso**.

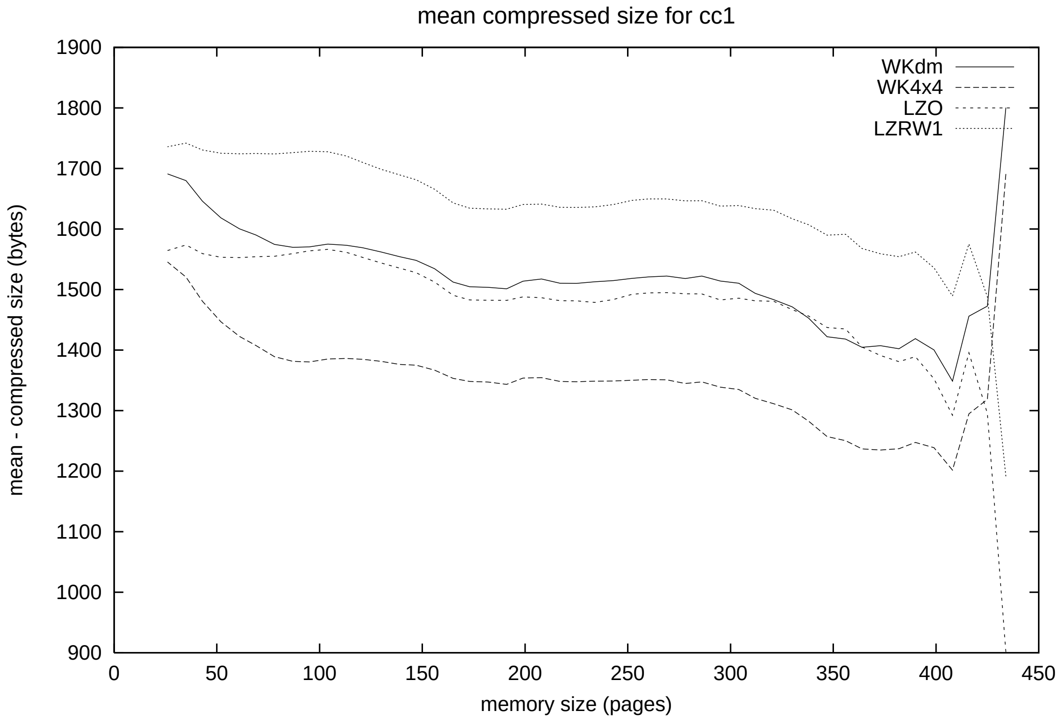

Figure 5.6: The means for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **cc1**.

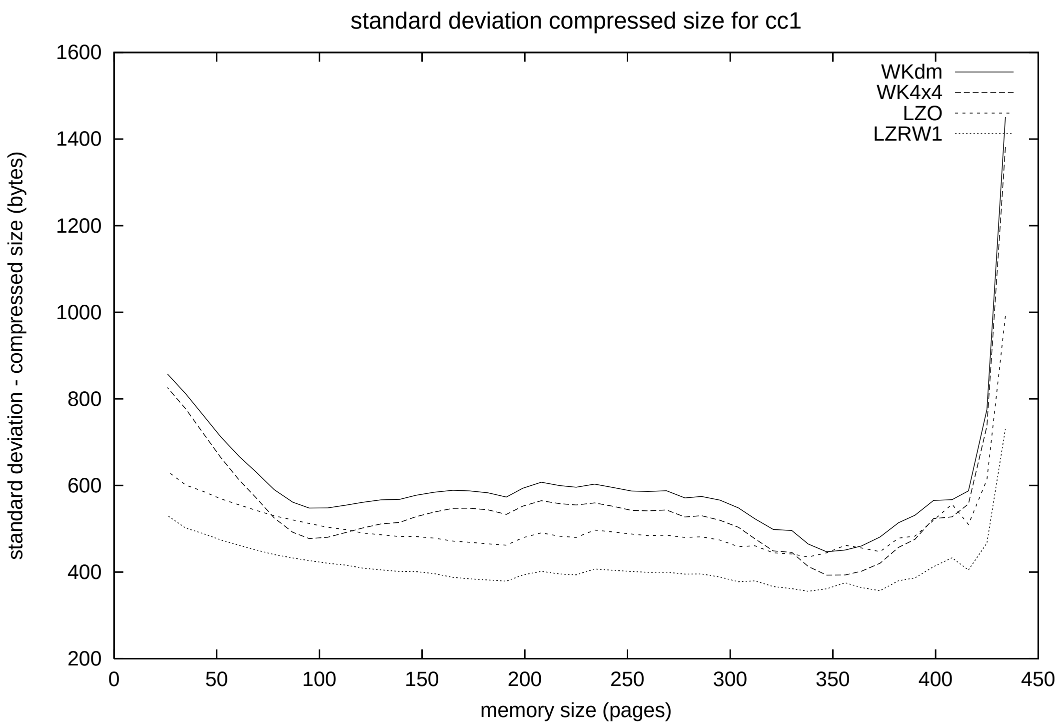

Figure 5.7: The standard deviations for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **cc1**.

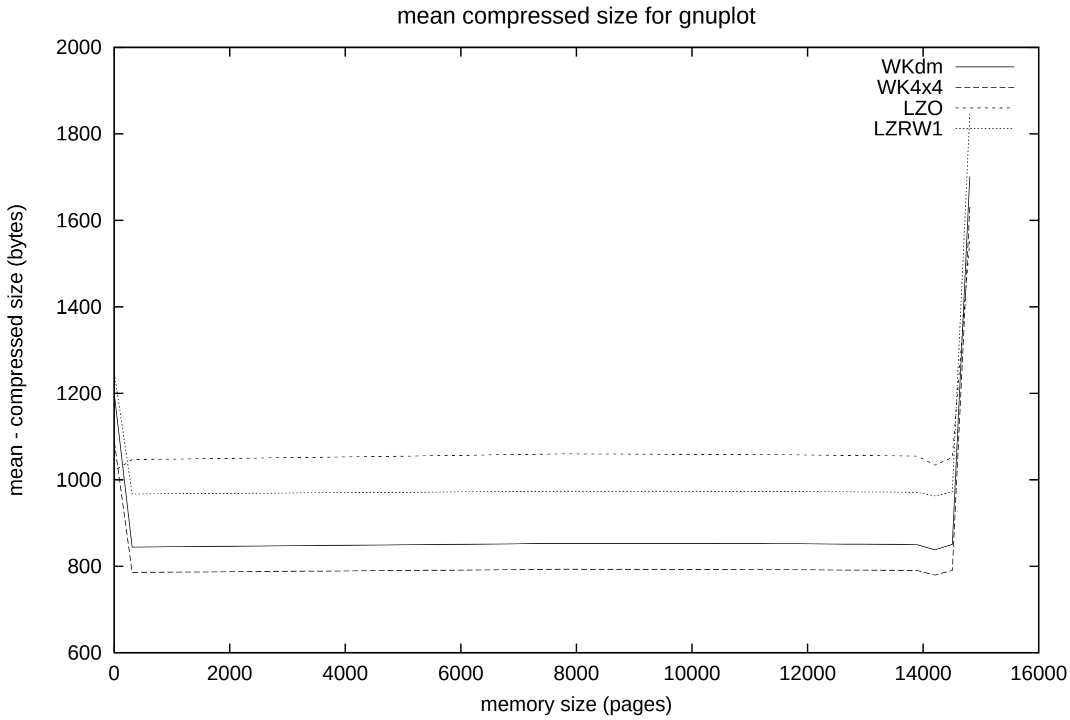

Figure 5.8: The means for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **gnuplot**.

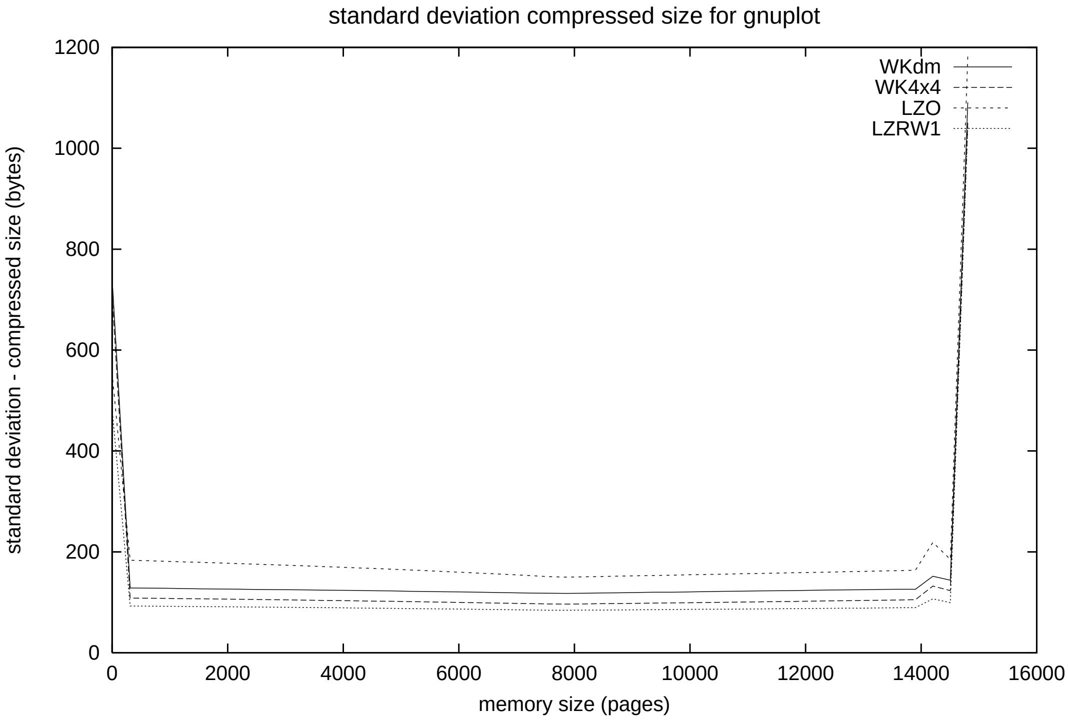

Figure 5.9: The standard deviations for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **gnuplot**.

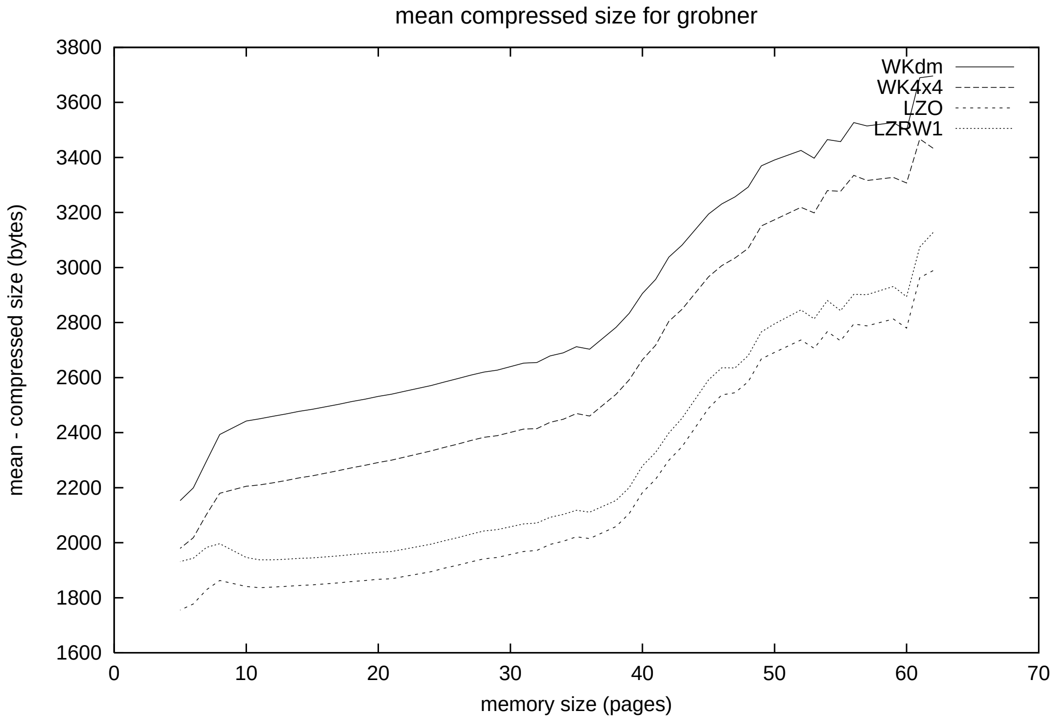

Figure 5.10: The means for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **grobner**.

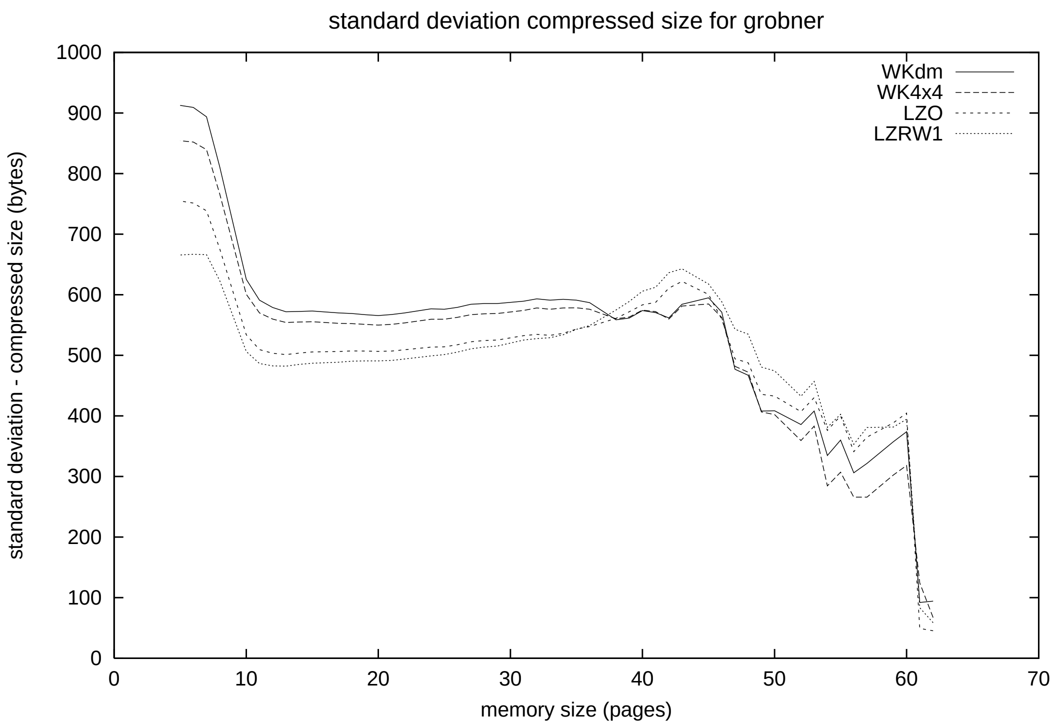

Figure 5.11: The standard deviations for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **grobner**.

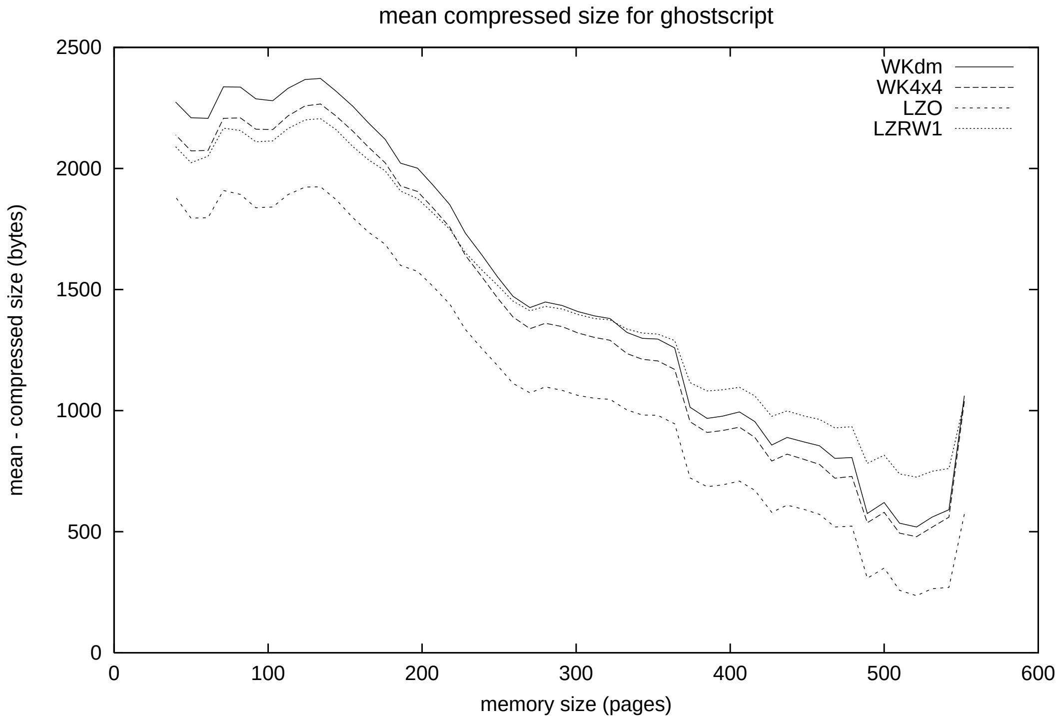

Figure 5.12: The means for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **Ghostscript**.

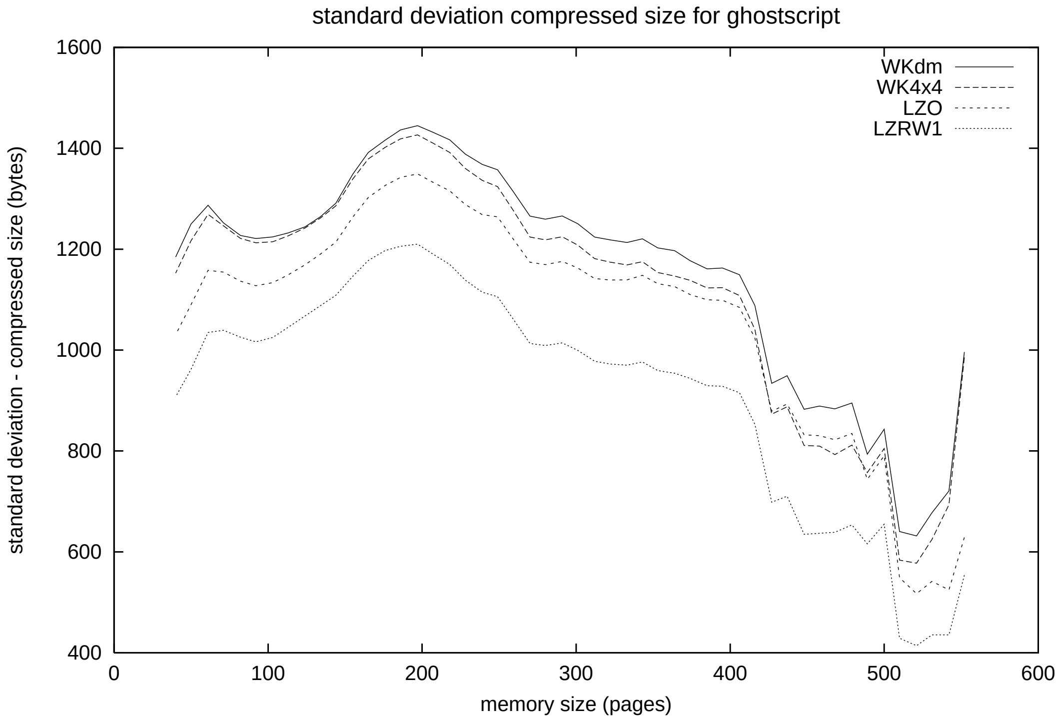

Figure 5.13: The standard deviations for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **Ghostscript**.

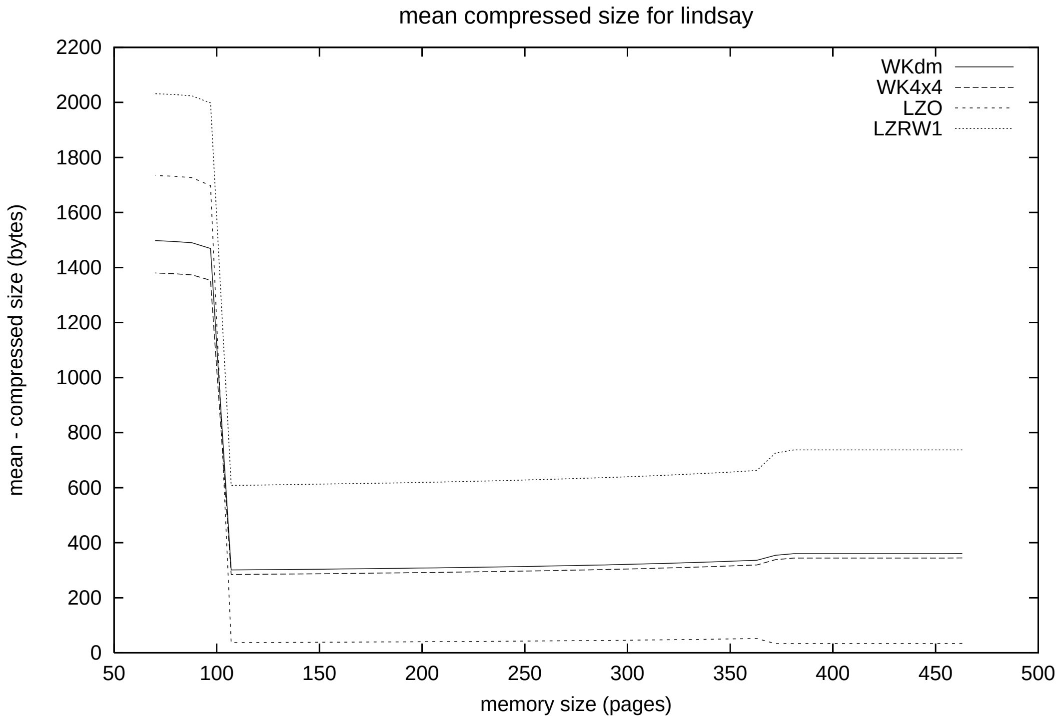

Figure 5.14: The means for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **lindsay**.

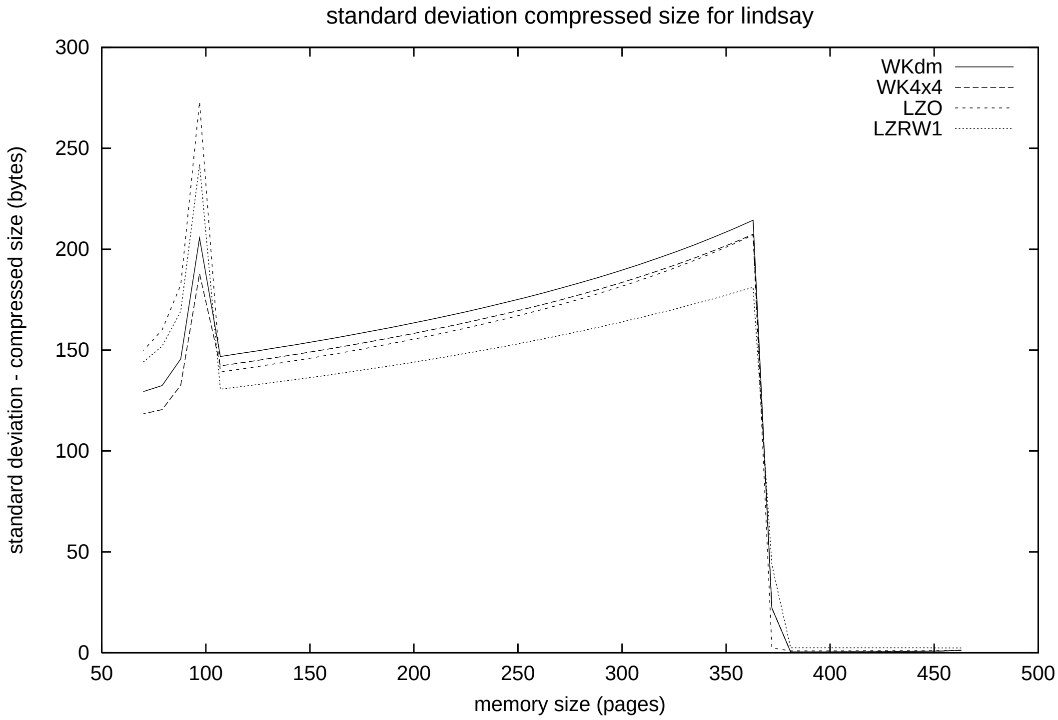

Figure 5.15: The standard deviations for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **lindsay**.

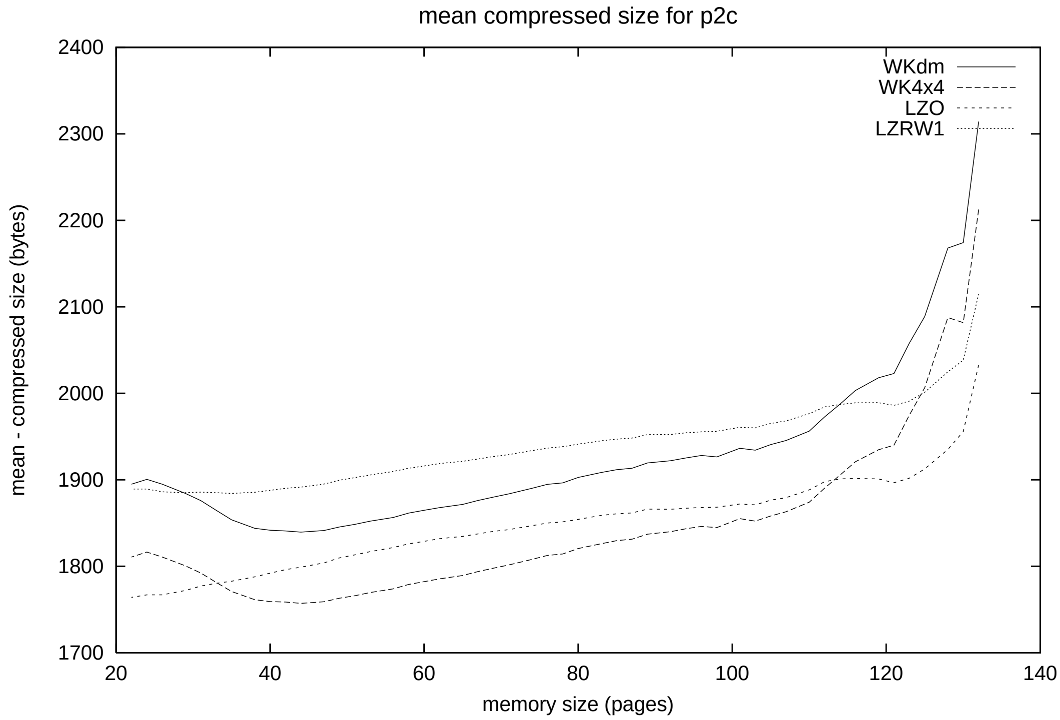

Figure 5.16: The means for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **p2c**.

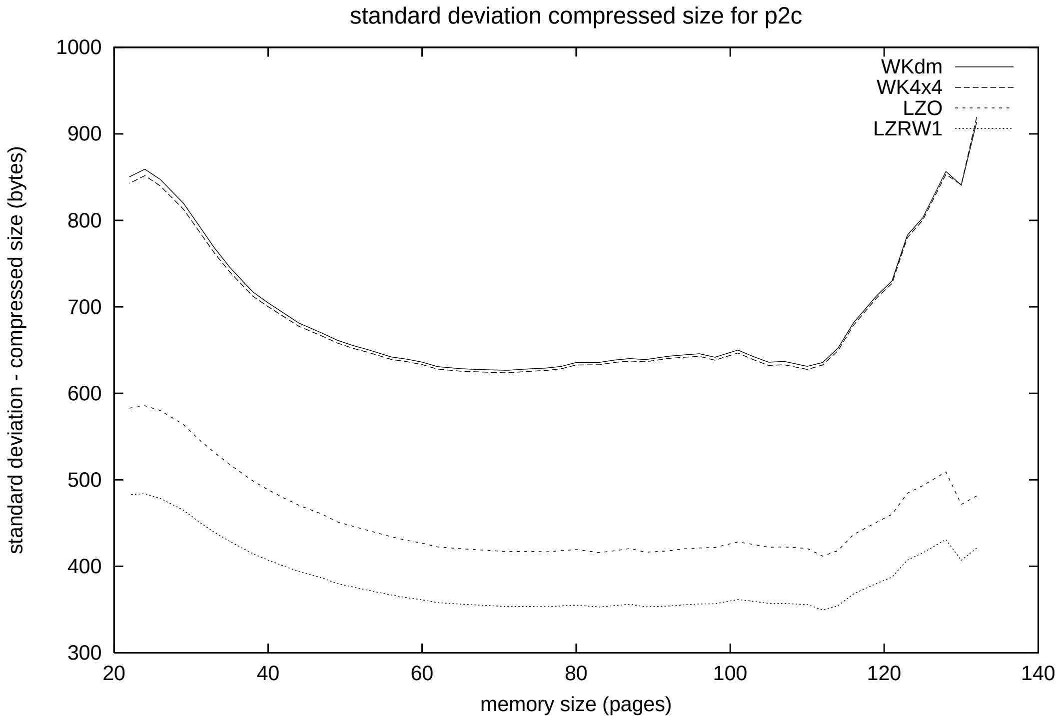

Figure 5.17: The standard deviations for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **p2c**.

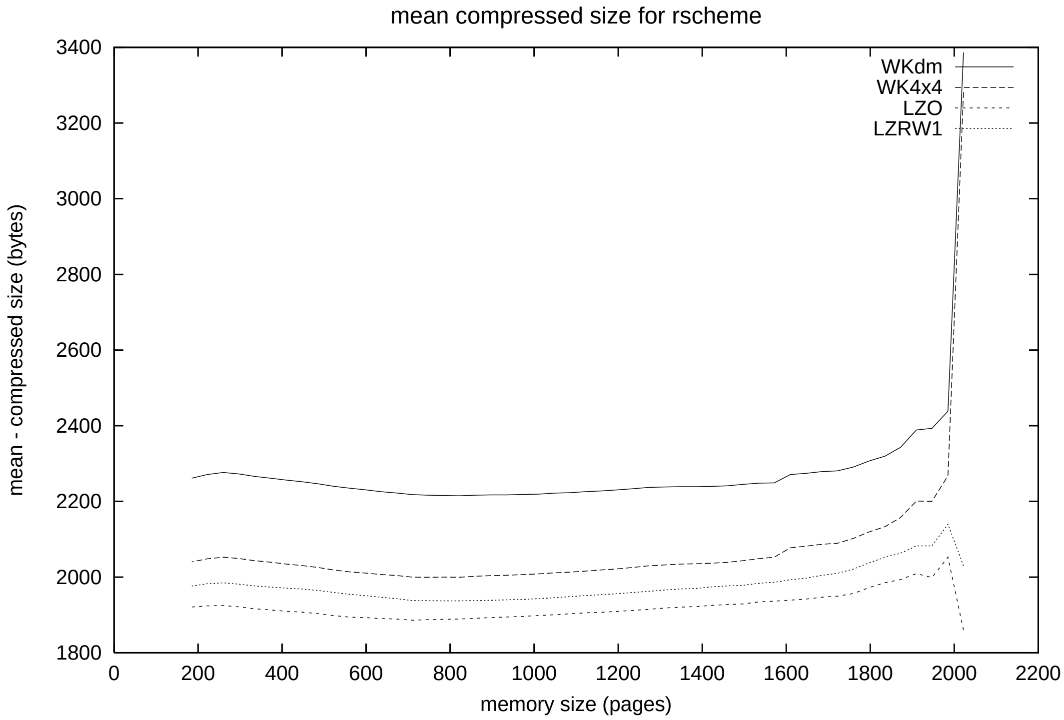

Figure 5.18: The means for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **RScheme**.

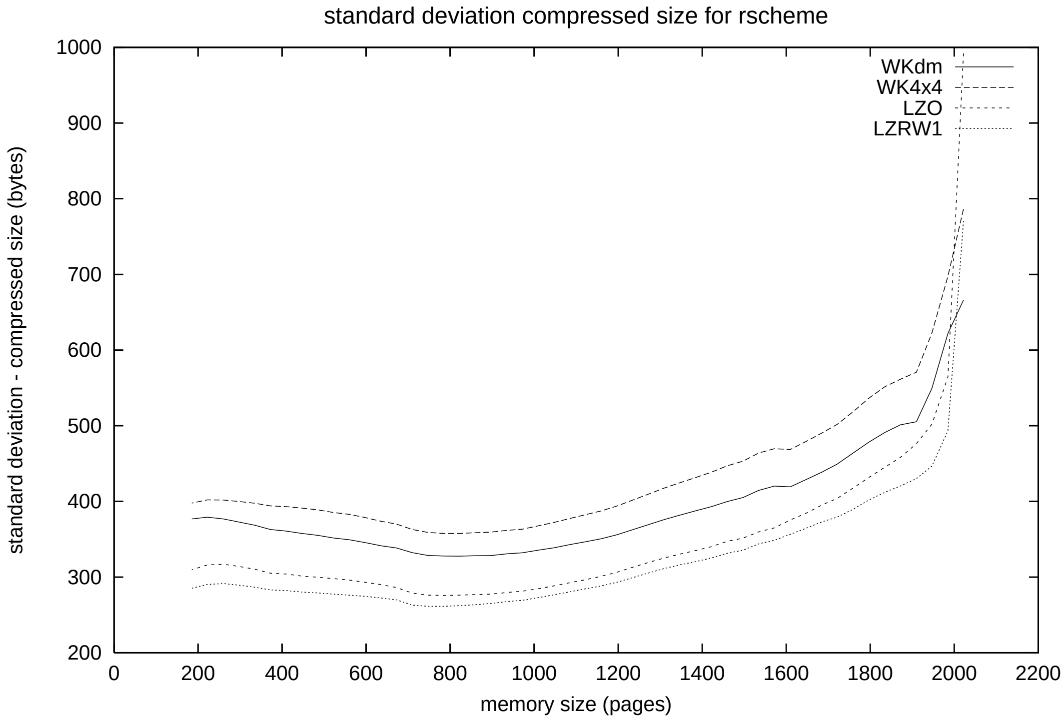

Figure 5.19: The standard deviations for the compressed sizes of the paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **RScheme**.

program, over all programs, pages are roughly compressible to 50% of their original size, and the deviation is only by a few hundred bytes. There are exceptions for very small or very large memories, where the compressibility can be much better or worse. However, such extreme memory sizes are not likely to be used for an uncompressed cache.

Another feature common to all of these plots is that the means and standard deviations of compressibility change identically across programs and memory sizes, irrespective of compression algorithm. We will note the exceptions to this pattern in a moment, but it is interesting to see that both LZ and WK approaches to modeling find enough of the same kinds of regularities that their compressibility is almost identically affected by changes.

One notable exception is for *espresso*, shown in Figure 5.4. As memory size varies, the changes in compressibility for the WK algorithms is fundamentally different from that of the LZ algorithms. With the exception of the traffic incurred around 40 to 45 pages, where the performance is very similar, the shapes of the curves for WK and LZ algorithms are dissimilar. Further, at larger memory sizes, the LZ algorithms compress significantly better. By modeling the input as bytes, instead of words, and by keeping a more extensive dictionary, the LZ compressors can capture regularities that the WK algorithms cannot. We will see later that capturing these regularities comes at the expense of slower LZ execution.

Differences between algorithms. Some of these algorithms compress the data better than others. Across all of the programs, WK4x4 provides better compression than WKdm, and LZO provides better compression than LZRW1. For **cc1**, **gnuplot**, and **p2c**, the WK algorithms provide slightly better compression, while the LZ algorithms are superior for the other five programs. The effectiveness of modeling depends on the program and the data that it stores.

It is interesting to see that in most cases, the relative rank of algorithms doesn't change as memory size changes. The data stored by a program is sufficiently similar on all of its pages that the best choice of algorithm for the tightest compression will not change. There are, however, exceptions:

- For **cc1** and **Ghostscript**, LZRW1 and WKdm change in their relative rank, depending on memory size. However, their changes in performance are similar; that is, the shapes of their curves are the same.
- For **gnuplot**, at the smallest memory sizes, compressibility seems to be very different. That is likely because the program uses about 10 pages of data very, very frequently, and the rest of the approximately 14,000 pages as part of a large loop over simple, integer data. Thus, the traffic at very small memory sizes is quite different in content than at other memory sizes.
- Similarly, with **lindsay**, the compressibility and relative rank of the compression algorithms changes for memories smaller than 100 pages. Again, about 100 pages of data are referenced quite often, while the remaining pages are part of a loop. For the memories smaller than about 100 pages, the WK algorithms perform best. However, for traffic at larger memory sizes, LZO performs best, while LZRW1 continues to provide the least compression. Note that the compressibility for all algorithms at memory sizes above 100 pages is excellent, with reduced representations that are around 10% of the original size. The more sophisticated approach to encoding used by LZO likely accounts for its superior performance.

Differences between programs. There is significant variation in the compressibility of data from each program. While the average reduction is roughly to 50% of the original size, the average for a particular program may be quite different. We see here that most programs have pages that can be compressed to less than 50%, but the few others have poorly compressible pages that raise the mean.

- **Poorly compressible data**
  - **grobner:** At small memory sizes, its pages can be compressed to roughly 45% to 50% of their original size. However, at sizes near the footprint of the program, pages are reduced only to 75% of their size with LZO, and to more than 90% of their size with WKdm. It is likely that grobner stores either byte-based or floating point values that the WK algorithms do not model well, and even LZ algorithms model only modestly better.
  - **RScheme:** The compressibility of this program's pages varies greatly from one compression algorithm to another. For WKdm, pages compress only to 55% to 60% of their original size. For the other algorithms, the reduction is between about 48% and 52%. The poor compressibility with WKdm is undesirable, as we will see it is the fastest algorithm, and as such it provides the best compressed cache performance overall.
- **Compressible data**
  - **espresso:** This program's pages compress only to about 56% and 61% for WK4x4 and WKdm, respectively, of their original size for small physical memories. However, the pages then compress at roughly 50% and 55% for these two algorithms at larger memory sizes. The LZ algorithms perform much better, reducing pages to 50% at larger memory sizes, to approximately 35% of their original size.
  - **cc1:** For this compiler, pages compress to about 35% to 41% of their original size, depending on the algorithm used. Notice that the compressibility doesn't change much with memory size.
  - **p2c:** These pages compress to approximately 45% to 50% of their original 4 Kbytes. As for cc1, the compressibility varies, but not drastically, with

the memory size. Both programs likely build up the same kinds of data structures throughout execution as each procedure is compiled, and so most pages contain the same kind of data.

- **Ghostscript:** The compressibility of this program's pages does vary with memory size. At smaller memory sizes, pages compress to about 50% of their uncompressed size; at larger memory sizes, they compress to 18% for the worst of the algorithms (LZRW1), and to less than 10% for the best (LZO). Such compressibility results imply that in-memory data compression can likely be applied to PostScript printers.

Very compressible data

- **gnuplot:** The data stored by this program is simple—many integers used for creating a scatter plot. It is therefore not surprising that the simple WK compressors provide excellent reduction. Pages for nearly all memory sizes can be reduced to 25% by the LZ algorithms, and to less than 20% for the WK algorithms.
- **lindsay:** This program's pages are the most compressible. At the smallest memory sizes, pages can be reduced to 50% by LZRW1, and 35% for WK4x4. For the majority of memory sizes, where more than roughly 100 pages are available, pages are compressible to less than 10% of their original size for all but LZRW1. The pages likely contain nearly all zeros, and it is the tightness of the encoding scheme used that determines the compressed size for each algorithm. There are other programs like lindsay that store mostly empty pages, like some scientific applications that work on sparse data.

There seems to be no clear correlation between compressibility and footprint size. It is our intuition that some programs with large footprints are tailored to use tight data representations to avoid paging. These representations tend to make programming complicated, and convolute the solution that the program implements. It is desirable that programmers write clearly, using direct and simple data representations, with a focus on the problem and its solution. Compressed caching allows them to do so without writing special, tight data representations to complicate their code.

Relationship between means and standard deviations. We can see, in each of the figures, that the plots of standard deviation across memory sizes is similar to the plots of means. We examine the standard deviation because it is important that it not be too high. If the deviation were exceedingly large, it would be difficult to maintain a compressed cache. The unpredictable size of compressed pages could make it difficult to determine whether or not compressed caching would provide benefit.

Fortunately, not only is in-memory data compressed efficiently and significantly, it is also compressed consistently. For most programs, the standard deviation is only hundreds of bytes—pages would rarely compress to larger than 75% of their original size, or to smaller than 25% of their original size. Only **Ghostscript** exhibited a standard deviation of greater than 1 Kbyte.[3] Given the good compressibility of that program's pages, such high deviation should not ruin the benefits that it could gain through compressed caching.

Variation of memory size seems to have little gross effect on the deviation. For each program, the relative rank among the compression algorithms usually doesn't change, irrespective of memory size.

Curiously, LZRW1 exhibits consistently lower deviation than LZO. Most of that consistency may be due to LZRW1's conservative encoding—on very com-

[3]Note again that there are extreme changes in compressibility at the largest memory sizes. These large variations are more the result of statistics based on the small sample size of paging events at large memory sizes, and less a result of a fundamental change in the compressibility of the pages.

pressible data, LZRW1 achieves only moderate reduction compared to LZO. Consequently, the compressed sizes for LZRW1 are more consistently near the mean. Dissimilarly, WK4x4 exhibits consistently lower deviation than WKdm, most likely because WK4x4's extra computational complexity allows it to achieve reduction on a wider range of inputs. WKdm can more easily fail to model the input well, and may suffer a larger number of pages that compress poorly.

By examining the pairs of mean and standard deviation plots for each program, we can observe surprising correlations between these two statistics, but no obvious relationship across programs. For some programs (**espresso**, **gnuplot**, **Ghostscript**, and **RScheme**), the two seem to be somewhat *directly* related; when the mean is smaller, so too is the standard deviation. That is, the shapes of the curves are roughly similar. For other programs (**cc1**, **grobner**, **lindsay**, **p2c**), no such relationship holds across the entire range of memory sizes.

What *is* consistent about the relationship between mean and standard deviation is that significant changes occur at the same memory sizes. For all of the programs, if the mean drastically increases or decreases at some particular memory size, the standard deviation also changes at that size. Given a change in the paging traffic, pages may become more or less compressible, *and* more or less consistent in their compressibility.

Compression and Decompression Times

The compression and decompression speeds of the four algorithms are consistent across memory sizes and across programs. There is some fluctuation in these times, as the set of operations required for a highly compressible page is different from those of poorly compressible page. However, the variation isn't large, and tends to affect the slower algorithms more than the faster ones.

Because the results are so consistent from program to program, in Fig-

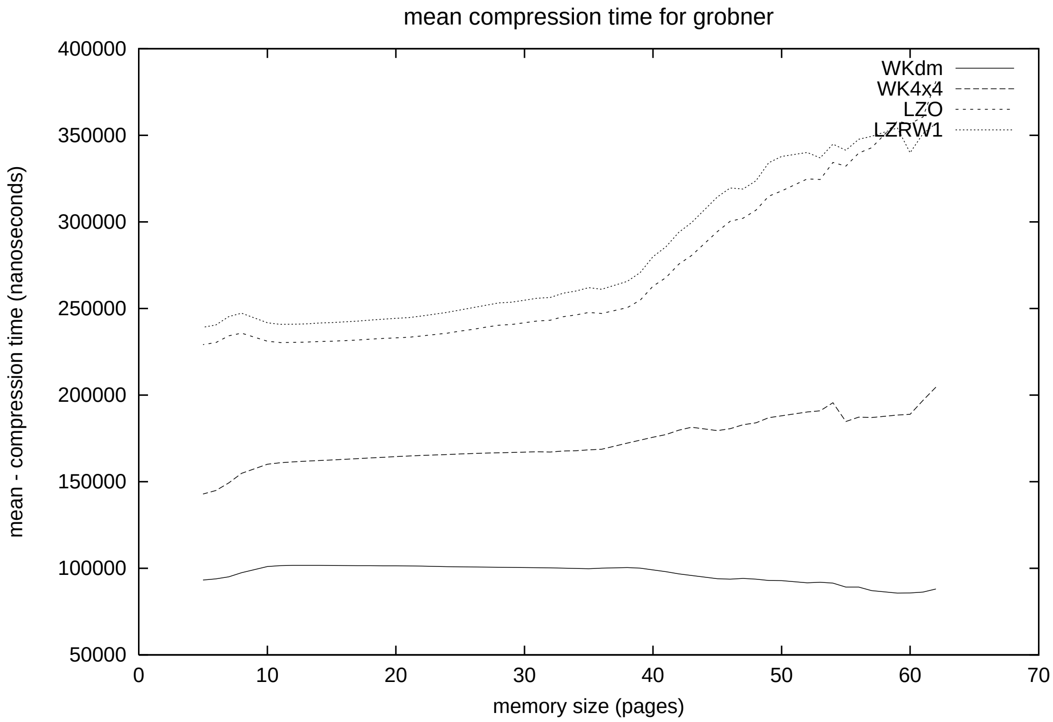

Figure 5.20: The means of the compression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **grobner**.

ures 5.20 to 5.35, we show the mean and standard deviation statistics for the compression and decompression times for only four of the eight programs. These results for the other four programs are similar to those of the four shown. The compression and decompression times were gathered on a **300 MHz UltraSPARC**, as it is representative of current processor speeds.

Mean compression speeds. Because they were designed to compress quickly, the WK algorithms consistently provided the fastest compression. WKdm was easily the fastest algorithm for all programs and at nearly all memory sizes, and WK4x4 was commonly about 50% slower. Both of the LZ algorithms were often twice as

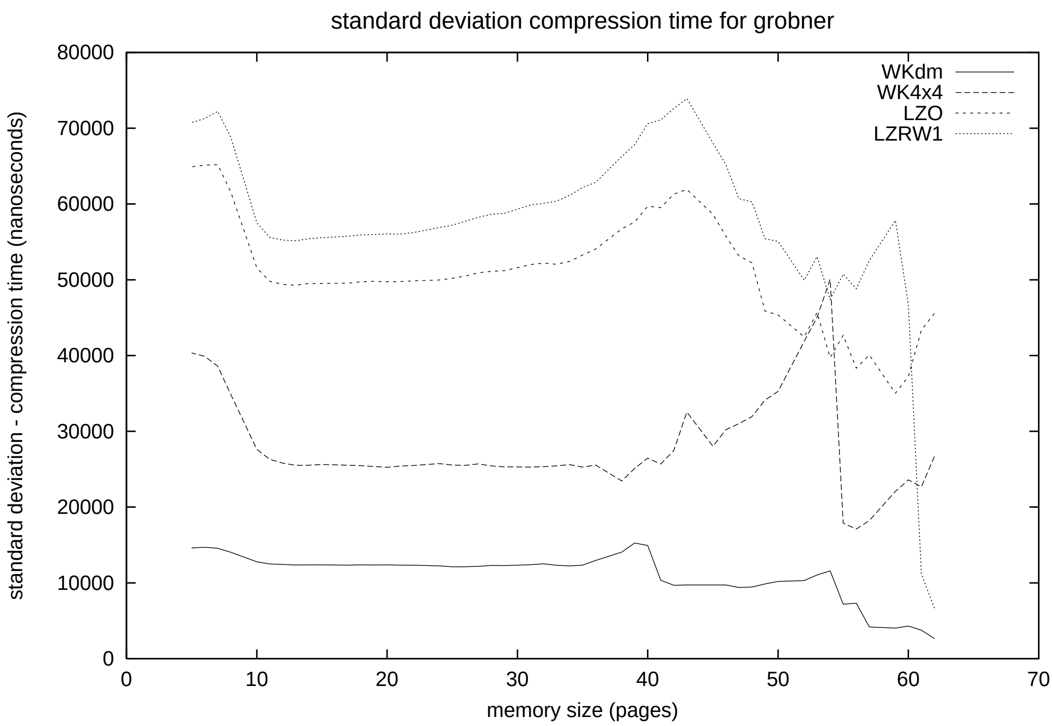

Figure 5.21: The standard deviations of the compression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **grobner**.

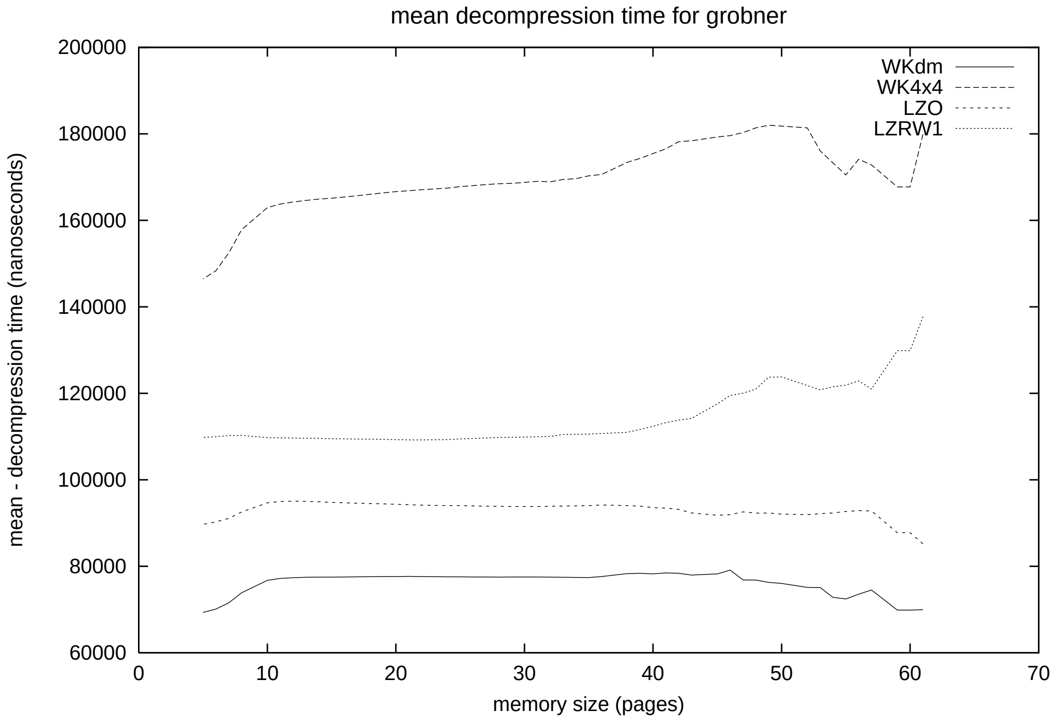

Figure 5.22: The means of the decompression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **grobner**.

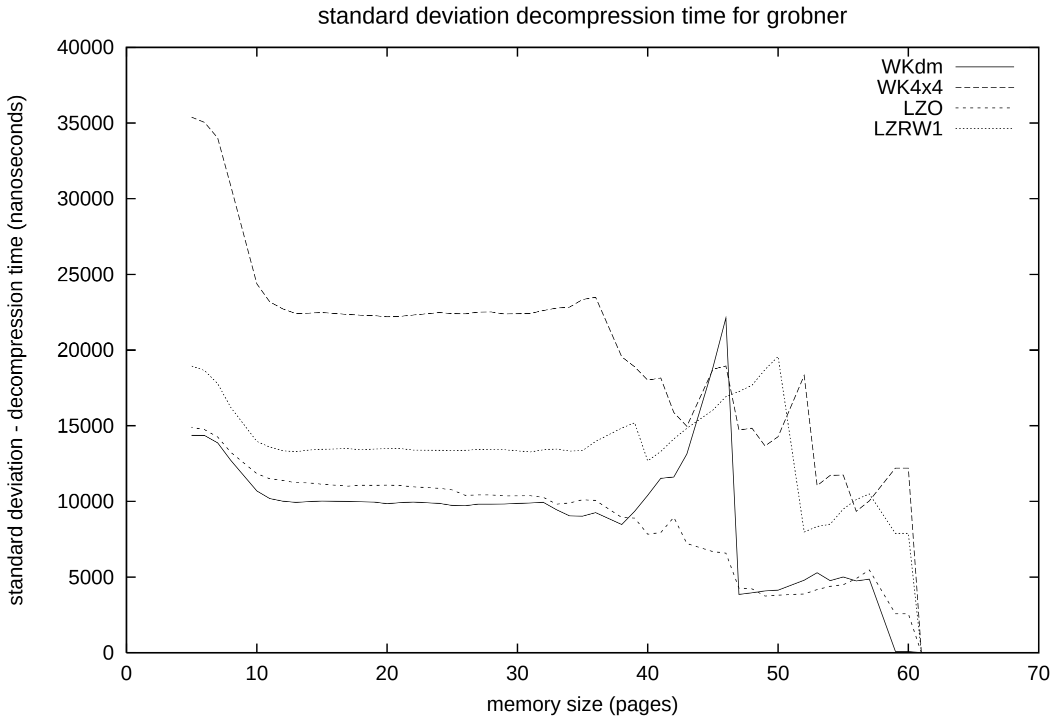

Figure 5.23: The standard deviations of the decompression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **grobner**.

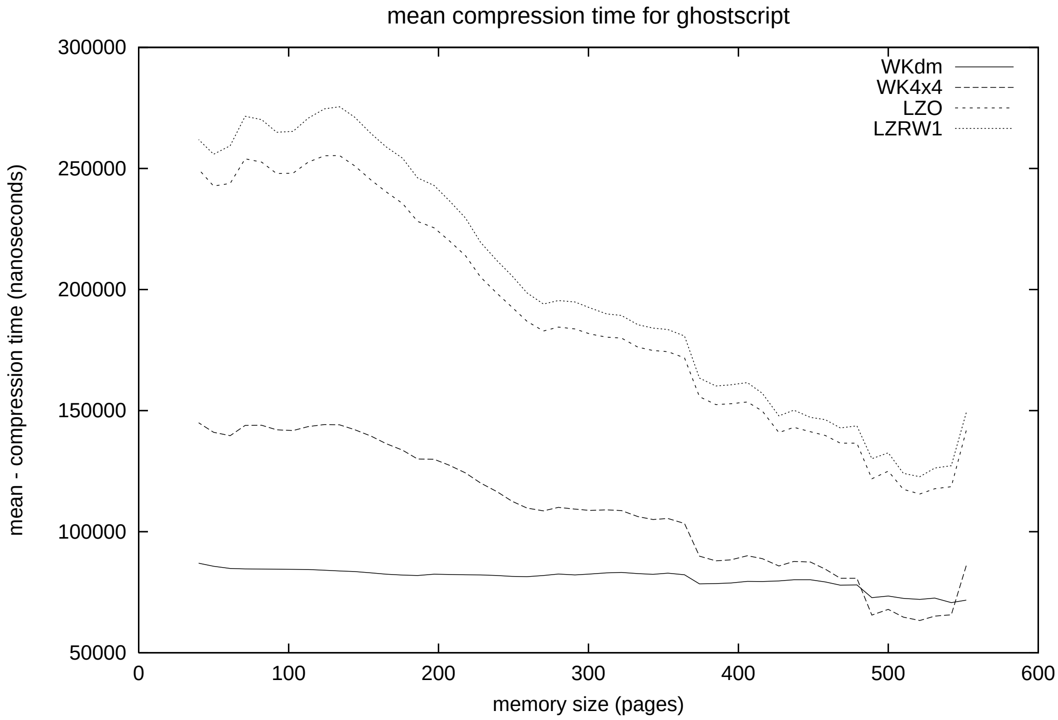

Figure 5.24: The means of the compression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **Ghostscript**.

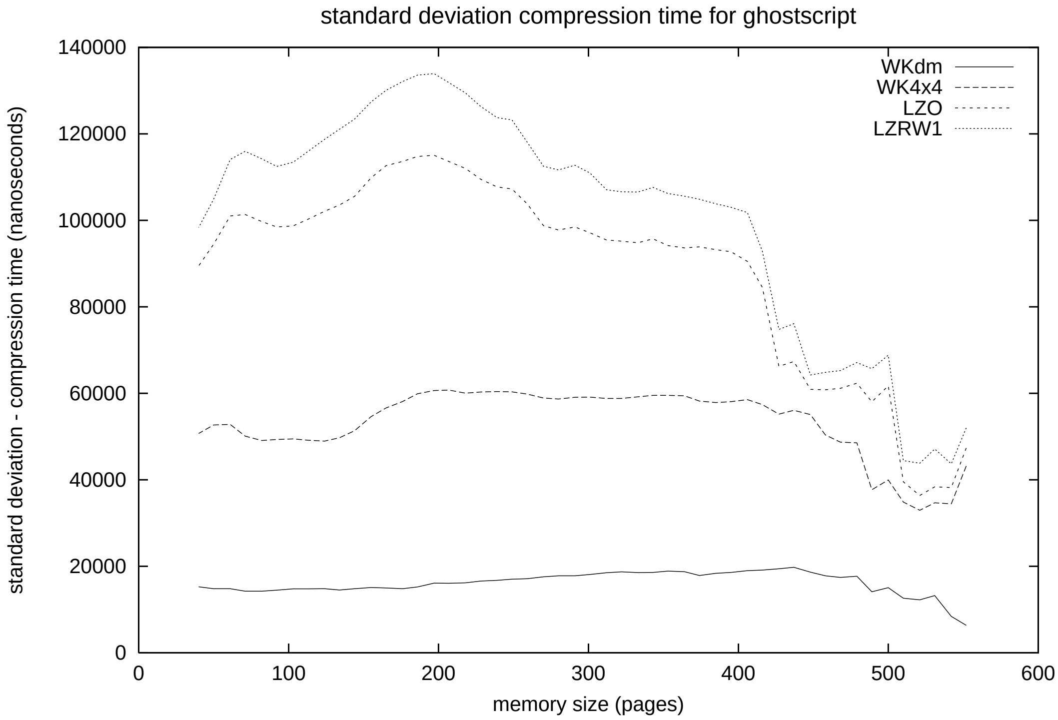

Figure 5.25: The standard deviations of the compression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **Ghostscript**.

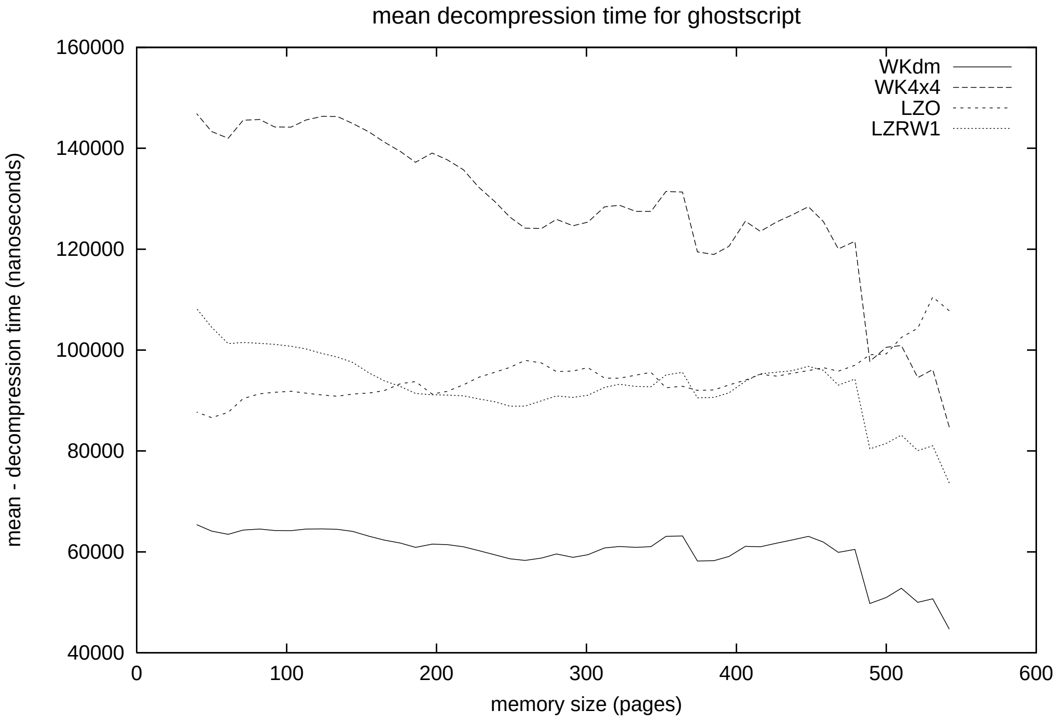

Figure 5.26: The means of the decompression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **Ghostscript**.

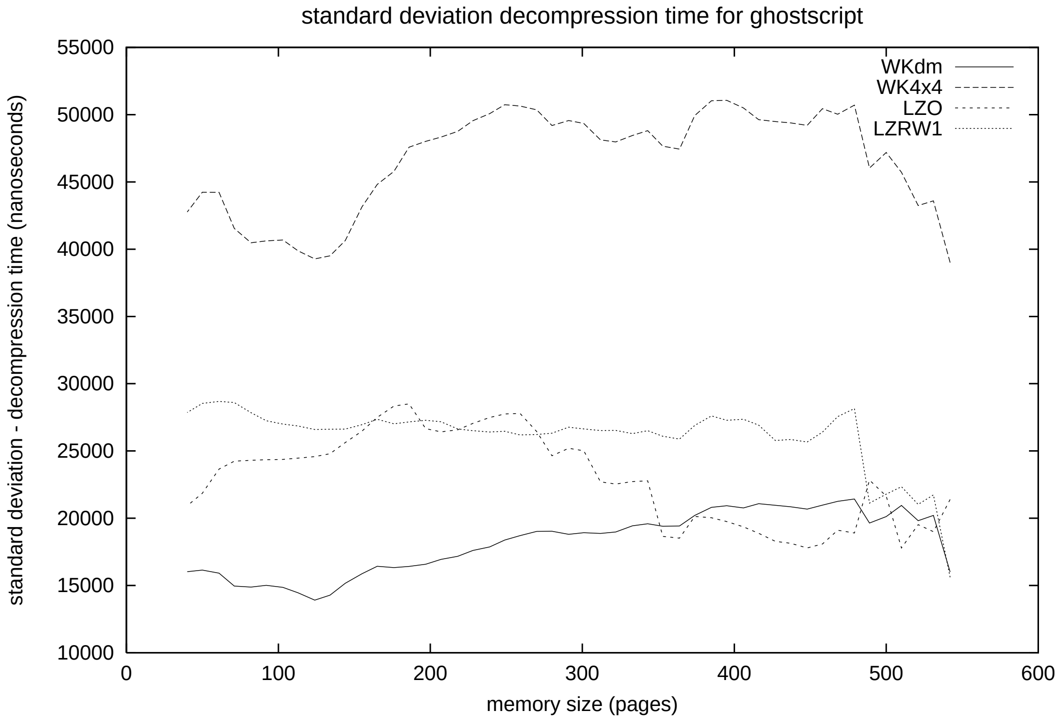

Figure 5.27: The standard deviations of the decompression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **Ghostscript**.

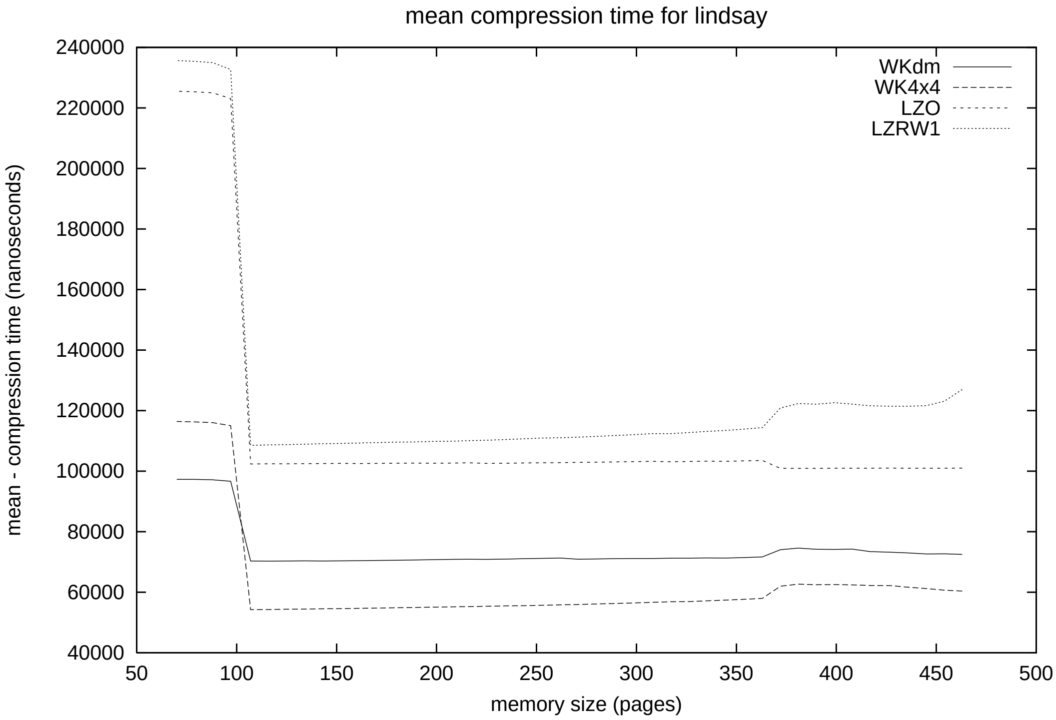

Figure 5.28: The means of the compression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **lindsay**.

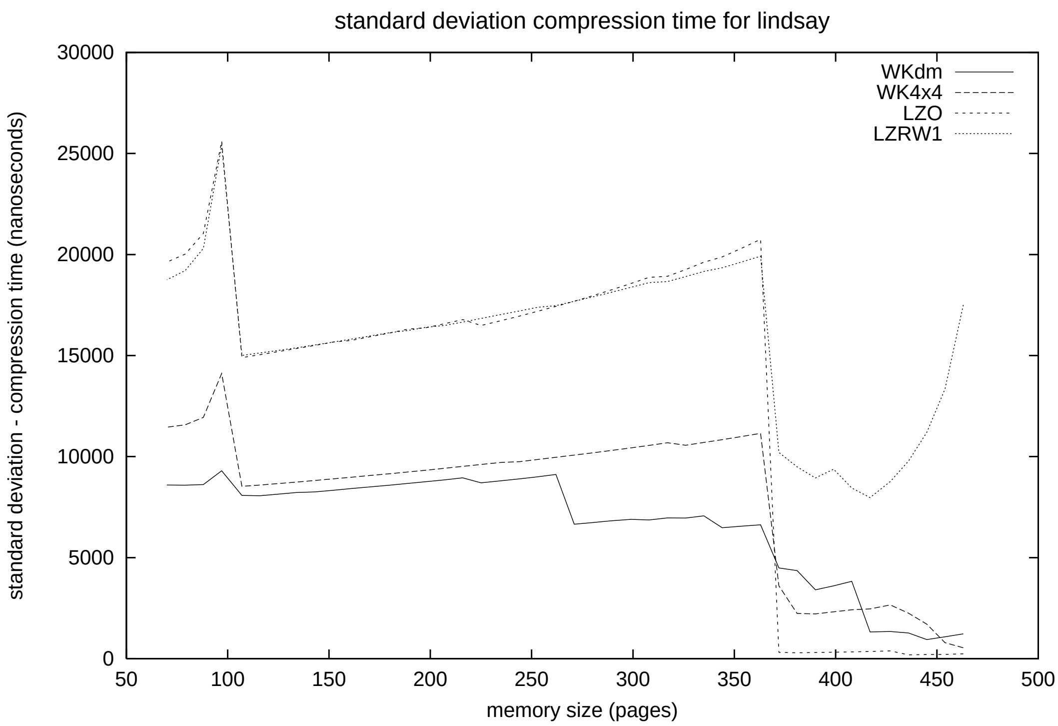

Figure 5.29: The standard deviations of the compression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **lindsay**.

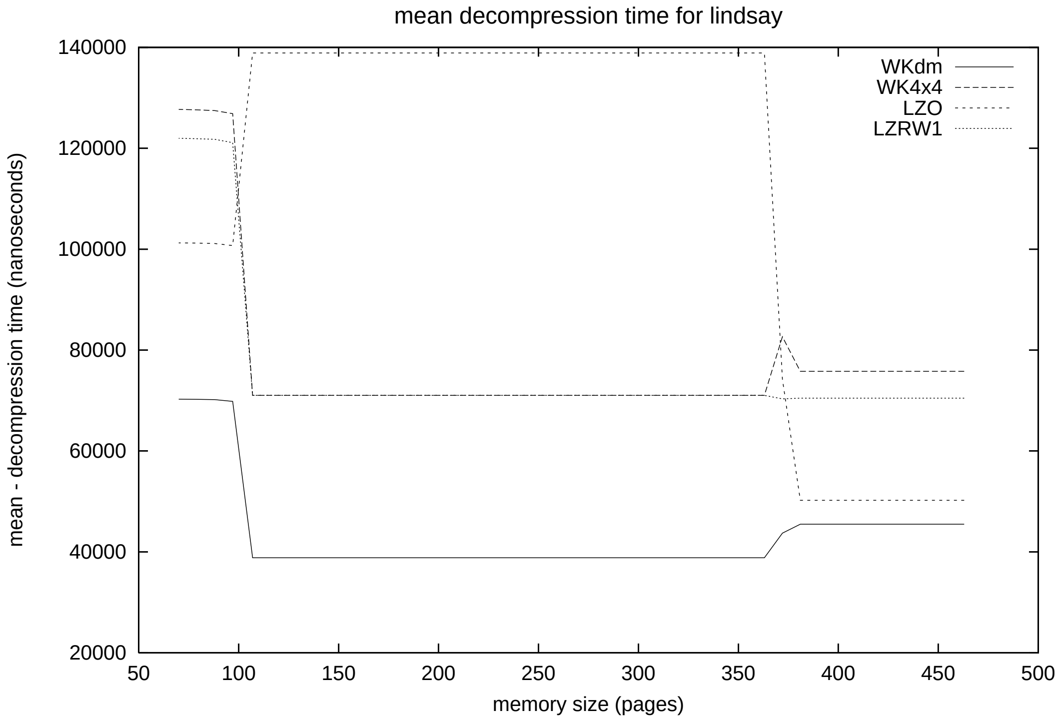

Figure 5.30: The means of the decompression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **lindsay**.

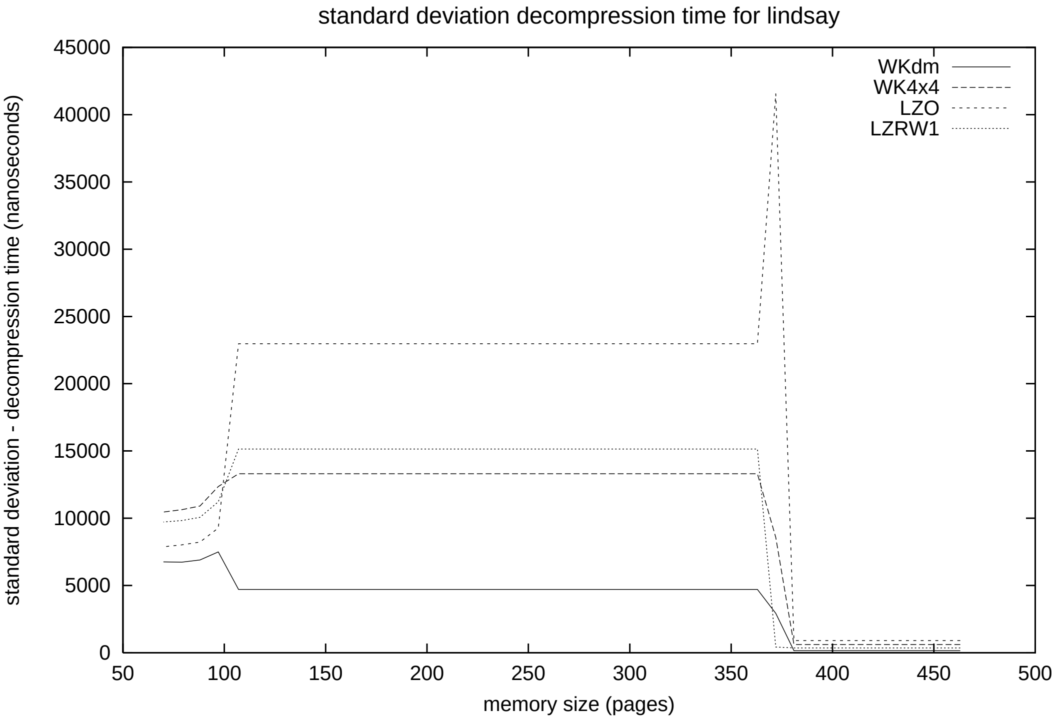

Figure 5.31: The standard deviations of the decompression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **lindsay**.

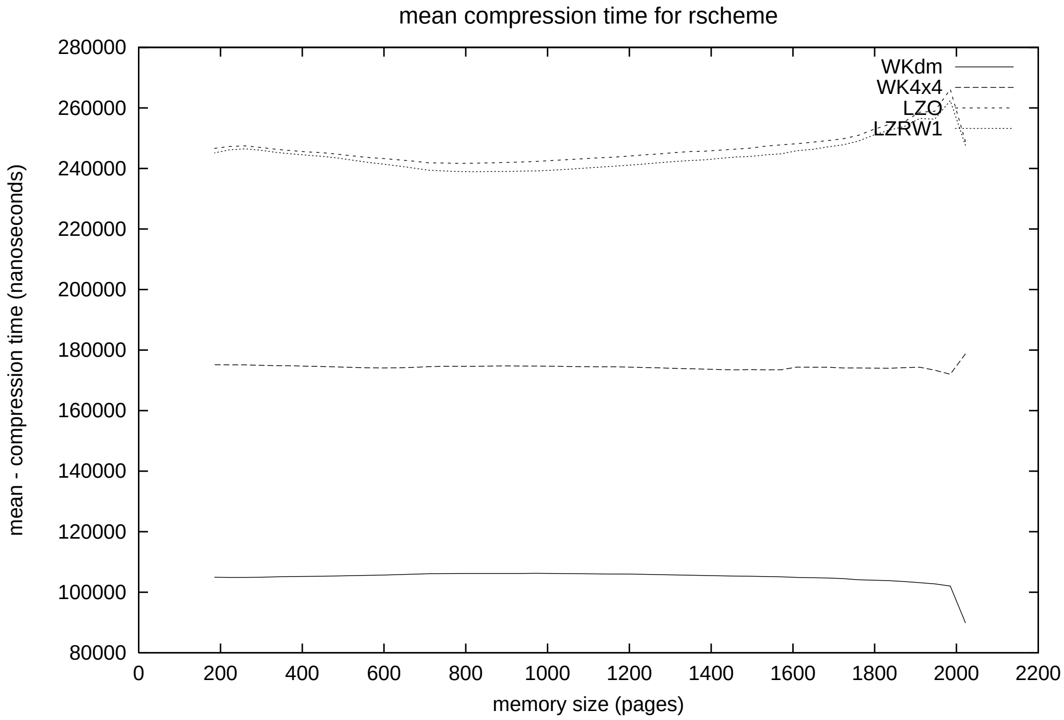

Figure 5.32: The means of the compression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **RScheme**.

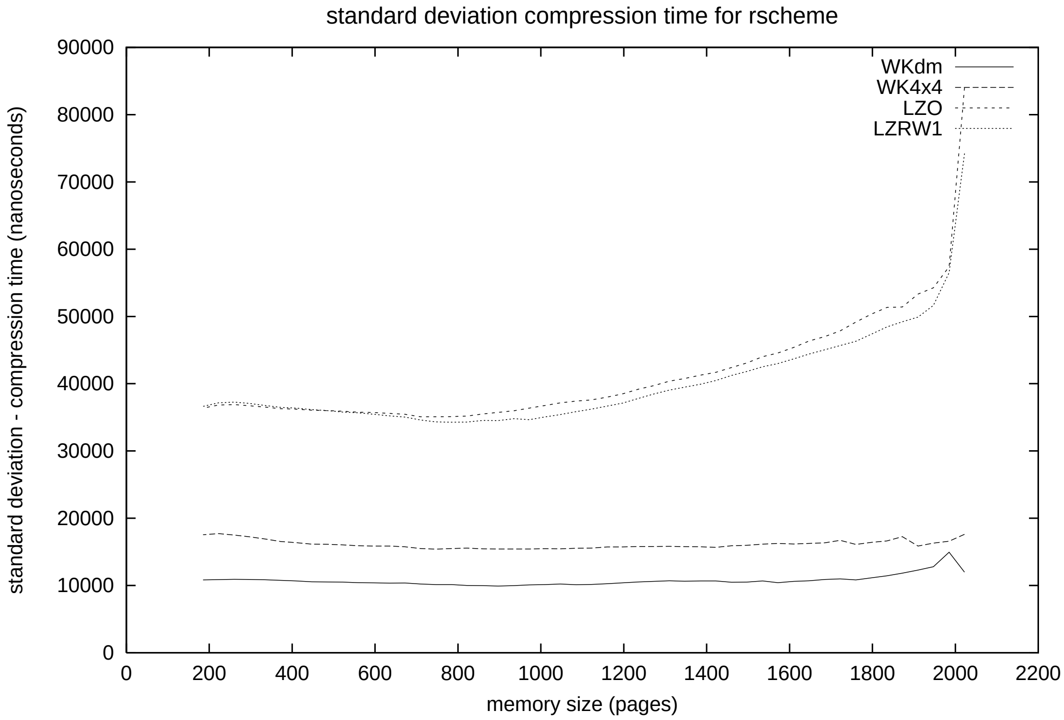

Figure 5.33: The standard deviations of the compression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **RScheme**.

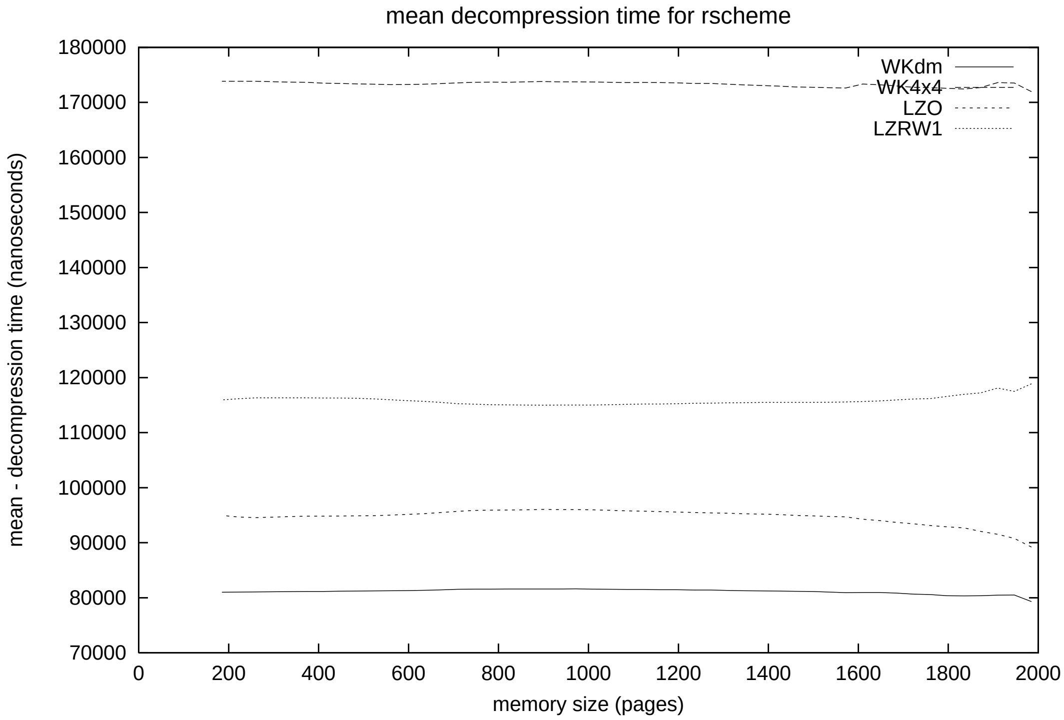

Figure 5.34: The means of the decompression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **RScheme**.

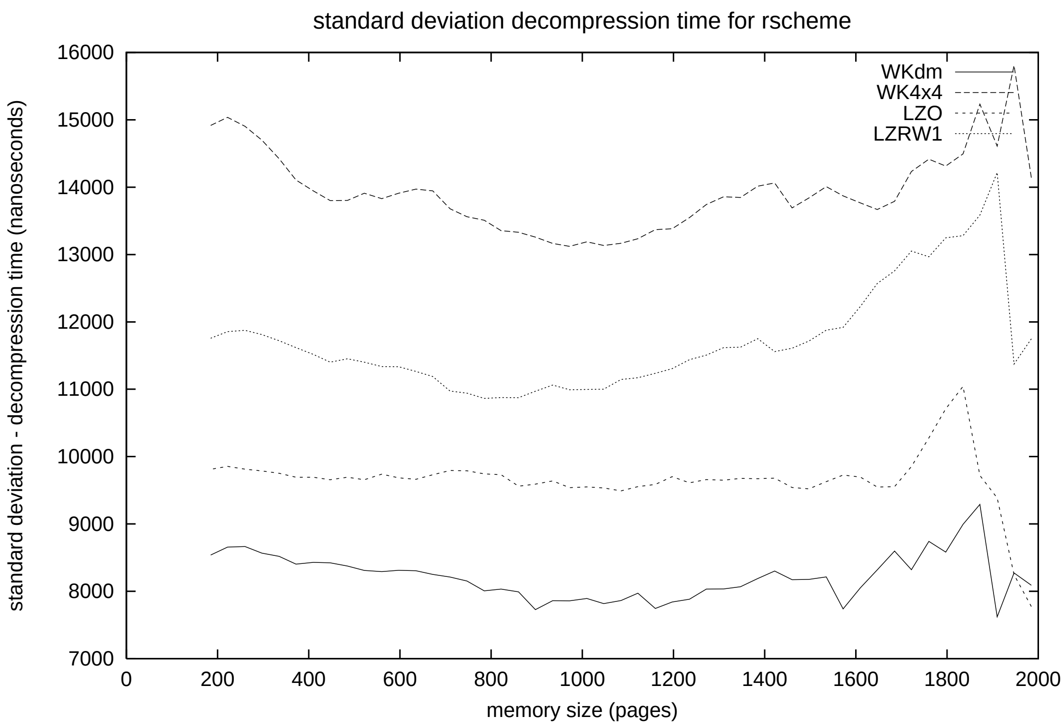

Figure 5.35: The standard deviations of the decompression times for paging traffic at different LRU memory sizes, using 4 Kbyte pages, for **RScheme**.

slow as WKdm. Interestingly, LZRW1 compressed 5% to 10% faster than LZO.

On this 300 MHz processor, WKdm is able to compress a page in approximately 0.1 milliseconds (30,000 processor cycles). WK4x4 is slightly slower, requiring about 0.15 ms (45,000 cycles) for the task. LZRW1 and LZO are significantly slower, compressing each page in a mean time of about 0.24 ms (72,000 cycles). The most extreme case is for **grobner**, shown in Figure 5.20. Although the LZ algorithms achieve much better compression than the WK algorithms at larger memory sizes, that reduction requires more time—0.35 ms (105,000 cycles) to compress each page.

It is important to note, though, that by using an UltraSPARC processor, we are timing the performance of the C implementation of LZO, rather than the hand-written assembly implementation that is available for the Intel x86 architecture. That more careful implementation would make LZO approximately 20% faster. However, that difference would change its relative rank only with LZRW1, as it would still not outperform the WK compressors. Moreover, the other algorithms would likely benefit as well from such a careful assembly implementation, and so the relative performance of the algorithms is more accurately reflected by using C implementations for all of them.

Standard deviations of compression speed. The standard deviations share the same general rankings as the means. The WK algorithms have much lower deviations than the LZ compressors, sometimes by more than a factor of five. However, none of the deviations are particularly high, with the highest begin around 0.06 ms (18,000 cycles).

WKdm, in particular, exhibits extremely low deviation, commonly around 0.01 ms (3,000 cycles). Not only is the compression time very consistent for WKdm, the deviation doesn't change much with the compressibility of the page. WKdm employs such a simple modeling technique that the time required to model a compressible page is not much different from the time needed to model a poorly compressible page.

Exceptions due to very high compressibility. In Figures 5.28 and 5.29 we can see that the compression performance can be qualitatively different if the data is extremely compressible. For the smallest memory sizes, the means and standard deviations for compression times are much like those for other programs. However, for memories larger than around 100 pages, the paging traffic comprises pages with mostly zeros.

As a result, the compression time for each algorithm is lower than for most paging traffic. The LZ algorithms are still slower than the WK algorithms, but not by as much. Both LZO and LZRW1 compress each 4 Kbyte page in approximately 0.11 ms (33,000 cycles)—more than twice as fast as they usually do for other paging traffic.

Interestingly, WK4x4 compressed these pages in less than 0.06 ms (18,000 cycles) each, also a 50% reduction in mean compression time, while WKdm improves only by only about 30% to 0.07 ms (21,000 cycles). In our implementations, WK4x4 checks for the special case of a zero first, while WKdm performs that comparison second, after comparing the input word to a dictionary entry. Thus, WK4x4 can compress a zero in fewer steps than WKdm.

Mean decompression speeds. The most striking feature of the plots of mean decompression speed is the lack of sensitivity to memory size. Decompression seems not to be effected by the contents and compressibility of a page, and performs about the same number of operations in any case.

The ranking of these algorithms changes with respect to decompression times. WKdm is still the fastest algorithm, requiring roughly 0.07 ms (21,000 cycles) to decompress a page. At 0.09 ms (27,000 cycles), LZO is about 25% slower. LZO was designed for fast decompression, and so it is no surprise that it is nearly as fast as WKdm.

WK4x4 suffers as a result of the slow approach to emitting its encoding, as described in 5.2.2. Because it performs many operations to parse each component of the compressed page, its decompression time is the worst of all four algorithms. The value of the fast packing used for WKdm is more evident on decompression than on compression.

Only in the unusual case of **lindsay**, where traffic for memory sizes beyond 100 pages comprises mostly zeros, do we see a significant change in mean decompression time. Only LZRW1 performs more slowly on this traffic than on the traffic at memory sizes less than 100 pages. The rest of the algorithms compress more than 40% faster due to the change in paging traffic.

Standard deviations of decompression speed. Given the consistency of mean decompression time, is it not surprising that the corresponding standard deviations are very low for all of the compression algorithms—about 0.04 ms (12,000 cycles) for the algorithms with greater variance, and under 0.01 ms (3,000 cycles) for both WKdm and LZO in many cases.

5.4 Related Work

There are other compression algorithms that have been designed for in-memory data compression. One of them has been implemented only in software, another only in hardware, and one has been implemented in both. While some of these algorithms resemble ours, others are quite different, and may capture meaningful regularities that could be combined with those modeled by the WK compressors.

5.4.1 The Original WK and The Newton

A precursor to the WK algorithms was used successfully in the Apple Newton—a personal digital assistant (or "palmtop") that had no backing store [SW91].[4] This algorithm was an early version of the WKfa algorithm. It was used successfully to increase the effective memory size of those machines.

5.4.2 X-match

The *X-match* algorithm presented in [KGJ96] is the most similar to our WK algorithms. X-match examines its input word by word. It keeps a dictionary of observed words, and so the dictionary grows with each new word encountered. When the dictionary ordered using the MTF algorithm. The output is created by using Huffman encoding.

Partial matching is also used by the X-match algorithm. Unlike the WK algorithms that rely on the similarity of high bits in many words, X-match attempts to match *any* two bytes of a word against dictionary entries. If it can find a dictionary entry that shares two byte values with the input word, then that is considered a partial match.

The *X-RL* variant of their algorithm is capable of detecting contiguous sequences of *zero* words, and then run-length encoding those sequences. Enough of in-memory data is zeros that this run-length encoding capability may provide real benefit.

X-match was designed with hardware implementation in mind. The researchers have implemented their algorithm in hardware, and have tested its applicability to in-memory data [KGJ98, KGJ99]. The hardware implementation does provide much faster execution. However, these first attempts at in-memory data compressors capture only some of the known regularities of the data. More regular-

[4]Also, personal communications between Paul Wilson and Walter Smith, 1994 and 1997.

ities could efficiently be captured, and more research is required to determine the efficacy of new techniques. Until more is known about this compression domain, it may be premature to commit solutions completely to hardware.

The conclusions reached by Kjelso, Gooch, and Jones are very similar to those presented here: There are strong, simple regularities to in-memory data that can be captured with straightforward models. In-memory pages can efficiently be compressed to 50% of its original size, and compression can be performed nearly as quickly as decompression.

5.4.3 Zero Removal

The extremely common occurrence of *zero* words led Rizzo to develop the *Zero Removal (ZR)*[5] compression algorithm for in-memory data [Riz97]. This compressor uses very simple encoding. All *zero* words are encoded by emitting a single 0 bit, and all other words are encoded by emitting a 1 bit, and then the literal bits of the non-zero word.

This approach to compression is very amenable to fast implementation. For programs with a good deal of sparse data, the compression may be acceptable. In experiments based on the compression of pages that have been evicted to the backing store of real machines, Rizzo found that ZR could reduce pages to approximately 60% of their original size.

Using the swap space of a real machine as a sample of the traffic that a compressed cache would see is not likely a good choice. The pages in such a sampling do not occur with the correct frequency, and the contents of the swap partition on a real disk may be largely unused. More importantly, the 10% of compression lost by modeling only zeros is a significant loss of reduction.

Rizzo asserts that detecting common word values other than zero would re-

[5]Rizzo did not name his algorithm, so we are providing this name for convenience.

quire too much computational time for a compressed caching context. However, the performance of the WK algorithms and X-Match algorithms demonstrate that finding regularities in non-zero values is possible and essential for effective compressed caching.

5.4.4 Parallel, Cooperative Dictionary Compression

As part of a research project at IBM, Franaszek, Robinson, and Thomas have developed new approaches to compression of small blocks [FRT96]. The intent of their work is to develop a hardware compressor that could be used in a compressed caching system, and possibly in other domains.

There are three novel ideas that they propose. None of these ideas are specific to in-memory data, but all contribute to the development of a fast, yet effective compressor.

1. Any input can be divided into a set of sub-blocks that can be concurrently compressed. By exploiting this available concurrency, a nearly linear-time speedup can be achieved over serial compression.

2. Each thread used to compress a sub-block of a concurrent compression scheme need not use independent dictionaries. The researchers describe the maintenance of a shared *(cooperative)* dictionary used by all compressing threads. This sharing can improve the effectiveness of modeling, and thus yield better compression.

3. Dictionary compressors like LZ77 store pointers to previously observed, repeated portions of the input, thus allowing a shorter representation. By storing pointers only to *previous* input sequences, we are guaranteed unambiguous decompression. The researchers note that pointers only to previous input is sufficient, but not necessary. They then describe an approach to matching

current portions of the input to *future* portions, while still guaranteeing an unambiguous decompression. By finding such "forward pointers", a cooperative dictionary, built by threads compressing different sub-blocks, provides more matches.

They implement their compression algorithm in software, and simulate the concurrent execution. They compare its performance on one text file of the Calgary Corpus *(paper1)* [BCW90] to LZSS using Huffman encoding of its pointers. They find that their parallel compression scheme can reduce the input to approximately 60% of its original size. They also find that by using a cooperative dictionary, parallel compression is nearly as effective as the serial compression provided by LZSS, an LZ77 variant.

Exploiting available parallelism for data compression seems a good approach to improving speed. However, this research does not consider that a concurrent approach could be taken with compression techniques other than those of Lempel and Ziv. Their compression techniques are not designed to find more efficiently the regularities that are common to in-memory data. As such, their algorithms are not as fast as they could be.

Their experiments also fail to provide useful information about the performance of their algorithms on representative data. By compression a text file, their experiments fails to demonstrate how well their algorithm would compress real in-memory data pages. Given that virtual memory pages are small, it is important to asses the compression performance of any algorithm, particularly LZ-like algorithms that may need larger blocks to accumulate useful dictionary contents.

Chapter 6

Compressed Caching

6.1 Introduction

As we have stressed, the gap between processor and disk speeds is large and is growing. Programs that run entirely in RAM benefit from improvements in CPU speeds, but the runtime of programs that page is likely to be dominated by disk accesses, and may run many times more slowly than CPU-bound programs. It would be desirable for virtual memory performance to improve with increases in processor speed. It would also be desirable if the performance decrease due to paging was gradual, such that small to medium amounts of paging were tolerable.

In [Wil90, Wil91b], Wilson proposed storing pages in compressed form in a main memory *compressed cache* to increase the effective memory size, and thus reducing disk paging. Appel also promoted this idea [AL91], and it has been evaluated empirically in a number of studies—first by Douglis [Dou93], and later by others [RC96, KGJ99, CCB99].

The study that was performed by Douglis was the most influential for many years. Unfortunately, that study was based on a kernel implementation in the Sprite operating system that showed reductions in paging time for some programs, but increase for others. In another study, Russinovich and Cogswell's mixed PC workload showed only a slight potential benefit. The result of these studies has been a widespread belief that compressed caching is attractive only for machines without a fast, local disk, such as diskless handheld computers, network computers, and laptops with slow disks.

As Douglis and we pointed out, however, compressed caching becomes more attractive as CPUs continue to get faster relative to disk speeds. The most recent studies [KGJ99, CCB99], performed concurrently with ours, show that compressed caching would provide benefit for current hardware configurations. Significantly, neither of these recent studies have presented methods of adaptively adjusting the size of the compressed cache. Douglis found that virtual memory systems are sensative to compressed cache size, and that an effective method of adaptivity must be employed. Unfortunately, his method of adaptivity does not respond to reference behavior in a timescale relative manner.

We aim to show that the discouraging results of former studies were primarily due to the use of machines that were quite slow by current standards. For current, fast, disk-based machines, compressed caching offers substantial performance improvements, and its advantages will only increase as processors get faster. We also study future trends in memory and disk bandwidths. Compressed caching will be increasingly attractive, regardless of other OS improvements (like sophisticated prefetching policies, which reduce the average cost of disk seeks, and log-structured file systems, which reduce the cost of writes to disk).

The concrete points in our analysis come from simulations of programs covering a variety of memory requirements and locality characteristics. Simulation allowed us to try easily many ideas in a controlled environment. It should be noted that all the simulation parameters have been chosen to be either perfectly realistic when possible, and conservative when realism was difficult to achieve. For instance, we conservatively assume a fast disk and perfect dirty-page cleaning in our experiments. We also measure with great accuracy the costs of compressions and decompressions for the actual paging traffic that would be incurred by a real compressed caching system.

The main value of the simulation results, however, is not in estimating the exact benefit of compressed caching (even though it is substantial). Instead, the most important result is that it is possible to detect reliably how much memory should be compressed during a phase of program execution. The same form of recency-based adaptivity that was applied to EELRU in Chapter 4 has been applied here, and the result is a compressed caching policy that adapts to program behavior. The exact amount of compressed memory crucially affects program performance: compressing too much memory when it is not needed can be detrimental, as is compressing too little memory when slightly more would prevent many memory faults. Unlike the simulations that used a fixed fraction of memory for the compressed cache, this adaptive scheme yields uniformly high benefits for all test programs and a wide range of memory sizes.

In this chapter, we will begin by demonstrating how to calculate the costs and benefits of keeping a compressed cache of a particular size. We will show that, given access patterns from three workloads, cost-benefit calculations indicate that compressed caching should provide benefit often. We will then show the results of simulating fixed-size compressed caches. These simulations will reveal that adapting the compressed cache size to improve performance is not difficult if the adaptivity method responds to simple recency information. We will discuss how to perform that adaptivity, and then analyze the results of simulating a size-adaptive compressed cache.

6.2 Calculating the Costs and Benefits

Recall that recency information reveals useful access patterns to which EELRU responded. The cost-benefit analysis presented here will also be based on the recency information that an LRU queue reveals about a program. By examining the LRU miss histogram, which indicates the number of misses across a range of memory sizes, we can determine the cost incurred and the benefit provided by a compressed cache.

When a compressed cache is introduced into the memory hierarchy, there is a cost associated with moving pages to and from that new level. However, because the compressed cache can reduce the number of disk accesses, there is also a benefit. In order to determine whether compressed caching improves system efficiency, the benefit must outweigh the cost.

The example shown below is for a specific (but artificial) LRU miss histogram and assumed system parameters. We illustrate how to calculate the cost and benefit based on this particular histogram. We then generalize that calculation and derive it for any set of values. That calculation is then used on real histograms from three different workloads to show that compressed caching should provide benefit on real systems.

6.2.1 An Example

Figure 6.1 shows an artificially constructed LRU miss histogram. Consider a system that has 100 pages of available RAM. This program would incur 100,000 disk accesses under a traditional virtual memory system—that is, one without a compressed cache.

Assume that half of RAM is used for a compressed cache; 50 pages are used for the uncompressed cache, and the other 50 pages are dedicated to the compressed cache. Also assume a compression ratio of 2:1 (that is, a reduction of 50%) for the compressed pages. The 50 page compressed cache can therefore store 100 compressed

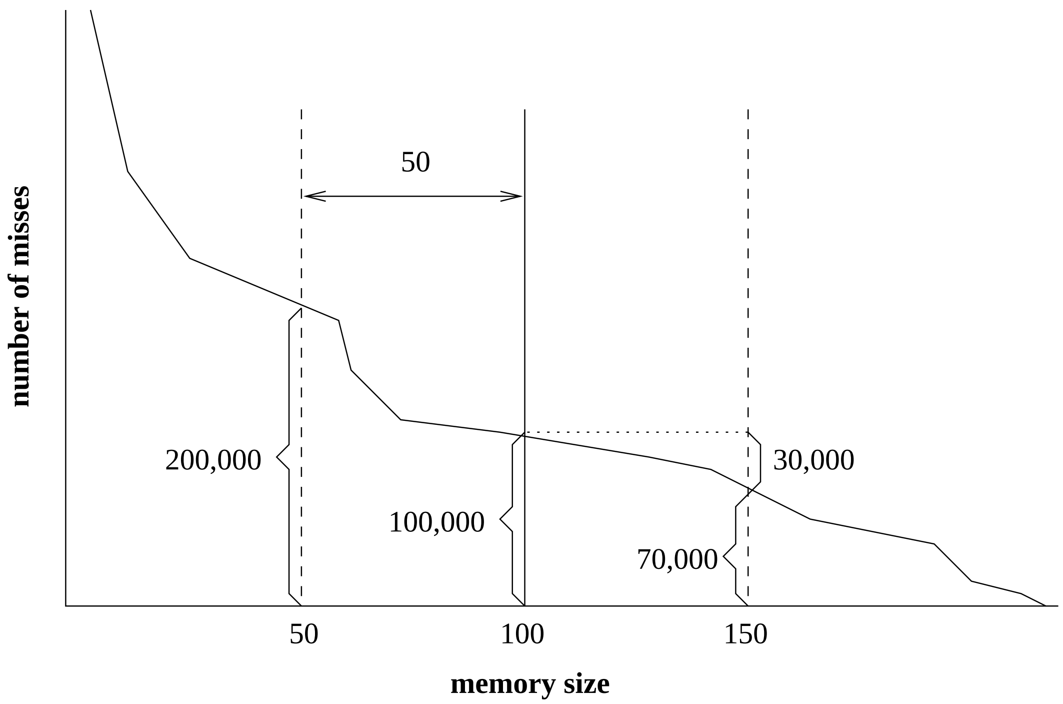

Figure 6.1: The LRU miss histogram provides needed information for the calculation of the cost and benefit of dividing some amount of RAM to hold both an uncompressed and a compressed cache.

pages. Thus, a total of 150 pages can be stored in RAM, and the compressed cache has effectively expanded the memory size by 50%.

Observe that the uncompressed cache will miss on 200,000 references. All of those missed references will require both the decompression of the referenced page and the compression of the page evicted from the uncompressed cache into the compressed cache. Assume that 0.5 ms are required to compress one page and decompress another. Servicing those 200,000 misses then requires 100,000 ms. That value is the *cost* of using a compressed cache.

Because the effective main memory size is larger, thanks to compression, the system only needs to access disk for 70,000 references instead of 100,000. Therefore, 30,000 disk accesses have been avoided. Assuming a disk access time of 5 ms, 150,000 ms of disk access time has been saved. The *benefit* of a compressed cache is that 150,000 ms.

So, the compressed cache has saved 150,000 ms of disk access time in exchange for 100,000 ms of compression and decompression time. Since the benefit was greater than the cost by 50,000 ms, this program would benefit from using a compressed cache.

Note that these numbers are chosen to make the example simple, and that the real values involved can be quite different. We also have not accounted for the cost of writing dirty pages to disk. The compressibility of pages is program and data dependent, as is the miss histogram. However, for any program, we can use this same kind of calculation to approximate the effect of a compressed cache.

6.2.2 Generalization and Derivation

Figure 6.2 shows an LRU miss histogram similar to the one in Figure 6.1[1], except we have labeled it with symbols so that we may derive general cost-benefit formulae.

Assume that M_{real} is the amount of main memory is available for executing a program. Some fraction f_{comp} of M_{real}, where $0 \leq f_{comp} < 1$, is used to store the compressed cache. The fraction of M_{real} dedicated to the *uncompressed cache* is the remaining portion of main memory, $f_{uncomp} = 1 - f_{comp}$. From these values, we calculate the amount of RAM M_{comp} dedicated to the compressed cache and the amount of RAM M_{uncomp} used for the uncompressed cache:

$$M_{comp} = M_{real} \times f_{comp}$$

$$M_{uncomp} = M_{real} \times f_{uncomp}$$

[1] Differences between the LRU miss histograms of two programs can be significant. Our cost-benefit analysis depends only on the monotonically non-increasing nature of the LRU miss histogram.

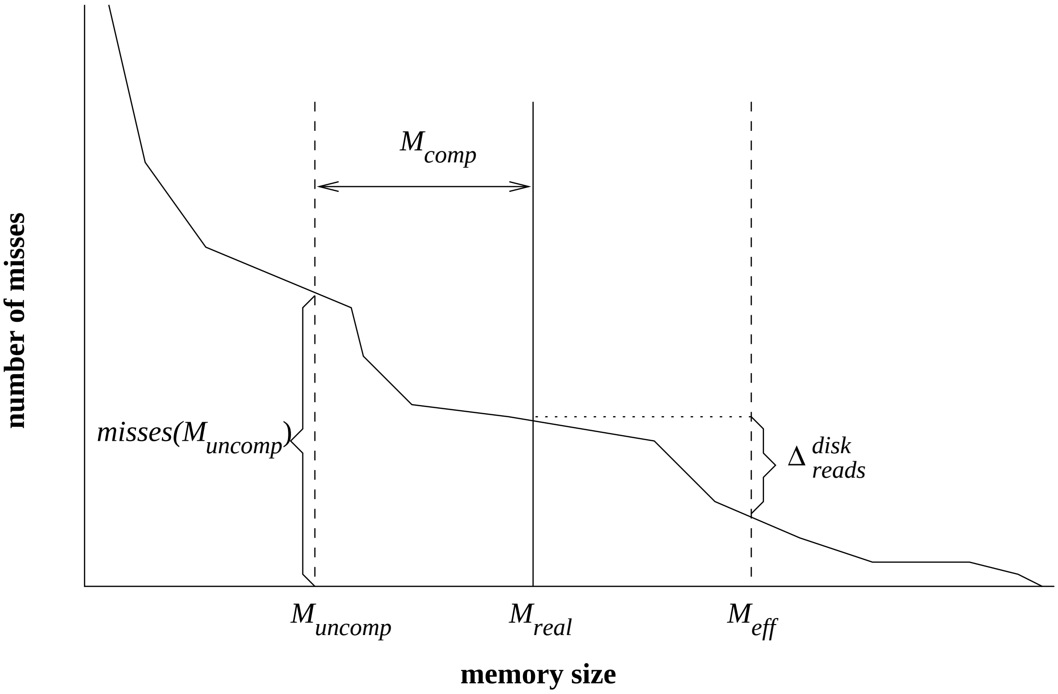

Figure 6.2: An LRU miss histogram, labeled for the derivation of cost-benefit formulae.

For a given set of pages stored in the compressed cache, we can refer to its mean compression ratio with the symbol $ratio$.[2] We can calculate M_{eff}, the effective memory size achieved with the compressed cache, as follows:

$$M_{eff} = M_{uncomp} + (M_{comp} \times ratio)$$

The disk is accessed for each miss at memory size M_{eff}. Assuming that compression is being achieved (that is, $ratio > 1$), then the number of disk accesses at M_{eff} is no greater (and is usually fewer) than at M_{real}. More specifically, if $misses(i)$ is the number of misses at memory size i, then the number of fetches from disk that are saved due to the increased effective memory size is:

[2]The figure is drawn with the mean compression ratio commonly achieved for in-memory data of $ratio = 2$.

$$\Delta^{disk}_{reads} = misses(M_{real}) - misses(M_{eff})$$

Similarly, each of the $misses(M_{uncomp}$ misses on the uncompressed cache will require one compression and one decompression. Note that pages on disk will also be in compressed form, as they will store pages evicted from the compressed cache. Each of the $misses(M_{eff})$ misses on the effective memory size will require not only a disk access, but also a compression and a decompression. However, those $misses(M_{eff})$ compression/decompression tasks are already counted in the $misses(M_{uncomp})$ misses on the uncompressed cache.

Lack of redundancy in the uncompressed and compressed caches. We have counted the number of pages that must be read from each memory level, but we have not been counting the number that must be written. Because disks are large, they store redundant copies of those pages that are also in RAM. Thus, only pages that have been modified while in RAM (i.e. pages that are *dirty*) must be written to disk upon eviction.

Compressed caches, however, should *not* store redundant copies of pages that are in the uncompressed cache, as that redundancy reduces the effective memory size and causes more disk faults[3]. All $misses(M_{uncomp})$ misses on the uncompressed cache require a compression operation, irrespective of the dirtiness of the evicted page.

Let $dirty(i)$ be the number of dirty pages that are evicted at memory size i. The number of disk write operations saved by expanding the effective memory size with a compressed cache is then:

$$\Delta^{disk}_{writes} = dirty(M_{real}) - dirty(M_{eff})$$

[3]Some past experiments with compressed caches *did* store redundant copies. Such redundancy may have contributed significantly to negative results.

Since service times are needed to calculate cost and benefit, let $service^{cc}_{read}$ be the time required to decompress a page as it is fetched from the compressed cache, and let $service^{cc}_{write}$ be the time required to compress a page as it is evicted into the compressed cache. Similarly, $service^{disk}_{read}$ and $service^{disk}_{write}$ are the times required to read a page from and write a page to disk, respectively.

Finally, we can calculate the cost and benefit. The cost, denoted by the symbol *cost*, is the time required to service all of the compressions and decompressions required for a compressed cache:

$$cost = (service^{cc}_{read} + service^{cc}_{write}) \times misses(M_{uncomp})$$

Note again that since redundant copies of pages are not stored in both the compressed and uncompressed cache, each miss on the uncompressed cache will require an eviction into the compressed cache irrespective of the dirtiness of the page being evicted.

The benefit, denoted by the symbol $benefit$, is the time that would have been required to service extra disk faults if a compressed cache had not been used:

$$benefit = (service^{disk}_{read} \times \Delta^{disk}_{reads}) + (service^{disk}_{write} \times \Delta^{disk}_{writes})$$

When $benefit > cost$—that is, when the the time saved by avoiding disk faults is greater than the time required to service compressed cache faults—then the compressed cache will reduce the program's paging time.

Because the number of misses on a smaller memory are usually much higher than on a larger one, the number of compressed cache faults is likely to be much larger than the number of disk faults. Compression and decompression must then be significantly faster than disk access. To calculate how much faster, we can algebraically transform $benefit > cost$ into:

$$\frac{service^{disk}_{read} + (service^{disk}_{write} \times \frac{\Delta^{disk}_{writes}}{\Delta^{disk}_{reads}})}{service^{cc}_{read} + service^{cc}_{write}} > \frac{misses(M_{uncomp})}{\Delta^{disk}_{reads}}$$

The factor $\frac{\Delta^{disk}_{writes}}{\Delta^{disk}_{reads}}$ is the fraction of disk writes saved to disk reads saved. Anywhere from none to all of the disk reads saved may have required a corresponding write operation. So, the range of that fraction is $0 \leq \frac{\Delta^{disk}_{writes}}{\Delta^{disk}_{reads}} \leq 1$. If no disk writes are saved by the compressed cache, then $\frac{\Delta^{disk}_{writes}}{\Delta^{disk}_{reads}} = 0$. Since disk write operations are sometimes queued, and then later written as a batch, assuming that this fraction is zero also allows us to conservatively assume perfect dirty-page cleaning.

Based on the inequality above, the ratio of disk speed to the combined compression and decompression speeds must be at least as large as the ratio of compressed cache accesses created to disk accesses saved. If the queuing of disk writes (a.k.a. *dirty-page cleaning*) is not perfect, then compression/decompression can be slower relative to disk while still satisfying the inequality.

6.2.3 Calculations for Real Programs

Now we can apply the cost-benefit calculation derived above to LRU miss histograms taken from real programs. We need to assume reasonable parameters for the CPU speeds and disk access times. So, we have chosen values based on one of the real machines used for other experiments, specifically a **180 MHz Pentium Pro**.

Chosen parameters. We assume a disk latency of 5 ms, which is a common average seek time for disks in fast workstations. For machines with slower disks, the benefit of compressed caching will be greater. Since a page is such a small unit of data for a disk to read or write, the time to read or write one page is dominated by the latency; the actual transfer time is negligible. For systems that perform effective prefetching, or systems that use a larger page size, the average latency per page may be somewhat lower, and compressed caching will offer less benefit. We will denote the time to read or write a page as $service^{disk}_{read} = service^{disk}_{write} = 5ms$.

We also assume a mean time for performing a compression and decompression. These numbers are approximately correct for our *WKdm* compression algorithm on the Pentium Pro 180 MHz processor, which performed well in the simulations presented in Sections 6.3 and 6.5. Decompression requires $service^{cc}_{read} = 0.15 ms/page$, and compression requires $service^{cc}_{write} = 0.25 ms/page$.

The LRU miss histograms show only misses, not the writing of dirty pages. Therefore, we calculate the *worst* case for a compressed cache by assuming that dirty-page cleaning is perfect, and none of the disk accesses saved were write operations; that is, $\frac{\Delta^{disk}_{writes}}{\Delta^{disk}_{reads}} = 0$. As a result, the calculation reflected in this section is:

$$\frac{service^{disk}_{read}}{service^{cc}_{read} + service^{cc}_{write}} > \frac{misses(M_{uncomp})}{\Delta^{disk}_{reads}}$$

It is important to notice that these LRU miss histograms are taken from representative workloads that would be executed in real computing environments. We have compiled LRU miss histograms from three different workloads, each having been collected from different platforms. Note that we are not using this cost-benefit calculation alone to provide a convincing case for compressed caching. Rather, we are trying to show that real histograms from different workloads indicate that compressed caching should work in real contexts, and thus merit further investigation.

Our UNIX workload. The first data set was taken from a group of six non-interactive programs run under Linux on Intel x86 machines. This set of programs is used for our simulations, and is more fully described in Section 6.3.1. We have chosen to use the histogram from *p2c*, a Pascal to C compiler, as a representative example.

The LRU miss histogram for p2c, plotted in Figure 6.3 provides the number of misses to each memory level. These miss numbers and the assumed hardware parameters allow us to calculate the execution time for every possible compressed cache size. The results of those calculations are shown in figure 6.4. The *non-paging* *runtime* is the total execution time for p2c if it had enough memory that it did not page. The *paging runtime* is the total execution time if no compressed cache were used and only 110 pages of main memory were available.

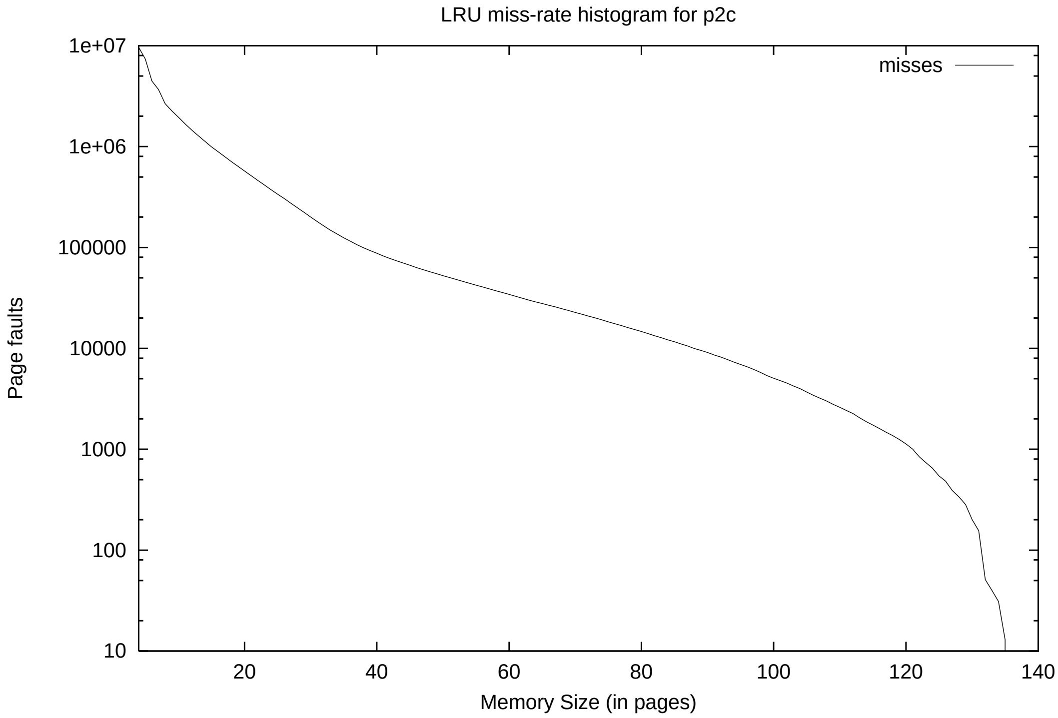

Figure 6.3: The LRU miss histogram for p2c, a Pascal to C compiler.

The results for different programs and different memory sizes have the same shape at the results shown in Figure 6.4. Notice that a large range of compressed cache sizes improve paging performance—from about 2% to 43% of the 110 pages. This finding corresponds well enough to simulation results that this calculation can be used to determine roughly the extent to which a fixed-size compressed cache would reduce paging time.

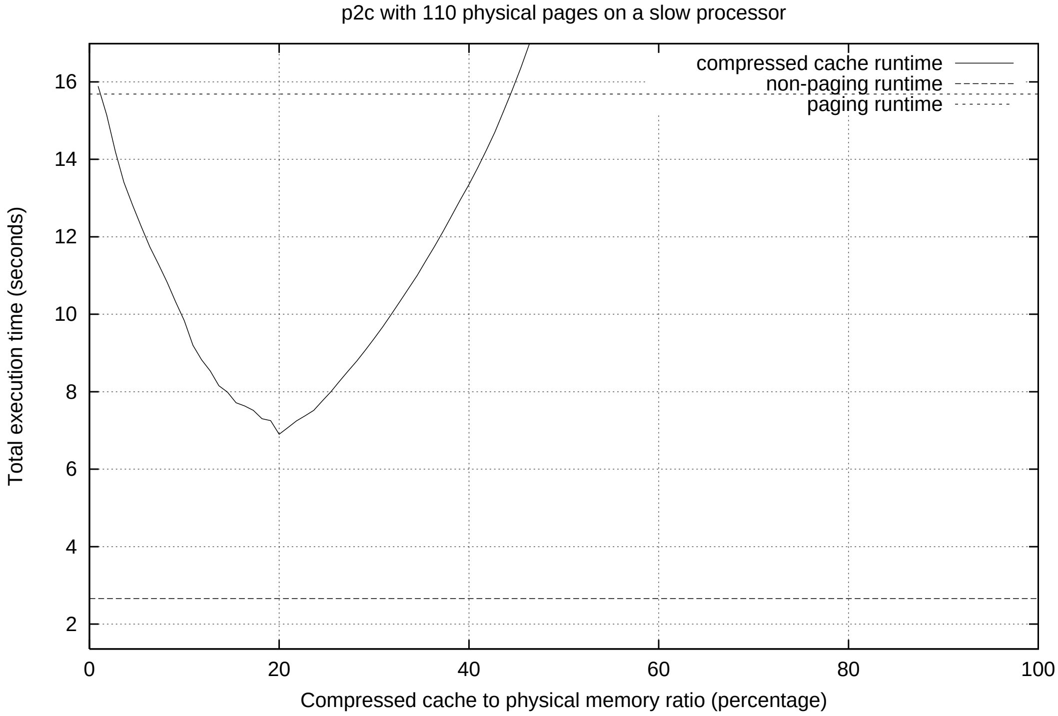

Figure 6.4: The LRU cost-benefit performance curve for p2c, which shows the total runtime for p2c over a range of compressed cache sizes for a 110 page memory.

A Windows NT, workload of batch and interactive programs. Researchers at the University of Washington used the Etch tracing tool to collect traces for both interactive and non-interactive programs running under Windows NT. Figure 6.5 is the LRU miss histogram for the popular web browser, Netscape Navigator. Based on that histogram, we can calculate the time this application would spend paging with and without a compressed cache. The results are shown in figure 6.6.

Navigator is an interactive program, unlike p2c, and yet the two have similarly shaped LRU miss histograms. The histograms are similar enough that a compressed cache would benefit both of these programs. If Navigator runs in 116 pages of RAM, we notice that the program would spend about 150 seconds paging

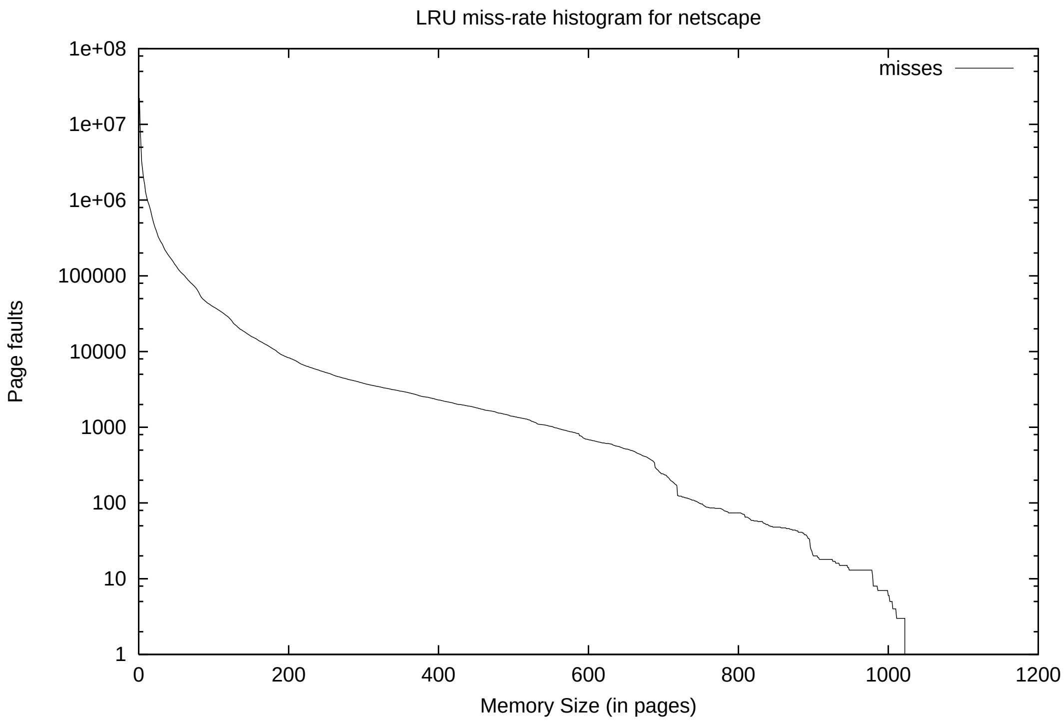

Figure 6.5: The LRU miss histogram for Netscape Navigator, a popular web browser.

without a compressed cache, but would need about 100 seconds of paging with one.

The Winstone benchmark. Finally, we examine the LRU miss histogram for the entire Winstone benchmark. This histogram was taken from [RC96], which presented an experiment in compressed caching under the Windows 95 operating system. The histogram data provided misses for only 6 to 16 Mbyte memories in one-Mbyte intervals. We extrapolated that histogram, and the results are shown in figure 6.7. It is similar to the other histograms shown, except that the extrapolated histogram never reaches zero misses as real histograms do when memory is large enough to hold the entire footprint.

With eight Mbytes available to run these programs on a traditional virtual

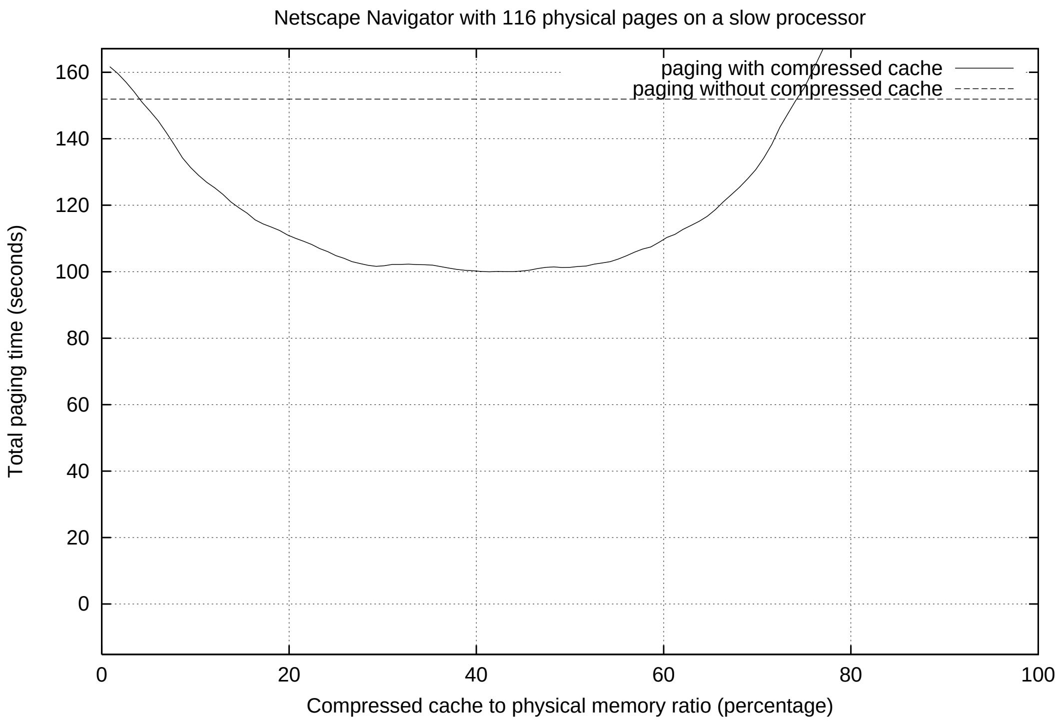

Figure 6.6: The LRU cost-benefit performance curve for Netscape Navigator, which shows the total paging time for a given range of compressed cache sizes in a 116 page memory.

memory system, the benchmark would suffer more than 130 seconds of paging time. Using slightly less than 40% of that eight Mbytes to store compressed pages, the total paging time would be reduced to about 70 seconds of paging—a reduction of about 40%.

Three different kinds of programs, gathered under different platforms, yield similar results. Although there may be workloads for which compressed caching cannot provide benefit, many real programs exhibit reference behavior for which compressed caching would be helpful.

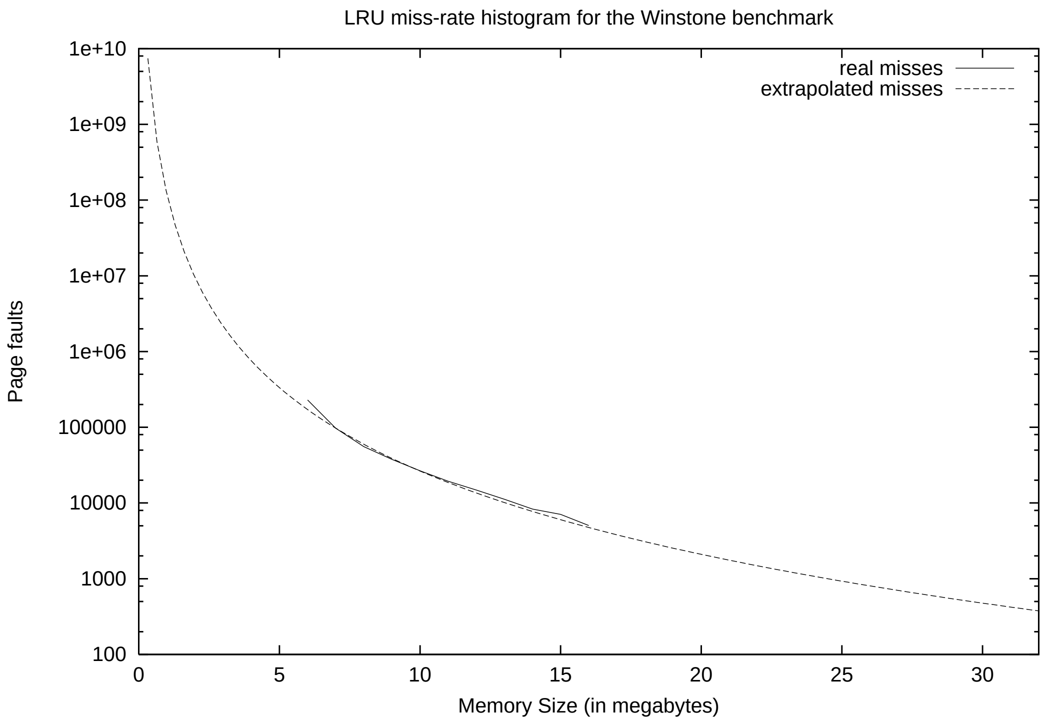

Figure 6.7: The LRU miss histogram for the Winstone benchmark. The actual data provided only part of the curve, and so the rest was extrapolated.

6.3 Fixed-Size Compressed Cache Simulations

Using the same set of real reference traces for a UNIX workload mentioned above and described in Section 6.3.1, we simulated compressed caches whose sizes were fixed. In particular, for each program, we simulated a range of physical memory sizes, and for each such physical memory size, we simulated a range of compressed cache sizes.

This group of simulations not only verified that our cost-benefit calculation was reasonable, but also helped to reveal the relationship between physical memory sizes and compressed cache sizes. Further, these simulations demonstrated that

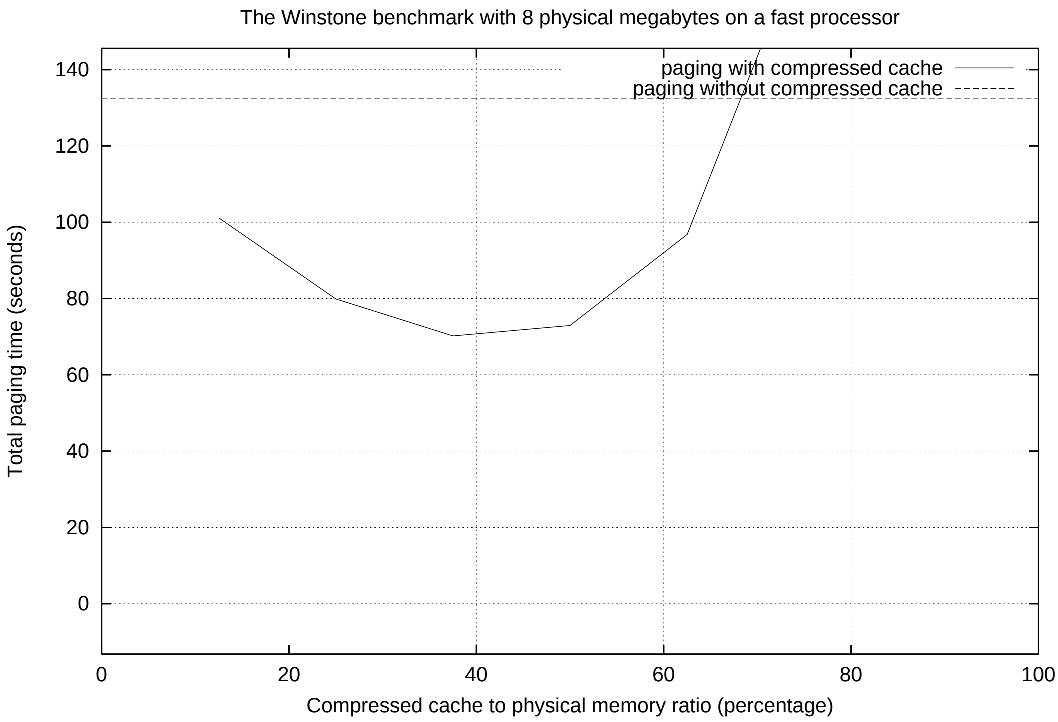

Figure 6.8: The LRU cost-benefit performance results for the Winstone benchmark, which shows the total paging time for a range of compressed cache sizes given an 8 Mbyte memory.

both compressibility and compression speed affect the ideal compressed cache size in different ways, and that neither characteristic is more important than the other.

6.3.1 Experimental Design

In this section, we describe the methodology and results of detailed simulations of compressed caching. We captured traces for six varied UNIX programs, which contain not only the sequence of pages touched *but also their contents*. These are the same page image traces used to measure the performance of various in-memory data compressors in Section 5.3.1, and are used here to simulate compressed caching in detail.

Note again that our traces do not contain references to executable code pages. We focus on data pages, because our main interest is in compressing in-memory data. As we will explain in Section 6.6, compressing code equally well should not be difficult. Several techniques complementary to ours have been proposed for compressing code. Further, the data from [RC96] indicate that references to code pages exhibit the same locality properties as references to data pages.

The test suite. For these simulations, we traced six programs on an Intel x86 architecture under the Linux operating system which uses a page size of 4 Kbytes. These programs were chosen to represent a number of different footprint sizes, as we did not know whether the reference patterns of small programs would be fundamentally different from those of large programs. We wanted to seek patterns in reference behavior and compressed cache performance that would scale to any memory size.

- **Espresso** is circuit simulator that is used to optimize programmable logic arrays. The input file was provided by Ben Zorn as part of a study of dynamic allocation behavior [Joh97].

- **cc1**, from the GNU C compiler version 2.7.2, is the heart of that compiler, and actually performs C compilation. The input file on which compilation was performed was the pre-processed `combine.c`, the largest source code file of this GCC distribution.

- **Ghostscript** version 3.33 is an interpreter of the PostScript page rendering language. The input file is a 127-page manual for the *Self* system. Some of the pages contain a mixture of graphics and text, and as such require more space to be rendered.

- **Grobner** is a program that rewrites a formula into its Gröbner basis functions.

- **Lindsay** *(Hyper)* is a communications simulator for a hypercube network of processors. It builds an internal model of the hypercube network, and then simulates the transmission of random messaging. The model of the hypercube is stored as a single, large array that is allocated early in execution and lives until termination. The messages are represented as small, dynamically allocated blocks of memory that live only a short time.
- **p2c** translates Pascal source code into C source code. The test input results in a C code file that is approximately 550 Kbytes long.

The data footprints of these programs vary widely: cc1, Ghostscript, and lindsay are larger (around 550 pages), while espresso, p2c, and grobner are small (around 100 pages).

We used the same four compression algorithms for these experiments as we used in the experiments on in-memory data compression, namely **WKdm**, **WK4x4**, **LZO**, and **LZRW1**. They are described more fully in Section 5.3.1.

Using different CPUs. One additional dimension to these experiments is that we recorded the results of compression and decompression times on different architectures and CPU speeds. Such results allowed us to demonstrate the effect of the improvements in CPUs that occur every year, as well as the applicability to different architectures. We used the following three processors:

1. **Pentium Pro at 180 MHz**: This processor represented an average desktop computer from one year ago. Compressed caching is not only for fast machines.
2. **UltraSPARC-10 300 MHz**: While one of the faster processors available now, it will soon be an average processor. Compressed caching works even better on a faster processor.

3. **UltraSPARC-2 168 MHz**: A slower SPARC machine which provides an interesting comparison to the Pentium Pro, due to its different architecture (e.g., faster memory subsystem).

Gathering reference traces. The simulator takes as input a trace of the pages a program touches, augmented with information about the compressibility of each page and the time required to perform compression and decompression. These results were needed for each compression algorithm. To create such a trace and keep the trace size manageable, we used several steps and several tracing and filtering tools.

We traced each program using the portable tracing tool *VMTrace* [WKB]. We added a module to VMTrace that made it emit a complete copy of each page as it was referenced. This was the mechanism used to capture the page image traces.

Each record of a page image trace is slightly over 4 Kbytes (assuming a 4 Kbyte page size). Since reference traces contain tens or hundreds of millions of references, page image traces can quickly exceed available storage. Even if they could be stored, it would be infeasible to use such traces as input to a simulator. So, the page image traces were reduced using Coffman and Randell's [CR70] technique to create behavior sequences, still augmented with page images.[4] (This technique is described more fully in Section 3.2.2.)

To record the actual effectiveness and time cost of compressing each page image, we created a set of *compressibility traces*. For each combination of compression algorithm and CPU, we created a trace augmented with the time required to compress and decompress pages from the page image trace, as well as the reduction achieved by compression. Since we have 6 test programs, 3 compression algorithms, and 3 CPU's, there were a total of 54 compressibility traces.

[4] We did not use our SAD and OLR techniques only because we had not yet developed them when we gathered these traces, but we had reinvented Coffman and Randell's behavior sequence technique.

The tool that creates compressibility traces is linked with a particular compressor and corresponding decompressor. It consumes a page image trace[5]. For each record in the input, this tool compresses and decompresses the page image and outputs a trace record. This record contains the page number, the times for compressing and decompressing the page's contents at that moment, and the resulting compressed size of the page. Each page image is compressed and decompressed several times, and the median times are recorded. Timing is very precise, using the Solaris high-resolution timer (as all of our compression timings were done under the Solaris operating system). To avoid favorable hardware caching effects, the caches are filled with unrelated data before each compression or decompression. (This approach is conservative, in that the burstiness of page faults will usually mean that some of the relevant memory is still cached in the second-level cache in a real system.)

Dynamic allocation and compressibility. Most of these programs were linked with Doug Lea's memory allocator, which was found to minimize fragmentation in [WJNB95]. If we had used an allocator that fragmented memory badly, pages would contain more wasted space that might compress well, unfairly flattering the compressed cache. The one exception is **cc1**. It uses a special *obstack* allocator, particularly well suited to the stack-like allocation patterns of compilation.

The range of physical memory sizes. For each of these programs, we chose *interesting* memory sizes to simulate. A loose definition of *interesting* is those memory sizes at which some paging is required to execute the program, but not so much paging that the program would be intolerably slow to run.

For a program whose footprint is five hundred pages, it is not interesting to simulate its execution on a six hundred page memory. No paging would occur,

[5] More specifically, our real implementation consumes a reduced page image LRU behavior sequence.

and so no events consequential to a virtual memory study would arise. Similarly, it would not likely be interesting to simulate that same program's execution on a five page memory, as it would likely page so much that its total runtime would be many orders of magnitude longer.

More specifically, we chose the memory size at which a 180 MHz Pentium Pro—one of our test processors—with a 5 ms disk would spend 1% of the total running time paging, and the remaining 99% executing on the CPU. We refer to this memory size as the *1/99 point*, and use it at the largest memory size for which we perform simulations. Note that the *0/100 point* is the memory size at which no paging occurred.

For the other end of our range, we chose the *99/1* point—a memory size so small that 99% of the execution time would be spent paging, and only 1% would be used executing the program on the CPU. In other words, the program's total runtime would be nearly 100 times longer than it would be at the *0/100* point. We believe that few people are interested in running a program if it is more than one hundred times slower than it should be due to paging.

Estimates used. During simulation, we had to estimate the costs for reading a page from disk or writing it to disk. We conservatively assumed that writing dirty pages to disk incurs *no cost.* Additionally, we again assumed a disk with a uniform seek latency of 5 ms.

6.3.2 Results

Understanding the plots. Some of the results of the simulations are shown in Figures 6.9 through 6.14. For each program, we selected memory sizes that incur a moderate amount of paging on a traditional virtual memory system. By "moderate", we mean that the program would execute at least twice as slowly, but no more than one hundred times slower than it would if enough RAM were available to avoid any

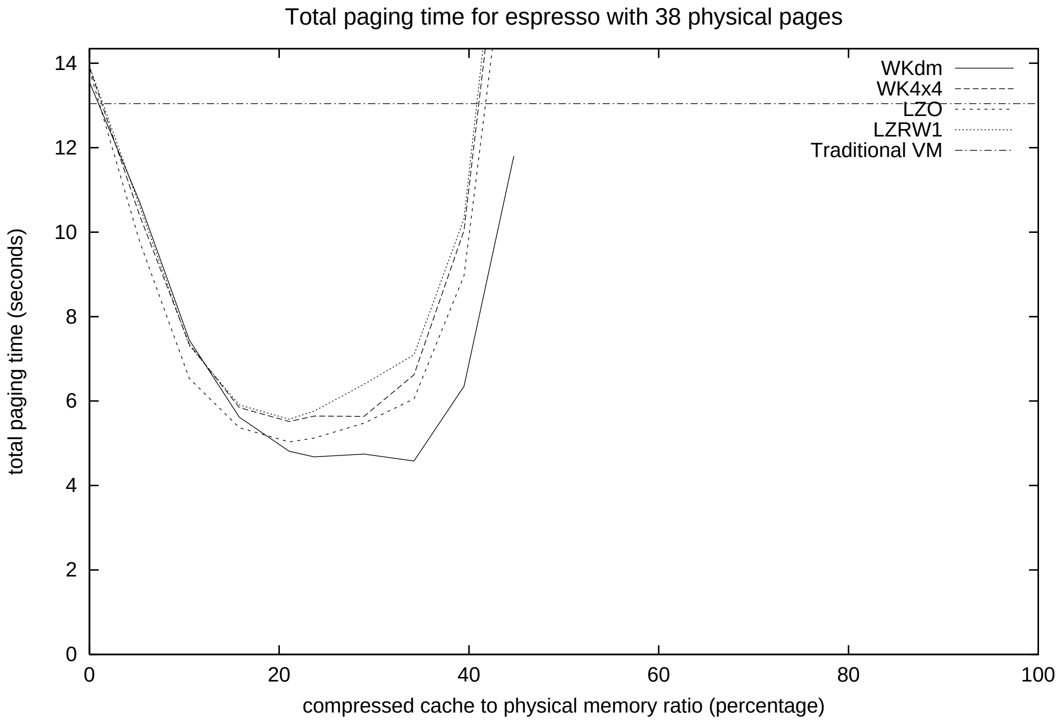

Figure 6.9: For a given physical memory size, the paging time for **espresso** with a compressed cache that is some percentage of that physical memory. For comparison, the "Traditional VM" time shows the paging time required for no compressed cache.

paging.

Note that these plots are chosen to show the relationship between compressed cache size, compression algorithm, and performance. While the results for other physical memory sizes are quite similar, they are not shown here. A more convincing argument for the applicability of compressed caching for nearly all physical memory sizes will be made later, in Section 6.4.

Note that many of the curves shown in these plots are truncated on the right. Because the traces had to be reduced, there is a limit to the information available to the simulator for larger compressed caches. In other words, the simulation requires

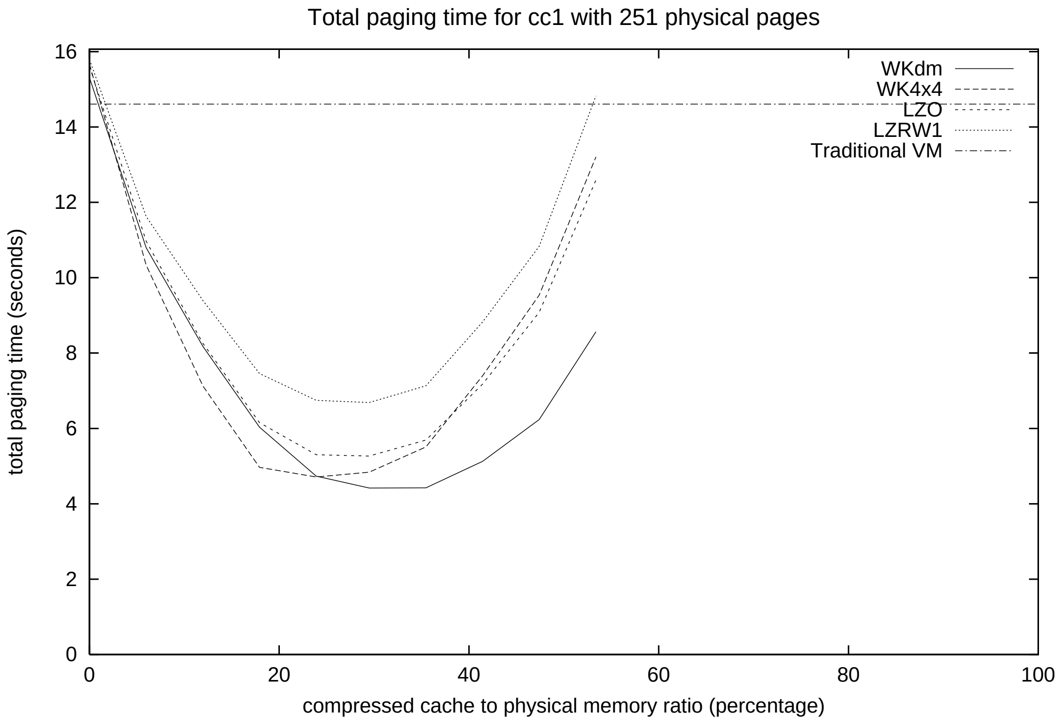

Figure 6.10: For a given physical memory size, the paging time for **cc1** with a compressed cache that is some percentage of that physical memory. For comparison, the "Traditional VM" time shows the paging time required for no compressed cache.

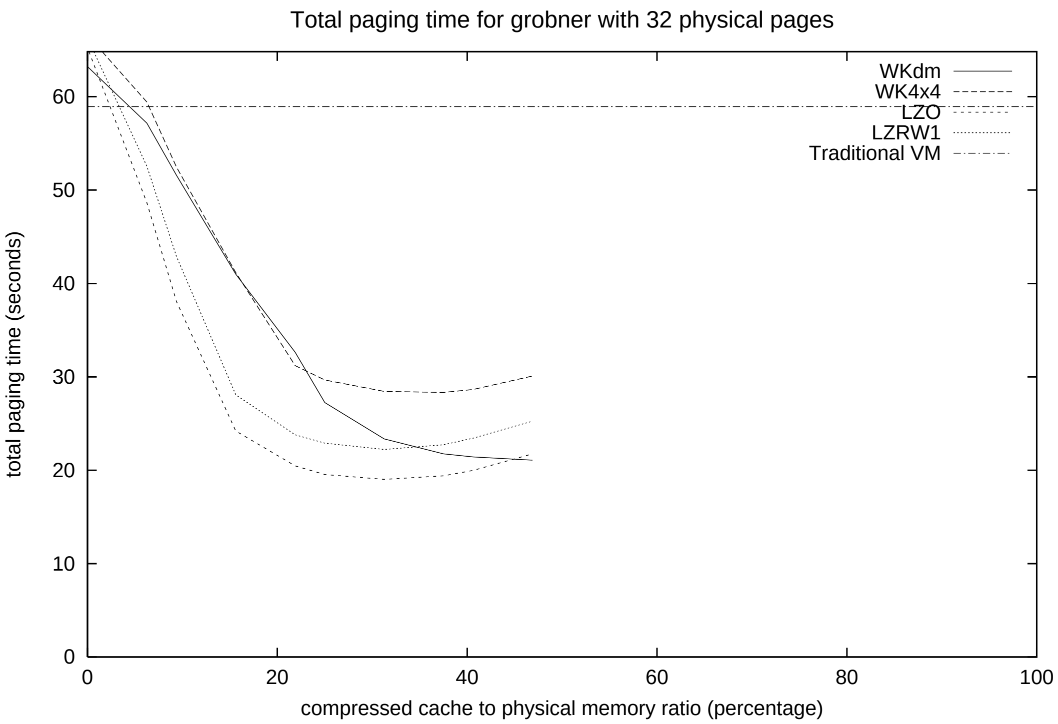

Figure 6.11: For a given physical memory size, the paging time for **grobner** with a compressed cache that is some percentage of that physical memory. For comparison, the “Traditional VM” time shows the paging time required for no compressed cache.

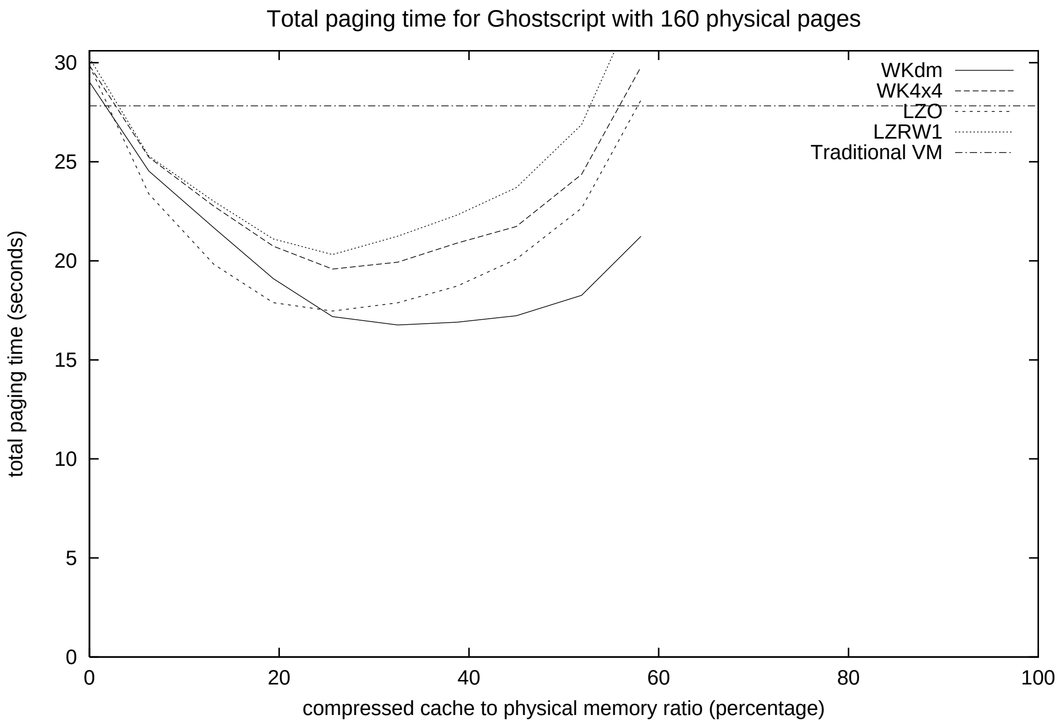

Figure 6.12: For a given physical memory size, the paging time for **Ghostscript** with a compressed cache that is some percentage of that physical memory. For comparison, the “Traditional VM” time shows the paging time required for no compressed cache.

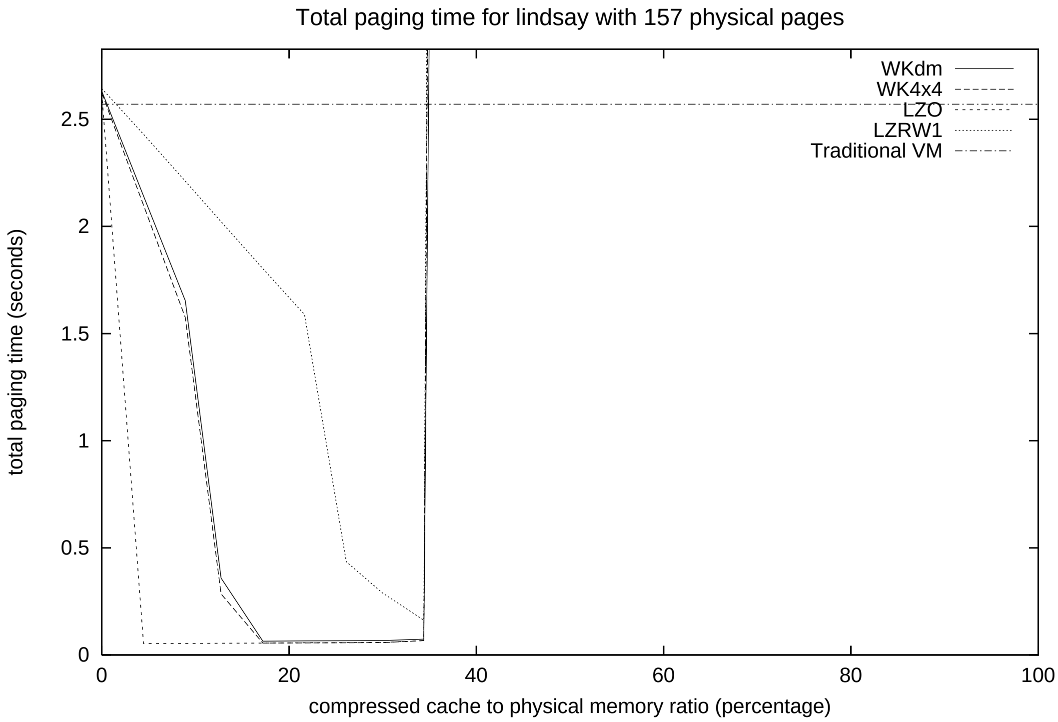

Figure 6.13: For a given physical memory size, the paging time for **lindsay** with a compressed cache that is some percentage of that physical memory. For comparison, the "Traditional VM" time shows the paging time required for no compressed cache.

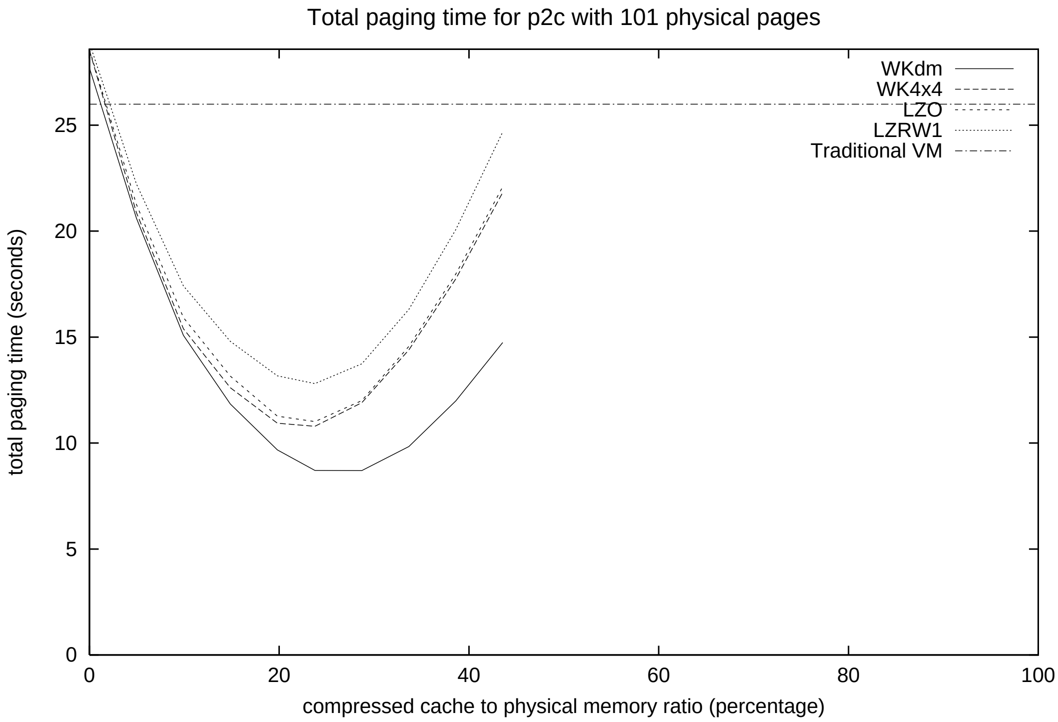

Figure 6.14: For a given physical memory size, the paging time for **p2c** with a compressed cache that is some percentage of that physical memory. For comparison, the "Traditional VM" time shows the paging time required for no compressed cache.

knowledge of which pages are fetched into and evicted from the uncompressed cache. As such, the simulation can only be performed for an uncompressed cache that is at least as large as the memory size specified for reduction.

Compressed cache sizes. With a well chosen compressed cache size, a significant reduction in paging costs is possible. If very few main memory pages are dedicated to the compressed cache, then the time spent compressing and decompressing cannot be beneficial, as the expansion in effective memory size is going to be too small. Similarly, if more than about 40% or 50% of the memory is used for a compressed cache, so much time is needed to perform compressions and decompressions that no benefit is provided.

The program **lindsay** is an unusual case. It's loop is sufficiently strict that for all memories beyond approximately 100 pages, very little paging time is required for a traditional virtual memory system. We can see, in Figure 6.13, that nearly all of that time can be eliminated with compressed caching. However, for any smaller memory, we would not easily be able to compress and decompress quickly enough to maintain a beneficial compressed cache of any size. The size of the compressed cache depends greatly on the locality and referencing behavior of the program.

Regularities in the results. The plots shown here, as well as those not shown for other physical memory sizes, all exhibit a very regular U shape. That is, for each curve, there is a single minimum. That minimum corresponds to the compressed cache size that would yield the least paging time given a particular physical memory size and compression algorithm.

Note that there is one very slight exception: For **espresso**, the first plot shown in Figure 6.9, WKdm and WK4x4 do not exhibit a perfect U shape. When approximately 30% of the 38 pages is used for a compressed cache, we see a slight change in concavity of the curve. This point is unique among our experiments, but shows that it is possible for the curves to have multiple, local minima. The compressibility of data is generally so uniform, however, that we do not see any other such cases of a non-U shape.

The uniformity of these plots provides very useful information about the problems in maintaining a compressed cache:

- All of the algorithms are capable of providing benefit on current hardware.
- The slower WK4x4 algorithm and the less effective LZRW1 algorithm are never the best algorithms to use.
- Algorithms that compress more tightly but execute more slowly, such as LZO, perform best with a smaller compressed cache than algorithms that compress less tightly but execute quickly, such as WKdm. Since both seem to provide roughly equal performance, given the appropriate compressed cache size, it is not clear that one should be chosen over the other.
- While the ideal compressed cache sizes for each compression algorithm differ, there tends to be a range of sizes that would yield nearly optimal performance for each. Adaptivity methods need not be sensitive to the compression algorithm being used in order to choose a good compressed cache size.
- If a compressed cache should be used at all, it should likely be between 10% and 50% of the total memory size.

These regularities indicate that a simple heuristic should be able to find the ideal compressed cache size. The smoothness of the curves indicate that such a heuristic could be somewhat inaccurate by choosing a compressed cache size that is not quite the ideal, and still provide a significant portion of the available benefit. Finally, these curves resemble closely those generated by the cost-benefit analysis.

Application of that simple calculation should be sufficient to make a compressed cache adapt its size.

6.4 Adaptively Adjusting the Compressed Cache Size

To perform well, a compressed caching system should adapt to the reference behavior of the programs for which it caches. If a program's data fits comfortably in RAM, few pages (or no pages) should be kept compressed, so that the overwhelming majority of pages can be kept in uncompressed form and accessed with no penalty. If a program's data is larger than the available RAM, and compressing pages would allow it to be kept in RAM, more pages should be compressed until the working set is "captured".

Douglis observed in his experiments that different programs needed compressed caches of different sizes. He implemented an adaptive cache-sizing scheme that varied the split between uncompressed and compressed RAM dynamically. Even with this adaptive caching system, his results were inconsistent; some programs ran faster, but others ran slower. We believe that Douglis' adaptive caching strategy may have been partly at fault. He used a fairly simple scheme in which the compressed cache competed not only with the uncompressed cache, but also the file system cache for main memory space. Resizing was performed by taking space away from the cache that held the globally least recently used page.

Given that the uncompressed cache *always* holds more recently-touched pages than the compressed cache, this scheme requires a bias to ensure that the compressed cache had any memory at all. We believe that this form of adaptivity can be maladaptive, as no single bias can commensurate the timescales of an uncompressed cache and compressed cache, since those timescales are determined by varying reference behavior at each memory size. In other words, no single bias value will be appropriate, as the strength of the bias must depend on the locality of the reference behavior for each program.

6.4.1 Online Cost-Benefit Analysis

Our own adaptive cache-sizing mechanism addresses the issue of adaptation by performing the an online cost-benefit analysis, based on recency information decayed in a timescale relative manner. This calculation is similar to the one outlined in Section 6.2, and the approach to keeping the statistics is nearly identical the one used for EELRU in Section 4.3.1. Assuming that behavior in the relatively near future will resemble behavior in the relatively recent past, our mechanism actually keeps track of aspects of program behavior that bear directly on the performance of compressed caching for different cache sizes, and compresses more or fewer pages to improve performance.

In general, recency information allows us to estimate the cost and benefit of a compressed cache of a given size, regardless of what the current size of the compressed cache actually is, and regardless of which pages are currently in memory. That is, we can do a "what if" analysis to find out if the current split of memory between uncompressed and compressed caches is a good one, and what might be a better one. We can simply count the number of touches to pages in different regions of the LRU queue, and interpret those as hits or misses relative to different sizes of uncompressed cache and corresponding sizes of compressed cache.

Multiple target sizes. We generalize the above scheme by using several different "target" compressed cache sizes, just as we used multiple potential early eviction points with EELRU. The adaptive component of our system computes the costs and benefits of each of the target sizes, based on recent counts of touches to regions of the LRU ordering, and chooses the target size with the lowest cost. Then the compressed cache size is adjusted in a *demand-driven* way: memory is compressed or uncompressed only when an access to a compressed page (either in compressed

RAM or on disk) occurs.

Actually, our system chooses a target *uncompressed* cache size, and the corresponding overall effective cache size is computed based on the number of page frames left for the compressed cache, multiplied by an estimate of the compression ratio for recently compressed pages. This means that the statistics kept by our adaptivity mechanism are not exact, as our past information may contain hits that are in a different recency region than that indicated by the current compressibility estimate. Nevertheless, this coarse-grained approach does not seem to matter much in our simulations; approximate statistics about which pages have been touched, and how recently they have been touched, are quite sufficient. Therefore, the system does not need to be sensitive to the details of the replacement policy used for the uncompressed cache; any normal LRU approximation should be sufficient.

6.4.2 Adapting to Recent Behavior

To adapt to recent program behavior, our statistics are decayed exponentially with virtual memory time. Events that advance this time are reference to pages that could affect our cost benefit analysis (i.e., would have been hits if we had compressed as much memory as any of our target compression sizes currently suggests).

Additionally, the decay factor used is inversely proportional to the size of memory (total number of page frames), so that time typically advances more slowly for larger memories than for small ones—small memories usually have a shorter replacement cycle, and need to decay their statistics at a faster rate than larger ones.

Extensive simulation results show that this strategy works as intended: our adaptivity ensures that for any memory size, the cache responds to changes in the recent behavior of a program relatively quickly, so that it can benefit from reasonably persistent program behavior. Moreover, it does not respond not so quickly that it is continually “distracted” by short duration behaviors.

A single setting of the decay factor (relativized automatically to the memory size) works well across a variety of programs, and across a wide ranges of memory sizes. Attempts to use different decay factors demonstrated that the adaptive heuristic was not particularly sensitive to this choice of factor. The use of the correct timescale is more critical to the rate of decay than the constant factor used.

6.5 Size-Adaptive Compressed Cache Simulations

We implemented this approach to adaptivity in a simulator. The test suite, as well as the traces gathered from that test suite, are a superset of those for the fixed-boundary compressed cache simulations, as two new programs were added. We performed experiments with the same set of compression algorithms, namely **WKdm**, **WK4x4**, **LZO**, and **LZRW1**. We also used the same three processors, specifically the **180 MHz Pentium Pro**, the **168 MHz UltraSPARC**, and the **300 MHz UltraSPARC**. By simulating a large range of physical memory sizes for each program, we have found that the adaptive method incurs little overhead, and for the vast majority of memory sizes, it chooses a compressed cache size that reduces paging time.

Two more programs in the test suite. Note that for these experiments, we added to the test suite two programs that have larger footprints:

- **Gnuplot** is a plotting program that can create two- and three-dimensional plots of functions and data files. As an input, we provided a 12 Mbyte file of two-dimensional data, and used it to generate a simple scatter plot. This input file was provided by Gideon Glass, and is the same one used for experiments on the page replacement algorithm *SEQ* [GC97].

Program name	**P-Pro 180 MHz**	**SPARC 168 MHz**	**SPARC 300 MHz**
gnuplot	46.89	32.99	20.61
rscheme	8.26	11.77	7.59
espresso	10.07	12.35	7.41
gcc	9.89	14.66	9.41
ghostscript	18.95	26.89	16.84
p2c	2.38	2.91	2.08
grobner	0.53	0.87	0.49
lindsay	7.74	11.01	6.69

Table 6.1: The processing times (in seconds) for each program in the test suite on each processor used in this study. If enough memory is available such that no paging occurs, these times will be the turnaround times.

- **RScheme** is bytecode-based interpreter for the *Scheme* programming language. As an input, we provided code that performed a number of lattice manipulation functions. Its performance is dominated by the runtime of a generational garbage collector.

Excluding one compression algorithm. We present the results for all but WK4x4, as that algorithm is very similar to WKdm, but its slowness yielded consistently lesser performance. We do believe, though, that a version of this algorithm that used our fast packing technique could perform better than WKdm in some cases.

The runtimes of the test suite. Our results are presented in terms of time spent paging, but it is helpful to know the processing time required to execute each program in the test suite. Table 6.1 shows the number of seconds required to execute each of our six programs on each of the three processors, when no paging occurs. These times can be added with paging time information to obtain total runtime time for a given architecture, memory size, and virtual memory configuration.

Simulation parameters. We used four different target compression sizes with values equal to 10%, 23%, 37%, and 50% of the simulated memory size. Thus, during persistent phases of program behavior (i.e., when the system has enough time to adapt) either none, or 10%, or 23%, or 37%, or 50% of our memory pages are holding compressed data. We chose this range of sizes, from 10% to 50%, based on what we had observed from the fixed-size compressed caching simulations in Section 6.3. Specifically, a compressed cache small than about 10% would rarely provide significant benefit, and one larger than about 50% would require too many compressions and decompressions to reduce paging time.

Limiting the number of target compression sizes to four guarantees that our cost-benefit analysis incurs a low overhead. This small number of target sizes also demonstrates that our adaptive heuristic is not sensitive to specific parameters, and that even a coarse-grained approximation will perform well. It is also a form of conservatism, as a few programs could benefit from larger compressed caches.

The decay factor used is such that the M-th most recent event (with M being the size of memory) has a weight equal to 20% of the most recent event. Our results were not particularly sensitive to the exact value of the decay factor, and this choice of decay factor worked well for every memory size with which we experimented.

In Section 6.5.2, we will examine the effect of using a faster disk (down to a seek time of 0.625 ms).

Allocation and fragmentation. This simulator does not consider the space lost to fragmentation as a result of variable-sized compressed pages. We believe that the fragmentation can be kept very low, and not significantly reduce the benefits shown by our simulator.

Compressed pages need not be stored contiguously. Instead, they could be held a linked list or similar data structure. The overhead introduced by this data structure would be small. Furthermore, the allocation for these compressed pages would be handled no differently than the allocation for other objects. In practice, fragmentation has been shown to be low for real programs [WJNB95], usually at most a few percent, and the allocation of compressed pages should be no different.

Another alternative would be to periodically compact the space used to store compressed pages. Such periodic compaction of space to eliminate fragmentation could be done in less time than is required for a single disk fault, during which hundreds of millions of processor cycles pass. Therefore, the space lost to fragmentation of a compressed cache can be kept low, and should not significantly reduce the benefits of having one.

6.5.1 Results of Detailed Simulations

Wide Range Results

For each of our test programs, we chose a wide range of memory sizes to simulate. The plots of this section show the entire simulated range for each program. Subsequent sections, however, concentrate on the *interesting* region of memory sizes. This range begins around the size where a program spends 90% of its time paging and 10% of its time executing on the CPU, and ends at a size where the program causes very little paging. Here, however, the range begins at memory sizes so small that the runtime would be hundreds to thousands of times slower, due to paging, than it would be if there were enough RAM to hold all of the program's data.

Figures 6.15 through 6.20 show log-scale plots of the paging time of each of our programs as a function of the memory size. Each line in the plot represents the results of simulating a compressed cache using a particular algorithm on a SPARC 168 MHz machine. The paging time of a regular LRU memory system (i.e., with no compressed cache) is shown for comparison. We can see that compressed caching yields benefits for a very wide range of memory sizes, indicating that our adaptivity mechanism reliably detects locality patterns of different sizes. Note that all com-

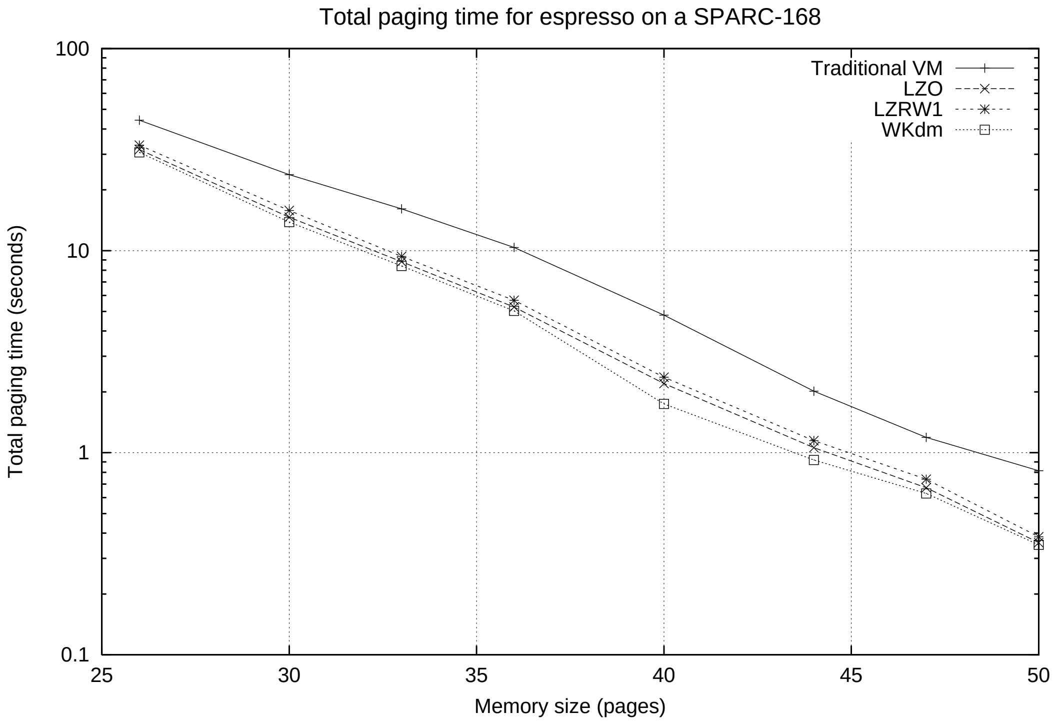

Figure 6.15: Results from simulating a compressed cache whose size is adapted for **espresso** across a wide range of memory sizes. *(Note that the y-axis is on a log scale.)*

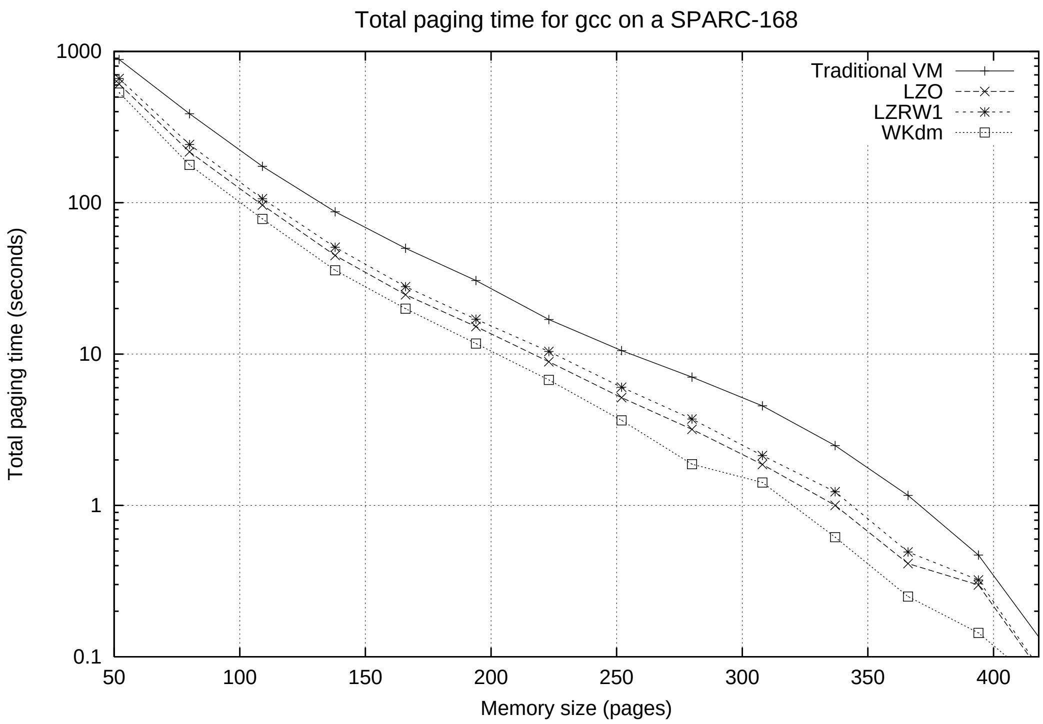

Figure 6.16: Results from simulating a compressed cache whose size is adapted for **cc1** across a wide range of memory sizes. *(Note that the y-axis is on a log scale.)*

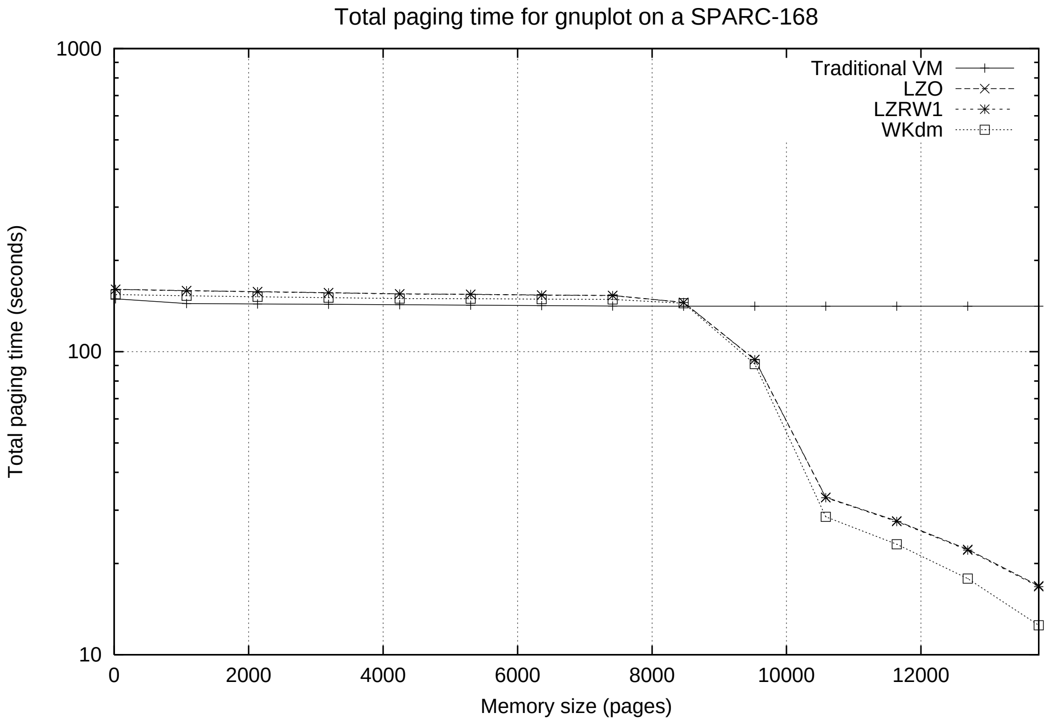

Figure 6.17: Results from simulating a compressed cache whose size is adapted for **gnuplot** across a wide range of memory sizes. *(Note that the y-axis is on a log scale.)*

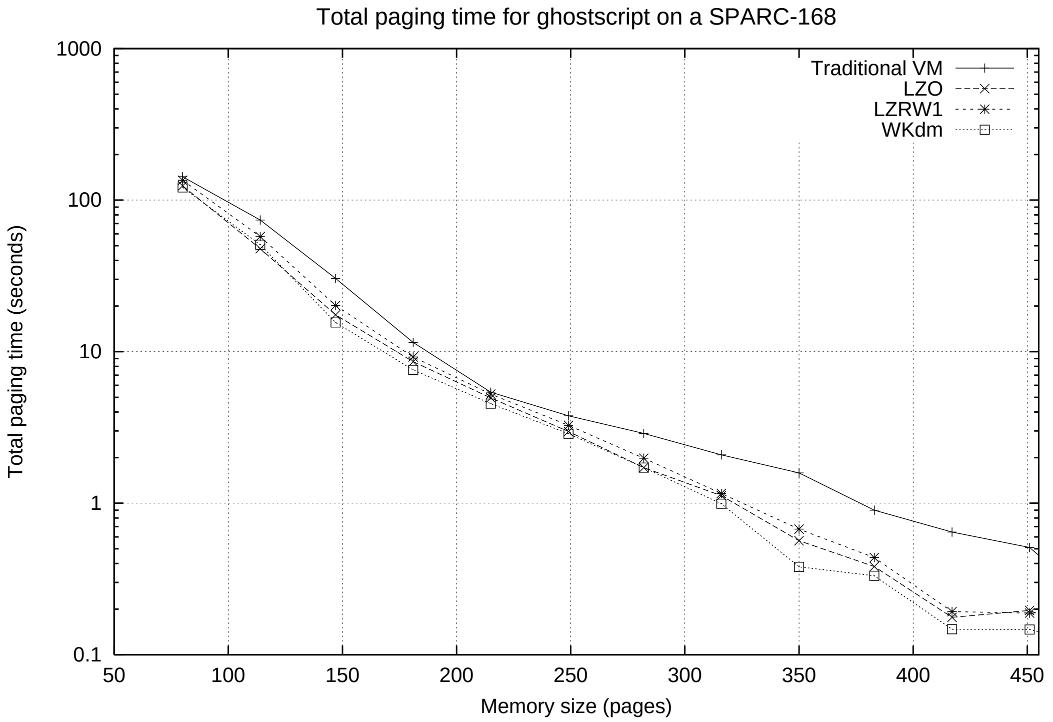

Figure 6.18: Results from simulating a compressed cache whose size is adapted for **Ghostscript** across a wide range of memory sizes. *(Note that the y-axis is on a log scale.)*

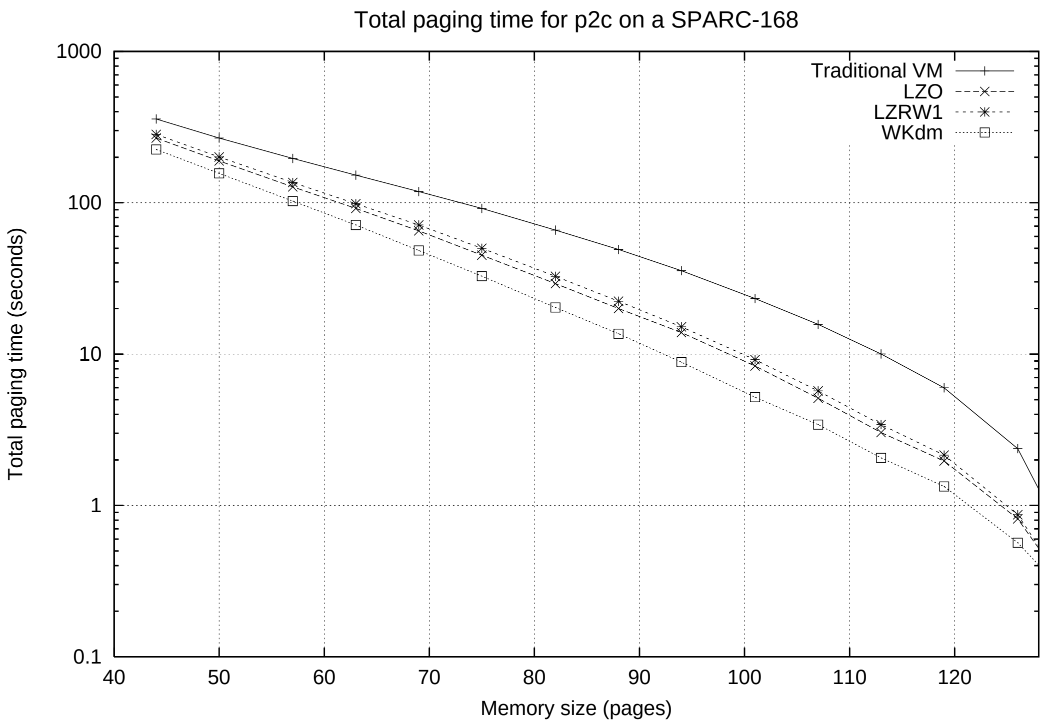

Figure 6.19: Results from simulating a compressed cache whose size is adapted for **p2c** across a wide range of memory sizes. *(Note that the y-axis is on a log scale.)*

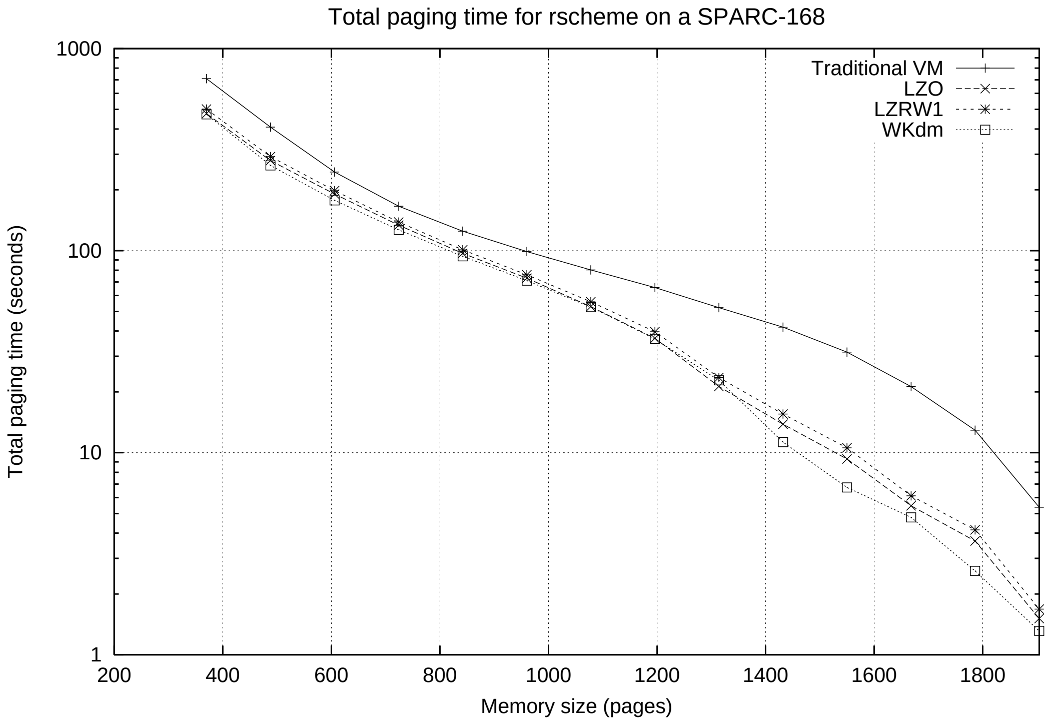

Figure 6.20: Results from simulating a compressed cache whose size is adapted for **RScheme** across a wide range of memory sizes. *(Note that the y-axis is on a log scale.)*

pression algorithms exhibit benefits, even though there are definite differences in their performance.

These figures only aim to convey the general idea of the outcome of our experiments. The same results are analyzed in detail in subsequent sections (where we isolate interesting memory regions, algorithms, architectures, and trends).

Normalized Benefits and the Effect of Compression Algorithms

We would like to quantify the benefits obtained by using compressed caching and to identify the effect of different compression algorithms on the overall system performance. It is hard to see this effect in plots that use a log scale for the y-axis, as they seem to indicate that all compression algorithms obtain similar results.

A more detailed plot reveals significant variations between algorithm performance. Figures 6.21 through 6.26 show plots of the normalized paging times for different algorithms in the interesting region. (Recall that this usually begins at the size where a program spends 90% of its time paging and 10% of its time executing on the CPU, and ends at a size where the program causes very little paging). By "normalized paging time" we mean the ratio of paging time for compressed caching over the paging time for a regular LRU replacement policy; the 100% line represents the paging time of a traditional virtual memory system.

All algorithms obtain significant benefit over traditional virtual memory systems for the interesting ranges of memory sizes. Benefits of over 40% are common for large parts of the plots shown. At the same time, losses are rare (only exhibited for gnuplot) and small. For gnuplot, the maximum loss in paging performance is about 15%. Roughly half of that loss is due to our conservative assumption that all disk traffic be compressed, even if no main memory is allocated for compressed cache use. The other half of that loss can be attributed to the adaptivity mechanism, which attempted to allocate a few pages to the compressed cache, only to

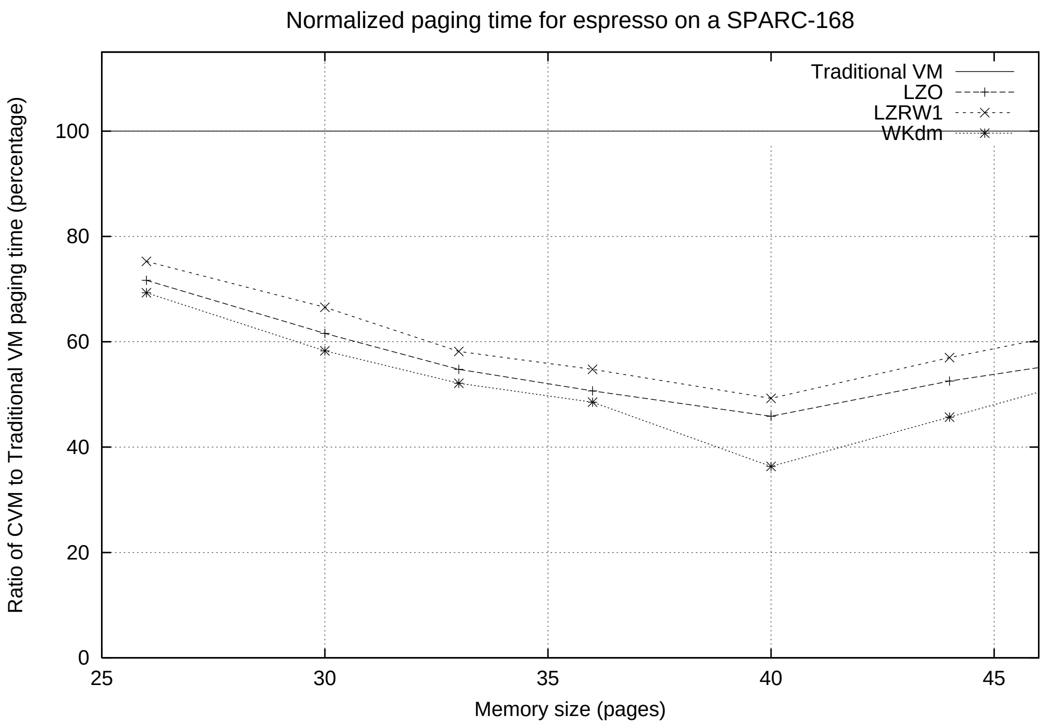

Figure 6.21: Normalized paging times for different compression algorithms with **espresso**. Even though all algorithms yield benefits, some are significantly better than others.

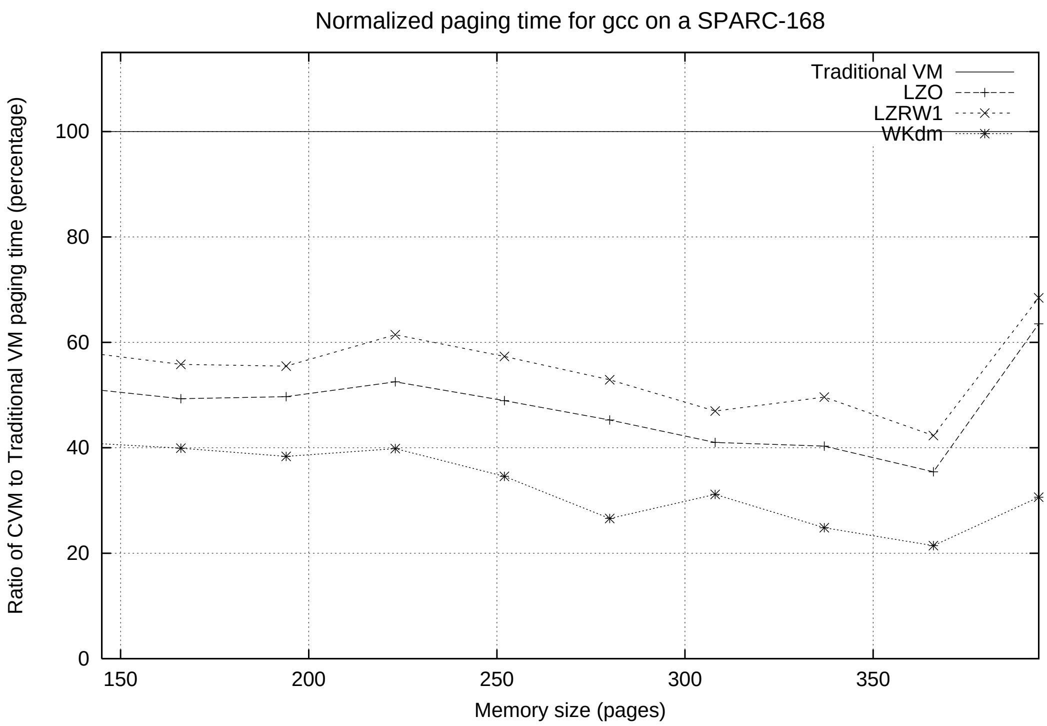

Figure 6.22: Normalized paging times for different compression algorithms with **cc1**. Even though all algorithms yield benefits, some are significantly better than others.

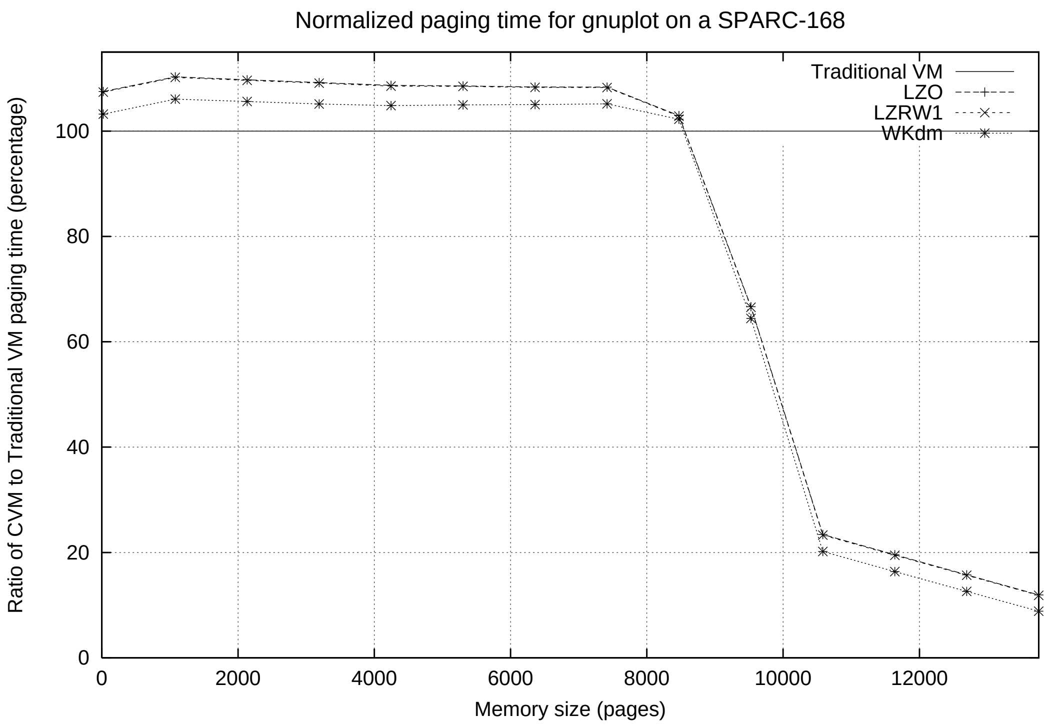

Figure 6.23: Normalized paging times for different compression algorithms with **gnuplot**. Even though all algorithms yield benefits, some are significantly better than others.

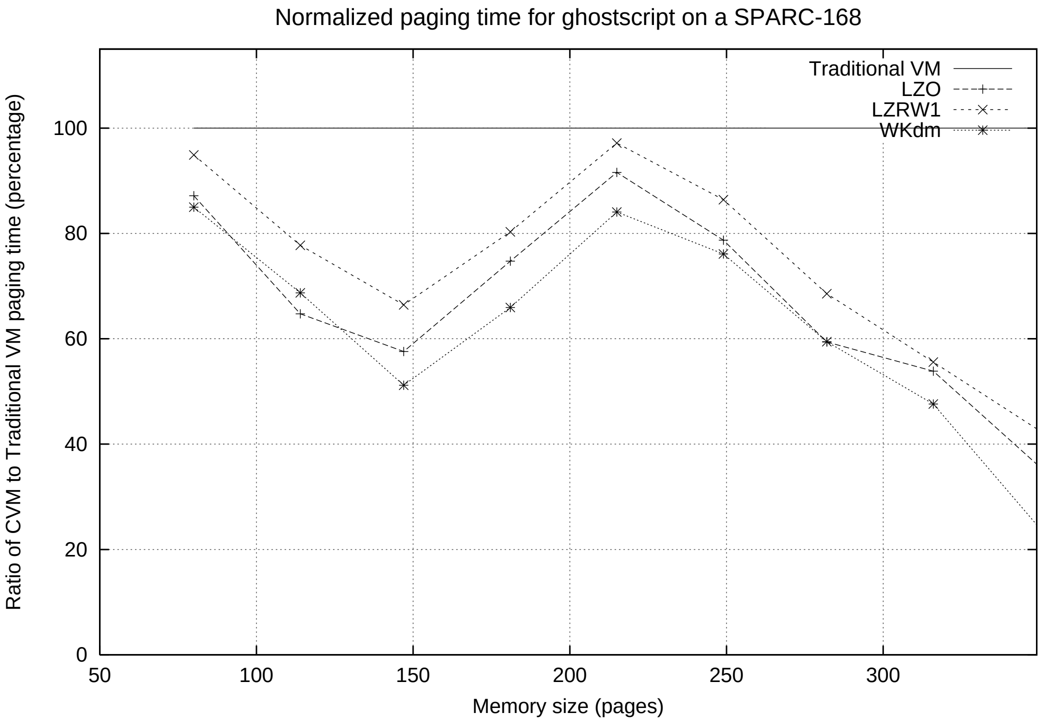

Figure 6.24: Normalized paging times for different compression algorithms with **Ghostscript**. Even though all algorithms yield benefits, some are significantly better than others.

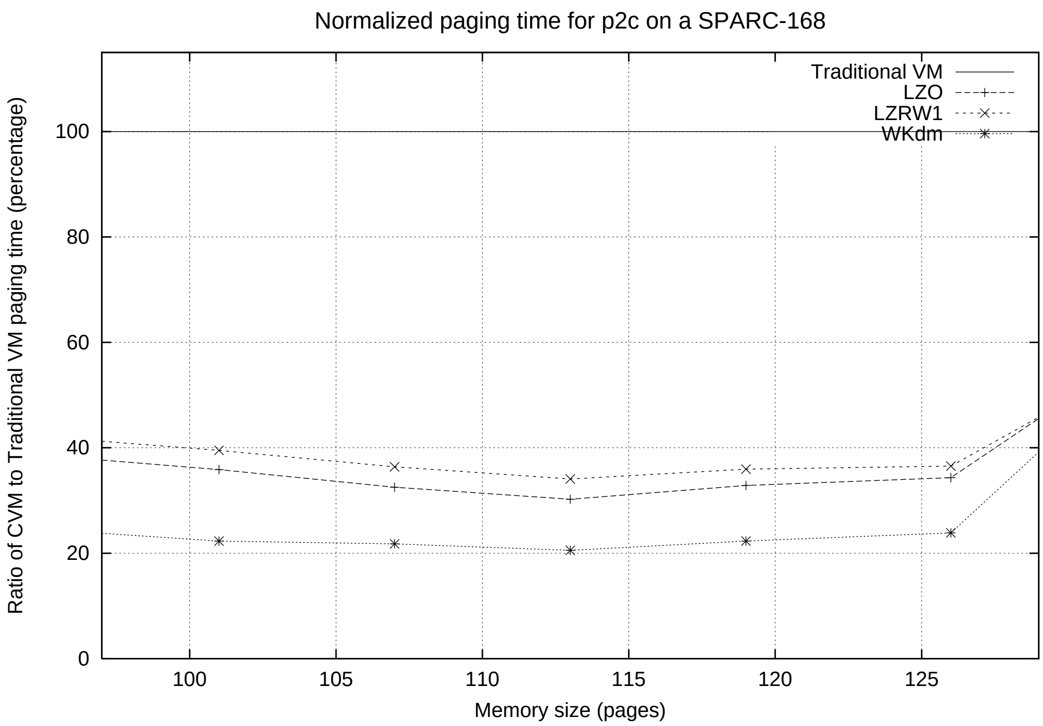

Figure 6.25: Normalized paging times for different compression algorithms with **p2c**. Even though all algorithms yield benefits, some are significantly better than others.

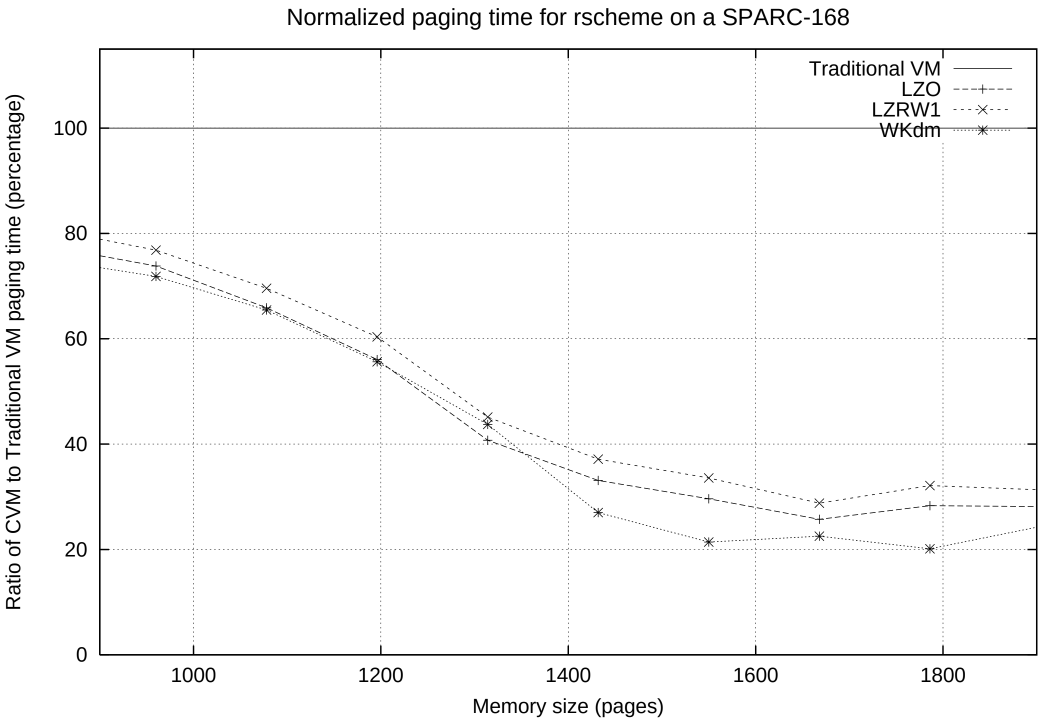

Figure 6.26: Normalized paging times for different compression algorithms with **RScheme**. Even though all algorithms yield benefits, some are significantly better than others.

discover that it could not help. This latter cost could be eliminated by refining the adaptivity implementation.

Also note that losses diminish for faster compression algorithms. That is, when our adaptivity does introduce overhead, that cost can be reduced by having a fast compression algorithm, since it is a direct function of performing unnecessary compressions and decompressions.

Gnuplot is an interesting program to study more closely. The program stores highly compressible data (exhibiting a ratio of over 4:1 on average). This way, the adaptive heuristic can consider large effective memory sizes, expecting that it can compress enough pages so that all the required data remains in main memory. Nevertheless, gnuplot's running time is dominated by a large loop iterating only twice on a lot of data. Hence, for small memory sizes the behavior that the compressed caching policy tries to exploit ends before any benefits can be seen. Further, our choice of possible uncompressed cache sizes limits the compressed cache to being 50% of the available RAM. For this program, an even larger compressed cache could have been used to capture the entire working set into a small total memory. For larger memory sizes, however, the benefit can be substantial, reaching over 80%.

As shown in these plots, the performance difference of compressed caching under different compression algorithms can often be over 15%. Our WKdm algorithm achieves the best performance for the vast majority of data points, due to its speed and comparable reduction rates with LZO. The LZRW1 algorithm, used by Douglis, consistently yields the worst results. This fact, combined with the slow machine used (for current standards) are at least partially responsible for the rather disappointing results that Douglis observed.

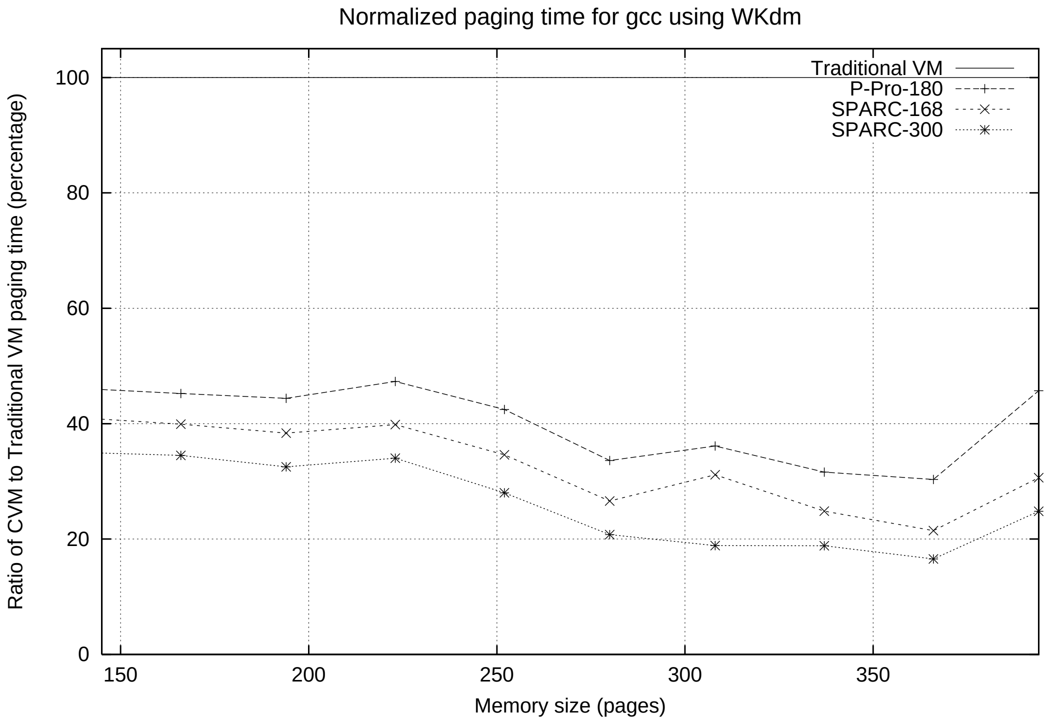

Figure 6.27: Normalized paging times for compressed caching on different processors with **cc1** and **WKdm**.

Implementation and Architecture Effects

In the past sections, we only showed results for our SPARC 168 MHz machine. As expected, the faster SPARC 300 MHz machine has a lower compression and decompression overhead and, thus, should perform better overall. The Pentium Pro 180 MHz machine is usually slower than both SPARC machines in compressing and uncompressing pages (not unexpectedly, as it is an older architecture—see also out later remarks on memory bandwidth).

Figures 6.27 through 6.32 show three of our test programs simulated under WKdm and LZO in all three architectures. For WKdm, the performance displayed

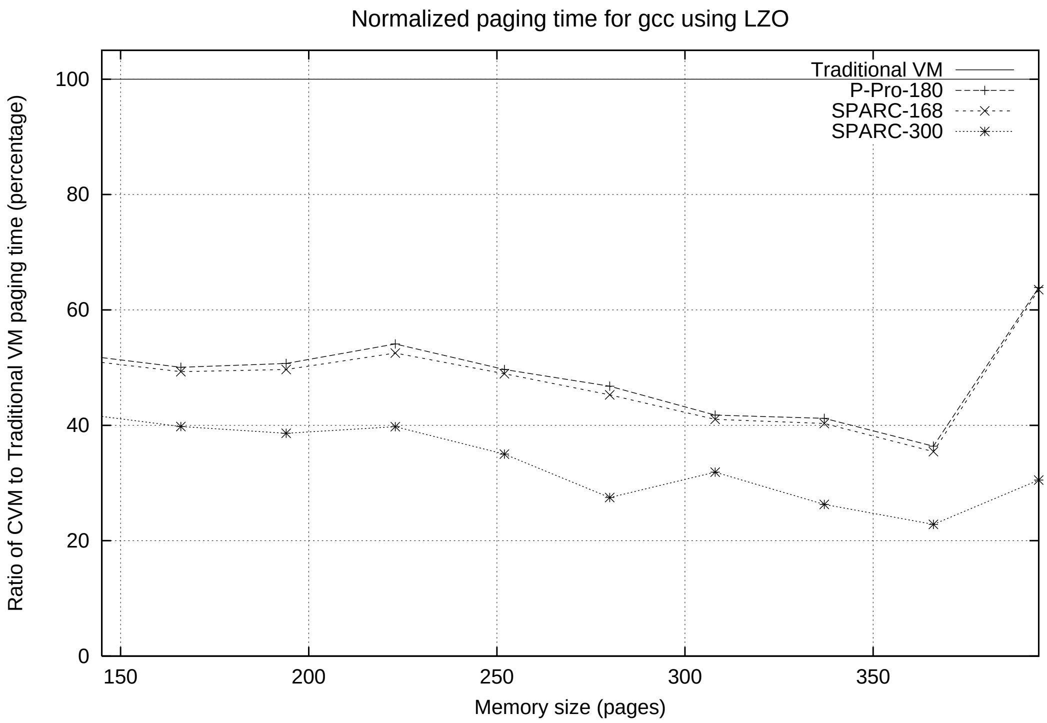

Figure 6.28: Normalized paging times for compressed caching on different processors with **cc1** and **LZO**.

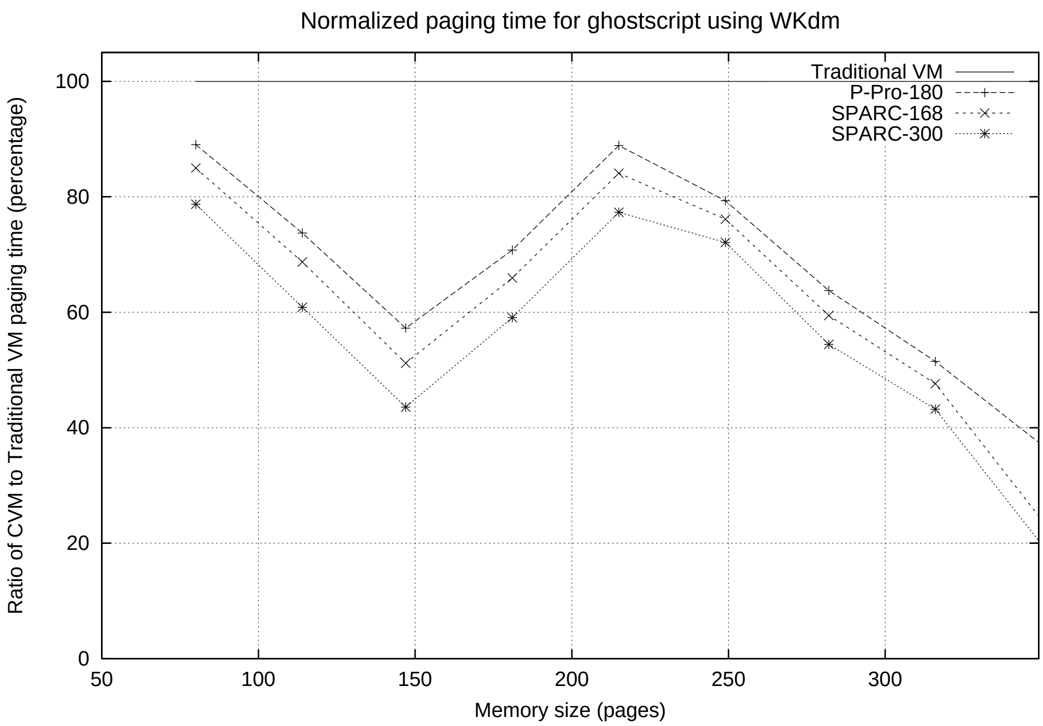

Figure 6.29: Normalized paging times for compressed caching on different processors with **Ghostscript** and **WKdm**.

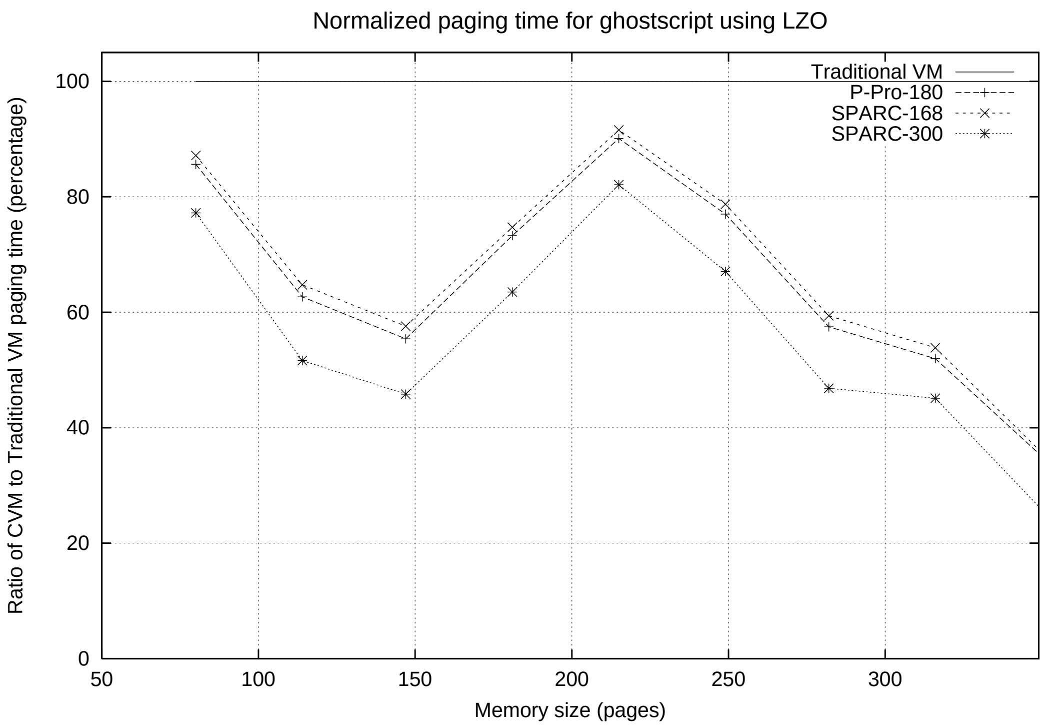

Figure 6.30: Normalized paging times for compressed caching on different processors with **Ghostscript** and **LZO**.

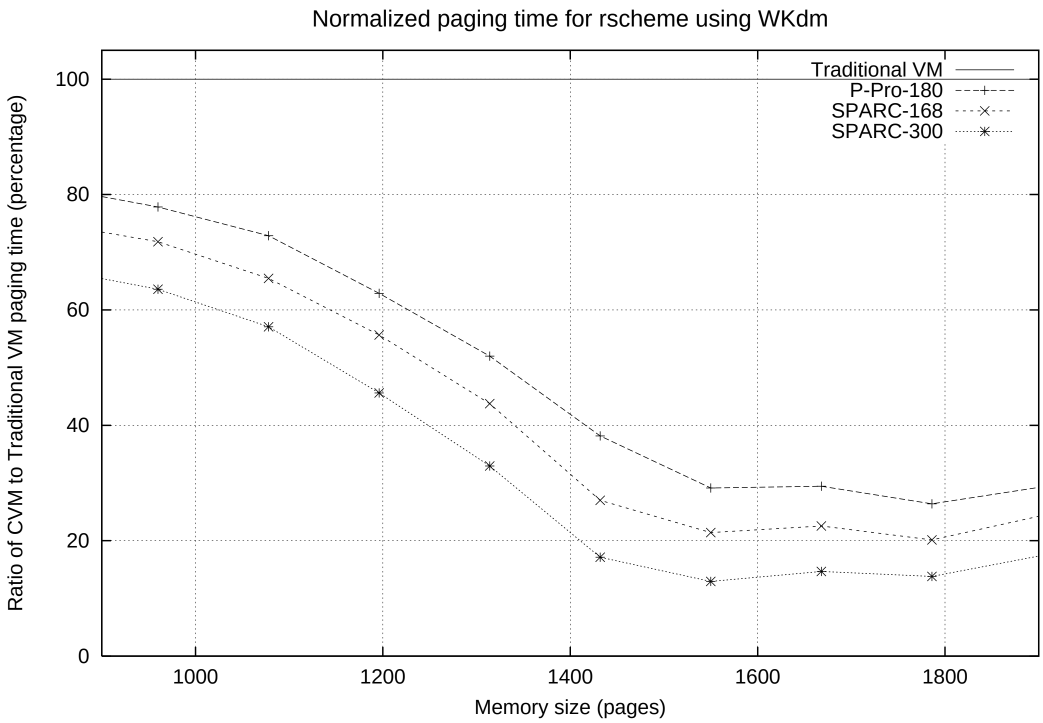

Figure 6.31: Normalized paging times for compressed caching on different processors with **RScheme** and **WKdm**.

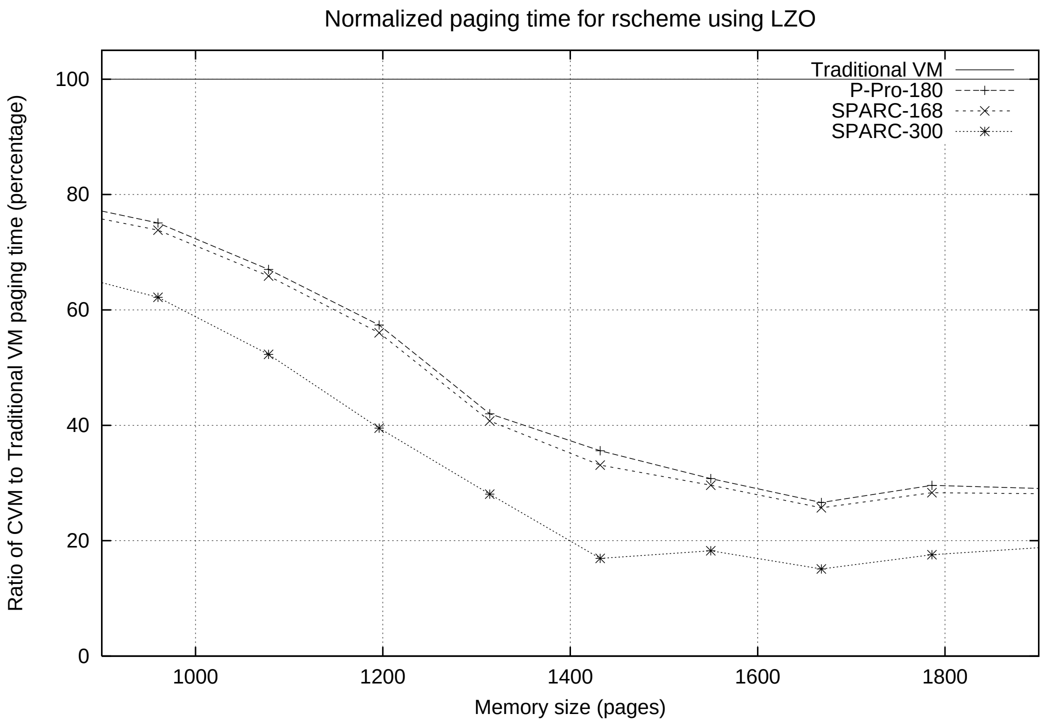

Figure 6.32: Normalized paging times for compressed caching on different processors with **RScheme** and **LZO**.

agrees with our observations on machine speeds. Nevertheless, the performance of LZO is significantly better on the Pentium Pro 180 MHz machine than one would expect based on the processor speed alone. LZO performs relatively better on this particular processor because, as pointed out earlier, it is hand optimized for speed in Intel x86 assembly language. Perhaps surprisingly, the effect of the optimization is quite significant, as can be seen. For Ghostscript, for instance, the Pentium Pro is faster than the SPARC 168 MHz using LZO.

6.5.2 Technology Trends

Is Memory Bandwidth a Problem?

Compressed caching mostly benefits from the increases of CPU speed relative to disk latency. Nevertheless, a different factor comes into play when disk and memory *bandwidths* are taken into account. A first observation is that moving data from memory takes at most one-third of the execution time of our WKdm compression algorithm. (This ratio is true for both the Pentium Pro 180 MHz machine, which has a slow memory subsystem, and the SPARC 300 MHz, which has a fast processor. It is significantly better for the SPARC 168 MHz machine.) Hence, memory bandwidth does not seem to be the limiting factor for the near future. Even more importantly, faster memory architectures (e.g., RAMBUS) will soon become widespread and compression algorithms can fully benefit as they only need to read contiguous data. The overall trend is also favorable. Memory bandwidths have historically grown at 40%, while disk bandwidths and latencies have only grown at rates around 20%.

Sensitivity Analysis

The cost and benefits of compressed caching are dependent on the relative costs of compressing (and decompressing) a page vs. fetching a page from disk. If compression is insufficiently fast relative to disk paging, compressed caching in virtual

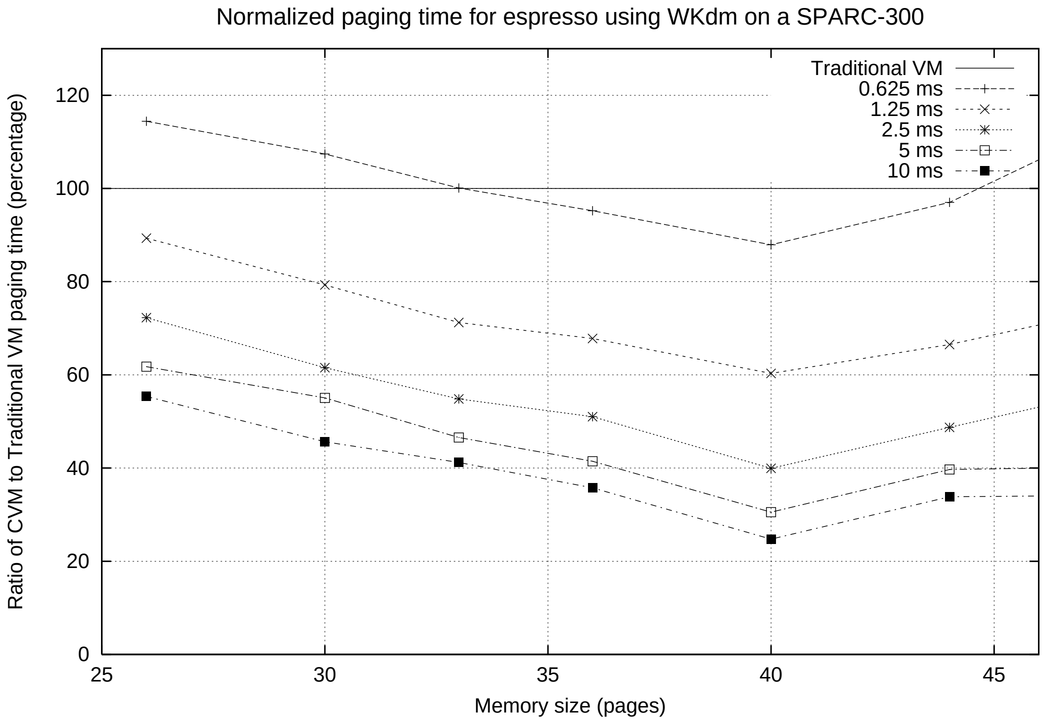

Figure 6.33: A sensitivity analysis of **espresso** with disks of various speeds. These plots can be interpreted not only for faster disks, but also for the cases of slower CPUs, perfect prefetching, and larger page sizes.

memory will not be worthwhile.

On the other hand, if CPU speeds continue to increase far faster than disk speeds, then compressed virtual memory will become increasingly effective and increasingly attractive. Over the last decade, CPU speeds have increased by about 60% a year, while disk latency and bandwidth have improved by only about 20% a year. This works out to an increase in CPU speeds *relative to disk speeds* of one third a year—or a doubling every two and a half years, and a quadrupling every five years.

Figures 6.33 through 6.38 show plots of simulated performance of our adap-

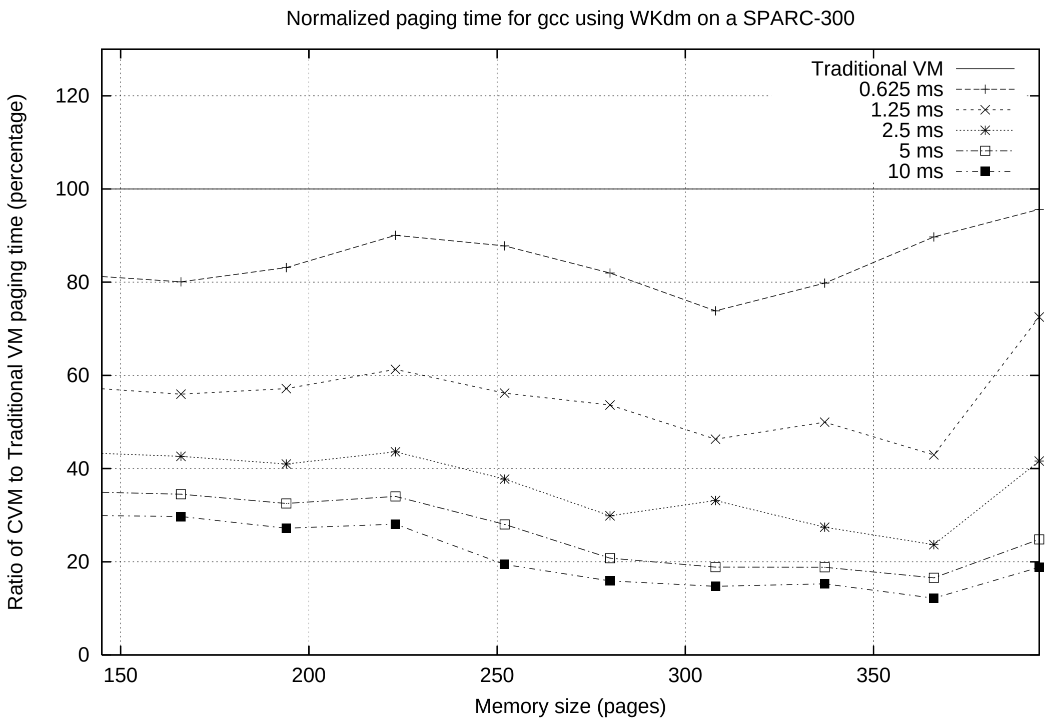

Figure 6.34: A sensitivity analysis of **cc1** with disks of various speeds. These plots can be interpreted not only for faster disks, but also for the cases of slower CPUs, perfect prefetching, and larger page sizes.

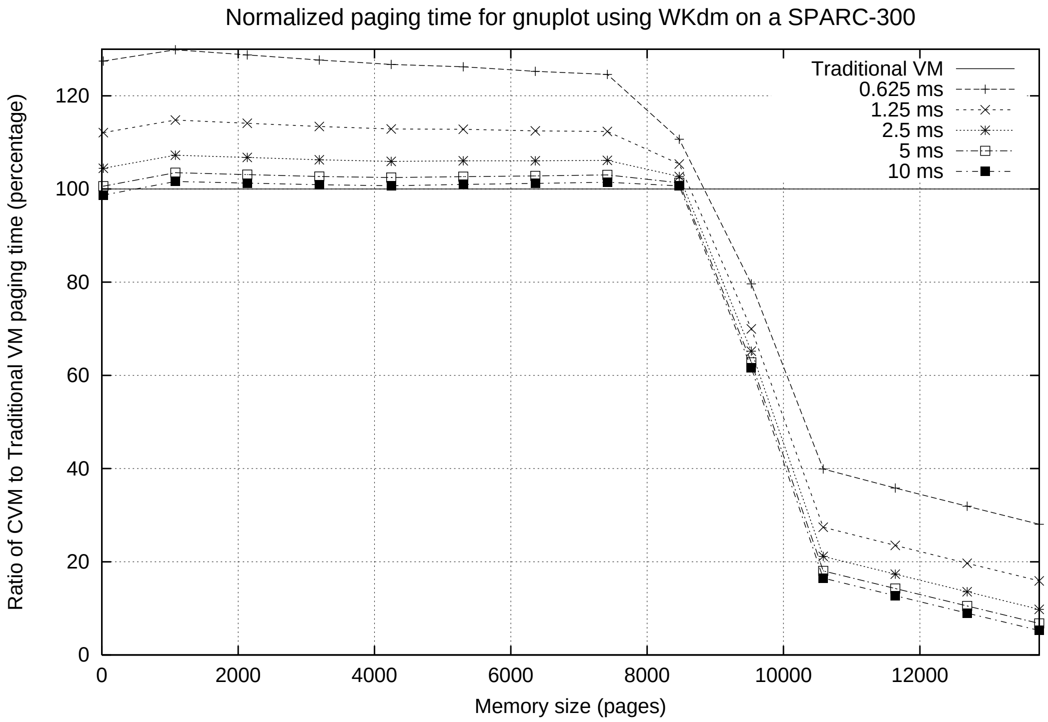

Figure 6.35: A sensitivity analysis of **gnuplot** with disks of various speeds. These plots can be interpreted not only for faster disks, but also for the cases of slower CPUs, perfect prefetching, and larger page sizes.

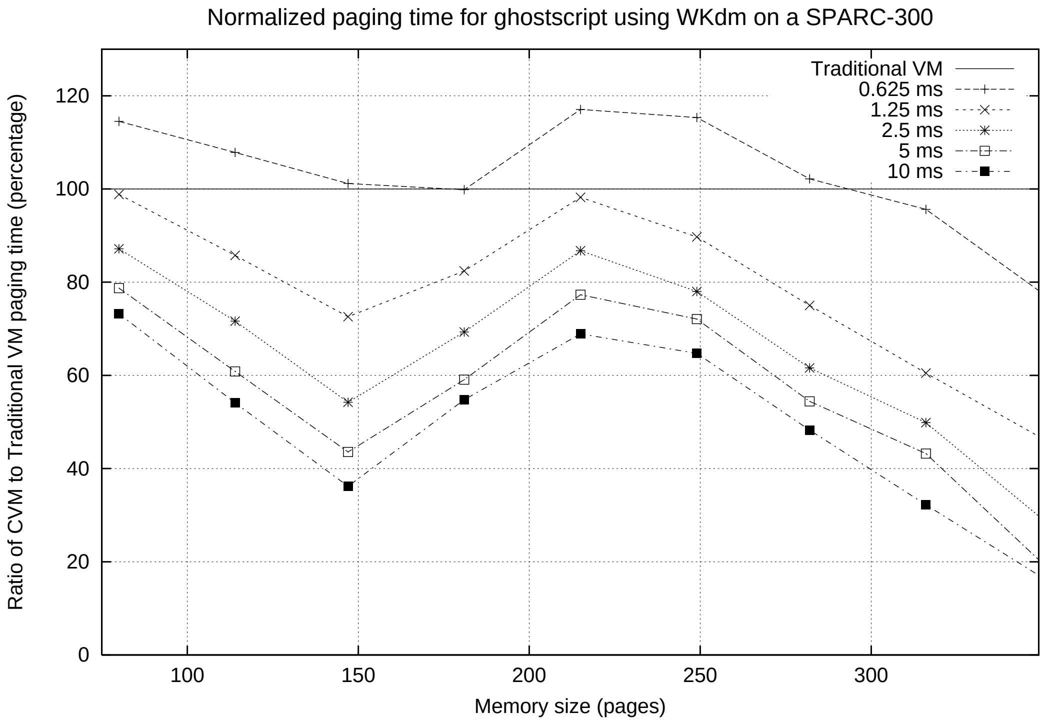

Figure 6.36: A sensitivity analysis of **Ghostscript** with disks of various speeds. These plots can be interpreted not only for faster disks, but also for the cases of slower CPUs, perfect prefetching, and larger page sizes.

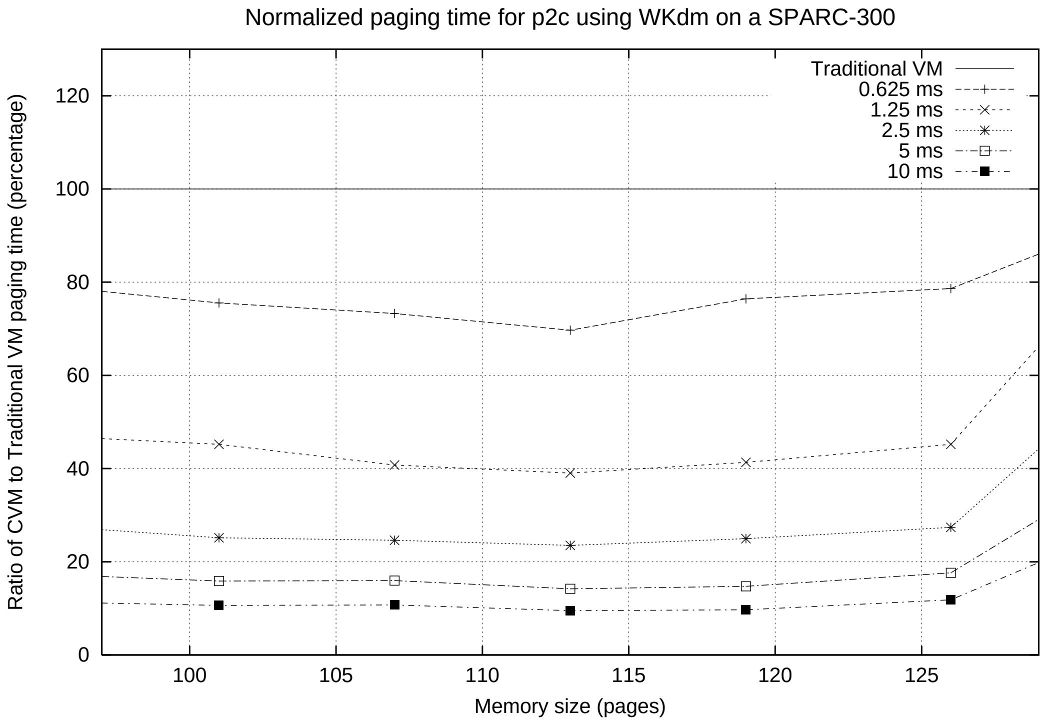

Figure 6.37: A sensitivity analysis of **p2c** with disks of various speeds. These plots can be interpreted not only for faster disks, but also for the cases of slower CPUs, perfect prefetching, and larger page sizes.

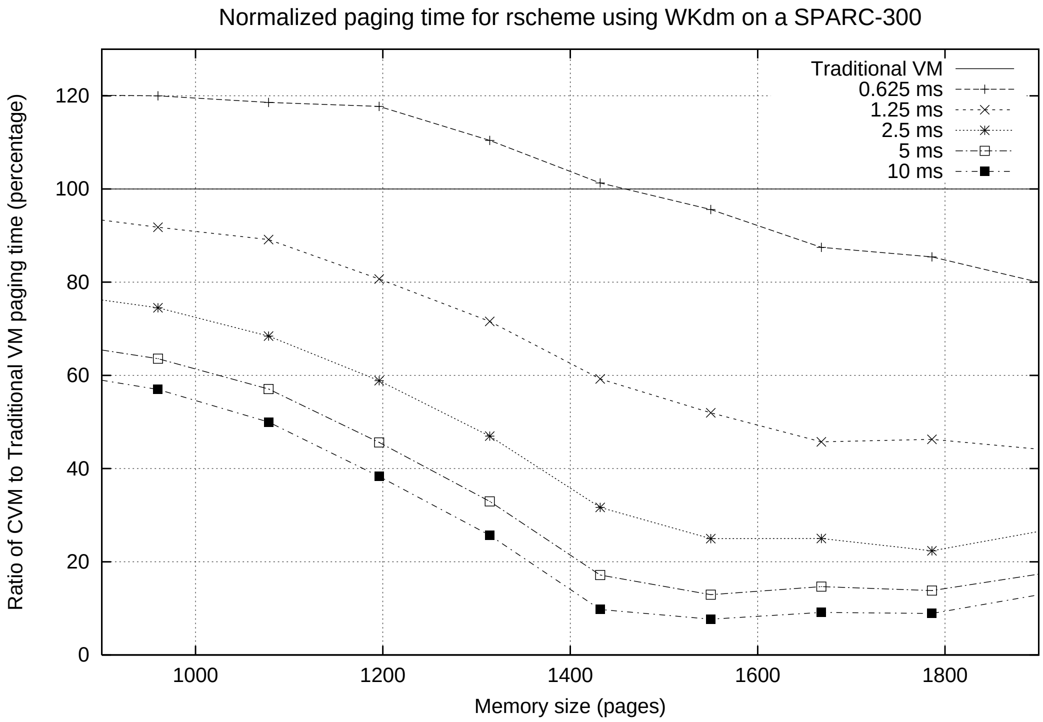

Figure 6.38: A sensitivity analysis of **RScheme** with disks of various speeds. These plots can be interpreted not only for faster disks, but also for the cases of slower CPUs, perfect prefetching, and larger page sizes.

tive caching system, using page compression timings measured on a 300 MHz UltraSPARC. Each line represents the paging costs for simulations using a given disk fault cost. Costs are normalized to the performance of a conventional LRU memory with the *same* disk page access time; that is, each curve represents the speedup or slowdown that comes from using compressed caching.

The middle line in each plot can be regarded as the performance of a machine the speed of a 300 MHz UltraSPARC with an average page fetch cost (for 4 Kbyte pages) of only 2.5 ms, about one half the average disk seek time of a fast disk. Note that, in normalized performance terms, assuming a disk that is twice as fast is exactly equivalent to assuming a CPU that is twice as slow. At the same time, studying the case of a fast disk conservatively covers the case of perfect prefetching of multiple pages (a disk that is twice as fast is equivalent to prefetching always the next page with one seek). This, in turn, conservatively covers the case of using larger page sizes. Hence, our sensitivity analysis (taking into account fast disks) also subsumes many other scenarios.

Looking at the middle line of each plot, we can see that with a disk page access cost of 2.5 ms, most programs show a reduction of paging times by 30 to 70 percent, averaged across the interesting range of memory sizes. Thus, compressed caching is a very clear win even for a disk access cost of 2.5 ms per 4 Kbyte page. The line above the middle one can be taken to represent a system with the same CPU speed and disk costs a factor of two lower, at 1.25 ms per 4 Kbyte page. Even though performance for this system is significantly worse, still much speedup is obtained. The top line represents a system where disk page accesses cost only 0.625 ms per 4 Kbyte page. For some programs, this degrades performance overall to the point that compressed caching is not clearly worthwhile.

For a different perspective, we can look at the next lower line to see the performance of a system with a disk of the same speed and a processor that is twice as fast. For most of our programs, each doubling of CPU speed offers a significant additional speedup, typically decreasing remaining paging costs by ten to forty percent.

6.6 Related Work

While we have not addressed the compression of machine code, others have shown that it is possible to compress machine code by a factor of 3 using a specially tuned version of a conventional compressor [Yu96] and by as much as a factor of 5 using a compressor that understands the instruction set [EEF+97]. We believe that similar techniques can be made very fast and achieve a compression ratio of at least 2, similar to the ratios we get for data. This ratio is within 20% of the size reduction found by Cogswell and Russinovich using an extremely fast, simple, and untuned "general purpose" compression algorithm on both code and data pages [RC96].

The significant previous study by Douglis, mention earlier, was performed by implementing a compressed caching scheme in the virtual memory system for the Sprite operating system. He evaluated it on a DECStation 5000, which is roughly 3 to 10 times slower than the machines we used in our experiments.

Douglis' results were mixed, in that compressed caching was beneficial for some programs and detrimental to others. As should be apparent from our discussion of performance modeling, we believe that this was primarily due to the slow hardware (by today's standards) used. This is supported by our sensitivity analysis, which showed that a machine that is eight times slower than a 300 MHz UltraSPARC would yield mixed results, even with better compression algorithms than those available to Douglis.

As discussed earlier, Russinovich and Cogswell's study [RC96] showed that a simple compressed cache was unlikely to achieve significant benefits for the PC application workload they studied. Nevertheless, their results do not seem to accurately reflect the trade-offs involved. On one hand, they reported compression overheads that seem unrealistically low (0.05 ms per compression on an Intel 80486 DX2/66, which is improbable, given both the process speeds and the memory bandwidth limitations). But the single factor most responsible for their results is the very high overhead for handling a page fault that they incurred (2 ms—this is overhead not containing the actual seek time). This overhead is certainly a result of using a slow processor but it is possibly also an artifact of the OS used (Windows 95) and their implementation.

A study on compressed caching, performed in 1997, but only very recently published, was done by Kjelso, Gooch, and Jones [KGJ99]. They, too, used simulations to demonstrate the efficacy of compressed caching. Additionally, they addressed the problem of memory management for the variable-size compressed pages. Their experiments used the LZRW1 compression algorithm in software and showed for most programs the same kinds of reduction in paging costs that we observed. These benefits become even greater with a hardware implementation of their X-match algorithm.

Kjelso, Gooch, and Jones did not, however, address the issue of adaptively resizing the compressed cache in response to reference behavior. Instead, they assumed that it is always beneficial to compress more pages to avoid disk faults. This is clearly not true as when more pages are compressed, many more memory accesses may suffer a decompression overhead, while only a few disk faults may be avoided. The purpose of our adaptive mechanism is to determine when the trade-off is beneficial and compression should actually be performed. Kjelso, Gooch, and Jones did acknowledge that some compressed cache sizes can damage performance. Indeed, their results strongly suggest the need for adaptivity: two of their four test programs exhibit performance deterioration under software compression for several memory sizes.

The most recent study in this area was performed by Cervera, Cortes, and Becerra [CCB99]. This group inserted a cache of compressed pages between an uncompressed cache and the disk. They do not consider this cache a "compressed cache" because pages that are fetched from disk are decompressed directly into the uncompressed cache, and no copy is kept in the compressed cache. As a matter of terminology, we disagree, as this approach to managing the fetching and decompression of disk faults is identical to ours, and still constitutes a "compressed cache", i.e. a RAM cache of compressed pages.

For their experiments, these researchers implemented a compressed cache in the Linux operating system on an x86 architecture. For compression, they used the LZO algorithm. As the other studies found, paging traffic was compressible, with pages being reduced to 50% of their original size, and sometimes much less. For their test suite, they observed anywhere from very slight loss of performance (4% overhead for attempting compressed caching) to great improvement (a factor of six improvement in runtime). On average, their runtime was reduced by 25% to 50%.

As with the Kjelso, Gooch, and Jones study, this one clearly addresses the need for adaptivity. They experimented with cache sizes that varied from 512 Kbytes to 2 Mbytes. A good choice of size yielded their beneficial results, but they note that the system was sensitive to this choice.

6.7 Conclusions

Compressed caching appears quite attractive on current machines, offering an improvement of tens of percent in virtual memory system performance. This improvement is largely due to increases in CPU speeds relative to disk speeds, but substantial additional gains come from better compression algorithms and successful adaptivity to program behavior.

For all of the programs we examined, on currently available hardware, a virtual memory system that uses compressed caching will incur significantly less paging cost. Given memory sizes for which running a program suffers tolerable amounts of paging, compressed caching often eliminates 20% to 80% of the paging cost, with an average savings of approximately 40%. As the gap between processor speed and disk speed increases, the benefit will continue to improve.

The recency based approach to adaptively resizing the compression cache provides substantial benefit at nearly any memory size, for many kinds of programs. In our tests, the adaptive resizing provided benefit over a very wide range of memory sizes, even when the program was paging little. The adaptivity is not perfect, as small cost may be incurred due to failed attempts to resize the cache, but performs well for the vast majority of programs. Moreover, it is capable of providing benefit for small, medium, and large footprint programs.

Chapter 7

Conclusions

As CPU speeds increase by about 60% each year, and disk latencies decrease by less than 15%, the number of computational cycles wasted by virtual memory paging will grow. When the initial studies of virtual memory and paging were conducted, the performance cost due to a poor caching decisions was not nearly as high as it is now. Memory management is becoming a critical issue to the performance of many kinds of computation, and paging is the most expensive of memory management operations.

Virtual memory paging is known for the drastic performance decrease that it causes. There has been a widespread belief that the predictive tasks required by on-line caching policies could not provide the graceful performance degradation that off-line management techniques do. While no on-line technique will ever be able to match the omniscience of off-line policies, we have shown that graceful degradation is possible.

By considering the concept of timescale relativity, we have been able to develop a better understanding of the performance of virtual memory systems for different reference patterns. This undersanding, in turn, has allowed us to develop simple yet effective methods to recognize various reference patterns at runtime. We have applied these methods to EELRU and size-adaptive compressed caching so that they can adapt to reference behavior in a timescale relative way. This form of adaptivity has yielded promising results that not only improve paging performance, but provide the graceful degradation that on-line paging sorely needs.

Too many potential advances have been lost because a new virtual memory idea was evaluated through a real implementation, and the results were unduly influenced by unforseen and uncontrolled aspects of that particular system. Compressed caching, as an example, provided little or no benefit, although the idea was valid and the implementation and hardware were to blame. Furthermore, other new ideas that do yield benefit for a real implementation are rarely examined in more depth, and cause of their benefit is not well understood. In response to these problems with evaluation through real implemention, the techniques that we have presented here have been developed and understood through trace-driven simulation.

7.1 Timescale Relativity and Adaptive Caching

Throughout this dissertation, we have tied our analyses and developments to the concept of timescale relativity. "Time" for a virtual memory system depends on reference behavior and memory size. It can advance very quickly or slowly depending on the phases of program execution. The relationships between virtual memory time, reference time, and real time can vary greatly within a single program's execution. Because these timescales cannot be made commensurate, we need to study virtual memory events on the virtual memory timescale. While the idea of studying a phenomena at the appropriate timescale seems trivial, is has rarely been used to understand paging behavior.

Soonness. Page replacement is at the heart of any virtual memory system. In order to understand how a given replacement policy performs, it is crucial to understand what predictions it is making about the future use of pages. A replacement policy predicts, in some fashion, how soon it believes a page will be used. In order to understand what that prediction means, we must have the appropriate notion of "soonness". Further, we must determine what notion of soonness is being employed by a given policy. It is the timescale at which events are recorded, and to which a policy responds, that dictates what "soon" means. By describing the notion of soonness employed by on-line and off-line policies, we can understand the performance of each policy under different reference patterns.

Aggregate behavior. By viewing reference behavior through recency information, we are taking an aggregate view of reference patterns. While it may seem that a coarse-grained approach to past reference patterns would be inferior to more fine-grained, specific information, aggregation is actually a strength. There is a good deal of variance in how programs repeat reference patterns. By seeking only to predict which region of the recency axis will be accessed, we can respond generally to many different kinds of access patterns. More specific predictions, often based on per-page access histories, tend to be brittle and perform poorly if the underlying assumptions are incorrect. Concentrating on aggregate behavior can make a virtual memory system more robust.

Furthermore, aggregate behavior makes it much easier to identify coarse-grained patterns. By examining references at this higher, "chunked" level, we can characterize the patterns and how policies adapt to those patterns.

Decay and adaption to phases. One of the most important developments presented here is the ability to decay reference information in a timescale relative manner. Policies that give equal weight to all past references, such as LFU, fail to notice changes in phase because past phases dominate the available statistics. Methods that give greater weight to virtual memory events of the recent past adapt much more easily to phase changes.

EELRU and our size-adaptive compressed caching scheme both take advantage of this kind of informational decay. By using the information collected by an LRU queue, both for resident and non-resident pages, this form of adaptivity can detect reference patterns to which they can adapt. If this recency information were not decayed at the proper rate, then this form of adaptivity would either "jump" to respond to transient behavior, or fail to respond to persistent behavior. Decay must be tied to the virtual memory timescale. We have observed that the constant used to perform exponential decay is not nearly as important as the use of virtual memory time to determine how quickly information should age.

This form of adaptivity has been applied here in a simple fashion. We have used recency information to perform cost-benefit analyses that allow a virtual memory scheme to choose a more beneficial management approach. Aging the information so that the recency information reflects recent patterns causes the cost-benefit analysis to predict more reliably patterns that will persist into the near future.

This form of timescale-relative adaptivity has played a large role in making the graceful decay of paging performance possible. Recency information can reveal reference patterns that cause disastrous amounts of paging. By decaying this information, a system can detect phases that exhibit these reference patterns. EELRU and compressed caching can both improve the paging performance of programs that reference more memory than is available by adapting to such phases.

There are other uses for this kind of adaptivity that we have not yet addressed. This kind of analysis could be used to aid in the on-line adaptivity of variable space management. Given this kind of recency information, better decisions can be made about the effect that a space increase or decrease would have on a process. Further, early-eviction and compression provide interesting alternatives to the addition or subtraction of the space alloted to a process. These are topics of future work that could advance the sophistication of scheduling and variable space management policies for the first time in decades.

7.2 Trace Reduction and Simulation

The ideas presented here have been evaluated through trace-driven simulation. In order to determine the effects of different variables on a virtual memory system, we wanted to control the experiments. We wanted our experiments to be reproducible, as any good scientific experiment should be. We also wanted to experiment with a wide range of configurations. However, reference traces are difficult to collect and to share because of their size.

The length of the reference traces we needed to collect were large for two reasons: First, processors can now perform tens and hundreds of millions of memory references per second; even simple traces, blocked for pages, can grow to gigabytes in length for a few seconds of execution. Second, for a compressed caching study, we wanted traces augmented with images of each page referenced, multiplying the length of a simple reference trace by approximately 1,000.

Trace reduction methods based on statistically sampling were designed for hardware caches, not for virtual memory systems, and introduce greater error than desirable. Existing reduction methods designed for virtual memory studies were either limited in accuracy (Smith's Stack Deletion), output something that was not a reference trace (Coffman and Randell's behavior sequences, the Glass and Cao technique), or were lossless and did not reduce simulation time (PDATS, Mache, Lempel-Ziv compressors).

We developed SAD and OLR to overcome these limitations. They can be used to gather traces that will yield exact results for LRU memories of a user-chosen, minimal size. Traces reduced by SAD can also be used simulate the OPT policy exactly for memories no smaller than that minimal size. LRU and OPT are used for nearly every paging study, as LRU-like policies dominate real page replacement, and OPT provides a standard against which on-line policies can be measured. SAD and OLR also introduce less error into the simulation of other policies than SD, which has been cited and used for decades.

Thanks to significant reductions, SAD and OLR traces can make the simulation of ideas like compressed caching possible; even if simulation requires traces to store extensive information such as page images, storage and simulation time are tractible.

7.3 Compressed Caching

Compressed caching has been a discarded idea, deemed useful only for machines with very slow or non-existent backing stores. Consideration of the increasing gap between processor and disk speeds, as well as improvements in the compression of in-memory data, have led us to re-evaluate compressed caching. Our simulations have shown that a significant portion of a program's paging time can be eliminated.

In-memory data and domain-specific compression. The improvements in the compression of in-memory data have contributed a good deal to our positive results. The WK algorithms demonstrate two important ideas: First, there are very strong and simple regularities for in-memory data that that can be captured without computational expense. There are a number of improvements over the current WK algorithms and their implementations, but even our first attempts provided better compression than any other algorithm for many programs.

Second, the domain-specificity of WK demonstrates that "general purpose" compressors may be widely applicable, but they do not necessarily provide the tightest compression, nor the fastest execution. The importance of compression tailored to a specific domain has been lost in the proofs of optimality under assumptions about the input data, when those assumptions are not the most applicable, nor highest level models for many domains.

Adapting the compressed cache size. The timescale relative adaptation used by EELRU to detect patterns in reference recency was also applied to the problem of choosing a compressed cache size. This adaptivity introduced small overhead, while selecting compressed cache sizes that led to reduced paging for many different programs over a very large range of memory sizes.

The advantages provided by compressed caching should only improve as the gap between processor speed and disk speed grows. Compression and decompression will become less expensive. Not only could a greater number of compressed cache hits be accommodated, but also the adaptivity method could attempt to be more aggressive, as mistakes would be less costly. These extra processor cycles could also be used to compress more tightly, providing even larger effective main memory sizes.

We must still study the effects of compressed caching in multiprogrammed environments. As mentioned earlier, scheduling and variable space allocation could change drastically given the information provided by recency-based adaptivity, as well as the options afforded by compressed caching. Compressed caching is no longer an idea relegated to small and poorly connected machines, but rather seems to be applicable to many different kinds of workloads and hardware configurations.

Bibliography

[ADU71] Alfred V. Aho, Peter J. Denning, and Jeffrey D. Ullman. Principles of optimal page replacement. *Journal of the ACM*, 18(1):80–93, January 1971.

[AH90] A. Agarwal and M. Huffman. Blocking: Exploiting spatial locality for trace compaction. In *Proceedings, ACM SIGMETRICS*, pages 48–57, 1990.

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 96–107, Santa Clara, California, April 1991.

[Bab81] Ozalp Babaoglu. Efficient generation of memory reference strings based on the LRU stack model of program behaviour. In *Proceedings, PERFORMANCE '81*, pages 373–383, 1981.

[BCW90] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression.* Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Bel66] L. A. Belady. A study of replacement algorithms for virtual storage. *IBM Systems Journal*, pages 5:78–101, 1966.

[BF83] Ozalp Babaoglu and Domenico Ferrari. Two-level replacement decisions in paging stores. *IEEE Transactions on Computers*, C-32(12):1151–1159, December 1983.

[BFH68] M. H. J. Baylis, D. G. Fletcher, and D. J. Howarth. Paging studies made on the I.C.T. ATLAS computer. *Information Processing, IFIP Congress Booklet D*, 1968.

[BSTW86] Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan, and Victor K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, April 1986.

[Car84] Richard W. Carr. *Virtual Memory Management*. UMI Research Press, Ann Arbor, Mich., 1984.

[CCB99] R. Cervera, T. Cortes, and Y. Becerra. Improving application performance through swap compression. In *1999 Proceedings FREENIX Track, USENIX Annual Technical Conference*, pages 207–218. USENIX, June 1999.

[CLR89] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill and The MIT Press, 1989.

[CR70] E. G. Coffman and B. Randell. Performance predictions for extended paged memories. *Acta Informatica*, 1:1–13, 1970.

[CV65] F. J. Corbato and V. A. Vyssotsky. Introduction and overview of the multics system. In *AFIPS Conf Proc 27*, pages 185–196, 1965.

[CV76] P. J. Courtois and H. Vantilborgh. A decompsable model of program paging behavior. *Acta Informatica*, 6:251–275, 1976.

[Den70] Peter J. Denning. Virtual memory. *Computing Surveys*, 3(2):153–189, September 1970.

[Den76] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 19(5):285–294, 1976.

[Den80] Peter Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, January 1980.

[Dou93] Fred Douglis. The compression cache: Using on-line compression to extend physical memory. In *Proceedings of 1993 Winter USENIX Conference*, pages 519–529, San Diego, California, January 1993.

[EEF+97] Jens Ernst, William Evans, Christopher W. Fraser, Steven Lucco, and Todd A. Proebsting. Code compression. In *Proceedings of the 1997 SIGPLAN Conference on Programming Language Design and Implementation*, Las Vega, Nevada, June 1997. ACM Press.

[FG74] M. A. Franklin and R. K. Gupta. Computation of pf probabilities from program transition diagrams. *Communications of the ACM*, 17:186–191, 1974.

[FLW78] E. B. Fernandez, T. Lang, and C. Wood. Effect of replacement algorithms on a paged buffer database system. *IBM Systems Journal*, 22(2):185–196, 1978.

[FRT96] Peter Franaszek, John Robinson, and Joy Thomas. Parallel compression with cooperative dictionary construction. In *1996 Proceedings of the Data Compression Conference*, pages 200–209. IEEE Computer Society Press, 1996.

[GC97] Gideon Glass and Pei Cao. Adaptive page replacement based on memory reference behavior. In *SIGMETRICS The 1997 ACM SIGMETRICS*

International Conference on Measurement and Modeling of Computer Systems, volume 25, pages 115–126. ACM Press, June 1997.

[JH94] E. E. Johnson and J. Ha. Pdats: Lossless address trace compression for reducing file size and access time. In *Proceedings, IEEE International Conference on Computers and Communications*, pages 213–219, 1994.

[Joh97] Mark S. Johnstone. *Non-compacting Memory Allocation and Garbage Collection.* PhD thesis, Department of Computer Sciences, University of Texas at Austin, December 1997.

[KGJ96] Morten Kjelso, M. Gooch, and S. Jones. Main memory hardware data compression. In *22nd Euromicro Conference*, pages 423–430. IEEE Computer Society Press, September 1996.

[KGJ98] Morten Kjelso, M. Gooch, and S. Jones. Empirical study of memory-data: Characteristics and compressibility. In *IEE Proceedings Comput. Digit. Tech.*, volume 145, pages 63–67. IEE, January 1998.

[KGJ99] M. Kjelso, M. Gooch, and S. Jones. Performance evaluation of computer architectures with main memory data compression. *Journal of systems Architecture*, 45:571–590, 1999.

[KPR92] Anna R. Karlin, S. J. Phillips, and P. Raghavan. Markov paging. In *IEEE Symposium on the Foundations of Computer Science*, pages 208–217. IEEE Computer Society Press, 1992.

[LCB+98] D. C. Lee, P. J. Crowley, J. L. Baer, T. E. Anderson, and B. N. Bershad. Execution characteristics of desktop applications on windows nt. In *25th Annual International Symposium on Computer Architecture.* IEEE Computer Society Press, 1998.

[LCK+99] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In SIGMETRICS99 [SIG99], pages 134–143.

[MGST70] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9:78–117, 1970.

[Nel95] Mark Nelson. *The Data Compression Book (2nd ed.)*. M & T Books, 1995.

[Obe97] Markus F. X. J. Oberhumer. Lzo real-time data compression library, 1997.

[Pha95] Vidyadhar Phalke. *Modeling and Managing Program References in a Memory Hierarchy*. PhD thesis, Rutgers University, 1995.

[Puz85] Thomas R. Puzak. *Analysis of Cache Replacement Algorithms*. PhD thesis, University of Massachussetts, Dept. of Electrical and Computer Engineering, February 1985.

[RC96] Mark Russinovich and Bryce Cogswell. RAM compression analysis, February 1996. O'Reilly Online Publishing Report available from http://ftp.uni-mannheim.de/info/OReilly/windows/win95.update/model.html.

[RD90] J. Robertson and M. Devarakonda. Data cache management using frequency-based replacement. In *SIGMETRICS*, 1990.

[Riz97] Luigi Rizzo. A very fast algorithm for RAM compression. In *Operating Systems Review 311997*, pages 36–45, 1997.

[Sam89] A. Dain Samples. Mache: No-loss trace compaction. In *ACM SIGMETRICS*, pages 89–97, May 1989.

[SIG99] *SIGMETRICS The 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems.* ACM Press, June 1999.

[SKW99] Yannis Smaragdakis, Scott F. Kaplan, and Paul R. Wilson. EELRU: Simple and efficient adaptive page replacement. In *SIGMETRICS The 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* [SIG99].

[Sma98] Yannis Smaragdakis. Trace reduction for LRU-based simulations. Technical Report 98-25, The University of Texas at Austin, 1998.

[Smi77] Alan J. Smith. Two methods for the efficient analysis of address trace data. *IEEE Transactions on Software Engineering*, SE-3(1), January 1977.

[Spi76] J. R. Spirn. Distance string models for program behavior. *IEEE Computer*, 9:14–20, 1976.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[SW91] Walter R. Smith and Robert V. Welland. A model for address-oriented software and hardware. In *25th Hawaii International Conference on Systems Sciences*, January 1991.

[TL81] R. Turner and H. Levy. Segmented fifo page replacement. In *SIGMETRICS The 1981 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems.* ACM Press, 1981.

[Tor98] E. Torng. A unified analysis of paging and caching. *Algorithmica*, 20:175–200, 1998.

[UM97] Richard A. Uhlig and Trevor N. Mudge. Trace-driven memory simulation: A survey. *Computing Surveys*, 29(2):128–170, 1997.

[WFL83] C. Wood, E. B. Fernandez, and T. Lang. Minimization of demand paging for the LRU stack model of program behavior. *Information Processing Letters*, pages 99–104, 1983.

[Wil90] Paul R. Wilson. Some issues and strategies in heap management and memory hierarchies. In *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, October 1990. Also appears in *SIGPLAN Notices 23*(3):45–52, March 1991.

[Wil91a] Ross N. Williams. An extremely fast Ziv-Lempel compression algorithm. In *Data Compression Conference*, pages 362–371, April 1991.

[Wil91b] Paul R. Wilson. Operating system support for small objects. In *International Workshop on Object Orientation in Operating Systems*, pages 80–86, Palo Alto, California, October 1991. IEEE Press.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *1995 International Workshop on Memory Management*, Kinross, Scotland, UK, 1995. Springer Verlag LNCS.

[WKB] Paul R. Wilson, Scott F. Kaplan, and V.B. Balayoghan. Virtual memory reference tracing using user-level access protections. In Preparation.

[WKM94] Paul R. Wilson, Sheetal V. Kakkad, and Shubhendu S. Mukherjee. Anomalies and adaptation in the analysis and development of prepaging policies. *Journal of Systems and Software*, 1994.

[Yu96] Tong Lai Yu. Data compression for PC software distribution. *Software Practice and Experience*, 26(11):1181–1195, November 1996.

Vita

Scott Frederick Kaplan was born in New Haven, Connecticut on April 25, 1973, the fourth son of Harold and Barbara Kaplan. After graduating from The Hopkins School in New Haven in 1991, he attended Amherst College. He received his Bachelor of Arts in Computer Science from that school in 1995, and then began the Ph.D. program with the Department of Computer Sciences at The University of Texas at Austin.

Permanent Address: Department of Mathematics and Computer Science
Amherst College, Box 2239
Amherst, MA 01002-5000

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin.