

2004

Efficient caching algorithms for memory management in computer systems

Song Jiang

College of William & Mary - Arts & Sciences

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

Recommended Citation

Jiang, Song, "Efficient caching algorithms for memory management in computer systems" (2004).
Dissertations, Theses, and Masters Projects. Paper 1539623446.
<https://dx.doi.org/doi:10.21220/s2-q8t1-e863>

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

Efficient Caching Algorithms
for Memory Management in Computer Systems

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William & Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philosophy

by

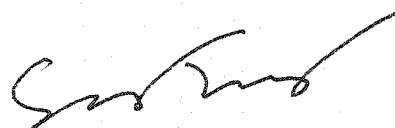
Song Jiang

2004

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

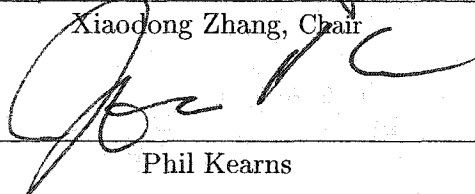


Song Jiang

Approved by the Committee, June 2004



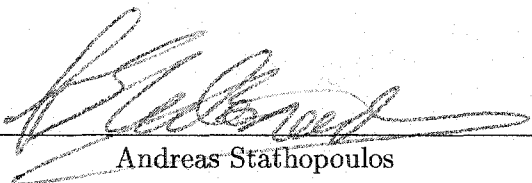
Xiaodong Zhang, Chair



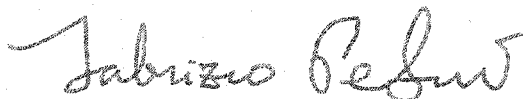
Phil Kearns



Bruce Lowekamp



Andreas Stathopoulos



Fabrizio Petrini
Los Alamos National Laboratory

To my mother, my wife and my son...

Table of Contents

Acknowledgments	x
List of Tables	xii
List of Figures	xiv
Abstract	xxiii
1 Introduction	2
1.1 Memory Hierarchies and Caching	3
1.1.1 Locality and Replacement algorithms	6
1.1.2 Replacement Policies for Virtual Memory	8
1.1.3 Global Replacement in Multiprogramming Environments	10
1.1.4 Placement and Replacement in Distributed File Buffer Caches	13
1.2 Contributions	15
1.3 Organization	17
2 General-Purpose Replacement Algorithms	19
2.1 Background	19

2.1.1	The Problems of the LRU Replacement Algorithm	19
2.1.2	An Executive Summary of my Algorithm	22
2.2	Related Work	23
2.2.1	User-level Hints	24
2.2.2	Tracing and Utilizing History Information of a Block	24
2.2.3	Detection and Adaptation of Access Regularities	27
2.2.4	Working Set Models	29
2.3	The LIRS algorithm	29
2.3.1	General Idea	29
2.3.2	The LIRS Algorithm Based on LRU Stack	32
2.3.3	A Detailed Description	34
2.4	Performance Evaluation	36
2.4.1	Experimental Settings	36
2.4.2	Access Pattern Based Performance Evaluation	38
2.4.2.1	Performance for the Looping Type	40
2.4.2.2	Performance for the Probabilistic Type	44
2.4.2.3	Performance for the Temporally-Clustered Type	46
2.4.2.4	Performance for the Mixed Type	48
2.4.3	LIRS Performance with High End Systems	49
2.4.4	LIRS versus Other Stack-Based Replacements	51
2.4.4.1	LIRS Threshold and Access Characteristics	53
2.4.4.2	LRU as a Special Member of the LIRS Family	55
2.5	Sensitivity and Overhead Analysis	57

2.5.1	Size Selection of List Q Holding Resident HIR Blocks (L_{hirs})	57
2.5.2	Overhead Analysis	58
2.6	Summary	60
3	Virtual Memory Replacement Policies	61
3.1	Background	61
3.1.1	The Research Status of Memory Replacement Policies	61
3.1.2	LRU/CLOCK and their Performance Disadvantages	63
3.1.3	LIRS and its Performance Advantages	66
3.2	Related Work	68
3.3	Description of CLOCK-Pro	71
3.3.1	Main Idea	71
3.3.2	Data Structure	73
3.3.3	Operations on Searching Victim Pages	74
3.3.4	Making CLOCK-Pro Adaptive	76
3.4	Performance Evaluation	78
3.4.1	Simulation on Buffer Cache for File I/O	78
3.4.2	Simulation on Memory for Program Executions	82
3.4.3	Simulation on Program Executions with Interference of File I/O	87
3.5	Summary	90
4	Thrashing in Multiprogramming Environments	91
4.1	Background	91
4.1.1	MPL versus System Thrashing	91

4.1.2	Thrashing and Page Replacement	92
4.1.3	Effectiveness of adaptive page replacement	93
4.1.4	Our work	94
4.2	Evolution of Page Replacement in Linux Kernel	95
4.2.1	Kernel 2.0	95
4.2.2	Kernel 2.2	96
4.2.3	Kernel 2.4	99
4.2.4	The Impact pf Page Replacement on CPU and Memory Utilizations	100
4.3	Evaluation of Page Replacement in Linux Kernels 2.2	101
4.3.1	Experimental environment	101
4.3.2	Page Replacement Behavior of Kernel 2.2.14	104
4.4	The Design and Implementation of TPF	109
4.4.1	The detection routine	110
4.4.2	The protection routine	113
4.4.3	State transitions in the system	113
4.5	Performance Measurements and Analysis	115
4.5.1	Observation and measurements of TPF facility	115
4.5.2	Experiences with TPF in the multiprogramming environment	119
4.6	Related Work	121
4.6.1	The Working Set Model and its Implementation Issues	122
4.6.2	Other Related Work	124
4.7	Summary	126

5	Multi-Level Buffer Cache Management	128
5.1	Background	128
5.1.1	Hierarchical Caching and its Challenges	128
5.1.2	Possible Solutions: Customized Second-Level Replacement and the Unified LRU	130
5.1.3	Our Principles to Address the Challenges	133
5.2	Quantifying Non-uniform Locality Strengths in Hierarchical Buffer Caching	134
5.2.1	Methods to Distinguish Locality Strengths	134
5.2.2	Comparisons of Locality Strength Quantification Methods	137
5.3	The Unified and Level-aware Caching (ULC) Protocol	144
5.3.1	An Executive Summary	144
5.3.2	A Detailed Description	145
5.3.2.1	The Single-client ULC Protocol	147
5.3.2.2	The Multi-client ULC Protocol	149
5.4	Performance Evaluation	152
5.4.1	Performance Metric	152
5.4.2	Simulation Environment	153
5.4.3	Comparisons of Multi-level Schemes in a Three-level Structure . . .	155
5.4.4	The Performance Implication of System Parameters	158
5.4.4.1	The Impact of Server Cache Size	159
5.4.4.2	The Impact of Client Cache Size	160
5.4.4.3	The Impact of Network Bandwidth	162
5.4.5	Comparisons of Caching Schemes for Multi-client Workloads	163

5.5	Related Work and Discussions	165
5.6	Summary	167
6	Conclusions and Future Work	168
6.1	General-Purpose Replacement Algorithms	169
6.2	Low Cost Virtual Memory Replacement Algorithms	170
6.3	Thrashing Prevention	172
6.4	Multi-Level Buffer Cache Management	173
	Bibliography	175

ACKNOWLEDGMENTS

Foremost, I give my thanks to my Lord, who strengthen me when I am weak, show me the way when I am lost, teach me the wisdom out of heaven, and love me in all the circumstances.

I would like to give my thanks to my adviser, Xiaodong Zhang, from my deepest heart. During the past five years, he has provided me every guidance and help for my research work and life needs. He always has the passion to motivate me with new research directions, to challenge me for better solutions, and to guide me through the difficulties in the process. He has always been discussing research issues with me open-mindedly and encouraged me to think in a broader background. The benefits I have so gratefully received from him are well beyond those on the academic. He has also put much effort to help me overcome the difficulties in my life and taken care of my well-being. I feel extremely lucky to have a person like Xiaodong to be my adviser, which makes my time at William and Mary a warm and happy memory.

I thank my committee, Phil Kearns, Bruce Lowekamp, and Andreas Stathopoulos at William and Mary, and Fabrizio Petrini at Los Alamos National Laboratory for their encouragement and advice on my research work. I learned a lot from the system course taught by Phil, which prepared me for my system implementation work. Fabrizio has provided me with his insightful comments on my research work and much help on my career development. I am really impressed by his dedication and passion as a researcher. I would also thank William Bynum for his help in reading almost every my manuscripts and giving his detailed comments and suggestions. I thanks Dimitrios Nikolopoulos for his valuable cooperations and discussions. I really appreciate the help and encouragement from Evgenia Smirni and Andreas Stathopoulos, who even gave so much baby stuff for my new-born son. I thank Vanessa Godwin, who, as the administrative director of the department, was so helpful and

thoughtful in providing me with the assistance I needed.

I also give my thanks to Shirong Zhen, who was my master thesis adviser in the University of Science and Technology of China (USTC). He had inspired my interests in research in the computer science field.

I wish to give my thanks to my colleagues and friends for making my educational life being so memorable, to name a few, Songqing Chen, Xin Chen, Lei Guo, Yongguang Liang, Hui Li, Ling Liu, Shuquan Nie, Shansi Ren, Tanping Wang, Qi, Zhang, and Donghua Zhou. I will miss the time I spent with them.

I would like to thank the warm-hearted friends I got to know over the years in the Williamsburg community. In particular, Harry Ambrose, who helped me with my English study for over three years, Debra Kemelek, who hosted me as an international student, as well as Eddie and Grace Liu, Walter and Elisabeth Kurth, Libby Von Fange, Connie and Richard Castor, Erwoom Chiou, and Florence Lee, who consistently showed their care and love to me and my family. They made my life in Williamsburg being so unforgettable.

Last, but not least, my deepest appreciation goes to my family. My wife, Shengli, has been with me shortly after I arrived at Williamsburg. Her commitment to our family has made my life full of happiness. No words can fully express my gratitude to her. Furthermore, I am really blessed to have my son, Caleb, who always reminds me of how beautiful a life can be! I also give my thanks to my parents, in particular, my mother, Yinghua Wang, who always loves me and supports me with all her heart under any circumstance. Without all of their love and supports, there would be no this dissertation.

List of Tables

2.1	An example to explain how a victim block is selected by the LIRS algorithm and how LIR/HIR statuses are switched. A “X” refers the block of the row is referenced at the virtual time of the column. The recency and IRR columns represent the values at the virtual time 10 for each block. We assume $L_{lirs} = 2$ and $L_{hirs} = 1$, and at the time 10 the LIRS algorithm leaves two blocks in the LIR set = {A, B}, and the HIR set is {C, D, E}. The only resident HIR block is E.	31
3.1	Hit ratios of the replacement algorithms OPT, CLOCK-Pro, LIRS, CAR, and CLOCK on workload <i>cpp</i>	81
3.2	Hit ratios of the replacement algorithms OPT, CLOCK-Pro, LIRS, CAR, and CLOCK on workload <i>sprite</i>	81
3.3	A brief description of the benchmark programs (“Size” is in number of millions of instructions)	83
3.4	The performance (number of page faults in one million of instructions) of algorithms CLOCK-Pro, CAR and CLOCK on program <i>m88ksim</i> with and without the interference of I/O file data accesses.	88

3.5	The performance (number of page faults in one million of instructions) of algorithms CLOCK-Pro, CAR and CLOCK on program <i>sor</i> with and without the interference of I/O file data accesses.	89
4.1	Execution performance and memory related data of the 3 benchmark programs.	104
5.1	Comparisons of the four measures on locality strengths by comparing their abilities to distinguish locality strengths, the stabilities of the distinctions, and if on-line measurements are possible.	143

List of Figures

1.1	Memory system is organized as a hierarchy, giving the user the illusion of a memory that is as large as largest level of memory and has the access speed as fast as the first level of cache.	3
1.2	The CLOCK replacement algorithm. The clock hand moves in the counter-clockwise direction. The reference bit of each page is either set (1) or unset (0).	9
1.3	CPU utilization is plotted against the number of processes in the system. Though increasing processes in the system could increase CPU utilization, too many processes could over-commit the limited memory and cause thrashing.	11
1.4	Multi-level buffer cache hierarchy. Caches are distributed along the clients, intermediate servers, and disk array, where accessed blocks can be buffered.	13
2.1	The LIRS stack S holds LIR blocks as well as HIRS blocks with or without resident status, and a list Q holds all the resident HIR blocks.	34

2.2	Illustration of the reference results in the example shown in Table 1 on the LIRS stack. In this figure, (a) corresponds to the state at virtual time 9. Accessing B, E, D, or C at virtual time 10 result in (b), (c), (d) and (e), respectively.	36
2.3	The time-space map (left) of cs and the hit rate curves by various replacement policies (right).	41
2.4	The time-space map (left) of glimpse and the hit rate curves by various replacement policies (right).	42
2.5	The time-space map (left) of postgres and the hit rate curves by various replacement policies (right).	43
2.6	The time-space map (left) of cpp and the hit rate curves by various replacement policies (right).	45
2.7	The time-space map (left) of <i>2-pools</i> and the hit rate curves by various replacement policies (right).	46
2.8	The time-space map (left) of sprite and the hit rate curves by various replacement policies (right).	47
2.9	The time-space map (left) of multi1 and the hit rate curves by various replacement policies (right).	48
2.10	The time-space map (left) of multi2 and the hit rate curves by various replacement policies (right).	49
2.11	The time-space map (left) of multi3 and the hit rate curves by various replacement policies (right).	50

2.12	The hit rate curves of workload <i>OpenMail</i> (left figure) and workload <i>Cello99</i> (right figure)	50
2.13	The IRRs of references of the workloads postgres (left) and sprite (right)	52
2.14	The rates of <i>Rmax</i> and cache size in blocks (<i>L</i>) for workloads postgres (left) and sprite (right). <i>Rmax</i> is the size of LIRS stack, which changes with virtual time. Cache size is 500.	54
2.15	The hit rate curves of workload postgres (left figure) and workload sprite (right figure) by varying the rates of threshold values for LIR/HIR status switching and <i>Rmax</i> in LIRS, as well as curves for OPT and LRU.	56
2.16	The hit rate curves of workload postgres (left figure) and workload sprite (right figure) by varying the size of list <i>Q</i> (<i>L_{hirs}</i> , the number of cache buffers assigned to HIR block set) of LIRS algorithm, as well as curves for OPT and LRU. “LIRS 2” means size of <i>Q</i> is 2, “LIRS x%” means size of <i>Q</i> is x% of the cache size in blocks.	58
2.17	The hit rate curves of workload postgres (left) and workload sprite (right) by varying the LIRS stack size limits, as well as curves for OPT and LRU. Limits are represented by rates of LIRS stack size limit in blocks and cache size in blocks (<i>L</i>).	59

3.1	There are three types of pages in CLOCK-Pro, hot pages marked as “H”, resident cold pages marked as “C” and non-resident cold pages marked as shadowed block with “C”. Around the clock, there are three hands: $HAND_{hot}$ pointing to the list tail (i.e. the last hot page) and searching a hot page to turn into a cold page, $HAND_{cold}$ pointing to the last resident cold page and searching for a cold page to replace out of memory, and $HAND_{test}$ pointing to the last cold page in the test period, terminating test periods of cold pages, and removing non-resident cold pages passing the test period out of the list.	
	The attached black dots represent the reference bits of 1.	73
3.2	Hit ratios of the replacement algorithms OPT, CLOCK-Pro, LIRS, CAR, and CLOCK on workloads <i>glimpse</i> and <i>multi2</i>	80
3.3	Adaptively changing the percentage of memory allocated to the cold pages in workloads <i>multi2</i> and <i>sprite</i>	82
3.4	Performance of CLOCK, CAR, CLOCK-Pro and OPT on programs with strong locality.	84
3.5	Performance of CLOCK, CAR, CLOCK-Pro and OPT on programs with moderate locality.	85
3.6	Performance of CLOCK, CAR, CLOCK-Pro and OPT on programs with weak locality.	86
4.1	The memory performance of gcc in a dedicated environment.	105
4.2	The memory performance of gzip in a dedicated environment.	105
4.3	The memory performance of vortex1 in a dedicated environment.	106

4.4	The memory performance of gzip (left figure) and vortex3 (right figure) during the interactions.	107
4.5	The memory performance of gcc (left figure) and vortex3 (right figure) during the interactions.	107
4.6	The memory performance of vortex1 (left figure) and vortex3 (right figure) during the interactions.	108
4.7	Dynamic transitions among normal, monitoring, and protection states in the improved kernel system.	114
4.8	The execution time comparisons (left figure) and comparisons of numbers of page faults (right figure) for the three group of program interactions in the Linux without TPF and with TPF.	115
4.9	The memory performance of gzip (left figure) and vortex3 (right figure) during the interactions in the Linux with TPF.	116
4.10	The memory performance of gcc (left figure) and vortex3 (right figure) during the interactions in the Linux with TPF.	118
4.11	The memory performance of vortex1 (left figure) and vortex3 (right figure) in the Linux with TPF.	118
4.12	Comparison of total interaction execution times for the three group of program interactions in the Linux with TPF, without TPF and the ideal interaction times.	119
5.1	Multi-level buffer cache hierarchy. Caches are distributed along the clients, intermediate servers, and disk array, where accessed blocks can be buffered.	129

- 5.2 In the two-level unified LRU scheme, there is an unified LRU stack corresponding to the two level of caches. The size of each individual LRU stack, N_1 or N_2 is equal to its respective cache size in terms of blocks. there are three type of accesses: (1) a hit in the L_1 cache. (2) a hit in the L_1 cache. (3) a miss in the two caches. If all the three cases, the accessed blocks are moved to the top of the stack. Except the first case, the block at the bottom of L_1 LRU stack is demoted onto the top of the L_2 stack. 132
- 5.3 In access stream $\{R_t, t = 0, 1, 2, \dots\}$, R_i , R_j , and R_l are three immediately consecutive references to block b . The current time is k . With these timing points, there are various measurements that can be used to quantify the locality strength of block b at time k , including the distance from R_k to R_l), called *OPT Distance (OD)*, the distance from R_j to R_k), called *Recency Distance (RD)*, the distance from R_j to R_l , called *Current Re-use Distance (CRD)*, and the distance from R_i to R_j , called *Last Re-use Distance (LRD)*. 135
- 5.4 In the LRU stack, for a given block, the position for the last access to the block corresponds to its LRD, its current position in the stack corresponds to its RD, and the position for its next access corresponds to its CRD. Before its current position exceeds its last access position (see left figure (a)), LRD-RD is LRD; after that (see right figure (b)), LRD-RD becomes RD. This allows LRD-RD to more accurately simulate CRD. The illustration also shows that RD and OD change with every reference. 137

5.5	Reference ratios to each of the segments (the ratios between the number of references to a segment and the number of all references in a workload). It also shows the accumulative reference ratios for the first N segments in each workload, where N is 1 through 10.	140
5.6	Movement ratio curves showing the ratios between the number of block movements across a segment boundary of the ordered lists and the number of total references for the four measures: OD, RD, CRD, and LRD-RD on various workloads. It shows that there are two groups of curves: OD and RD with high movement ratios, NRD and LRD-RD with low movement ratios. . . .	142
5.7	An example to show the data structure of ULC for a 3-level hierarchy. The blocks with their recencies less than that of yardstick Y_3 are kept in <i>uniLRU-stack</i> . The level status (L_1 , L_2 or L_3) of a block is determined by its position between two yardsticks where it was accessed last time. Its recency status (R_1 , R_2 or R_3) is determined by its position between two yardsticks where it sits currently. To decide which block should be replaced in each level, the blocks in the same level can be viewed to be organized in a separate LRU stack (LRU_1 , LRU_2 , or LRU_3), and the bottom block is for replacement. .	146

5.8	An example to explain how a requested block is cached in the server cache, and how the allocation scheme adjusts the size of the server cache used by various clients in a multi-client two-level caching structure. Originally in (a) server stack <i>gLRU</i> holds all the L_2 blocks from clients 1 and 2, which are also in their LRU_2 stacks, respectively. Then block 9 is accessed in client 1. Because block 9 is between yardstick Y_1 and Y_2 in its <i>uniLRU stack</i> , it turns into L_2 block and needs to be cached in the server. Because the server cache is full, the bottom block of <i>gLRU</i> , block 14, is replaced, which will be notified to its owner, client 2, through a piggyback on the next retrieved block going to client 2 (delayed notification). After the server buffers re-allocation (b), the size of server cache for client 1 is increased by 1 and that for client 2 is decreased by 1. So the clients and the server cooperate to make the server cache efficiently allocated with the aim of high performance for the entire system.	150
5.9	hit ratios in each of the three levels, demotion rates at each of two boundaries (between L1 and L2, and between L2 and L3 cache), and average access time for each workload with the multi-level caching schemes indLRU, uniLRU and ULC.	156
5.10	The average access times for schemes ULC, uniLRU, MQ and indLRU with various server cache sizes. The client cache size is fixed. It is 256MB for <i>zipf</i> , and 128MB for <i>httpd</i> and <i>dev1</i>	159

5.11	The average access times for schemes ULC, uniLRU, MQ and indLRU with various client cache sizes. The server cache size is fixed. It is 200MB for <i>zipf</i> and <i>dev1</i> , and 150MB for <i>httpd</i>	161
5.12	The average access times for schemes ULC, uniLRU, MQ and indLRU with various block transfer times. The client and server cache sizes are fixed, and are 100MB each for all the workloads.	162
5.13	The average access times of multi-client traces <i>httpd</i> , <i>openmail</i> , and <i>db2</i> with various server cache sizes. Among them <i>httpd</i> is with 7 clients, <i>openmail</i> is with 6 clients, and <i>db2</i> is with 8 clients. Each client contains 8MB, 1GB, or 256MB respectively.	163

ABSTRACT

As disk performance continues to lag behind that of memory systems and processors, fully utilizing memory to reduce disk accesses is a highly effective effort to improve the entire system performance. Furthermore, to serve the applications running on a computer in distributed systems, not only the local memory but also the memory on remote servers must be effectively managed to minimize I/O operations. The critical challenges in an effective memory cache management include: (1) Insightfully understanding and quantifying the locality inherent in the memory access requests; (2) Effectively utilizing the locality information in replacement algorithms; (3) Intelligently placing and replacing data in the multi-level caches of a distributed system; (4) Ensuring that the overheads of the proposed schemes are acceptable.

This dissertation provides solutions and makes unique and novel contributions in application locality quantification, general replacement algorithms, low-cost replacement policy, thrashing protection, as well as multi-level cache management in a distributed system. First, the dissertation proposes a new method to quantify locality strength, and accurately to identify the data with strong locality. It also provides a new replacement algorithm, which significantly outperforms existing algorithms. Second, considering the extremely low-cost requirements on replacement policies in virtual memory management, the dissertation proposes a policy meeting the requirements, and considerably exceeding the performance existing policies. Third, the dissertation provides an effective scheme to protect the system from thrashing for running memory-intensive applications. Finally, the dissertation provides a multi-level block placement and replacement protocol in a distributed client-server environment, exploiting non-uniform locality strengths in the I/O access requests.

The methodology used in this study include careful application behavior characterization, system requirement analysis, algorithm designs, trace-driven simulation, and system implementations. A main conclusion of the work is that there is still much room for innovation and significant performance improvement for the seemingly mature and stable policies that have been broadly used in the current operating system design.

Efficient Caching Algorithms
for Memory Management in Computer Systems

Chapter 1

Introduction

With the ongoing dramatic increase in processor speeds, and relatively stable disk speed, the disparity between processor speeds and disk access times are keeping widened. For memory-intensive applications and those with frequent file data accesses, their execution times are often dominated by I/O latency. Since disk access times are improving slowly, these applications are receiving decreasing benefits from the rapid advance of processor technology, and I/O latency is accounting for an increasing proportion of their execution times. This technology trend makes memory play an increasingly important role to serve as a cache for I/O file data and virtual memory swap files. So, fully utilizing memory to reduce disk accesses is an important issue concerning to the entire system performance. To serve the applications running on a computer in a distributed system, not only the local memory but also the memories distributed on remote servers, even on other clients have to be effectively managed to minimize I/O operations.

In this dissertation, we examine four challenging issues in the effective memory management to reduce I/O accesses, including (1) General-purpose memory replacement algorithms; (2) Low-cost virtual memory replacement policies; (3) Thrashing prevention for running multiple memory-intensive programs; (4) Multi-level distributed cache manage-

ment. Our dissertation provides solutions to these challenging issues, and use trace-driven simulation or implementation techniques to demonstrate their effectiveness in terms of both performance and cost. Our dissertation demonstrates that innovative methods can significantly improve the utilization of available memory and reduce I/O accesses by effectively exploiting the locality in the access requests.

1.1 Memory Hierarchies and Caching

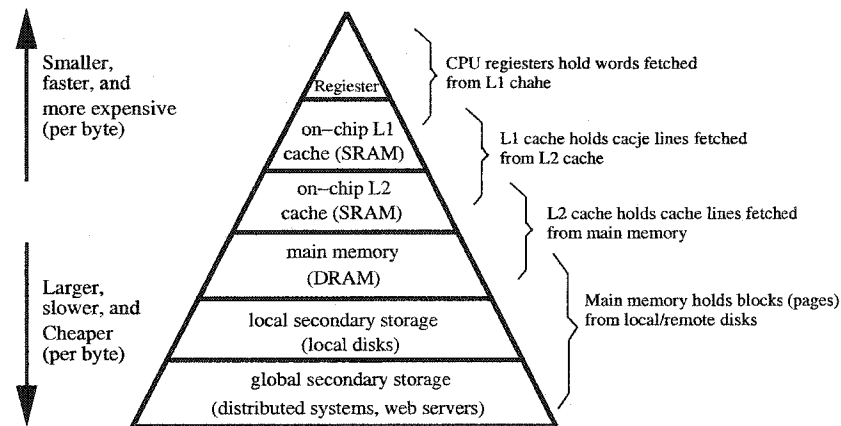


Figure 1.1: Memory system is organized as a hierarchy, giving the user the illusion of a memory that is as large as largest level of memory and has the access speed as fast as the first level of cache.

In computer systems, memory is organized as a memory hierarchy. A memory hierarchy consists of multiple levels of memory with different speed, size and unit cost (see Figure 1.1). In the hierarchy, the layers close to processors are two or three levels of fast and expensive SRAM (Static RAM) cache memory, with their size from 128K to a few Megabytes. The next layer is main memory made of DRAM (Dynamic RAM), which is of a higher capacity in the same size of chip area, and less costly, but is slower in access time. Currently the typical size of main memory is from 128MB to 1GB. A layer below the main memory, which

is further away from processors, is the mechanical disk. The size of disk can be hundreds of Gigabytes, but its speed is several magnitudes slower than the memory made of RAM chips. The goal of organizing memory into the hierarchy is try to present the user with a fast, large and affordable memory, with its speed close to the CPU caches, and its size close to the disk.

When a program is running at a processor, it always tries to fetch the data it needs from the memory close to it. If the data is found there, called a hit, the program will continue its execution with the fetched data. However, if the data is not found there, called a miss, the request has to be sent to the next layer of the memory hierarchy to retrieve the data.

CPU caches are implemented primarily in hardware to match the processor speed. The speed of the caches, especially for the first-level cache, is critical to the processor speed, because they directly affect the performance of load and store instructions. Because of the critical importance of hit times of hardware cache, its design is severely constrained – only very simple and low cost operations are allowed, so that most of them can be wired in the hardware. For this purpose, direct mapped or set associative mapping are used to minimize the addressing cost both in time and extra parts. The associativity of the set-associativity is typically from 2 to 16. Further increasing associativity is not worthwhile because of the diminishing hit rate increase and rapidly increased comparator cost.

It is a different case for main memory and disk. On one hand, misses are much more expensive than those in CPU caches. So even a small decrease of misses could considerably reduce execution time. On the other hand, the main memory can afford fully associative mapping and more sophisticated management algorithms to reduce misses.

To run programs with their total memory demand larger than the amount of main

memory available on the machine, *virtual memory* (VM) is devised to use the main memory as a cache that contains only the active portions of one or multiple programs. The rest of programs are saved on the swap areas on disks. Conventionally, a virtual memory block is called a page, and a virtual memory miss is called a page fault. To support a fully associative virtual memory system, each program is equipped with a page table to index its virtual memory space. When a page fault occurs with a memory access, which means the virtual page can not be mapped onto a page resident in the main memory, the program issuing the memory access request has to stop and wait for the page to be retrieved from the disk, which is much more expensive than a memory hit. Even though the size of both memory and disk have rapidly increased, their speed gap remains largely unchanged.

With the dramatically decreased memory price, the memory installed on the computer has been significantly increased. However, this does not relent the pressure on memory used as a cache of swap area. Parkinson's Law [58] states that "Work expands to fill the time available." In the virtual memory case, the law actually reflects the fact that programs expand to fill the memory available to hold them. The most obvious example is that Microsoft continues to increase the memory demand of its operating systems and office software to include more advanced functionalities and increased performance. In the field of scientific computations, many of the computational problems of interest to scientists and engineers involve data sets that are much larger than physical memory. Increases in processor power and the available memory capacity make it feasible to solve larger problems, or to solve the same problem at a finer granularity, and the size of the data set grows with the problem being solved. For example, the visualization of Computational Fluid Dynamics (CFD), input data sets today can surpass 100 Gbytes, and are expected to scale with the

ability of supercomputers to generate them. Despite the continuing trend toward larger memories, it is unlikely that these data sets will fit entirely within the main memory. How to make an effective use of main memory as a cache of data on disks to reduce disk accesses is a ever-existing challenge to operating system designers. The goal of caching is to keep those active pages in memory, so that their next references are hits without the need of disk accesses. The effectiveness of the caching in main memory depends on how effectively to identify active portions of program address space for storing in memory. This is also the theme of this dissertation.

1.1.1 Locality and Replacement algorithms

Caching works because of the existence of program access locality, which states that “most of the time, a program tends to reference only a few of its pages and the set of pages being referenced changes slowly [12]”. The locality consisting of a small portion of program address space is of two types. The first type is temporal locality, which states that if an item is accessed, it will tend to be accessed again soon. The second type is spatial locality, which states that if an item is accessed, items whose addresses are close to it will tend to be accessed soon. The locality information exhibited in the memory access provides a very useful hints to predict which set of pages are probably to be used soon and should be prefetched or kept in memory. The spatial locality is mostly exploited by large page size and prefetching. Prefetching is to fetch pages in advance into memory before there are access requests on the pages. In contrast, demand paging states that a page is brought into memory only on a page fault. The temporal locality is used in caching to decide which pages should be kept in memory and which pages should be evicted out of memory to make

room for faulted pages. The algorithm used in the decision process is called replacement algorithm.

The study on replacement algorithms has a long history since 1950s and generate numerous papers on the algorithms, modeling, implementation and performance evaluation. However, the problem is still far from being effectively solved and continues to draw attention from the industry and academia. Meanwhile, the changes of the program memory access behaviors and system configurations generate new demands on replacement algorithms.

Traditionally, the metric to evaluate replacement algorithms is hit ratio, which is defined as the ratio of the number of misses and the number of accesses. Using the metric, the optimal algorithm is the one called OPT [1] or MIN [7, 63], which replaces the page that will not be used for the longest period of time. It is easy to describe, but unfortunately, it is a off-line, unimplementable algorithm, because it requires future knowledge of the reference requests. As a result, OPT is used mainly for comparison studies.

There are two types of history locality information used in the general-purpose replacement algorithms: recency and frequency. Recency of a page refers to the time of its last reference. Least Recently Used (LRU) [46, 7] is the most well known replacement algorithm. It assumes that a page not accessed recently will not be accessed in the near future. Thus it chooses the page whose last reference is the farthest to replace. LRU is very successful due to its simplicity, low-cost and good performance in most cases and is widely used in various systems. However, because it considers very limited history accesses, and makes a assumption that does not hold for certain access patterns, LRU performance could be unacceptably poor. In contrast, Least Frequently Used (LFU) replacement algorithm [20] uses frequency, the number of times a page has been accessed, to select victim pages. How-

ever, LFU is rarely used in practice because of its severe drawbacks: it requires a very high running cost, cause *cache pollution*, in which pages that have accumulated large frequencies in history and will not be used are hard to be replaced.

Most recently proposed replacement algorithms take both recency and frequency factors into consideration. For example, Least Recently/Frequently Used (LRFU) algorithm [45] uses a parameter to dictate how much more weight given to the recent history than to the past history. Other algorithms considering more history information include LRU-2 [57], 2Q [37], EELRU [67], MQ [82] LIRS [33] and ARC [51]. These algorithms differ in their hit ratios with different access patterns, their overhead, and adaptivity to access pattern changes.

The key challenge for a high performance and low-cost replacement algorithm is to accurately quantify locality strength and make an efficient use of the locality information. The first part of this dissertation provide solution to meet the challenge.

1.1.2 Replacement Policies for Virtual Memory

We have stated that LRU is the most widely used replacement algorithm. Because of a very stringent cost requirement on the policy from virtual memory (VM) management, actually it is the LRU approximations that are used for VM page replacement. It requires the cost be associated with the number of page faults or a moderate constant. An algorithm with its cost proportional to the number of memory references would be prohibitively expensive. This causes the user program to incur a trap to the operating system every few instructions, the CPU would spend much more time on page replacement work than doing useful work for the user application even when there are not paging requests.

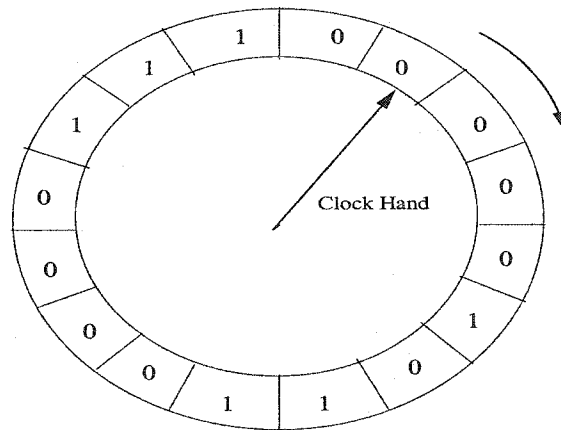


Figure 1.2: The CLOCK replacement algorithm. The clock hand moves in the counter-clockwise direction. The reference bit of each page is either set (1) or unset (0).

There are several low-cost VM replacement algorithm. most of them attempt to simulate LRU behavior. The FIFO (Fist-In, First-Out) replacement policy maintains a list of all pages currently in memory, where the page at the head of the list represents the oldest one, and the page at the tail the most recently accessed one. On a page fault, the page at the head is removed for the replacement and the faulted page is placed at the list tail. This simple algorithm does not allow actively accessed pages to always stay in memory. To make recent access information considered, it is evolved into the Second-Chance (SC) algorithm [70]. In the SC algorithm, there is a reference bit associated with each resident page, which is set by hardware with every memory access. When a page moves to the head of the list, its reference bit is checked. If its bit is set, the page is given a second chance and move to the list tail. Otherwise, the page is replaced. So SC is looking for an old page that has not been referenced in the previous clock interval. One way to implement the algorithm is to maintain the list as a circular queue called CLOCK. A pointer called clock hand indicates which page to be replaced next (see Figure 1.2). When a free page is needed, the hand advances until it finds a page with an unset reference bit. The implementation

is usually called the CLOCK algorithm. Experiences and experiments have shown that CLOCK has effectively simulated LRU and has the performance very close to that of LRU. In a generalized CLOCK version called GCLOCK[69], a counter is associated with each page rather than a single bit. The counter will be incremented if the page is hit. The circulating clock hand sweeps through the page decrementing the counter until a page with its count of zero is found for replacement.

The CLOCK algorithm and its alternatives have been dominating the VM replacements for more than three decades. Though their performance is satisfactory in general, they inherit the performance drawbacks from LRU and seriously under-perform for some commonly observed access patterns. On one hand, there are many general-purpose replacement algorithms improving LRU performance. On the other hand, due to the extremely low cost requirement of VM management, the performance advantages of the algorithms are difficult to transfer to VM performance. So the challenge is to design a VM replacement algorithm that has a cost comparable to CLOCK and overcomes the performance disadvantages of LRU and CLOCK. The second part of the dissertation is to address this challenge.

1.1.3 Global Replacement in Multiprogramming Environments

In a multiprogramming environment, when multiple processes compete for page frames, page replacement algorithms can be classified into two broad categories: local replacement and global replacement. Local replacement requires that each process select from only its own set of allocated page frames for replacement to satisfy its page fault. Global replacement allows a process to select a page frame belonging to any processes for replacement and load

its faulted page into the frame ¹. So one process can take a frame from another. Memory allocation can be re-distributed according to the competition among processes.

A local replacement uses a memory scheme to assign the allocation to each process. The assignment can be based on the estimation of the need of each process. However, the static allocation can not capture dynamical changing memory demand of each program [38]. As a result, memory space is not well utilized. If we dynamically adjust the allocation to the current demand of individual process, the local replacement will essentially evolve into a global one. Researchers and system practitioners seem to have agreed that a local policy is not an effective solution for virtual memory management, and it is rarely used nowadays. Global replacement can automatically implement memory allocation adapting to the memory demands of processes through their page replacement interaction. This would make memory better utilized in a global replacement than that in a local replacement.

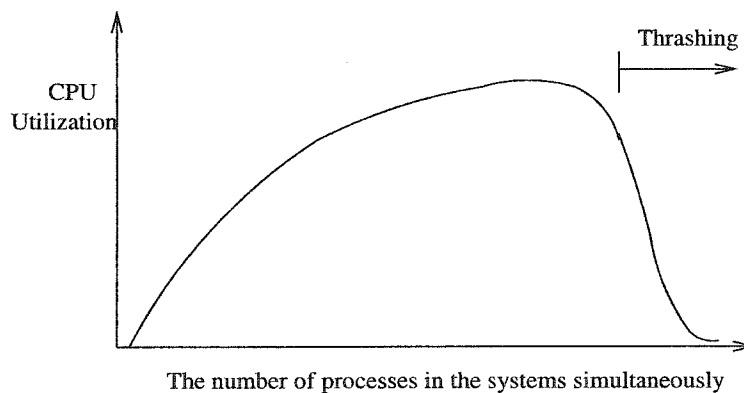


Figure 1.3: CPU utilization is plotted against the number of processes in the system. Though increasing processes in the system could increase CPU utilization, too many processes could over-commit the limited memory and cause thrashing.

¹Actually in practice the page frame used for the current page fault may not be the frame just replaced. Normally, operating systems do not wait to start the search free pages until all the free pages are running out. Instead, they set a threshold for the minimal available free pages. Once the threshold is reached, they start to search proper replacement candidates to fill up the pool of free pages. And these free pages are ready for use whenever needed.

One problem with global replacement algorithm is thrashing among multiple processes. A primary objective of memory management is to maximize the effectiveness of main memory in meeting the overall goals of sharing, throughput, and responsiveness. For this purpose, we need to maintain a proper number of processes active simultaneously in memory. If there are not enough active processes, main memory is underutilized, and the possibility of all processes being blocked, leaving the CPU idle, is increased. If there is an excess of active processes in memory, the main memory will be over-committed, excessive number of page faults will take place, also CPU idling. This is called thrashing (see Figure 1.3).

Now let's have a brief look into how a thrashing is developed among multiple processes. The set of recently used, active pages of a process are called its working set [24], which is used to estimate the current memory demand of a running program in the system. Now suppose a process enters a new phase in its execution and needs more page frames. It starts faulting and could take pages away from other processes under a global page replacement policy. The replaced pages may belong to their working sets because of the memory overcommitment. So these processes need these pages, they also fault, taking pages from other processes. which escalates the problem further. The situation can be worsen until the system ends up spending most of its time in page fault handling, and the processes can make little progress. Thus a thrashing occurs.

This problem may be addressed by reducing the number of active processes, thus controlling the system load. This is called load control. However, by abruptly suspending, even killing active processes, the brute-force mechanism could introduce an unnecessary working set reloading overhead, excessively reducing active processes, and reduce user interactivity. Actually the thrashing is directly related to the global replacement policy. Without having

to resort to local replacement policies, a global replacement policy that can adapt its replacement behavior to the current CPU utilization would be a promising idea to overcome the aforementioned difficulties, and as well alleviate or even solve thrashing problem. The third part of the dissertation is to develop techniques to address the thrashing problem.

1.1.4 Placement and Replacement in Distributed File Buffer Caches

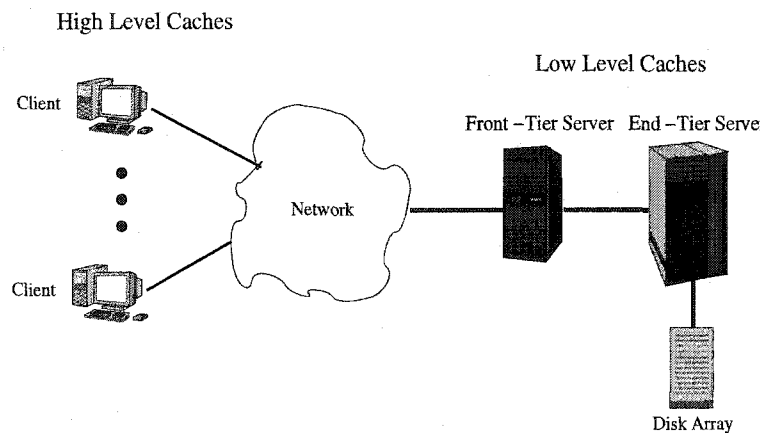


Figure 1.4: Multi-level buffer cache hierarchy. Caches are distributed along the clients, intermediate servers, and disk array, where accessed blocks can be buffered.

When a user requests a remote data item in a client-server distributed environment, the retrieved data is cached at the client file buffer cache, it could also be cached at intermediate server buffer caches and disk built-in caches, which forms a multi-level buffer cache hierarchy (see Figure 1.4). For example, disk arrays use a significant amount of cache RAM as a data buffer attempting to provide as much as re-accessed data as possible from the cache without access disks. As an example case, EMC 8830 disk array supports up to 64 GBytes cache for this purpose. We might naively expect a large amount of cache memory invested on the data retrieving path in the distributed system would automatically gain steady performance increase. However, in the distributed situation, the issue can be more complicated than we

thought because of the existing buffer cache management.

Unlike the processor cache hierarchy, where the multi-level inclusivity [3] between L_1 , L_2 , .. L_n cache could be accepted as a principle to simplify the cache coherence protocol and the cache behaviors at different levels are well coordinated, the multi-level caches here are much loosely connected. The placement of cached file block in the hierarchy and the replacement at each level of cache are determined by local policies independently from each other. Any client requested blocks are cached by intermediate caches, when they are on their way to the clients passing through the caches. This causes accessed blocks be redundantly cached and makes caches under-utilized. Only block misses from the high level caches, which are close to clients, appear at the low level caches, which are far away from clients. This causes the locality, which the replacement algorithm depend on for its replacement decision, is weakened and makes the hit ratios at the low level caches significantly deteriorate.

There are several possible approaches to attack the problems. One approach is to make the replacement algorithms at each level coordinate with each and allow one block be cached at one place at most. Its potential problem is that it could incur excessive amount of communication overhead on the network for the coordination. Another approach is to re-design local replacement algorithms, to improve their hit ratios even with weakened locality information. This is certainly inadequate in the whole system point of view, because each cache still makes replacement decisions independently and the block redundant caching problem is not solved. Without the coordination among the caches, its performance potential is greatly limited. In the fourth part of the dissertation, we will address the aforementioned problems in a distributed environment.

1.2 Contributions

The contribution of the dissertation in cache management algorithms are fourfold: general-purpose replacement algorithm [33], low-cost replacement policy for virtual memory, thrashing prevention in multiprogramming environments [34], and file block placement and re-placement in multi-level buffer caches [36], and are outlined as follows:

- This dissertation proposes an efficient general-purpose replacement algorithm, called Low Inter-reference Recency Set (LIRS). We designed the algorithm based on a locality qualification metric called Low Inter-Reference Recency (IRR), or re-use distance in the previous studies in the fields such as compiler [28] and CPU cache [64]. It describes the time between two consecutive references to a block. In the trace-driven simulation, We compared the hit ratios of LIRS with LRU, LRU-2, 2Q, LRFU, EELRU, ARC, and UBM. Without tuning sensitive parameters and assuming specific properties of access patterns, LIRS outperforms all the other replacement algorithms across a large number of real-life and synthetic traces with different memory sizes. In many cases, its hit ratios are very close to the optimal ones. The data structure and operations of LIRS are very simple but effective. Its running cost is as low as that of LRU. Its unique performance and cost advantages have made LIRS very attractive to the industry [73, 51].
- Inspired by the general-purpose LIRS replacement algorithm and the demanding need of a new virtual memory page replacement policy to improve the performance of the dominating CLOCK policy, this dissertation proposes an enhanced CLOCK replacement policy, called CLOCK-Pro. By additionally keeping track of a limited num-

ber of replaced pages, CLOCK-Pro works in a similar fashion as CLOCK with a VM-affordable cost. In the meanwhile, it brings all the much-needed performance advantages from LIRS to CLOCK. CLOCK-Pro even eliminates the only tunable parameter in LIRS and makes itself a policy adapting to the changing access locality to serve a broad spectrum of workloads. With the access patterns where CLOCK is able to achieve high hit ratios, CLOCK-Pro behaves much like CLOCK. For the access patterns such as memory scan, large-scale loop accesses, where CLOCK performs unacceptably poor, CLOCK-Pro significantly reduces the page faults, thus makes system more robust to various memory access behaviors. We also compared CLOCK-Pro with other recently proposed VM page replacement policies, such as CAR [6] and show that CLOCK-Pro consistently outperforms CAR.

- To deal with thrashing in multiprogramming environments, this dissertation provide a scheme, called *Thrashing Protection Facility* (TPF), which protects the system from thrashing once a thrashing is detected. The scheme deals with thrashing by adaptively making adjustments on global page replacement policies. The adjustments are based carefully analyzing the correlation between global page replacement behaviors and CPU utilizations. Implementation in Linux kernels shows that the scheme can reduce the program execution times by up to 67% when there is thrashing.
- In the area of multi-level buffer cache management, this dissertation proposes a client-directed, coordinated file block placement and replacement protocol called *Unified Level-aware Caching* (ULC), where the strengths of locality are dynamically quantified at the client level, where full locality information is available. The quantification

results are used to direct servers on placing or replacing file blocks at different levels of the buffer caches. So that the locality of block accesses dynamically matches the caching layout of the blocks in the hierarchy. The effectiveness of our proposed protocol comes from achieving the following three goals: (1) The multi-level cache retains the same hit rate as that of a single level cache whose size equals to the aggregate size of multi-level caches. (2) The non-uniform locality strengths of blocks are fully exploited and ranked to fit into the physical multi-level caches. (3) The communication overheads between caches are also reduced. Our trace-driven simulation results show that ULC significantly and consistently outperforms existing multi-level caching schemes.

In this long-term comprehensive study of caching algorithms under the above four situations, the dissertation demonstrates that there is still much room for innovation and significant performance improvement for the seemingly mature and stable policies broadly used in the system design, such as LRU replacement and load control. The algorithms proposed and evaluated in the dissertation are valuable in making the system more capable to handle large-scale, more complicated applications running on variously-configured systems.

1.3 Organization

Chapter 2 describes our study on general-purpose cache replacement algorithms. Chapter 3 continues the replacement work and customizes the proposed replacement algorithm in the virtual memory management with an extremely low cost policy. Chapter 4 discusses an experimental study on thrashing prevention in multiprogramming environments by adaptively

adjusting global page replacements. Chapter 5 describes our study on the management of distributed, multi-level buffer caches through an effective block placement and replacement protocol. Chapter 6 provides the conclusions and future work of the dissertation.

Chapter 2

General-Purpose Replacement Algorithms

Replacement algorithms play important roles in buffer cache management, and their effectiveness and efficiency are crucial to the performance of file systems, databases, and other data management systems. In this chapter, we will review previous work on improving the performance of replacement algorithms and introduce the design of a novel replacement algorithm.

2.1 Background

2.1.1 The Problems of the LRU Replacement Algorithm

The Least Recently Used (LRU) replacement is widely used to manage buffer cache due to its simplicity, but many anomalous behaviors have been found with some typical workloads, where the hit rates of LRU may only slightly increase with a significant increase of cache size. The observations reflect LRU's inability to cope with access patterns with weak locality such as file scanning, regular accesses over more blocks than the cache size, and accesses

on blocks with distinct frequencies. Here are some representative examples reported in the research literature, to illustrate how LRU poorly behaves.

1. Under the LRU policy, a burst of references to infrequently used blocks, such as “sequential scans” through a large file, may cause replacement of commonly referenced blocks in the cache. This is a common complaint in many commercial systems: sequential scans can cause interactive response time to deteriorate noticeably [57]. A wise replacement policy should prevent “hot” blocks from being evicted by “cold” blocks.
2. For a cyclic (loop-like) pattern of accesses to a file that is only slightly larger than the cache size, LRU always mistakenly evicts the blocks that will be accessed soonest, because these blocks have not been accessed for the longest time [67]. A wise replacement policy should maintain a miss rate close to the buffer space shortage.
3. In an example of multi-user database application [57], each record is associated with a B-tree index. There are 20,000 records. The index entries can be packed into 100 blocks, and 10,000 blocks are needed to hold records. We use $R(i)$ to represent an access to Record i , and $I(i)$ to Index i . The access pattern of the database application alternates references to random index blocks and record blocks by $I(1), R(1), I(2), R(2), I(3), R(3), \dots$. Thus, index blocks will be referenced with a probability of 0.005, and data blocks are with a probability of 0.00005. However, LRU will keep an equal number of index and record blocks in the cache, and perhaps even more record blocks than index blocks. A wise replacement should select the resident blocks according to the reference probabilities of the blocks. Only those blocks with relatively high

probabilities deserve to stay in the cache for a long period of time.

The reason for LRU to behave poorly in these situations is that LRU makes a bold assumption – a block that has not been accessed the longest would wait for relatively longest time to be referenced again. This assumption cannot capture the access patterns exhibited in these workloads with weak locality. Generally speaking, there is less locality in buffer caches than that in CPU caches or virtual memory systems [65].

However, LRU has its distinctive merits: simplicity and adaptability. It only samples and makes use of very limited information – recency. However, while addressing the weakness of LRU, existing policies either take more history information into consideration, such as LFU (Least Frequently Used)-like ones in the cost of simplicity and adaptability, or switch temporarily from LRU to other policies whenever regularities are detected. In the switch-based approach, these policies actually act as supplements of LRU in a case-by-case fashion. To make a prediction, these policies assume the existence of relationship between the future reference of a block with the behaviors of those blocks in its temporal or spatial locality, while LRU only associates the future behavior of a block with its own history reference. This additional assumption increases the complexity of implementations, as well as their performance dependence on the specific characteristics of workloads. My LIRS only samples and makes use of the same history information as LRU does – recency, and mostly retains the simple assumption of LRU. Thus it is simple and adaptive. In our design, LIRS is not directly targeted at specific LRU problems but fundamentally addresses the limitations of LRU.

2.1.2 An Executive Summary of my Algorithm

we use recent Inter-Reference Recency (IRR) as the recorded history information of each block, where IRR of a block refers to the number of other distinct blocks accessed between two consecutive references to the block. In contrast, the recency refers to the number of other distinct blocks accessed from last reference to the current time. We call IRR between last and penultimate (second-to-last) references of a block as recent IRR, and simply call it IRR without ambiguity in the rest of the paper. We assume that if the IRR of a block is large, the next IRR of the block is likely to be large again. Following this assumption, we select the blocks with large IRRs for replacement, because these blocks are highly possible to be evicted later by LRU before being referenced again under our assumption. Note that these evicted blocks may also have been recently accessed, i.e. each has a small recency.

Similar definition to IRR for measuring data access locality have been found in literature as early as in 1970. Mattson et al in [46] define “stack distance” by measuring the number of distinct virtual memory pages accessed between two consecutive accesses of the same page in a stack. Recently, this concept has been generalized as “reuse distance” [77] referring to the number of distinct data elements accessed between two consecutive uses of the same data element.

In comparison with LRU, by adequately considering IRR in history information in our policy, we are able to eliminate negative effects caused by only considering recency, such as the problems presented in the above three examples. When deciding which block to evict, our policy utilizes the IRR information of blocks. It dynamically and responsively distinguishes low IRR (denoted as LIR) blocks from high IRR (denoted as HIR) blocks, and

keep the LIR blocks in the cache, where the recency of blocks is only used to help determine LIR or HIR statuses of blocks. We maintain an LIR block set and an HIR block set and manage to limit the size of the LIR set so that all the LIR blocks in the set can fit in the cache. The blocks in the LIR set are not chosen for replacement, and there are no misses with references to these blocks. Only a very small portion of cache is assigned to store HIR blocks. Resident HIR blocks may be evicted at any recency. However, when the recency of an LIR block increases to a certain point, and an HIR block gets accessed at a smaller recency than that of the LIR block, the statuses of the two blocks are switched. We name the proposed policy “Low Inter-reference Recency Set” (denoted as LIRS) replacement, because the LIR set is what the algorithm tries to identify and keep in cache. The LIRS policy aims at addressing three issues in designing replacement policies: (1) how to effectively utilize multiple sources of access information; (2) how to dynamically and responsively distinguish blocks by comparing their possibility to be referenced in the near future; and (3) how to minimize implementation overheads.

2.2 Related Work

LRU replacement is widely used for the management of virtual memory, file caches, and data buffers in databases. The three typical problems described in the previous section are found in different application fields. A lot of efforts have been made to address the problems of LRU. We classify existing schemes into three categories: (1) replacement schemes based on user-level hints; (2) replacement schemes based on tracing and utilizing history information of block accesses; and (3) replacement schemes based on regularity detections.

2.2.1 User-level Hints

Application-controlled file caching [11] and application-informed prefetching and caching [59] are the schemes based on user-level hints. These schemes identify blocks with low possibility to be re-accessed in the future based on available hints provided by users. To provide appropriate hints, users need to understand the data access patterns, which adds to the programming burden. In [53], Mowry et. al. attempt to abstract hints from compilers to facilitate I/O prefetching. Although their methods are orthogonal to our LIRS replacement, the collected hints may help us to ensure the existence of the correlation of consecutive IRRs. However, in most cases, the LIRS algorithm can adapt its behavior to different access patterns without explicit hints.

2.2.2 Tracing and Utilizing History Information of a Block

Realizing that LRU only utilizes limited access information, researchers have proposed several schemes to collect and use “deeper” history information. Examples are LFU-like algorithms such as FBR, LRFU, as well as LRU-K and 2Q. We take a similar direction by effectively collecting and utilizing access information to design the LIRS replacement.

Robinson and Devarakonda propose a frequency-based replacement algorithm (FBR) by maintaining reference counts for the purpose to “factor out” locality [65]. However it is slow to react to reference popularity changes and some parameters have to be found by trial and error. Having analyzed the advantages and disadvantages of LRU and LFU, Lee et. al. combine them by weighing recency factor and frequency factor of a block [45]. The performance of the LRFU scheme largely depends on a parameter called λ , which decides the weight of LRU or LFU, and which has to be adjusted according to different

system configurations, even according to different workloads. However, LIRS does not have a tunable parameter that is sensitive to the target workload.

The LRU-K scheme [57] addresses the LRU problems presented in the Examples 1 and 3 in the previous section. LRU-K makes its replacement decision based on the time of the K th-to-last reference to the block. After such a comparison, the oldest resident block is evicted. For simplicity, the authors recommended $K = 2$. By taking the time of the penultimate reference to a block as the basis for comparison, LRU-2 can quickly remove cold blocks from the cache. However, for blocks without significant differences of reference frequencies, LRU-2 does not work well. In addition, LRU-2 is expensive: each block access requires $\log(N)$ operations to manipulate a priority queue, where N is the number of blocks in the cache.

Johnson and Shasha propose the 2Q scheme that has overhead of a constant time [37]. The authors showed that the scheme performs as well as LRU-2. The 2Q scheme can quickly remove sequentially-referenced blocks and loopingly-referenced blocks with long periods from the cache. This is done by using a special buffer, called the *A1in* queue, in which all missed blocks are initially placed. When the blocks are replaced from the *A1in* queue in the FIFO order in a short period of time, the addresses of those replaced blocks are temporarily placed in a ghost buffer called *A1out* queue. When a block is re-referenced, if its address is in the *A1out* queue, it is promoted to a main buffer called *Am*. That is, only blocks have short re-use distance measured by the *A1in* queue and *A1out* queue can be cached for a long period of time in *Am*. In this way they are able to distinguish frequently referenced blocks from those infrequently referenced. By setting of the sizes of *A1in* and *A1out* queues as constants K_{in} and K_{out} , respectively, 2Q provides a victim block either from *A1in* or

from Am . However, K_{in} and K_{out} are pre-determined parameters in 2Q scheme, which need to be carefully tuned, and are sensitive to the types of workloads. Although both the 2Q and the LIRS algorithms have simple implementations with low overheads, LIRS has overcome the drawbacks of 2Q by a properly updating of the LIR block set. Another recent algorithm, ARC, maintains two variable-sized lists [51]. Their combined size is two times of the number of pages that are held in the cache. one half of the lists contain the blocks in the cache and another half are for the history access information of replaced blocks. The first list contains blocks that have been seen only once recently and the second list contains blocks that have been seen at least twice recently. The cache spaces allocated to the blocks in these two lists are adaptively changed, depending on in which list recent misses happen. More cache spaces will serve cold blocks (resp. hot blocks) if there are more cold block (resp. hot block) accesses. However, though the authors advocate the superiority of the ARC algorithm by its adaptiveness and excluding tunable parameters, the locality of blocks in the two lists, quantified by recency or frequency, can not directly and consistently be compared. For example, a block that is regularly accessed with an IRR a little bit more than the cache size may have no hits at all while a block in the second list can stay in cache without any accesses since it has been accepted into the list.

Inter-Reference Gap (IRG) for a block is the number of the references between consecutive references to the block, which is different from IRR on whether duplicate references on a block are counted. Phalke and Gopinath considered the correlation between history IRGs and future IRG [62]. The past IRG string for each block is modeled by Markov chain to predict the next IRG. However, as Smaragdakis et. al. indicate, replacement algorithms based on a Markov models fail in practice because they try to solve a much harder problem

than the replacement problem itself [67]. An apparent difference in their scheme from our LIRS algorithm is on how to measure the distance between two consecutive references on a block. My study shows that IRR is more justifiable than IRG in this circumstance. First, IRR only counts the distinct blocks and filters out high-frequency events, which may be volatile with time. Thus the IRR is more relevant to the next IRR than the IRG to the next IRG. Moreover, it is the “recency” but not “gap” information that is used by LRU. An elaborate argument favoring IRR in the context of virtual memory page replacement can be found in [67]. Secondly, IRR can be easily dealt with under the LRU stack model [20], on which most popular replacements are based.

2.2.3 Detection and Adaptation of Access Regularities

More recently, researchers took another approach to detect access regularities from the history information by relating the accessing behavior of a block to those of the blocks in its temporal or spatial locality scope. Then different replacements, such as MRU, can be applied to blocks with specific access regularities.

Glass and Cao propose adaptive replacement SEQ for page replacement in virtual memory management[30]. It detects sequential address reference patterns. If long sequences of page faults are found, MRU is applied to such sequences. If no sequences are detected, SEQ performs LRU replacement. Smaragdakis et. al. argued that address-based detection lacks generality, and advocated using aggregate recency information to characterize page behaviors [67]. Their EELRU examines aggregate recency distributions of referenced pages and changes the page eviction points using an on-line cost/benefit analysis by assuming the correlation among temporally contiguously referenced pages, unlike LRU, which actually

always set the eviction point in the bottom of LRU stack. However, EELRU has to choose a eviction point from a pre-determined set of LRU stack positions. And how to select the set could affect its performance. Moreover, by aggregate analysis, EELRU can not quickly respond to the changing access patterns. Without spatial or temporal detections, our LIRS uses independent recency events of each block to effectively characterize their references.

Choi et. al. propose a new adaptive buffer management scheme called DEAR that automatically detects the block reference patterns of applications and applies different replacement policies to different applications based on the detected reference patterns [19]. Further, they propose an Application/File-level Characterization (AFC) scheme in [18], which first detects the reference characteristics at the application level, and then at the file level, if necessary. Accordingly, appropriate replacement policies are used to blocks with different patterns. The Unified Buffer Management (UBM) scheme by Kim et. al. also detects patterns in the recorded history [42]. Unlike the detection method proposed in [19], which associates the backward distance and frequency with the forward distances of blocks between two consecutive detection invocation points, UBM track the reference information such as the file descriptor, start block number, end block number, and loop period if re-reference occurs. Though their elaborate detection of block access patterns provide a large potential to high performance, they address the problems in a case-by-case fashion and have to cope with the allocation problem, which does not appear in LRU. To facilitate the on-line evaluation of buffer usage, certain pre-measurements are needed to set pre-defined parameters used in the buffer allocation scheme [18, 19]. My LIRS does not have these design challenges. Just as LRU does, it chooses the victim block in the global stack. However, it can use the advantages provided by the detection based schemes.

2.2.4 Working Set Models

Lastly, we would like to compare our work with the working set model, an early work by Denning [24]. A working set of a program is a set of its recently used pages. Specifically, at virtual time t , the program's working set $W_t(\theta)$ is the subset of all pages of the program, which have been referenced in the previous θ virtual time units (the working set window). A working set replacement algorithm is used to ensure that no pages in the working set of a running program will be replaced [25]. Estimating the current memory demand of a running program in the system, the model does not incorporate the available cache size. When the working set is greater than the cache size, working set replacement algorithm would not work properly. Another difficulty with the working set model is its weak ability to distinguish recently referenced "cold" blocks from "hot" blocks. My LIRS algorithm ensures that LIR block set size is less than the available cache size and keeps the set in the cache. IRR helps to distinguish the "cold" blocks from "hot" ones: a recently referenced "cold" block could have a small recency, but would have a large IRR.

2.3 The LIRS algorithm

2.3.1 General Idea

We divide the referenced blocks into two sets: High Inter-reference Recency (HIR) block set and Low Inter-reference Recency (LIR) block set. Each block with history information in cache has a status – either LIR or HIR. Some HIR blocks may not reside in the cache, but have metadata in the cache recording their statuses as non-resident HIR blocks. We also divide the cache, whose size in blocks is L , into a major part and a minor part in terms of

their sizes. The major part with the size of L_{lirs} is used to store LIR blocks, and the minor part with the size of L_{hirs} is used to store blocks from HIR block set, where $L_{lirs} + L_{hirs} = L$. When a miss occurs and a block is needed for replacement, we choose an HIR block that is resident in the cache. The LIR block set always resides in memory, i.e., there are no misses for the references to LIR blocks. However, a reference to an HIR block would be likely to encounter a miss, because L_{hirs} is very small (its practical size can be as small as 1% of cache size).

We use Table 2.1 as a simple example to illustrate how a replaced block is selected by the LIRS algorithm and how LIR/HIR statuses are switched. In Table 1, symbol “X” denotes a block access at a virtual time unit ¹. For example, block A is accessed at time units 1, 6, and 8. Based on the definition of recency and IRR in Chapter 2.1.2, at time unit 10, blocks A, B, C, D, E have their IRR values of 1, 1, “infinite”, 3, and “infinite”, respectively, and have their recency values of 1, 3, 4, 2, and 0, respectively. We assume $L_{lirs} = 2$ and $L_{hirs} = 1$, thus at the time 10 the LIRS algorithm leaves two blocks in the LIR set = {A, B}. The rest of the blocks go to the HIR set = {C, D, E}. Because block E is the most recently referenced, it is the only resident HIR block due to $L_{hirs} = 1$. If there is a reference to an LIR block, we just leave it in the LIR block set. If there is a reference to an HIR block, we need to know whether we should change its status to LIR.

The key to successfully make the LIRS idea work in practice rests on whether we are able to dynamically and responsively maintain the LIR block set and HIR block set. When an HIR block is referenced, it gets a new IRR equal to its recency. Then we determine whether the new IRR is small compared with reference statistics of existing LIR blocks, so

¹Virtual time is defined on the reference sequence, where a reference represents a time unit.

Blocks / Virtual time	1	2	3	4	5	6	7	8	9	10	Recency	IRR
E									x		0	inf
D		x					x				2	3
C				x							4	inf
B			x		x						3	1
A	x					x		x			1	1

Table 2.1: An example to explain how a victim block is selected by the LIRS algorithm and how LIR/HIR statuses are switched. A “X” refers the block of the row is referenced at the virtual time of the column. The recency and IRR columns represent the values at the virtual time 10 for each block. We assume $L_{lirs} = 2$ and $L_{hirs} = 1$, and at the time 10 the LIRS algorithm leaves two blocks in the LIR set = {A, B}, and the HIR set is {C, D, E}. The only resident HIR block is E.

that we can decide whether we need to change its status to LIR. Here we have two options: to compare it either with the IRRs or with the recencies of the LIR blocks. We choose the recencies for the comparison. There are two reasons for this: (1) The IRRs are generated before their respective recencies and are outdated, which are not directly relevant to the new IRR of the HIR block. A recency of a block is determined not only by its own reference activity, but also the recent activities of other blocks. The result of comparison of the new IRR and recencies of the LIR blocks determines the eligibility of the HIR block to be considered as a “hot block”. Though we claim that IRRs are used to determine which block should be replaced, it is the new IRRs that are directly used in the comparisons. (2) If the new IRR of the HIR block is smaller than the recency of an LIR block, it will be smaller than the upcoming IRR of the LIR block. This is because the recency of the LIR block is a part of its upcoming IRR, and not greater than the IRR. Thus the comparisons with the recencies are actually the comparisons with the relevant IRRs. Once we know that the new IRR of the HIR block is smaller than the maximum recency of all the LIR blocks, we switch the LIR/HIR status of the HIR block and the LIR block with the maximum recency.

Following this rule, we can (1) allow an HIR block with a relatively small IRR to join LIR block set in a timely way by removing an LIR block from the set; (2) keep the size of LIR block set no larger than L_{lirs} , thus the entire set can reside in the cache.

Again in the example of Table 1, if there is a reference to block D at time 10, then a miss occurs. LIRS algorithm evicts resident HIR block E, instead of block B, which would be evicted by LRU due to its largest recency. Furthermore, because block D is referenced, its new IRR becomes 2, which is smaller than the recency of LIR block B (=3), indicating that the upcoming IRR of block B will not be smaller than 3. So the status of block D is switched to LIR, and the block joins the LIR block set, while block B becomes an HIR block. Since block B becomes the only resident HIR block, it is going to be evicted from the cache once another free block is requested. If at virtual time 10, block C with its recency 4, rather than block D with its recency 2, gets referenced, there will be no status switching. Then block C becomes a resident HIR block, though the replaced block is still E at virtual time 10. The LIR block set and HIR block set are formed and dynamically maintained in this way.

2.3.2 The LIRS Algorithm Based on LRU Stack

The LIRS algorithm can be efficiently built on the model of LRU stack, which is an implementation structure of LRU. The LRU stack is a cache storage containing L entries, each of which represents a block². In practice, L is the cache size in blocks. LIRS algorithm makes use of the stack to record the recency, and to dynamically maintain the LIR block set and

²For simplicity, in the rest of the dissertation we just say without ambiguity “a block in the stack” instead of “the entry of a block in the stack” .

HIR block set. In contrast to the LRU stack, where only resident blocks are managed by LRU replacement in the stack, we store LIR blocks, and HIR blocks with their recency less than the maximum recency of LIRS blocks in a stack called LIRS stack S . S is similar to the LRU stack in operation but has variable size. With this implementation, we do not need to explicitly keep track of the IRR and recency values and to search for the maximum recency value. Each entry in the stack records the LIR/HIR status and residence status indicating whether or not the block resides in the cache. To facilitate the search of resident HIR blocks, we link all these blocks into a small list Q with its maximum size L_{hirs} . Once a free block is needed, the LIRS algorithm removes a resident HIR block from the front of the list for replacement. However, the replaced HIR block remains in the stack S with its residence status changed to non-resident, if it is originally in the stack. We ensure the block in the bottom of the stack S is an LIR block by removing HIR blocks below it. Once an HIR block in the LIRS stack gets referenced, which means there is at least one LIR block, such as the one at the bottom, whose upcoming IRR will be greater than the new IRR of the HIR block, we switch the LIR/HIR statuses of the two blocks. The LIR block at the bottom is evicted from the stack S and goes to the end of the list Q as a resident HIR block. This block will soon be evicted from the cache due to the small size of the list Q (at most L_{hirs}).

Such a scheme is intuitive from the perspective of LRU replacement behavior: if a block gets evicted from the bottom of LRU stack, it means the block occupies a buffer during the period of time when it moves from the top to the bottom of the stack without being referenced. Why should we afford a buffer for another long idle period when the block is loaded again into the cache? The rationale behind this is the assumption that temporal

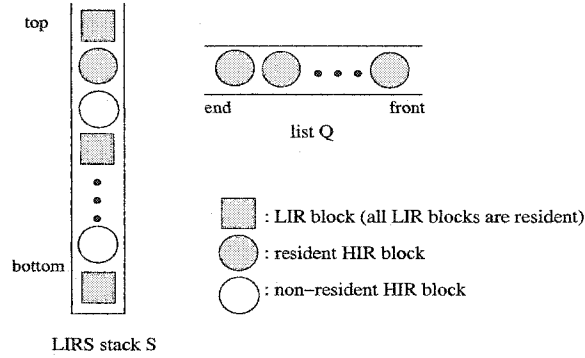


Figure 2.1: The LIRS stack S holds LIR blocks as well as HIRS blocks with or without resident status, and a list Q holds all the resident HIR blocks.

IRR locality holds for block references.

2.3.3 A Detailed Description

We define an operation called “stack pruning” on LIRS stack S , which removes the HIR blocks in the bottom of the stack until an LIR block sits in the stack bottom. This operation serves two purposes: (1) We ensure the block in the bottom of the stack always belongs to the LIR block set. (2) After the LIR block in the bottom is removed, those HIR blocks contiguously located above it will not have chances to change their statuses from HIR to LIR, because their recencies are larger than the new maximum recency of LIR blocks.

When LIR block set is not full, all the referenced blocks are given LIR status until its size reaches L_{lirs} . After that, HIR status is given to any blocks that are referenced for the first time, and to blocks that have not been referenced for a long time so that they are not in stack S any longer.

Figure 2.1 shows a scenario where stack S holds three kinds of block: LIR block, resident HIR block, non-resident HIR block, and a list Q holds all of the resident HIR blocks. An HIR block may either be in the stack S or not. Figure 2.1 does not depict non-resident HIR

blocks that are not in the stack S . There are three cases with various references to these blocks.

1. **Upon accessing an LIR block X :** This access is guaranteed to be a hit in the cache. We move it to the top of the stack S . If the LIR block is originally located at the bottom of the stack, we conduct a stack pruning. This case is illustrated in the transition from state (a) to state (b) in Figure 2.2 based on the example shown in Table 1.
2. **Upon accessing an HIR resident block X :** This is a hit in the cache. We move it to the top of the stack S . There are two cases for block X : (1) If X is in the stack S , we change its status to LIR. This block is also removed from list Q . The LIR block at the bottom of S is moved to the end of list Q with its status changed to HIR. A stack pruning is then conducted. This case is illustrated in the transition from state (a) to state (c) in Figure 2.2. (2) If X is not in the stack S , we leave its status in HIR and move it to the end of list Q .
3. **Upon accessing an HIR non-resident block X :** This is a miss. We remove the HIR resident block at the front of list Q (it then becomes a non-resident block), and evict it from the cache. Then we load the requested block X into the freed buffer and place it at the top of stack S . There are two cases for block X : (1) If X is in the stack S , we change its status to LIR and move the LIR block at the bottom of stack S to the end of list Q with its status changed to HIR. A stack pruning is then conducted. This case is illustrated in the transition from state (a) to state (d) in Figure 2.2. (2) If X is not in the stack S , we leave its status in HIR and place it at the end of list Q .

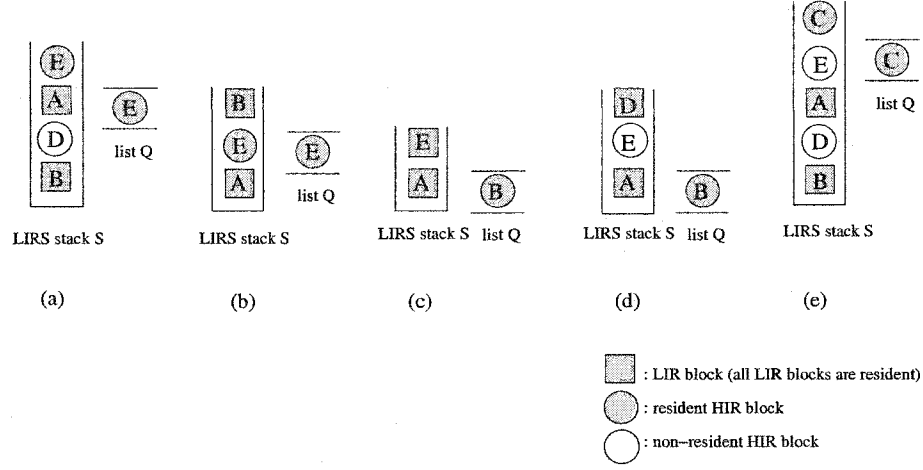


Figure 2.2: Illustration of the reference results in the example shown in Table 1 on the LIRS stack. In this figure, (a) corresponds to the state at virtual time 9. Accessing B, E, D, or C at virtual time 10 result in (b), (c), (d) and (e), respectively.

This case is illustrated in the transition from state (a) to state (e).

2.4 Performance Evaluation

2.4.1 Experimental Settings

To validate our LIRS algorithm and to demonstrate its strength, we use trace-driven simulations with various types of workloads to evaluate and compare it with other algorithms. We have adopted many application workload traces used in previous literature aiming at addressing limitations of LRU. We have also generated a synthetic trace. Among these traces, *cpp*, *cs*, *glimpse*, and *postgres* are used in [18, 19] (*cs* is named as *cscope* and *postgres* is named as *postgres2* there), *sprite* is used in [45], *muulti1*, *multi2*, *multi3* are used in [42], *OpenMail* and *Cello99* are used in [76]. We briefly describe the workload traces here. These traces represent a wide range of access patterns, sizes, sources and collecting times.

1. *2-pools* is a synthetic trace, which simulates application behavior of the example 3 in Chapter 2.1.1 with 100,000 references.
2. *cpp* is a GNU C compiler pre-processor trace. The total size of C source programs used as input is roughly 11 MB.
3. *cs* is an interactive C source program examination tool trace. The total size of the C programs used as input is roughly 9 MB.
4. *glimpse* is a text information retrieval utility trace. The total size of text files used as input is roughly 50 MB.
5. *postgres* is a trace of join queries among four relations in a relational database system from the University of California at Berkeley.
6. *sprite* is from the Sprite network file system, which contains requests to a file server from client workstations for a two-day period.
7. *mulit1* is obtained by executing two workloads, *cs* and *cpp*, together.
8. *multi2* is obtained by executing three workloads, *cs*, *cpp*, and *postgres*, together.
9. *multi3* is obtained by executing four workloads, *cpp*, *gnuplot*, *glimpse*, and *postgres*, together.
10. *OpenMail* is a trace of a production e-mail system running the HP OpenMail application.
11. *Cello99* is a trace of every disk I/O access for the month of April 1999 from an HP 9000 K570 server.

Because a well-designed replacement algorithm should perform well under various access patterns exhibited in workloads, we select traces 1-9, which are in relatively small scales, but cover a wide range of file access patterns to compare the performance of LIRS with other proposed algorithms. Then we use the two large scale traces, *OpenMail* and *Cello99*, to test the effectiveness of LIRS with applications on state-of-art, high end server systems. The only parameter of the LIRS algorithm, L_{hirs} , is set as 1% of the cache size, or $L_{lirs} = 99\%$ of the cache size. This selection results from a sensitivity analysis to L_{hirs}/L_{lirs} , which is described in Chapter 2.5.1.

2.4.2 Access Pattern Based Performance Evaluation

Through an elaborate investigation, Choi et. al. classify the file cache access patterns into four types [18]:

- Sequential references: all blocks are accessed one after another, and never re-accessed;
- Looping references: all blocks are accessed repeatedly with a regular interval (period);
- Temporally-clustered references: blocks accessed more recently are the ones more likely to be accessed in the future;
- Probabilistic references: each block has a stationary reference probability, and all blocks are accessed independently with the associated probabilities.

The classification serves as a basis for their access pattern detections and for adapting to different replacement policies. For example, MRU applies to sequential and looping patterns, LRU applies to temporally-clustered patterns, and LFU applies to probabilistic

patterns. Though our LIRS policy does not depend on such a classification, we would like to use it to present and explain our experimental results. Because a sequential pattern is a special case of looping pattern (with infinite interval), we only use the last three groups: looping, temporally-clustered, and probabilistic patterns.

Policies LRU, LRU-2, 2Q, ARC, LRFU, and LIRS belong to the same category of replacement policies. In other words, these policies take the same technical direction — predicting the access possibility of a block through its own history access information. Thus, we focus our performance comparisons between ours and these policies. As representative policies in the category of regularity detections, we choose two schemes for comparisons: UBM for its spatial regularity detection, and EELRU for its temporal regularity detection. UBM simulation requires file IDs, offsets, and process IDs of a reference. However, some traces available to us only consists of logical block numbers, which is an unique number for each accessed block. Thus, we only include the UBM experimental results for the traces used in paper [42], which are *multi1*, *multi2*, *multi3*. We also include the results of OPT, an optimal, off-line replacement algorithm [20] for comparisons.

We divide traces 1-9 into 4 groups based on their access patterns. Traces *cs*, *postgres*, and *glimpse* belong to the looping type, traces *cpp* and *2-pools* belong to the probabilistic type, trace *sprite* belongs to the temporally-clustered type, and traces *multi1*, *multi2*, and *multi3* belong to the mixed type.

We present performance results for each trace by a pair of figures: the time-space maps and the hit rate curves. In a time-space map, the x axis represents virtual time, the reference sequence of a given workload, and the y axis plots the logical block numbers of those referenced. The hit rate curves show the hit rates as the cache size increases for

various replacement policies on a workload trace.

2.4.2.1 Performance for the Looping Type

Figures 2.3 to 2.5 plot four pairs of time-space maps (left figures) and the hit rate curves (right figures) generated by the various replacement policies for traces *cs*, *glimpse*, and *postgres*, respectively. The time-space maps show that all the 4 programs have looping patterns with long intervals. As expected, LRU performs poorly for these workloads with the lowest hit rates. Let us take *cs* as an example, which has a pure looping pattern. Each block is accessed almost at the same frequency (see the left figure in Figure 2.3). Since all blocks in a loop have the same eligibility to be kept in cache, it is desirable to keep the same set of blocks in cache no matter what blocks are referenced currently. That is just what LIRS does: the same LIR blocks are fixed in the cache because HIR blocks do not have IRRs small enough to change their statuses. In the looping pattern, recency predicts the opposite of the future reference time of a block: the larger the recency of a block is, the sooner the block will be referenced. The hit rate of LRU for *cs* is almost 0% until the cache size approaches 1,400 blocks, which can hold all the blocks referenced in the loop. It is interesting to see that the hit rate curve of LRU-2 overlaps with the LRU curve. This is because LRU-2 chooses the same victim block as the one chosen by LRU for replacement. When making a decision, LRU-2 compares the penultimate reference time, which is the recency plus the recent IRG. However, the IRGs are the same for all the blocks at any time after the first reference. Thus, LRU-2 relies only on recency to make its decision, the same as LRU does. Generally, when recency makes a major contribution to the penultimate reference time, LRU-2 behaves similarly to LRU.

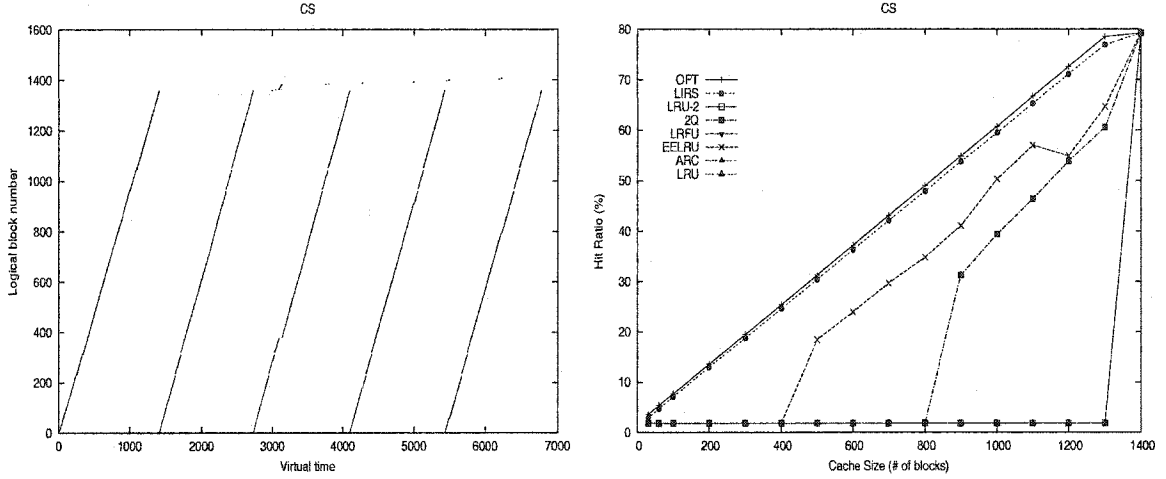


Figure 2.3: The time-space map (left) of *cs* and the hit rate curves by various replacement policies (right).

Except for *cs*, the other three workloads have mixed looping patterns with different intervals. LRU presents a stair-step curve to increase the hit rates for those workloads. LRU is not effective until all the blocks in its locality scope are brought into the cache. For example, only after the cache can hold 355 blocks does the LRU hit rate of *postgres* have a sharp increase from 16.3% to 48.5% (see the right figure in Figure 2.5). Because LRU-2 considers the last IRG in addition to the recency, it is easier for it to distinguish blocks in the loops with different intervals than LRU does. However, LRU-2 lacks the capability to deal with these blocks when varying recency is involved. My experiments show that the achieved performance improvements by LRU-2 over LRU is limited. (see the right figures in Figures 2.4 and 2.5).

It is illuminating to observe the performance difference between 2Q and LIRS, because both employ two linear data structures following a similar principle that only re-referenced blocks deserve to be in cache for a long period of time. We can see that the hit rates of 2Q are significantly lower than those of LIRS for all the three workloads (see the right figures in

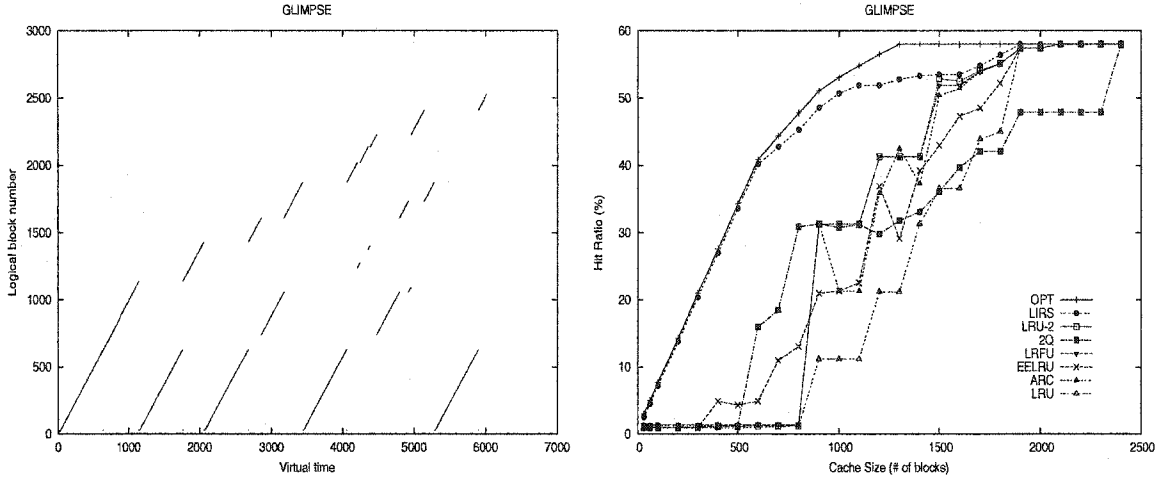


Figure 2.4: The time-space map (left) of *glimpse* and the hit rate curves by various replacement policies (right).

Figures 2.3, 2.4, and 2.5). As the cache size increases, 2Q even performs worse than LRU for workloads *glimpse* and *postgres*. Another observation for 2Q on *glimpse* and *postgres* is a serious “Belady’s anomaly” [8]: increasing the size of cache size may increase the number of misses. Though ARC is an adaptive algorithm without tunable parameters, it actually shares the same problem as that of 2Q. The performance improvement of ARC over LRU is very limited. Belady’s anomaly also appears in the workload *glimpse* for ARC. This is mainly caused by the inconsistent quantification and comparison of locality of blocks in two lists, which is effectively addressed in LIRS. We will provide a in-depth analysis on this issue in Section 2.4.4.

LRFU, which combines LRU and LFU, is not effective on a workload with a looping pattern, because reference frequencies are hard to distinguish for looping references. The LRFU and LRU hit rate curves for workload *cs* are overlapped, which is shown in Figure 2.3.

My trace-driven simulation results show LIRS significantly outperforms all of the other

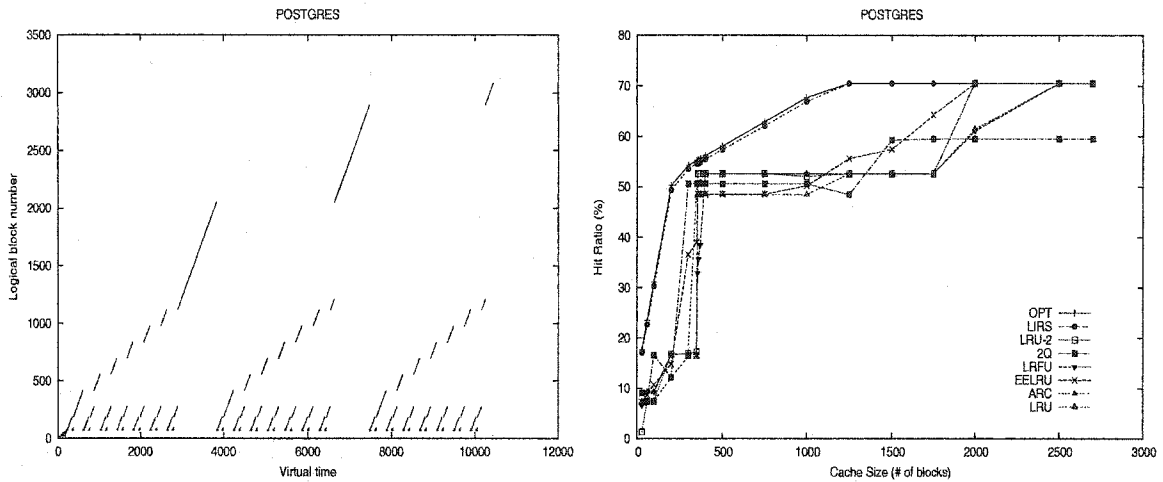


Figure 2.5: The time-space map (left) of **postgres** and the hit rate curves by various replacement policies (right).

policies, and its hit rate curves are very close to that of OPT. LIRS can make a more accurate prediction on the future LIR/HIR status of each block for *cs* and *postgres* than *glimpse*, because the intervals of loops in *cs* and *postgres* are of less variance, thus the consecutive IRRs are of less variance (See the performance difference among *cs*, *postgres* in Figures 2.3, and 2.5 and *glimpse* in Figure 2.4. However, the LIRS algorithm is not sensitive to the variance of IRRs, which is reflected by its good performance on workload *glimpse*. We explain it as follows.

We denote the recency of the LIR block in the bottom of LIRS stack S as $Rmax$. When there are no free block buffers, $Rmax$ is larger than the cache size in blocks. Only when the two consecutive IRRs of references to a block vary across value $Rmax$, is the status prediction of the LIRS algorithm based on the last IRR wrong, including two cases: (1) an IRR less than $Rmax$ is succeeded by another IRR greater than $Rmax$, and (2) an IRR greater than $Rmax$ is succeeded by another IRR less than $Rmax$. All other IRR variances, no matter how much they are, would impose no mishandling of the LIRS replacement. Let

us take a close look at the penalty from a wrong LIR/HIR status decision: (1) Suppose a block is labeled as LIR (due to its previous, small IRR) when it should be labeled as HIR. The block will be evicted by LIRS after L_{lirs} references (i.e., when the block reaches the bottom of stack S), instead of being evicted after L_{hirs} references. Since L_{lirs} is almost as large as L, the performance penalty imposed by the LIRS mis-classification is no worse than that imposed by LRU. (2) Suppose a block is labeled as HIR (due to its previous, large IRR) when it should be labeled as LIR. The block will be evicted by LIRS far before it reaches the stack bottom, instead of being hit by a reference before it reaches the stack bottom. Thus LIRS would incur an extra miss if the block had been evicted from HIR resident list Q. However, because the number of block buffers assigned to list Q (L_{hirs}) is very small, which is only 1% of total cache size in our experiments, HIR blocks would be replaced very soon, which reduces the chance for the replaced block to be re-referenced shortly after its eviction. The free block buffer for the period between the early eviction and its next reference helps to reduce the penalty from the extra misses.

2.4.2.2 Performance for the Probabilistic Type

Figures 2.6 and 2.7 plot two pairs of time-space maps (left figures) and the hit rate curves (right figures) generated by the various replacement policies for workloads *cpp* and *2-pools*, respectively. According to the detection results in [18], workload *cpp* exhibits probabilistic reference patterns. The right figure in Figure 2.6 shows that before the cache size increases to 100 blocks, the hit rate of LRU is much lower than that of LIRS for *cpp*. For example, when the cache size is 50 blocks, hit rate of LRU is 9.3%, while hit rate of LIRS is 55.0%. This is because holding a major reference locality needs about 100 blocks (see the left figure

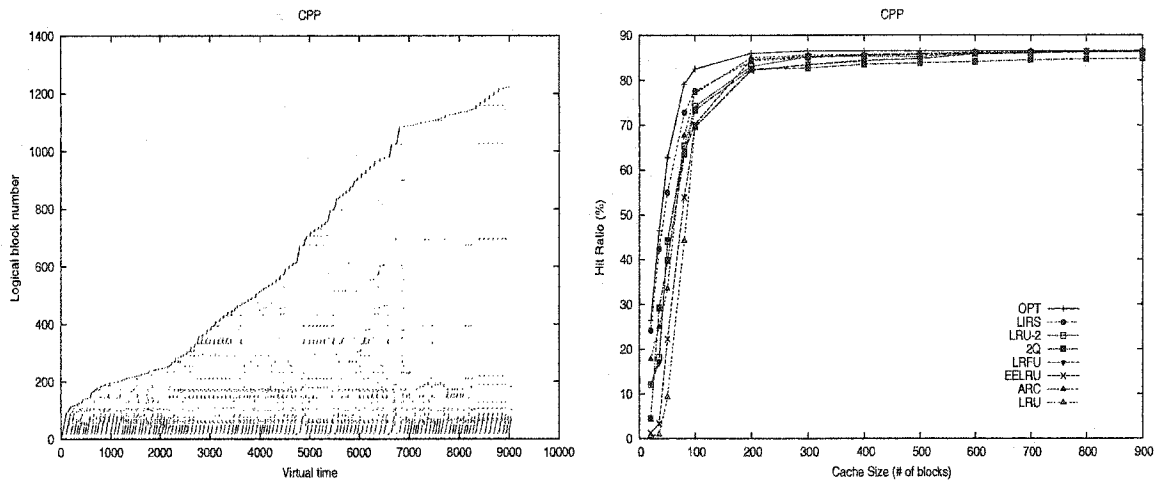


Figure 2.6: The time-space map (left) of *cpp* and the hit rate curves by various replacement policies (right).

of Figure 2.6). LRU can not exploit locality until enough cache space is available to hold all the recently referenced blocks. However, the capability for LIRS to exploit locality does not depend on the cache size – when it is identifying the LIR set to keep them in the cache, it always let the set size match the cache size. Workload *2-pools* is generated to evaluate the replacement policies on their abilities to recognize the long-term reference behavior. Though the reference frequencies are largely different between the record blocks and the index blocks, it is hard for LRU to distinguish them when the cache size is relatively small to the number of referenced blocks, because LRU takes only recency into consideration. LRU-2, 2Q, and LIRS algorithms take one more previous references into consideration — the time for the penultimate reference on a block is involved. Even though the reference events to a block are randomized (the IRRs on a block are random with a certain fixed frequency, which is unfavorable to LIRS.), LIRS still outperforms LRU-2 and 2Q (see the right figure in Figure 2.7). However, LRFU utilizes “deeper” history information. Thus, the constant long-term frequency becomes more visible, and is ready to be utilized by the

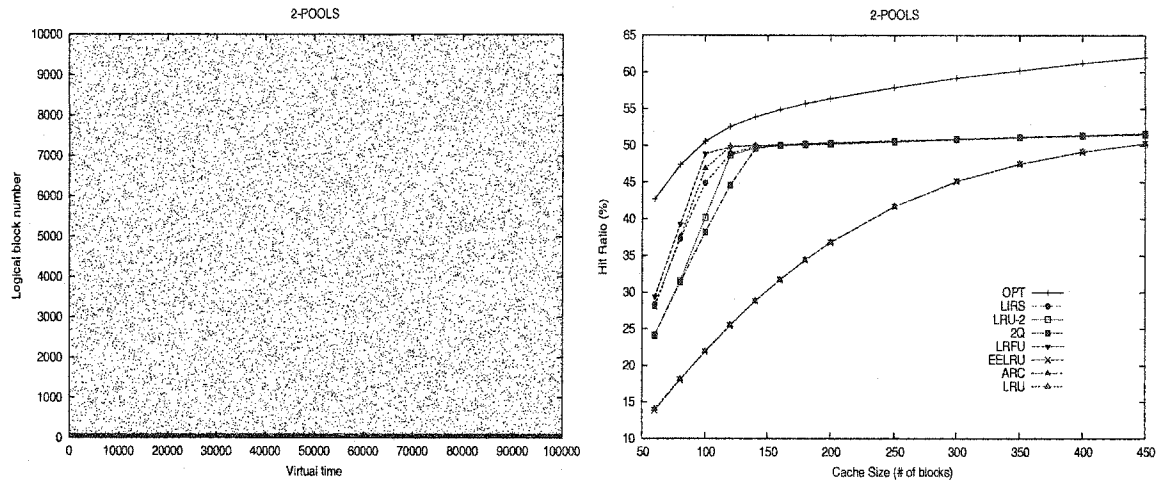


Figure 2.7: The time-space map (left) of *2-pools* and the hit rate curves by various replacement policies (right).

LFU-like scheme. The performance of LRFU is slightly better than that of LIRS. It is not surprising to see the hit rate curve of EELRU exhibits the poor performance and overlaps with that of LRU, because EELRU relies on an analysis of a temporal recency distribution to decide whether to conduct an early point eviction. In workload *2-pools*, the blocks with high access frequency and the blocks with low access frequency are alternatively referenced, thus no sign of an early point eviction can be detected.

2.4.2.3 Performance for the Temporally-Clustered Type

Figure 2.8 presents the time-space map (left figure) and the hit rate curves (right figure) generated by the various replacement policies for workload *sprite*, which exhibits temporally-clustered reference patterns. The right figure in Figure 2.8 shows that the LRU hit rate curve smoothly climbs with the increase of the cache size. Although there is still a gap between the LRU and OPT, the slope of the LRU is close to that of OPT. *Sprite* is a so called LRU-friendly workload [67], which seldom accesses more blocks than the cache size

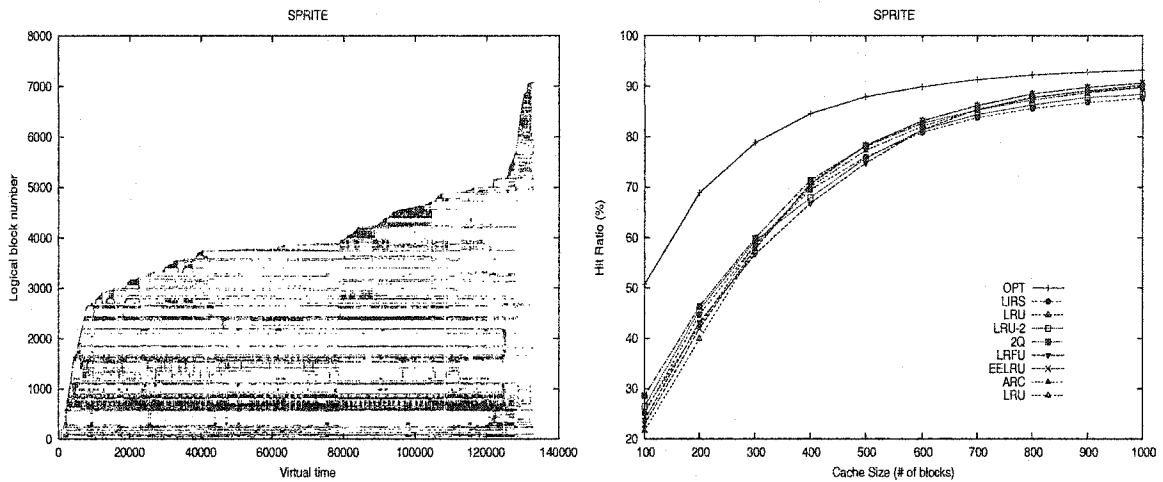


Figure 2.8: The time-space map (left) of *sprite* and the hit rate curves by various replacement policies (right).

over a fairly long period of time. For this type of workload, the behavior of all the other policies should be similar to that of LRU, so that their hit rates could be close to that of LRU. Before the cache size reaches 350 blocks, the right figure in Figure 2.8 shows that the hit rate of LIRS is higher than that of LRU. After this point, the hit rates of LRU is slightly higher. Here is the reason for the slight performance degradation of LIRS beyond that cache size: whenever there is a locality scope shift or transition, i.e. some HIR blocks get referenced, one more miss than would occur in LRU may be experienced by each HIR block. Only the next reference to the block in the near future after the miss makes it switch from HIR to LIR status and then remain in the cache. However, because of the strong locality, there are not frequent locality scope changes. So the negative effect of the extra misses is very limited.

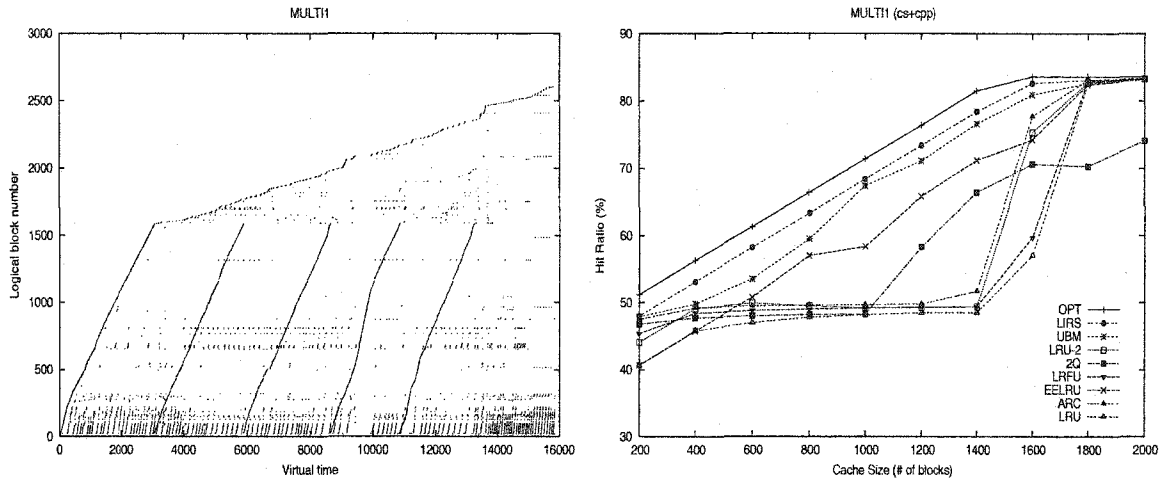


Figure 2.9: The time-space map (left) of *multi1* and the hit rate curves by various replacement policies (right).

2.4.2.4 Performance for the Mixed Type

Figures 2.9 to 2.11 present three pairs of time-space maps (left figures) and the hit rate curves (right figures) generated by the various replacement policies for workloads *multi1*, *multi2*, and *multi3*, respectively. The authors in [42] provide a detailed discussion why their UBM shows the best performance among the policies they have considered – UBM, LRU-2, 2Q, and EELRU. Here we focus on performance differences between LIRS and UBM. UBM is a typical spatial regularity detection-based replacement policy that makes an exhaustive reference pattern detections. UBM tries to identify sequential and looping patterns and applies MRU to the detected patterns. UBM further measures looping intervals and conducts period-based replacements. For unidentified blocks, LRU is applied. A dynamical buffer allocation among blocks managed by different policies is employed. Without devoting specific effort to specific regularities, LIRS outperforms UBM for all the three mixed type workloads, which shows that our assumption on IRR well holds and LIRS is able to cope with weak locality reference in the workloads with mixed type patterns.

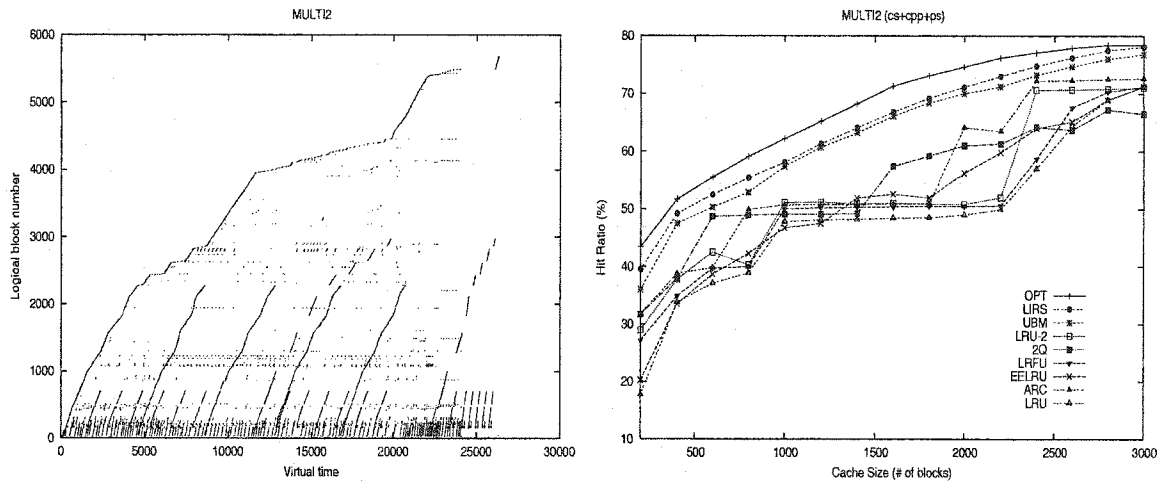


Figure 2.10: The time-space map (left) of *multi2* and the hit rate curves by various replacement policies (right).

2.4.3 LIRS Performance with High End Systems

Modern high end server systems can have a couple of giga-bytes of memory. Moreover, state-of-art high-end disk arrays typically have several giga-bytes of cache RAMs, which are mainly used as low-latency pools of data that is accessed multiple times by the connected servers. We have two issues to investigate for the high end systems: (1) whether LRU becomes competent enough to deal with the workloads on those systems equipped with large amount of memory? (2) whether LIRS can help in such as a system environment once LRU under-performs? We use two large-scale workload traces for this investigation, *OpenMail* and *Cello99*, which have been used in [76] by Wong and Wilkes for their study of the effective use of cache RAM in disk arrays.

OpenMail was collected on an HP OpenMail email system for 25,700 users, 9,800 of whom were active during the hour-long trace, containing about 5.4M I/O requests. The system is configured by six HP 9000 K580 servers running HP-UX 10.20, each with 6 CPUs, 2GB of memory, and 7 SCSI interface cards. The original traces are collected on the six

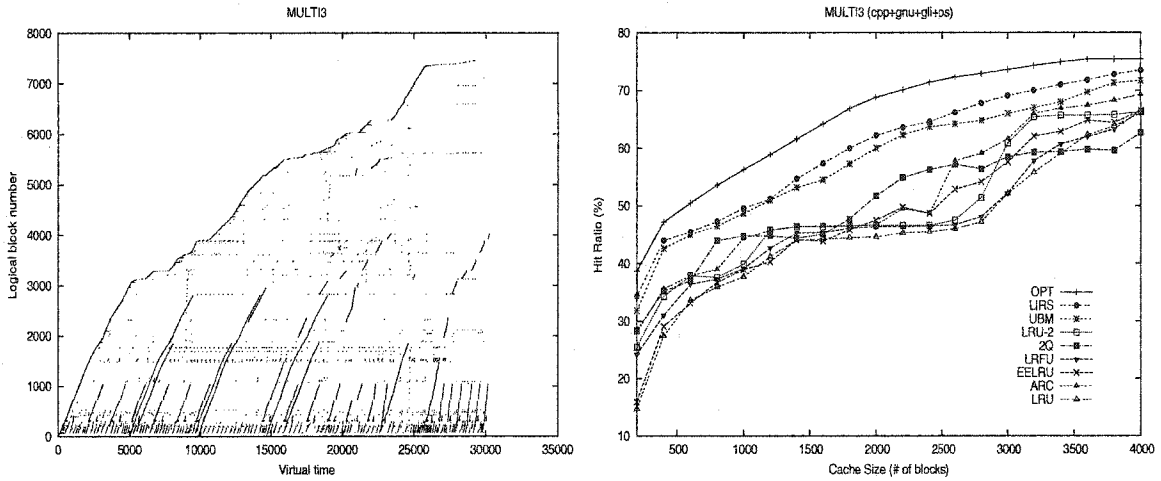


Figure 2.11: The time-space map (left) of *multi3* and the hit rate curves by various replacement policies (right).

nodes separately. We aggregated the six request streams into a single stream in the order of their request times. Trace *Cello99* is a collection of recorded disk I/O accesses for the month of April 1999 from an HP 9000 K570 server. The trace contains about 61.9M I/O requests. The server has 4 CPUs, about 2GB of main memory, two HP AutoRAID arrays and 18 directly connected disk drives. The system ran a general time-sharing load under HP-UX 10.20.

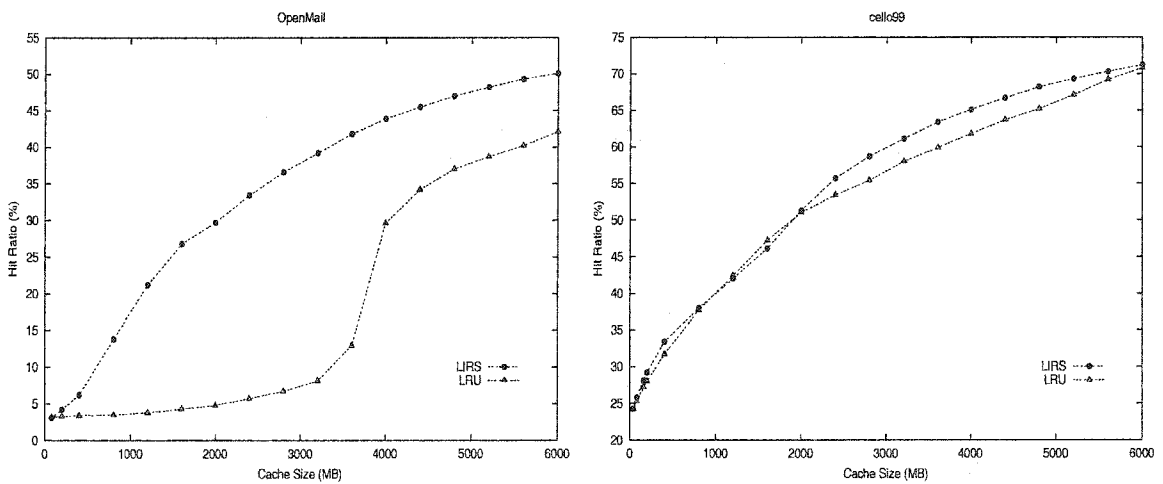


Figure 2.12: The hit rate curves of workload *OpenMail* (left figure) and workload *Cello99* (right figure)

Figure 2.12 shows the hit rates of *OpenMail* and *Cello99* with a range of large file cache size. For *OpenMail*, LRU seriously under-performs, where significantly increasing of cache size before cache size reaches 3GB yields little improvements on hit rates, while LIRS shows much better performance than LRU by its steadily increasing hit rates. While examining the trace, We found that 60.3% of its blocks are accessed only once, while the references to other blocks exhibit random access characteristics. LRU allows each of those once-accessed blocks holding a buffer space for at least L reference times, where L is the cache size in blocks. This actually reduces the number of buffers used for caching re-use blocks, which can contribute to the hit rates. LIRS replaces those once-accessed blocks shortly after they are accessed, so it makes more buffers available to the re-use blocks. In general, *Cello99* is an LRU-friendly workload, where its LRU hit rates get a steady increase with the increase of cache size. LIRS performs on the workload closely to LRU. Note that LRU becomes a little less effective after the cache size exceeds 2GB: the contribution of increased cache sizes to its hit rate is reduced. In comparison, LIRS produces better performance than LRU, which implies that LIRS can effectively overcome LRU inability. The experiments on the two large scale workload traces shows that the performance of LRU is susceptible to the workload access characteristics. LRU could under-perform on various system settings when workload access patterns are not friendly to it. They also show the effectiveness of LIRS to overcome LRU's inability on high end systems.

2.4.4 LIRS versus Other Stack-Based Replacements

To get insights of superiority of LIRS over other stack-based replacement algorithms, including LRU, 2Q, we would like to use time-IRR graph to observe their actions on the

blocks accessed at different recencies. In the graph, x axis plots virtual time, references in the access stream, y axis plots IRR, the recency where the reference at a specific time takes place. For the first time accessed blocks, their IRR is infinite, which we do not plot in the graph. We select two representative workloads, a non-LRU-friendly one, *postgres*, and an LRU-friendly one, *sprite*, for this study. Their IRRs are depicted in Figure 2.13.

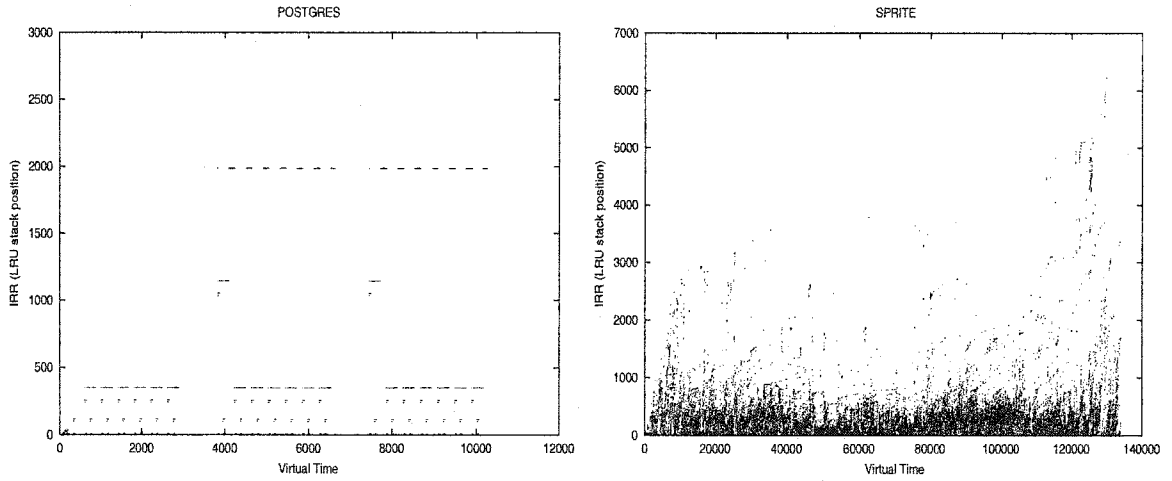


Figure 2.13: The IRRs of references of the workloads **postgres** (left) and **sprite** (right)

The stack size in LRU, which is determined by the cache size in blocks, is fixed. If the stack size is L , all the references shown in the graphs with their IRRs less than L are hits, and those with IRRs larger than L are misses in LRU. Thus the hit rates of LRU are directly determined by the IRR distribution. If most of IRRs are concentrated in the low recency area such as what is shown in the graph for *sprite*, LRU will get a high hit rate. For workloads with dispersed recency distributions, LRU is incompetent in achieving high hit rates. For example, in workload *postgres* there are two IRR concentrations at around IRRs 350, 1150 and 1950 in the left figure of Figure 2.13. In corresponding to the IRR distribution, there are obvious “lift ups” in the LRU hit rate curve when the cache size

reach these values, which is shown in Figure 2.5. However, if the number of blocks with their IRRs less than L is significantly less than stack size L , a large number of blocks with low recencies but high IRRs hold the stack spaces (residing in the cache) without being accessed before being replaced from the stack. The occupied buffers do not contribute to the hit rate. Thus what really matters is IRR, not recency. To improve LRU, the criterion to determine which accessed blocks are to be cached should be the L blocks with smallest IRRs, rather than the L blocks with their recencies less than L . Following this criterion, LIRS algorithm uses the LIRS stack to dynamically predict the L blocks which will have the smallest IRRs. The LIRS stack serves for two purposes: (1) holding the L blocks with smallest IRRs, called LIR blocks; (2) providing a threshold for being a LIR block. In our algorithm the threshold is $Rmax$, the recency of the LIR block in the LIRS stack bottom. The LIRS stack contains blocks with their recencies less than $Rmax$. Thus the threshold is also the LIRS stack size.

2.4.4.1 LIRS Threshold and Access Characteristics

To get the insights between the relationship of the threshold used by LIRS and workload access characteristics, we plot the ratio of the LIRS stack size, $Rmax$, and the size of the LRU stack L in Figure 2.14, when we fix the cache size at 500 blocks for the two workloads *postgres* and *sprite*. We find that the threshold is an inherent reflection of the LRU capability to exploit locality. If the references have a strong locality, most of the references are to the blocks with small recencies. Thus LRU stack still hold these blocks while they get re-accessed, and LRU achieves a high hit rate. At the same time, these blocks are low IRR blocks, i.e. most of the references go to the LIR blocks, which would

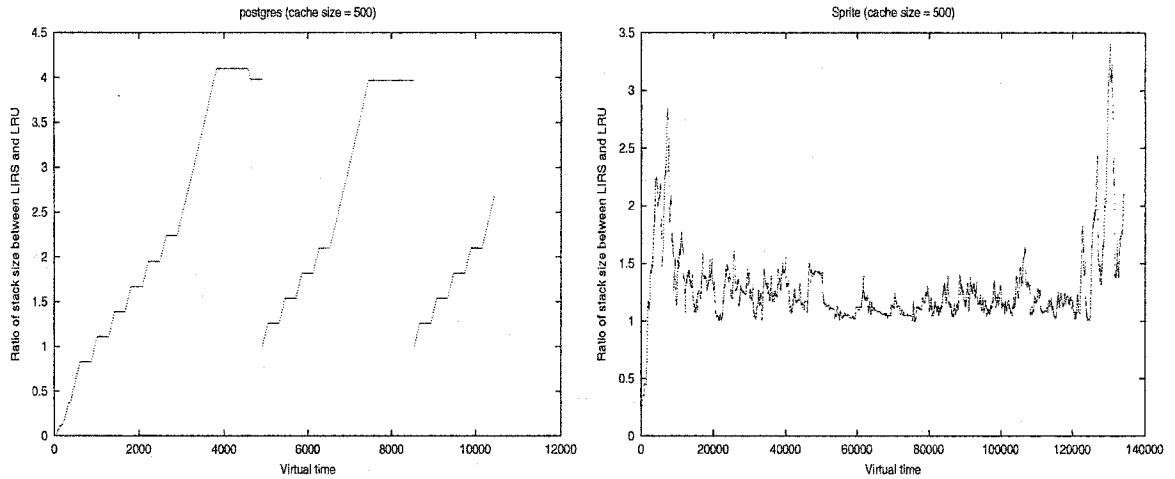


Figure 2.14: The rates of R_{max} and cache size in blocks (L) for workloads *postgres* (left) and *sprite* (right). R_{max} is the size of LIRS stack, which changes with virtual time. Cache size is 500.

leave only a small number of HIR blocks in the LIRS stack and cause the stack to shrink. This is the case for workload *sprite*. With 500 buffer blocks LRU stack is able to hold most the frequently referenced blocks (see right figure of Figure 2.13). On the other hand, LIRS can find enough low IRR blocks within the recency range also covered by LRU stack. Thus there is no need for LIRS to raise its stack size significantly to hold large number of blocks with high recencies in the cache. This is evidenced in the right figure of Figure 2.14, where the ratios of LIRS and LRU stack sizes are not far from 1 for the most of period of time. However, once LIRS can not find enough low IRR blocks within the size of LRU stack, it will raise its size accordingly. We observe that the thresholds of *postgres* are significantly increased in several phases during the periods when more references went to the blocks with high recencies than to those with low recencies (see left figure of Figure 2.14). With 500 buffer blocks LRU with its fixed stack size can not capture the locality distinction among blocks with high recencies, and causes their references all missed. By increasing the stack size according to the current access characteristics, LIRS can make the distinction among

blocks with weak locality and make a wiser decision on the replacement than LRU. The experiments also hint that the threshold is a good indicator of the LRU-friendliness of a workload.

Replacement algorithm 2Q also tries to identify blocks with small IRRs and to hold them in cache. It relies on queue *Alout* to decide whether a block is qualified to promote to stack *Am* so that it can be cached for a long time, or consequently to decide whether a block in *Am* should be demoted out of *Am*. In 2Q, the size of *Alout* serves as a threshold to identify blocks with small IRRs, and *Am* holds these blocks. Because the threshold is intended to predict the blocks with L smallest IRRs among all accessed blocks, it should be related to the access characteristics of blocks in *Am*. Unfortunately, it is not in 2Q. The recommended size of *Alout* in paper [37] is 50% of the cache size. So the threshold used in 2Q is a constant $1.5L$, which would be a straight horizontal line with its x axis value at $1.5L$ in a time-IRR graph. This threshold would be too tight to let blocks join in *Am* when LIRS threshold is larger than $1.5L$, and be too loose to allow blocks to stay outside of *Am*. This explains why 2Q can not provide a consistently improved performance over LRU.

2.4.4.2 LRU as a Special Member of the LIRS Family

In LIRS algorithm, any HIR block with a new IRR smaller than the LIRS threshold can change into LIR status, and may demote a LIR block into HIR status. The threshold controls how easily a HIR block may become a LIR block, or how difficult it is for a LIR block to become a HIR one. We would like to vary the threshold value so we will have a family of LIRS algorithms with various thresholds in order to get insights into the relationship of LRU and LIRS. Lowering the threshold value, we are able to strengthen the

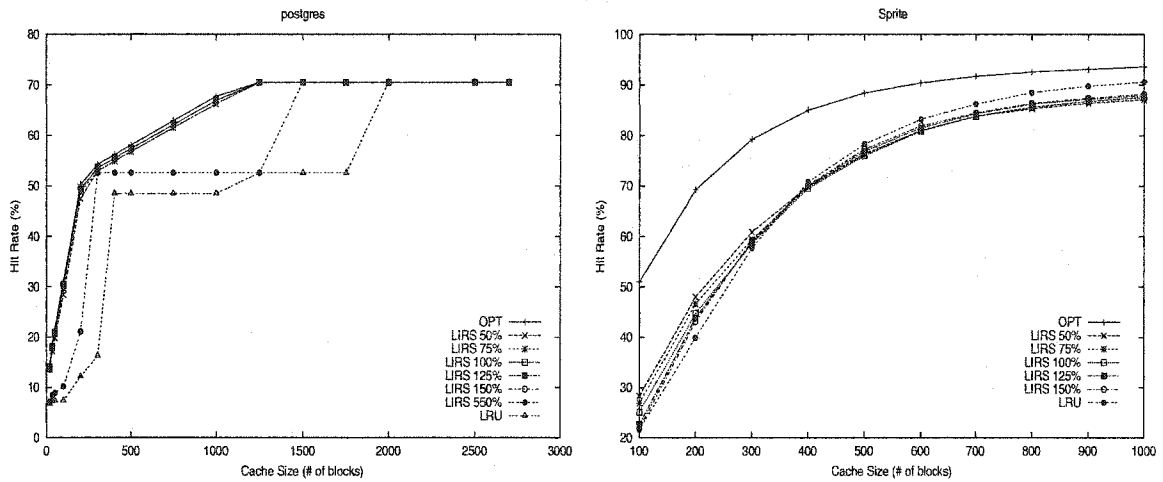


Figure 2.15: The hit rate curves of workload *postgres* (left figure) and workload *sprite* (right figure) by varying the rates of threshold values for LIR/HIR status switching and R_{max} in LIRS, as well as curves for OPT and LRU.

stability of the LIR block set by making it more difficult for HIR blocks to switch their status into LIR. It also prevents LIRS algorithm from responding to the relatively small IRR variance. Increasing the threshold value, we go in the opposite direction. Then LRU becomes a special member of the LIRS family – a LIRS algorithm with an indefinitely large threshold, which always gives any accessed block LIR status and keeps it in cache until it is evicted from the bottom of stack.

Figure 2.15 presents the results of a sensitivity study of the threshold value. We again use workloads *postgres* and *sprite* to observe the effect of changing the threshold values from 50%, 75%, 100%, 125% to 150% of R_{max} . For *postgres*, we include a very large threshold value – 550% of R_{max} to highlight the relationship between LIRS and LRU. We have two observations. First, LIRS is not sensitive to the threshold values across a large range. In *postgres*, curves for the threshold values of 100%, 125%, 150% of R_{max} are overlapped, and curves for 50%, 75% of R_{max} are slightly lower than that of the curve with 100% of R_{max} threshold. Specifically for *sprite*, an LRU-friendly workload, increasing the

threshold value, the LIRS hit rate curves move very slowly close to that of LRU. Secondly, the LIRS algorithm can simulate LRU behavior by largely increasing the threshold. As the threshold value increases to 550% of $Rmax$, LIRS curve of workload *postgres* is very similar to that of LRU in its shape, and close to it (See the left figure of Figure 2.15). Further increasing the threshold value, the LIRS curve overlaps with that of LRU.

2.5 Sensitivity and Overhead Analysis

2.5.1 Size Selection of List Q Holding Resident HIR Blocks (L_{hirs})

L_{hirs} is the only parameter in the LIRS algorithm. The blocks in the LIR block set can stay in the cache for longer time than those in the HIR block set and experience less page faults. An sufficiently large L_{lirs} (the cache size for LIR blocks) ensures there are a large number of LIR blocks. For this purpose, we set L_{lirs} to be 99% of the cache size, L_{hirs} to be 1% of the cache size in our experiments, and achieve expected performance. From the other perspective, an increased L_{hirs} may be beneficial to the performance: it reduces the first time reference misses. For a longer list Q (larger L_{hirs}), it is more likely that an HIR will be re-accessed before it is evicted from the list, which can help the HIR block change into LIR status without experiencing an extra miss. However, the benefit of large L_{hirs} is very limited, because the number of such kind of misses is small.

We also use the two workloads, *postgres* and *sprite*, to observe the effect of changing the size. We change L_{hirs} from 2 blocks, to 1%, 10%, 20%, and 30% of the cache size. Figure 2.16 presents the results of a sensitivity study of L_{hirs} for *postgres* (left figure) and *sprite* (right figure). For each workload, we measure the hit rates of OPT, LRU, and LIRS with different

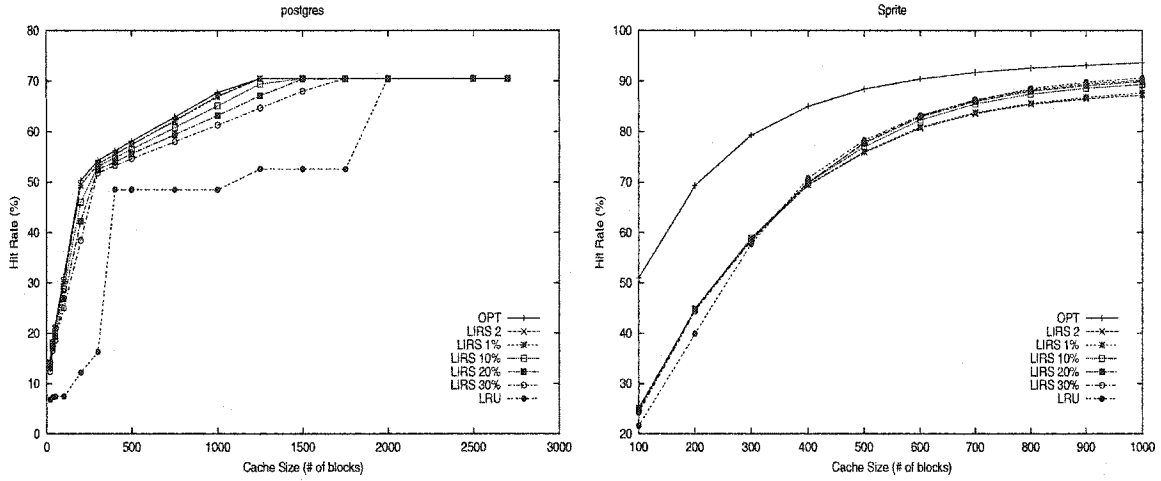


Figure 2.16: The hit rate curves of workload *postgres* (left figure) and workload *sprite* (right figure) by varying the size of list Q (L_{hirs} , the number of cache buffers assigned to HIR block set) of LIRS algorithm, as well as curves for OPT and LRU. “LIRS 2” means size of Q is 2, “LIRS x%” means size of Q is x% of the cache size in blocks.

L_{hirs} sizes by increasing the cache size. We have following two observations. First, for both workloads, we find that LIRS is not sensitive to the increase of L_{hirs} . Even for a very large L_{hirs} that is not in favor of LIRS, the performance of LIRS with different cache sizes is still quite acceptable. With the increase of L_{hirs} , the hit rate of LIRS approaches that of LRU. Secondly, our experiments indicate that increasing L_{hirs} reduces the performance benefits of LIRS to workload *postgres*, but slightly improves performance of workload *sprite*.

2.5.2 Overhead Analysis

LRU is known for its simplicity and efficiency. Comparing the time and space overhead of LIRS and LRU, we show that LIRS keeps the LRU merit of low overhead. The time overhead of LIRS algorithm is $O(1)$, which is almost the same as that of LRU with a few additional operations such as those on the list Q for resident HIR blocks. The extended portion of the LIRS stack S is the additional space overhead of the LIRS algorithm.

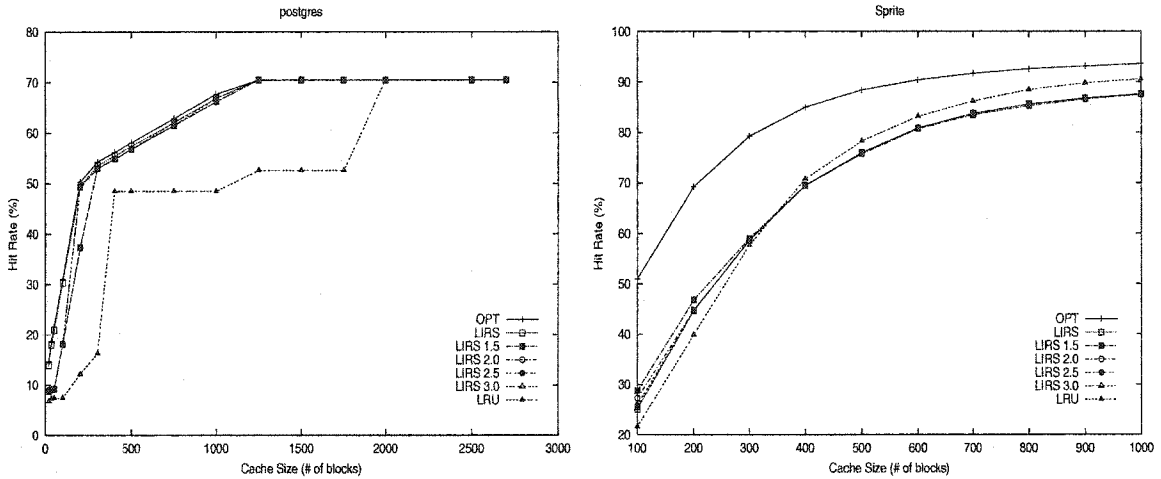


Figure 2.17: The hit rate curves of workload **postgres** (left) and workload **sprite** (right) by varying the LIRS stack size limits, as well as curves for OPT and LRU. Limits are represented by rates of LIRS stack size limit in blocks and cache size in blocks (L).

The stack S contains metadata for the blocks with their recency less than $Rmax$. When there is a burst of first-time (or “fresh”) block references, the LIRS stack could be extended to be unacceptably large. To give a size limit is a practical issue in the implementation of the LIRS algorithm. In an updated version of LIRS, the LIRS stack has a size limit that is larger than L , and we remove the HIR blocks close to the bottom from the stack once the LIRS stack size exceeds the limit. We have tested a range of rather small stack size limits, from 1.5 times to 3.0 times of L . From Figure 2.17, we can observe that even with these strict space restrictions, LIRS retains its desired performance. The effect of limiting LIRS stack size is equivalent to reducing the threshold values in Chapter 2.4.4.2. As expected, the results are consistent with the ones presented in Chapter 2.4.4.2. In addition, a stack entry only consists of several bytes, it is easily affordable to have LIRS stack size limit much more than 3 times of LRU stack size. With such large limits, there is little negative effect on LIRS performance by removing HIR block entries close to the stack bottom because of the size limit. By moderately extending the LRU stack size, LIRS makes a large difference on

its performance. This is because our solution fundamentally address the critical limitations of LRU.

2.6 Summary

We make two contributions in this work by proposing LIRS algorithm: (1) We show that LRU limitations with weak locality workloads can be successfully addressed without relying on the explicit regularity detection. By not depending on the detectable pre-defined regularities in the reference stream of workloads, my LIRS catches more opportunities to improve LRU performance. (2) We show earlier work on improving LRU such as LRU-K or 2Q can be evolved into one algorithm with consistently superior performance, without tuning or adaptation of sensitive parameters. The effort of these algorithms, which only trace their own history information of each referenced block, is promising because it is very likely to produce a simple and low overhead algorithm just like LRU. We have shown the LIRS algorithm accomplishes this goal.

My LIRS algorithm can be effectively applied in the virtual memory management for its simplicity and its LRU-like assumption on workload characteristics. In the next chapter, we will describe my design of a LIRS approximation, called CLOCK-Pro, with its reduced overhead comparable to that of LRU approximations, such as the CLOCK and second chance algorithms.

Chapter 3

Virtual Memory Replacement Policies

With the ever-growing performance gap between memory systems and disks, and rapidly improving CPU performance, virtual memory (VM) management becomes increasingly important for overall system performance. Because of the very stringent cost requirement on the replacement policies from VM management, almost all the general-purpose replacement algorithms cannot be directly applied here. The research of VM replacement policies is of special interests to operating system designers.

3.1 Background

3.1.1 The Research Status of Memory Replacement Policies

Memory management has always been one of the most active research areas for decades since it was introduced in the computer systems. On one frontier, to make the installed memory effectively used, much work has been done on memory allocation, recycling, and the management in various programming languages. Many breakthroughs have been made in

both theory and practice. On another frontier, to reduce the page paging between memory and disks, researchers and practitioners in both academia and industries are working hard to improve the performance of page replacement to reduce I/O paging, especially to avoid the worst performance cases. A significant advance in this regard becomes increasingly demanding with the continuously growing gap between memory and disk access times, and rapidly improved CPU performance. Unfortunately, an approximation of LRU, the CLOCK replacement policy [21], which was developed at least 35 years ago, is still dominating almost all the major operating systems including MVS, Unix, Linux and Windows¹, even though it has apparent performance problems inherited from LRU with certain commonly observed memory access patterns.

We believe there are two factors responsible for the lack of significant improvements of VM page replacements. First, there is a very stringent cost requirement on the policy from VM management. It requires the cost be associated with the number of page faults or a moderate constant. As we know, a page fault incurs a penalty worth of hundreds of thousands of CPU cycles. This allows a replacement policy to do its job without intrusively intervening application executions. A policy with its cost proportional to the number of memory references would be prohibitively expensive. This causes the user program to incur a trap to the operating system every few instructions, and the CPU would spend much more time on the page replacement work than doing useful work for the user application even when there are not paging requests. From the cost perspective, even LRU, a well-recognized low-cost and simple replacement algorithm, is unaffordable, because it has to maintain the

¹This generally covers many CLOCK variants, including Mach-style active/inactive list, FIFO list facilitated with hardware reference bits. These CLOCK variants share the same performance problems plaguing LRU.

LRU ordering of pages at any time. The second factor is that most proposed replacement algorithms attempting to improve LRU performance turn out to be too complicated to produce their approximation versions with their costs meeting the requirements of VM. This is mainly because the weak cases for LRU mostly attribute to its minimal use of history access information, which motivates other researchers to make an opposite approach by adding more bookkeeping and access statistic analysis work to make their algorithms more intelligent in dealing with some access patterns unfriendly to LRU.

3.1.2 LRU/CLOCK and their Performance Disadvantages

LRU is designed based on the assumption that a page would be re-accessed soon if it has been accessed recently. It manages a data structure conventionally called LRU stack, in which the Most Recently Used (MRU) page is at the stack top and the Least Recently Used (LRU) page is at the stack bottom. Other in-between pages in the stack strictly follow the ordering of their last access times. To maintain the stack, LRU algorithm has to move an accessed page from its current position in the stack (assume it has been in the stack) to the stack top. The LRU block at the stack bottom is the one to be replaced if there is a page fault and no free spaces are available. In CLOCK, the memory spaces holding the pages can be regarded as a circular buffer and the replacement algorithm cycles through the ordering of the pages, like the minute hand of a clock. Each page is associated with a bit, called reference bit, which is set by hardware whenever the page is accessed. When it is necessary to replace a page to service a page fault, the page pointed by the hand is checked. If its reference bit is unset, the page is replaced. Otherwise, the algorithm unsets its reference bit and continues moving the hand to the next page. Research and experience have shown

that CLOCK is a close approximation of LRU, and its performance characteristics are very similar to those of LRU. So all the performance disadvantages discussed about LRU in the following are also applied to CLOCK.

The LRU assumption is valid for a significant portion of workloads, and LRU works well for these workloads, which we call LRU-friendly workloads. The distance of a page in the LRU stack from the stack top to its current position is called **recency**, which shows the number of other distinct pages accessed after the last reference to the page. Assuming an unlimitedly long LRU stack, the position the page is in when it is accessed is called its **re-use distance**, indicating the number of other distinct pages accessed between its last access and its current access. LRU-friendly workloads have two distinct characteristics: (1) There are much more references with small re-use distances than those with large re-use distances. (2) Most references have re-use distances smaller than the available memory size in terms of the number of pages. The locality exhibited in this type of workloads is regarded as strong, which ensures a high hit ratio and a steady increase of hit ratio with the increase of memory size.

However, there do exist occasions that this assumption does not hold, where LRU performance could be unacceptably degraded. One example access pattern is memory scan, which consists of a sequence of one-time page accesses. These pages actually have infinitely large re-use distance and cause no hits. More seriously, in LRU the scan could flush all the previously active pages out of memory. Linux, which uses a variant of CLOCK as its replacement policy, faces a serious challenge on the memory management due to the scan effect by accessing one-time or infrequently used file data on disks.

In Linux the memory management for process-mapped program memory and file I/O

buffer cache is unified, so that the memory can be flexibly allocated between them according to their respective needs. The allocation balancing between program memory and buffer cache poses a big problem because of the unification. Here is a quote from Rik van Riel at Red Hat Inc.[73] to describe this problem. *“... the amount of data on the file systems tends to be several magnitudes larger than the amount of memory taken by the processes in the system. This means that the number of accesses to pages from the file cache could overwhelm the total number of accesses to the pages of the processes, even though the individual pages of the processes get accessed more frequently than most file cache pages. In other words, the system can end up evicting frequently accessed pages from memory in favor of a mass of recently but far less frequently accessed pages.”* An example scenario on this is that after one extracts a large tarball, he/she could feel the computer gets much slower because the previous active working set is replaced and has to be faulted in. To address this problem in a simple way, current Linux versions have to introduce some “magic parameters” to enforce the buffer cache allocation within the range of 1% and 15% of memory size. However, this approach does not fundamentally solve the problem, because one major factor to cause this allocation unbalancing between process memory and buffer cache is the inefficient replacement policy to deal with infrequent accessed pages in buffer caches.

Another example access pattern defeating LRU is loop, where a set of pages are accessed cyclically. Loop and loop-like access patterns dominate the memory access behavior of many programs, particularly in scientific computation applications. If pages involved in the loop along with other pages accessed in a cycle cannot completely fit in the memory, there could be repeated page faults and no hit at all. The most cited example [30, 67] for the loop problem is that even if you have a memory of 100 pages to hold 101 page data, the hit ratio

would be ZERO if you loop over this data set!

3.1.3 LIRS and its Performance Advantages

A recently proposed breakthrough replacement algorithm, namely LIRS (Low Inter-reference Recency Set)[33], removes all the aforementioned LRU performance disadvantages while still maintaining a low cost close to LRU. It can not only overcome the side-effects of scan and loop accesses, but also accurately differentiate the pages based on their locality strengths quantified by re-use distance.

The key different approach in handling history access information in LIRS from LRU is that it uses re-use distance rather than recency in LRU for the replacement decision. A page with a large re-use distance will be replaced even if it has a small recency. For instance, when a one-time-use page is recently accessed in a memory scan, LIRS will replace it quickly because its re-use distance is infinite, even though its recency is very small. To retain the LRU low-cost merit, LIRS does not explicitly bookkeep and compare re-use distances of accessed pages, but dynamically categorizes the pages into two sets, one for pages with small re-use distance, called LIR set, and another for pages with large re-use distance, called HIR set. In LIRS, only pages in LIR set are cached and cannot be replaced until it proves to be ineligible to stay in the LIR set due to its large recency. On the other hand, the pages in the HIR set will be replaced soon after they are faulted in. A HIR page must generate a relatively small re-use distance to turn into LIR page and then enjoy the privilege of staying in memory for a relatively long period of time. In contrast, LRU lacks the insights of LIRS: all accessed pages are indiscriminately cached until they are either re-accessed when they are in the stack or replaced at the bottom of the stack, without considering which of the

two cases is more possible to happen. For the infrequently accessed pages, which are most possible to be replaced at the stack bottom without being re-accessed in the stack, holding them in memory (as well as in stack) certainly means a waste of the memory resources. This explains the LRU misbehavior with the access patterns of weak locality.

The performance advantages of LIRS are impressive while it is compared with other recently proposed replacement algorithms, including DEAR[19], AFC [18], UBM [42], 2Q [37], LRU-2[57], SEQ [30], LRFU [45], EELRU[67] and ARC [51]. The advantages include (1) Unlike DEAR, AFC, UBM, and SEQ, LIRS does not depend on the explicit detection of access regularity on which LRU is possible to fail in order to improve LRU performance. (2) Unlike LRU-2, LRFU, and EELRU, LIRS has an $O(1)$ overhead and its cost is actually very close to LRU. (3) Unlike 2Q, SEQ, ARC, LIRS is able to remove LRU problems in a broad spectrum of workloads with scan, loop and various changing access patterns. The advantages of LIRS to effectively and intelligently replace infrequently accessed pages in buffer caches have drawn the attention from the industry. Here is a brief comment of Rik van Riel on LIRS [73]: *“..., the facts that LIRS would make the file cache vs process memory balancing automatic and that LIRS would also do the right thing as a second level cache ... make the implementation of LIRS for Linux a promising future experiments.”*

In this chapter, we will describe a VM page replacement algorithm, called CLOCK-Pro, to take the place of CLOCK, which meets both the performance demand from application users and the low overhead requirement from system designers. CLOCK-Pro integrate the principle of LIRS and the way in which CLOCK works. CLOCK-Pro has the following features: (1) CLOCK-Pro works in a similar fashion as CLOCK and its cost is easily affordable in VM management. (2) CLOCK-Pro brings all the much-needed performance

advantages from LIRS into CLOCK. (3) Without any pre-determined parameters, CLOCK-Pro adapts to the changing access patterns to serve a broad spectrum of workloads. (4) Through extensive simulations on real-life I/O and VM traces, we have shown the significant performance improvement of CLOCK-Pro over CLOCK as well as other representative VM replacement algorithms.

3.2 Related Work

There have been a large number of new replacement algorithms proposed for many years, especially in the last fifteen years. Almost all of them are proposed to target at the performance problems of LRU. In general there are three approaches taken in these algorithms. (1) Requiring applications to explicitly provide future access hints, such as Application-controlled file caching [11], and application-informed prefetching and caching [59]; (2) Explicitly detecting the access patterns failing LRU and adaptively switching to other effective replacements, such as SEQ [30], EELRU [67], AFC [18], and UBM [42]. (3) Tracing and utilizing deeper history access information such as FBR [65], LRFU [45], LRU-2 [57], 2Q [37], MQ [81], LIRS [33], and ARC [51]. More elaborate description and analysis on the algorithm can be found in [33]. The algorithms taking the first two approaches usually place too much constraint on the applications they are designed to serve to be applicable in the VM of a general-purpose OS. For example, SEQ is designed to work in VM managements, and it only does its job when there is a page fault. However, its performance depends on an effective detection of long sequential address reference patterns, on which LRU could behave poorly. Thus, the mechanism it uses makes SEQ lose the generality. For instance,

SEQ is hard to detect the loop access over linked lists or the accesses to a sequence of pages by an application but the sequence is randomly interleaved with the accesses to the pages of other applications. Among the algorithms taking the third approach, FBR, LRU-2, LRFU and MQ are too costly even compared with LRU. The performance of 2Q has been shown to be very sensitive to its parameters and could be much worse than LRU. LIRS and ARC are the two most promising candidate algorithms that could be applied in VM, because they use the data structure and operations similar to LRU and their cost is also close to that of LRU. Both have the potential to produce approximation versions for VM, while keeping their respective performance advantages.

ARC maintains two variable-sized lists holding history access information of referenced pages. Their combined size is two times of the memory in terms of pages. So ARC not only records the information of cached pages, but also keeps track of the same number of replaced pages. The first list contains pages that have been touched only once recently (cold pages) and the second list contains pages that have been touched at least twice recently (hot pages). The cache spaces allocated to the pages in these two lists are adaptively changed, depending on in which list recent misses happen. More cache spaces will serve cold pages (resp. hot pages) if there are more misses in the first list (resp. in the second list). However, though ARC allocates memory to hot/cold pages adaptively to the ratio of cold/hot page accesses and excludes tunable parameters, the locality of pages in the two lists, supposed to hold cold and hot pages respectively, can not directly and consistently be compared. So hot pages in the second list could have a weaker locality in terms of re-use distance than cold pages in the first list. For example, a page that is regularly accessed with a re-use distance a little more than the memory size has no hits at all in ARC while a page in the second list can

stay in memory without any accesses since it has been accepted into the list. This does not happen in LIRS, because any pages supposed to be hot (LIR pages) or cold (HIR pages) are placed in the same list and compared in a consistent fashion. Any LIR/HIR status changes are responsively conducted. There is one pre-determined parameter in LIRS algorithm on the amount of memory allocation for HIR pages. In CLOCK-Pro, the parameter is removed and the allocation becomes fully adaptive to the current access patterns.

Compared with the research on the general replacement algorithm targeting at LRU, the work specific to the VM replacements and targeting at CLOCK is much less and inadequate. While Second Chance (SC)[70] as the simplest kind of CLOCK algorithm utilizing only one reference bit to indicate recency, other CLOCK variants introduce a finer distinction between page access history. In a generalized CLOCK version called GCLOCK[69], a counter is associated with each page rather than a single bit. The counter will be incremented if the page is hit. The circulating clock hand sweeps through the page decrementing the counter until a page with its count of zero is found for replacement. In Linux and FreeBSD, a similar mechanism called page aging is used. The counter is called age in Linux or act_count in FreeBSD. When scanning through memory for pages to replace, the page age is increased by a constant if its reference bit is set. Otherwise its age is decreased by a constant. One problem for this kind of designs is that their performance improvements are not consistent, and “can be either better or worse than LRU”[55]. The parameters for setting the maximum value of counters or adjusting ages are mostly empirically decided. Another problem is that they will consume too many CPU cycles and adjust to changes in the access patterns slowly, which is evidenced in Linux kernel 2.0. Recently, Bansal and Modha provided an approximation version of ARC, called CAR [6], which has a cost close to CLOCK. Their simulation

tests on I/O traces indicate that CAR has a performance similar to ARC. Our simulation experiments on I/O and VM traces both show that CLOCK-Pro has a significantly better performance than CAR.

While the design VM replacements is difficult to take much benefit from the work on improving LRU due to the strict VM cost requirement, it remains as a demanding challenge in the OS design.

3.3 Description of CLOCK-Pro

3.3.1 Main Idea

CLOCK-Pro takes the same principle as that of LIRS – it uses the re-use distance (called IRR in the LIRS replacement algorithm) rather than recency in its replacement decision. When a page is accessed, the re-use distance is the period of time in terms of the number of other distinct pages accessed since its last access. Although there is a re-use distance between any two consecutive references to a page, only the most current distance is relevant in the replacement decision. We use the re-use distance of a page right at the time of its access to characterize it either as a cold page if it has a large re-use distance, or as a hot page if it has a small re-use distance. Then we mark its status as either cold or hot. We place all the accessed pages, either hot or cold, into the same list ² in the order of their accesses ³ with the pages with small recency at the list head and the pages with small recency at the list tail.

²Actually it is the directory entries that are placed in the list. However, for simplicity we say “a page in the list” instead of explicitly “the entry of a page in the list”

³Actually we can only maintain an approximate access order because we cannot update the list with a hit access in a VM replacement algorithm and thus lose the exact access orderings between page faults.

To give the cold pages a chance to compete with the hot pages and ensure their cold/hot statuses accurately reflect their current access behaviors. We grant each cold page a test period once it is accepted into the list. If it is accessed during its test period, the cold page turns into a hot page. If the cold page passes the test period without a re-access, it will leave the list. It is noted that the cold page in its test period can be replaced out of memory, but its page entry will remain in the list for the test purpose until the end of the test period or being re-accessed.

The key question here is how to set the time of the test period. When a cold page is in the list and there is still at least one hot page after it (i.e. with a larger recency), it can turn into a hot page if it is accessed, because it has a new re-use distance smaller than the hot page(s) after it. Accordingly, the hot page with the largest recency should turn into a cold page. So the test period should be set as the largest recency of the hot pages. If we make sure that the hot page with the largest recency is always at the list tail, and all the cold pages that pass this hot page terminate their test periods, then the test period of a cold page is equal to the time before it passes the tail of the list. So all the non-resident cold pages can be removed from the list right after it reaches the tail of the list. In practice, we could shorten the test period and limit the number of cold pages in the test period to save the space cost. By implementing this test mechanism, we ensure that “cold/hot” are defined based on relativity and constant comparison, not on a fixed threshold. This makes CLOCK-Pro distinctive from the prior work including 2Q and ARC, which attempts to use a constant threshold to distinguish the two types of pages, and treat them differently in the separate lists. Unfortunately this will make these algorithms share the performance weakness of LRU.

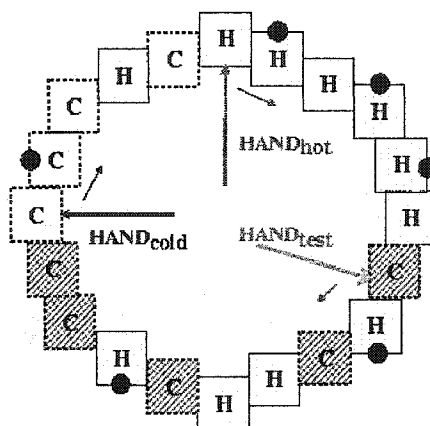


Figure 3.1: There are three types of pages in CLOCK-Pro, hot pages marked as “H”, resident cold pages marked as “C” and non-resident cold pages marked as shadowed block with “C”. Around the clock, there are three hands: $HAND_{hot}$ pointing to the list tail (i.e. the last hot page) and searching a hot page to turn into a cold page, $HAND_{cold}$ pointing to the last resident cold page and searching for a cold page to replace out of memory, and $HAND_{test}$ pointing to the last cold page in the test period, terminating test periods of cold pages, and removing non-resident cold pages passing the test period out of the list. The attached black dots represent the reference bits of 1.

When it is necessary to generate a free space, we replace a resident cold page.

3.3.2 Data Structure

Let us first assume the memory allocations for hot and cold pages, m_h and m_c , respectively, are fixed, where $m_h + m_c$ is the total memory size m ($m = m_h + m_c$). The number of hot pages is also m_h , so all the hot pages are cached at any time. For a hot page to be replaced, it must first change into a cold page. Except hot pages, all the other accessed pages are categorized as cold pages. Among the cold pages, m_c pages are cached, another at most m non-resident cold pages also have their history access information cached. So totally there are at most $2m$ directory entries for page access history in the list. The same as CLOCK, all the page entries are organized as a circular linked list, shown in Figure 3.1. For each page, there is a cold/hot status associated with it. For each cold page, there is a flag to

indicate if it is in the test period.

In CLOCK-Pro, there are three hands. The hand $HAND_{hot}$ points to the hot page with the largest recency. The position of this hand actually serves as a threshold being a hot page. Any hot pages swept by the hand turn into cold pages. For the convenience of the presentation, we call the page pointed by $HAND_{hot}$ as the tail of the list, and the page immediately before the tail page in the clockwise direction as the head of the list. The $HAND_{cold}$ points to the last resident cold page (i.e. the furthest one to the list head). Because we always select the cold page for replacement, this is the position where we start to look for a victim page, an equivalent to the hand in CLOCK. The hand $HAND_{test}$ points to the last cold page in the test period. This hand serves to terminate the test period of cold pages. The non-resident cold pages swept by this hand will leave the list. All the hands move in the clockwise direction.

3.3.3 Operations on Searching Victim Pages

Just like CLOCK, there are no operations in CLOCK-Pro for page hits, only the reference bits of the accessed pages are set by hardware. Before we see how a victim page is generated, let's examine how the three hands move around the list (clock), because the victim page is searched by coordinating the movements of the hands.

The reason to move $HAND_{hot}$ is that a cold page is accessed in its test period and thus turns into a hot page. Accordingly we need to change the hot page with the largest recency to turn into a cold page. If the reference bit of the hot page pointed by the hand is unset, we can simply change its status and then move the hand forward. However, if the bit is set, which indicates the page has been re-accessed, we spare this page, reset its reference bit

and keep it as a hot page. This is because the actual access time of the hot page could be earlier than the cold page. Then we move the hand forward to examine the next page until it encounters a hot page with its reference bit of zero. Then the hot page with its reference of zero turns into a cold page. Whenever it encounters a cold page, it will terminate its test period and remove the cold page out of list if it is non-resident (the most probable case). This actually does the work assigned to hand $HAND_{test}$. Finally the hand stops at a hot page.

We keep track of the number of non-resident cold pages. Once the number exceeds m , the memory size in the number of pages, we remove the cold page pointed by $HAND_{test}$ out of the list if it is non-resident. We terminate its test period. Because the cold page has used up its test period without a re-access and has no chance to turn into a hot page with its next access. $HAND_{test}$ will then move forward and stop at the next cold page.

$HAND_{cold}$ is used to search a resident cold page for replacement. If the reference bit of the resident cold page currently pointed by $HAND_{cold}$ is unset, we replace the cold page for a free space. Otherwise, if its bit is set and it is in its test period, we turn the cold page into a hot page, move it to the list head, and ask $HAND_{hot}$ for its actions, because an access during test period indicates a competitively small re-use distance. Note that the replaced cold page will remain in the list as non-resident cold page until it runs out of its test period. The hand will keep moving until it encounters a cold page eligible for replacement, and stops at the next resident cold page.

When there is a page fault, the faulted page must be a cold page. We first run $HAND_{cold}$ for a free space. If the cold page is not in the list, its re-use distance is highly possible to

be larger than the recency of hot pages⁴. So the page is still categorized as a cold page and is placed at the list head. It also initiates its test period. If the number of cold pages is larger than the threshold, we run $HAND_{test}$. If the cold page is in the list⁵, the faulted page turns into a hot page and is placed on the head of the list. We run $HAND_{hot}$ to make a hot page with large recency turn into a cold page.

3.3.4 Making CLOCK-Pro Adaptive

Until now, we have assumed that the memory allocations for hot and cold pages are fixed. In LIRS, there is a pre-determined parameter, called L_{hirs} , to determine the percentage of memory that are used by HIR pages. As it is shown in [33], L_{hirs} actually affects how LIRS behaves differently from LRU. When L_{hirs} approaches 100%, LIRS's replacement behavior as well as its hit ratios are close to those of LRU. Although the evaluation of LIRS algorithm indicates that its performance is not sensitive to L_{hirs} variations within a large range between 1% and 30%, it also shows that the hit ratios of LIRS could be moderately lower than LRU for LRU-friendly workloads (i.e. with strong locality) and increasing L_{hirs} could eliminate the performance gap.

In CLOCK-Pro, resident cold pages are actually managed the same as CLOCK. $HAND_{cold}$ behaves the same as what the clock hand in CLOCK does: sweeping across the pages while sparing the page with its reference bit of 1 and replacing the one with its reference bit of 0. So increasing m_c , the size of the allocation for cold blocks, makes CLOCK-Pro behave more like CLOCK. Let's see the performance implication of changing memory allocation

⁴We cannot guarantee the largeness because there are no operations on hits in CLOCK-Pro and we limit the number of cold pages in the list. But our experiment results show this approximation minimally affects the performance of CLOCK-Pro.

⁵The cold page must be in its test period. Otherwise, it must have been removed from the list.

in CLOCK-Pro. To overcome the CLOCK performance disadvantages with weak access patterns such as scan and loop, a small m_c value means a quick eviction of cold pages just faulted in and the strong protection of hot pages from the interference of cold pages. However, for strong locality accesses, almost all the accessed pages have relatively small re-use distance. But some of the pages have to be categorized as cold pages. With a small m_c , these pages would have to be replaced out of memory soon after its being loaded in, then with an additional fault access during its test period to be loaded in the memory again as a hot page. Increasing m_c would allow these cold pages to be cached for a longer time and to be more possible to be re-accessed before being replaced. So they can save the additional page faults.

For a given re-use distance of an accessed cold page, m_c decides the probability of a page to be re-accessed before being replaced during its test period. For a cold page with its re-use distance larger than its test period, retaining the page in the memory with a large m_c is a waste of buffer spaces. On the other hand, for a page with a small re-use distance, retaining the page in the memory for more time with a large m_c would save an additional page fault. In the adaptive CLOCK-Pro, we allow m_c to dynamically adjust to the current re-use distance distribution. If a cold page is accessed during its test period, we increment m_c by 1. If a cold page passes its test period without a re-access, we decrement m_c by 1. Note the aforementioned cold pages include resident and non-resident cold pages. By making the adaptation, CLOCK-Pro could take both LRU advantages with strong locality and LIRS advantages with weak locality.

3.4 Performance Evaluation

To evaluate our CLOCK-Pro and to demonstrate its performance advantages, we use trace-driven simulations on various types of workloads to compare it with other algorithms, including CLOCK, LIRS, CAR, and OPT. CAR [6] is an approximation of ARC [51]. OPT is an optimal, offline, but unimplementable replacement algorithm [7].

Our simulation experiments are conducted in three steps with different kinds of workload traces. Because LIRS is originally proposed as I/O buffer cache replacement algorithm, in the first step, we test the replacement algorithms on the I/O traces to see how well CLOCK-Pro can retain the LIRS performance advantages, as well as its performance with typical I/O access patterns. In the second step, we test the algorithms on the VM traces of application program executions. Because the integrated VM management on file cache and program memory such as what is implemented in Linux, is always desired, but has the concern of mistreatment of file data and process pages as mentioned in Chapter 3.1.2. In the third step, we test the algorithms on the aggregated VM and I/O traces to see how these algorithms respond to the integration.

3.4.1 Simulation on Buffer Cache for File I/O

The I/O traces used in this section are from [33] used for the LIRS evaluation. In their comprehensive performance evaluation, the traces are categorized into four groups based on their access patterns, namely, loop, probabilistic, temporally-clustered and mixed patterns. Here we selected one representative trace from each of the groups for the replacement evaluation, and briefly describe them here.

1. **glimpse** is a text information retrieval utility trace. The total size of text files used as input is roughly 50 MB. The trace belongs to the loop pattern.
2. **cpp** is a GNU C compiler pre-processor trace. The total size of C source programs used as input is roughly 11 MB. The trace belongs to the probabilistic pattern.
3. **sprite** is from the Sprite network file system, which contains requests to a file server from client workstations for a two-day period. The trace belongs to the temporally-clustered pattern.
4. **multi2** is obtained by executing three workloads, cs, cpp, and postgres, together. The trace belongs to the mixed pattern.

These are small-scale traces with clear access patterns. We use them to investigate the implications of various access patterns on the algorithms. The hit ratios of *glimpse* and *multi2* are shown in Figure 3.2. To help readers clearly see the hit ratio difference of the algorithms, we list the hit ratios of *cpp* and *sprite* in Tables 3.1 and 3.2, respectively. For LIRS, the memory allocation (L_{hirs}) to HIR pages is set as 1% of memory size, the same value as it is used in [33]. There are several observations we can make in the experiments.

First, even though CLOCK-Pro does not responsively deal with hit accesses to meet the cost requirement of VM management, the hit ratio of CLOCK-Pro and LIRS are very close, which shows that CLOCK-Pro effectively retains the performance advantages of LIRS. For workloads *glimpse* and *multi2*, which contain many loop accesses, LIRS with a small L_{hirs} is most effective. The hit ratios of CLOCK-pro are a little lower than LIRS. However, for the LRU-friendly workload, *sprite*, which consists of strong locality accesses, the performance of

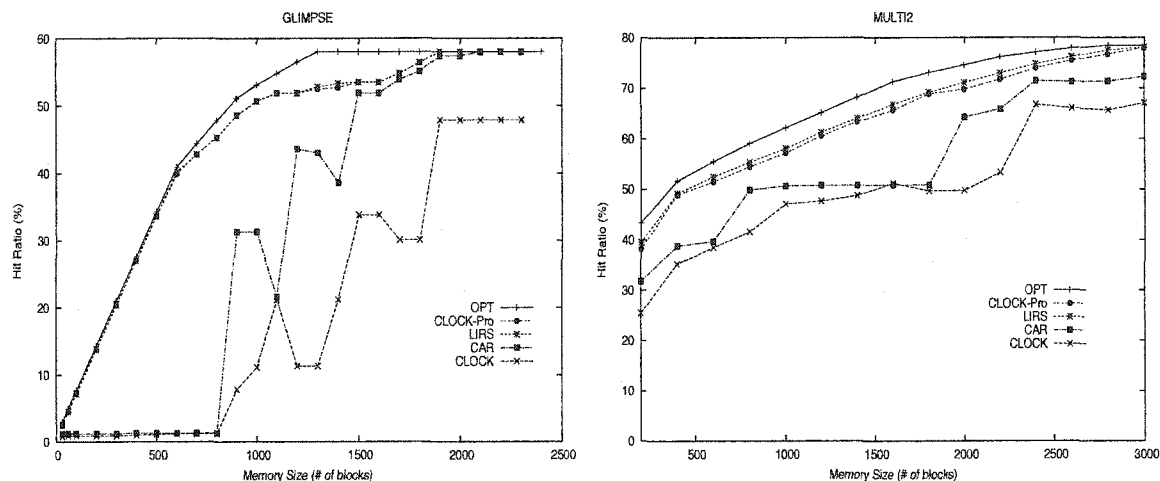


Figure 3.2: Hit ratios of the replacement algorithms OPT, CLOCK-Pro, LIRS, CAR, and CLOCK on workloads *glimpse* and *multi2*.

LIRS could be lower than CLOCK (see Table 3.2). With its memory allocation adaptation, CLOCK-Pro improves LIRS performance.

Figure 3.3 shows the percentage of memory allocated to the cold pages during the execution courses of *multi2* and *sprite* for a memory size of 600 pages. We can see that for *sprite* the allocations for cold pages are much larger than 1% of memory used in LIRS, and the allocation fluctuates over the time adaptively to the changing access patterns. It sounds paradoxical that we need to increase cold page allocation when there are many hot page accesses in the strong locality workload. Actually only the real cold pages with large re-use distances should be managed in a small cold allocation for their quick replacements. The so-called “cold” pages could also be hot pages in strong locality workloads because the number of so called “hot” pages are limited by its allocation. So these pseudo-cold pages should be avoided to be quickly replaced by increasing the cold page allocation. We can see that cold page allocations for *multi2* are lower than *sprite*, which is consistent with the fact that *multi2* access patterns consist of many long loop, weak locality accesses.

Pages	OPT	CLOCK-Pro	LIRS	CAR	CLOCK
20	26.4	23.9	24.2	17.6	0.6
35	46.5	41.2	42.4	26.1	4.2
50	62.8	53.1	55.0	37.5	18.6
80	79.1	71.4	72.8	70.1	60.4
100	82.5	76.2	77.6	77.0	72.6
200	86.0	84.0	84.3	84.8	81.8
300	86.5	85.1	85.0	85.6	83.5
400	86.5	85.7	85.6	85.7	84.3
500	86.5	85.9	85.9	85.8	84.7
600	86.5	86.2	86.2	86.0	85.0
700	86.5	86.3	86.3	86.3	85.4
800	86.5	86.4	86.4	86.4	85.2
900	86.5	86.4	86.4	86.4	85.7

Table 3.1: Hit ratios of the replacement algorithms OPT, CLOCK-Pro, LIRS, CAR, and CLOCK on workload *cpp*.

Pages	OPT	CLOCK-Pro	LIRS	CAR	CLOCK
100	50.8	24.8	25.1	26.1	22.8
200	68.9	45.2	44.7	43.0	43.5
300	78.8	58.8	58.6	59.1	59.5
400	84.6	70.1	69.5	70.5	70.9
500	87.9	77.5	76.0	77.7	78.3
600	89.9	82.4	80.9	82.1	83.3
700	91.3	85.3	83.8	85.3	86.0
800	92.2	87.6	85.6	87.3	88.1
900	92.8	88.8	86.8	88.8	89.4
1000	93.2	89.7	87.6	89.6	90.4

Table 3.2: Hit ratios of the replacement algorithms OPT, CLOCK-Pro, LIRS, CAR, and CLOCK on workload *sprite*.

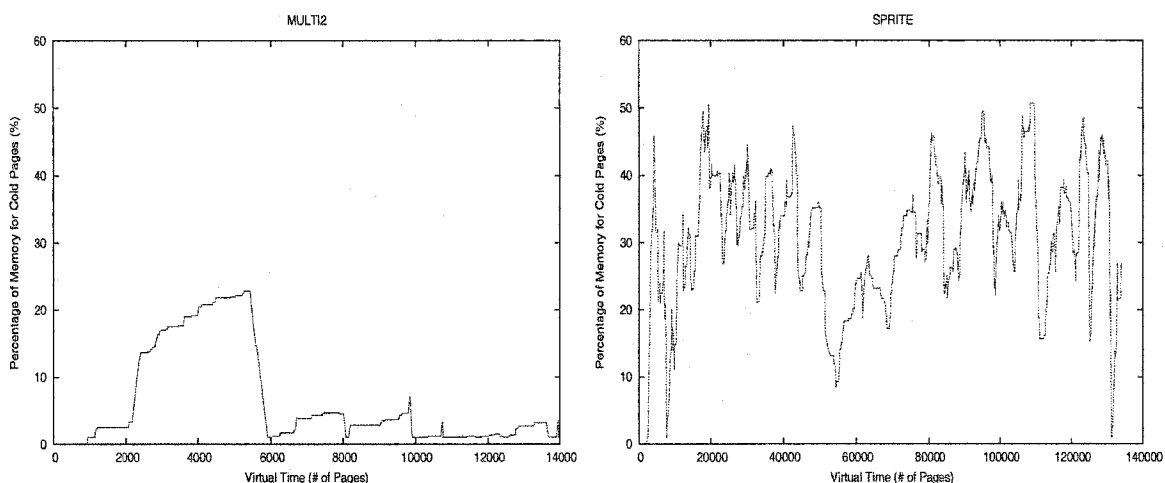


Figure 3.3: Adaptively changing the percentage of memory allocated to the cold pages in workloads *multi2* and *sprite*.

Second, regarding the performance difference of the algorithms, CLOCK-Pro and LIRS have much higher hit ratios than ARC and CLOCK for *glimpse* and *multi2*, and are close to the optimal ones. For strong locality accesses like *sprite*, there are little improvements either for CLOCK-Pro or ARC. This is the case for CLOCK to win its popularity considering its extremely simple implementation and low cost.

Third, even with a built-in memory allocation adaption mechanism, CAR cannot provide consistent improvements over CLOCK, especially for weak locality accesses, on which a fix is most needed in LRU. As we have analyzed, this is because CAR as well as ARC lack a consistent locality strength comparison mechanism.

3.4.2 Simulation on Memory for Program Executions

In this section, we use the traces of memory accesses of the program executions to evaluate the performance of the algorithms. All the traces used here are also used in [29] and many of them are also used in [30, 67]. However, we do not include the performance results of

Program	Description	Size	Max. Mem. Demand(KB)
applu	Solve 5 coupled parabolic/elliptic PDE	1,068	14,524
blizzard	Binary rewriting tool for software DSM	2,122	15,632
coral	Deductive database evaluating query	4,327	20,284
gnuplot	PostScript graph generation	4,940	62,516
jpeg	Image conversion into IJPEG format	42,951	8,260
m88ksim	Microprocessor cycle-level simulator	10,020	19,352
murphi	Protocol verifier	1,019	9,380
perl	Interpreted scripting language	18,980	39,344
sor	Successive over-relaxation on a matrix	5,838	70,930
swim	Shallow water simulation	438	15,016
trygtsl	Tridiagonal matrix calculation	377	69,688
wave5	plasma simulation	3,774	28,700

Table 3.3: A brief description of the benchmark programs (“Size” is in number of millions of instructions)

SEQ and EELRU, because of the generality or cost concerns of them for VM management. Interested readers are referred to the respective papers for a detailed performance details of SEQ and EELRU, and make a comparison of them with CLOCK-Pro and CAR. Here we simply say that CLOCK-Pro provides better or comparable performance over SEQ and EELRU.

Table 3.3 summarizes all the program traces used in this chapter. For detailed program descriptions, space-time memory access graphs, and trace collection methodology, readers are referred to papers [29, 30]. These traces cover a large range of access patterns. After observing their memory access graphs drawn from the collected traces, the authors of paper [30] categorized programs *coral*, *m88ksim*, and *murphi* as having “no clearly visible patterns” with all accesses temporarily clustered together, categorized programs *blizzard*, *perl*, and *swim* as having “patterns at a small scale”, and categorized the rest of programs as having “clearly-exploitable, large-scale reference patterns”. If we examine the program

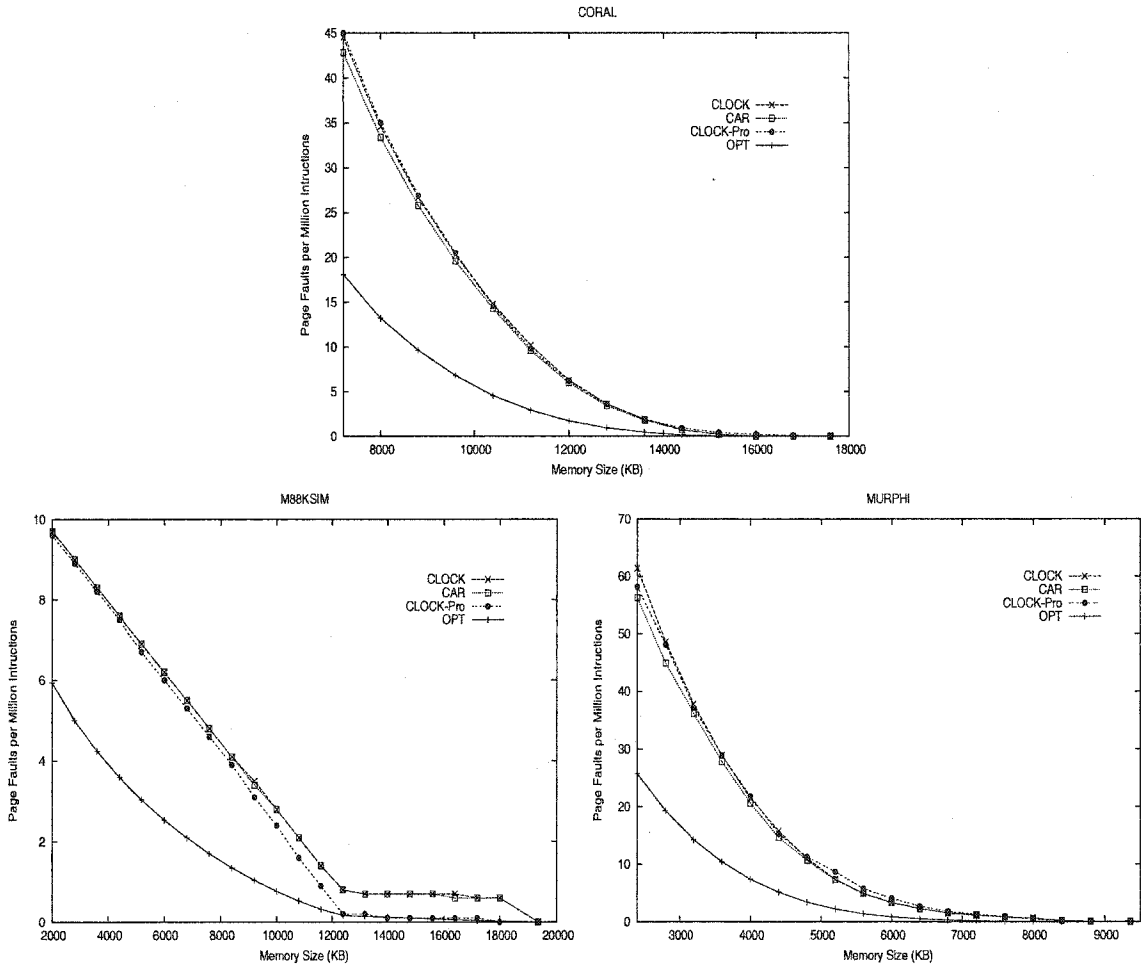


Figure 3.4: Performance of CLOCK, CAR, CLOCK-Pro and OPT on programs with strong locality.

access behaviors in terms of re-use distance, the programs in the first category belong to the strong locality workloads. Those in the second category belong to the moderate locality workloads. And the rest programs in the third category belong to the weak locality workloads. Figure 3.4, Figure 3.5, and Figure 3.6 show the number of page faults per million instruction executed for each of the programs, denoted as page fault ratio, as its memory increases up to its maximum memory demand. We exclude the cold page faults which occur on their first time accesses. The algorithms considered here are CLOCK, CLOCK-

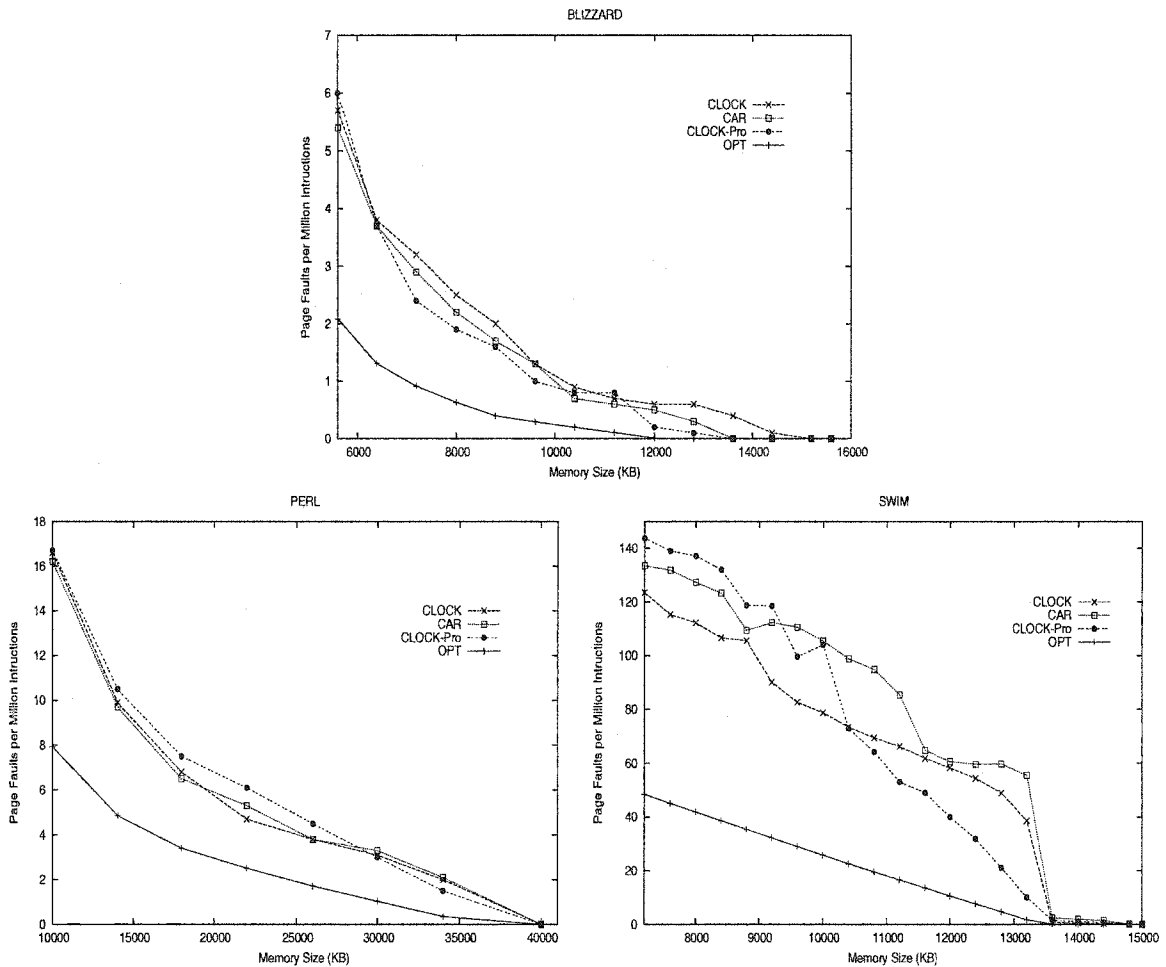


Figure 3.5: Performance of CLOCK, CAR, CLOCK-Pro and OPT on programs with moderate locality.

Pro, CAR and OPT.

The experiment results clearly show that CLOCK-Pro significantly outperforms CLOCK for the programs with weak locality, including programs *applu*, *gunplot*, *ijpeg*, *sor*, *trygtsl*, and *wave5*. For *gunplot* and *sor*, which have very large loop accesses, the page fault ratios of CLOCK-Pro are almost equal to those of OPT. The improvements of CAR over CLOCK are far from being consistent and significant. In many cases, it performs worse than CLOCK. The most inability of CAR appears on traces *gunplot* and *sor* – it cannot correct the LRU

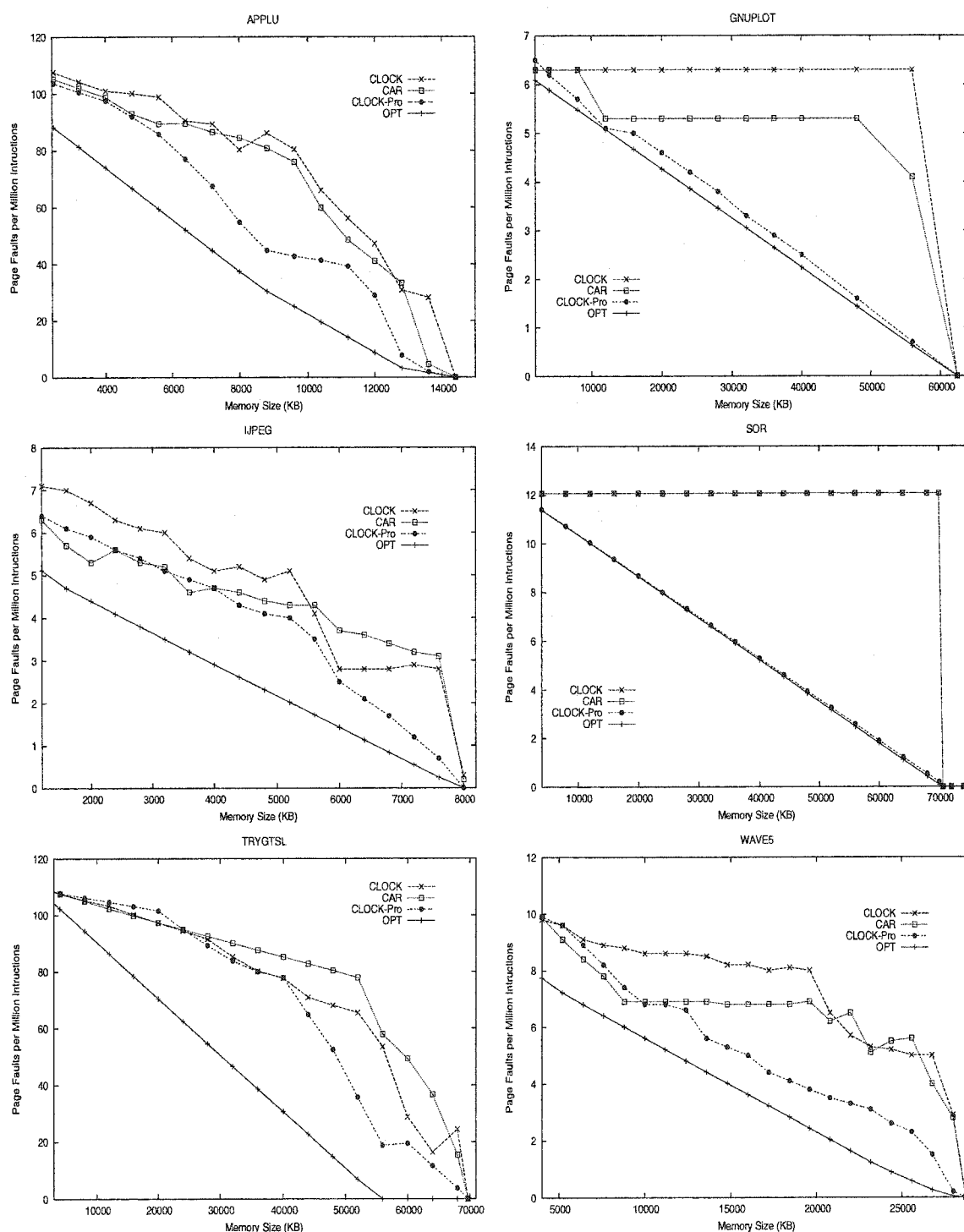


Figure 3.6: Performance of CLOCK, CAR, CLOCK-Pro and OPT on programs with weak locality.

problems with loop accesses and its page fault ratios are almost as high as those of CLOCK.

For the programs with strong locality accesses, including *coral*, *m88ksim* and *murphi*, there is little room for other replacement algorithms to do a better job than CLOCK/LRU. The good things are that both CLOCK-Pro and ARC retain the LRU performance advantages for the type of programs, and CLOCK-Pro even does a little bit better than CLOCK.

For the programs with moderate locality accesses, including *blizzard*, *perl* and *swim*, the results are mixed. Though we see the improvements of CLOCK-Pro and CAR over CLOCK in the most cases, there does exist a case in *swim* with small memory sizes where CLOCK performs better than CLICK-Pro and CAR. Though in most cases CLOCK-Pro performs better than CAR, for *perl* and *swim* with small memory sizes, CAR performs moderately better.

To summarize, we found that CLOCK-Pro can effectively remove the performance disadvantages of CLOCK with weak locality accesses, retains its performance advantages with strong locality. It exhibits apparently more impressive performance than CAR which was proposed to have the same objectives as CLOCK-Pro.

3.4.3 Simulation on Program Executions with Interference of File I/O

In an unified memory management system, file buffer cache and process memory are managed with a common replacement policy. As we have stated in Chapter 3.1.2, the memory competition from a large number of file data accesses in the shared space could interfere with the program execution. Because of the file data is far less frequently accessed than process VM, a process should be more competitive in keeping its memory from being taken away as file cache buffer. However recency-based replacement algorithms like CLOCK allow

Memory(KB)	CLOCK-Pro	CLOCK-Pro w/IO	CAR	CAR w/IO	CLOCK	CLOCK w/IO
2000	9.6	9.94	9.7	10.1	9.7	11.23
3600	8.2	8.83	8.3	9.0	8.3	11.12
5200	6.7	7.63	6.9	7.8	6.9	11.02
6800	5.3	6.47	5.5	6.8	5.5	10.91
8400	3.9	5.22	4.1	5.8	4.1	10.81
10000	2.4	3.92	2.8	4.9	2.8	10.71
11600	0.9	2.37	1.4	4.2	1.4	10.61
13200	0.2	0.75	0.7	3.9	0.7	10.51
14800	0.1	0.52	0.7	3.6	0.7	10.41
16400	0.1	0.32	0.6	3.3	0.7	10.31
18000	0.0	0.22	0.6	3.1	0.6	10.22
19360	0.0	0.19	0.0	2.9	0.0	10.14

Table 3.4: The performance (number of page faults in one million of instructions) of algorithms CLOCK-Pro, CAR and CLOCK on program *m88ksim* with and without the interference of I/O file data accesses.

these file pages to replace the process memory even if they are not frequently used, and to pollute the memory. To provide a preliminary study on the effect, we select an I/O trace [22] (WebSearch1) from a popular search engine and use its first 900 second accesses as a sample I/O accesses to co-occur with the process memory accesses in a shared memory space. This segment of I/O trace contains extremely weak locality – among the total 1.12 millions page accesses, there are 1.00 million unique pages accessed. We first scale the I/O trace onto the execution time of a program and then aggregate the I/O trace with the program VM trace in the ordering of access times. We select a program with strong locality accesses, *m88ksim*, and a program with weak locality accesses, *sor*, for the study.

Tables 3.4 and 3.5 show the number of page faults per million of instructions (only the instructions for *m88ksim* or *sor* are counted) for *m88ksim* and *sor*, respectively, with various memory sizes. We are not interested in the performance of I/O accesses. There would be few page hits even for a very large dedicated memory because there is almost no locality in the accesses.

Memory(KB)	CLOCK-Pro	CLOCK-Pro w/IO	CAR	CAR w/IO	CLOCK	CLOCK w/IO
4000	11.4	11.9	12.1	12.2	12.1	12.2
12000	10.0	10.7	12.1	12.2	12.1	12.2
20000	8.7	9.6	12.1	12.2	12.1	12.2
28000	7.3	8.6	12.1	12.2	12.1	12.2
36000	5.9	7.5	12.1	12.2	12.1	12.2
44000	4.6	6.5	12.1	12.2	12.1	12.2
52000	3.2	5.4	12.1	12.2	12.1	12.2
60000	1.9	4.4	12.1	12.2	12.1	12.2
68000	0.5	3.4	12.1	12.2	12.1	12.2
70600	0.0	3.0	0.0	12.2	0.0	12.2
74000	0.0	2.6	0.0	12.2	0.0	12.2

Table 3.5: The performance (number of page faults in one million of instructions) of algorithms CLOCK-Pro, CAR and CLOCK on program *sor* with and without the interference of I/O file data accesses.

From the simulation results shown in the tables, we found that:

(1) For the strong locality program, *m88ksim*, both CLOCK-Pro and ARC can effectively protect the program execution from the I/O access interference, while CLOCK is not able to reduce its page faults with the increase of memory.

(2) For the weak locality program, *sor*, only CLOCK-Pro can protect the program execution from the interference, though its page faults are moderately increased compared with its dedicated execution on the same size of memory. However, CAR and CLOCK cannot reduce its faults even when the memory size exceeds the program memory demand, and the number of faults on the dedicated executions has been zero.

We did not see a devastating influence on the program executions with the co-existence of intensive file data accesses. This is because even the weak accesses of *m88ksim*, are strong enough to fend off the memory competition from file accesses with their page re-accesses, and actually there are almost no page re-uses in the file accesses. However, if there are quiet periods during program active executions, such as waiting for the user interactions, the

program working set would be flushed by the file accesses under recency-based replacement algorithms. However, re-use distance based algorithms such as CLOCK-Pro will not have the problems, because the file accesses have to generate small re-use distances to qualify the file data a long-term memory stay, and to replace the program memory.

3.5 Summary

In this chapter, we proposed a new VM replacement policy, CLOCK-Pro, which is intended to take the place of CLOCK currently dominating various OS designs. We believe it is a promising replacement policy in future OS designs because (1) It has a low cost that can be easily accepted by current systems. Though it could move up to three pointers (hands) during one victim page search, the total number of the hand moves is comparable to that of CLOCK. Keeping track of the replaced pages in CLOCK-Pro doubles the size of the linked list used in CLOCK. However considering the marginal memory consumption of the list in CLOCK, the additional cost is well acceptable. (2) CLOCK-pro provides a systematic solution to the CLOCK problems. It is not just a quick and experience-based fix to a problem of CLOCK in a specific situation, but is designed based on a more accurate locality definition – re-use distance and addresses the source of the LRU problem. (3) It is fully adaptive to the strong or weak access patterns without any pre-determined parameters. (4) Extensive simulation experiments on real-life I/O and VM traces show significant and consistent performance improvements. We believe that CLOCK-Pro would be very attractive to the VM system designers in industry.

Chapter 4

Thrashing in Multiprogramming Environments

Improvement of CPU and memory utilizations has been a fundamental consideration in the design of operating systems. The interaction of memory management and CPU utilization is much more involved in the multiprogramming environments than in a dedicated execution environment. Studies of page replacement policies have a direct impact on memory and CPU utilization, which have continued for several decades (e.g. a representative and early work in [1], and recent work in [30, 67]).

4.1 Background

4.1.1 MPL versus System Thrashing

Multiprogramming level, simplified as MPL, is defined as the number of active processes in a system. We refer to these active processes in an multiprogramming environment as *interacting processes*, because they are competing for CPU and memory resources interactively. How to dynamically maintain an optimal MPL to keep a high CPU utilization has been a

fundamental issue in the design of operating systems [60]. Operating system designers aim at providing an optimal solution to the problem of using the CPU and memory resources effectively in multiprogramming, while avoiding the thrashing that multiprogramming can cause. CPU utilization can be increased by increasing MPL — running more processes. However, as MPL increases to a certain degree, the competition for memory pages among processes becomes serious, which can eventually cause system thrashing, and CPU utilization will then be significantly lowered. Considering large variations of memory demands from multiple processes and dynamical memory requirements in their lifetimes of the processes, it is not practically possible to set a pre-defined optimal MPL in order to avoid thrashing while allowing a sufficient number of processes in the system. Existing operating systems, such as BSD and Solaris, provide load control facility to swap out and in processes, if necessary, for thrashing protection. This facility allows the systems to adaptively lower MPL, but process swapping can be quite expensive for both systems and user programs.

4.1.2 Thrashing and Page Replacement

Thrashing events can be directly affected by how page replacement is conducted. Most operating systems adopt global LRU replacement to allocate the limited memory pages among competing processes according to their memory reference patterns. With an increase in MPL, memory allocation requests become more demanding. To keep more processes active, limited memory space should be fully utilized. The global LRU page replacement policy follows this principle. However, the effort to improve memory utilization could cause low CPU utilization.

In a multiprogrammed environment, global LRU replacement selects an LRU page for

replacement throughout the entire user memory space of the computer system. The risk of low CPU utilization increases if the memory page shortage happens all over the interacting processes. For example, a process is not able to access its resident memory pages when the process is resolving page faults. These already obtained pages may soon become LRU pages when memory space is being demanded by other processes. When the process is ready to use these pages in its execution turn, these LRU pages may have already been replaced to satisfy memory requests of other processes. The process then has to request the virtual memory system to retrieve these pages by replacing LRU pages of others. The page replacement may become chaotic, and could cascade among the interacting processes, eventually causing system thrashing. Once all interacting processes are in the waiting queue due to page faults, the CPU is doing little useful work.

4.1.3 Effectiveness of adaptive page replacement

Existing operating system protects thrashing at the process scheduling level by load controls. A commonly used mechanism is to suspend/reactivate or swapping out/in programs to free more memory space after the thrashing is detected. For example, the 4.4 BSD operating system [50] initially suspends a program after thrashing. If the thrashing continues, additional programs are suspended until enough memory become available. Our experiments and analysis show that there are several system performance advantages for conducting adaptive page replacement over process scheduling to eliminate thrashing. First, since improper page replacement during process interactions is a major and internal source of system thrashing, a solution to adaptively adjust page replacement behavior to current system needs can be fundamentally effective to address the problem. Second, the alter-

natives of load controls are limited to suspend or remove existing processes. Since this approach is expensive and can dramatically degrade user program interactivity, it is only used when the system is seriously thrashing. Finally, using the adaptive page replacement in the first place, we are able to eliminate the thrashing in its early stage, or significantly delay the usage of load controls. With adaptive page replacement and load controls guarding at two different levels and two different stages, the system performance will become more stable and cost-effective.

4.1.4 Our work

The objective of our study is to provide highly responsive and cost-effective thrashing protection by dynamically detecting and adaptively taking necessary actions at the kernel level of page replacement. It can also be regarded as page replacement adaptive to the system situation. We have designed a dynamic system Thrashing Protection Facility (TPF) in the system kernel considering the trade-off between CPU and memory utilizations. Once TPF detects system thrashing, one of the interacting processes will be identified for protection. The identified process will have a short period of privilege during which it does not contribute its LRU pages for removal. This allows the process to quickly establish its working set. With the support of TPF, early thrashing can be eliminated at the level of page replacement, so that process swapping will be avoided or delayed until it is truly necessary. TPF also improves the system stability when memory is dynamically and competitively demanded by interacting processes. We take the Linux kernel as a case study to illustrate why TPF is needed and how it works.

4.2 Evolution of Page Replacement in Linux Kernel

Linux, like most other systems, uses an approximate LRU scheme to keep the working set of a process in the system, and to contribute already allocated pages which may not be used in the near future to other interacting ones. A clock algorithm [70] is used, because it provides acceptable approximation of LRU, and it is cheap to implement, where NRU (Not Recently Used) pages are selected for replacement.

Current page replacement implementation in Linux is based on the following framework. The interacting processes are arranged in an order to be searched for NRU pages when few free pages are available in the user space, and/or they are demanded by interacting processes. The system examines each possible process to see if it is a candidate from which NRU pages can be found for replacement. The kernel will then check through all of the virtual memory pages in the selected process. In a moderately loaded system, we could hardly observe execution performance differences due to the different page replacement implementations. However, when processes are competitively demanding memory allocations, interacting processes may chaotically replace pages among themselves, leading to the thrashing. We take the three recent Kernel versions to illustrate how the thrashing potential is introduced and why a non-adaptive replacement policy is hard to deal with it.

4.2.1 Kernel 2.0

In Kernel 2.0, the NRU page contributions are proportionally distributed among interacting processes. There is a “swap_cnt” variable for each process, which is initialized with a quantity (RSS/1MB) proportional to its resident set size (RSS). Once an NRU page is

taken away from the process, its “swap_cnt” will be decreased by one. Only when its “swap_cnt” becomes zero, or the searching for an NRU page fails in resident space of the process, is the next process in the process list examined. When a process with “swap_cnt” of zero is encountered, it will be re-initialized using the same proportion rule. This strategy effectively balances memory usage by making all the processes provide proportional NRU pages. However, a major disadvantage of this approach is its high potential for thrashing, resulting low CPU utilization. This is because when all the memory-intensive processes are struggling to build its working set under heavy memory loads, all are requesting more pages through page faults, and no one will be given a priority for the purpose of thrashing protection.

4.2.2 Kernel 2.2

In order to address the limit, Kernel 2.2 makes each identified process continuously contribute its NRU pages until no NRU pages are available in the process. Attempting to increase CPU utilization, this strategy allows the rest of the interacting processes to build up their working sets more easily by penalizing the memory usage of one process at a time. Here is the major section of code to select a process for page replacement in the kernel function “swap_out” in mm/vmscan.c [47].

```
for (; counter >= 0; counter--) {  
    max_cnt = 0;  
    pbest = NULL;  
select:  
    read_lock(&tasklist_lock);
```



```
p = init_task.next_task;

for (; p != &init_task;
    p = p->next_task) {
    if (!p->swappable)
        continue;

    if (p->mm->rss <= 0)
        continue;

    /* Refresh swap_cnt? */

    if (assign == 1)

        p->mm->swap_cnt = p->mm->rss;

    if (p->mm->swap_cnt > max_cnt) {

        max_cnt = p->mm->swap_cnt;

        pbest = p;

    }

}

read_unlock(&tasklist_lock);

if (assign == 1)

    assign = 2;

if (!pbest) {

    if (!assign) {

        assign = 1;

        goto select;

    }

}
```

```
        goto out;

    }

    if (swap_out_process(pbest, gfp_mask))

        return 1;

}

out:

return 0;
```

In this section of code, the “swap_cnt” variable for a process’s data structure can be thought as a “shadow RSS”, which becomes zero when a swap-out operation of a process fails. The “swap_cnt”s of all the swappable processes will be re-assigned with the respective RSS in the second pass through the process list in the inner loop when they all become zeros. This inner loop will select the swappable process with the maximal RSS that has not yet been swapped out. Variable “counter” is used to control how many processes are searched before finding an NRU page. We can see that once a process provides an NRU page, which means it is the one with the maximum “swap_cnt” currently, the process will be selected for swapping upon the next request. This allows its NRU pages continuously to be replaced until a failure on finding an NRU page in the process occurs. Compared with previous kernel version, in addition to the changes in the selection of processes for NRU pages, there has been another major change in this kernel. In kernel 2.0, there is an “age” associated with each page, which is increased by 3 when it is referenced called page aging and decreased by 3 each time the page is examined. Once the “age” decreases to zero, it will become an NRU page and be ready to be replaced. The Kernel 2.2 greatly simplifies

the structure by eliminating the “age” and only making use of the reference bit of each page in the PTE (Page Table Entry). The bit is set when the page is referenced and reset when the page is examined. The pages with reference bits of 0s are NRU pages and ready to be replaced. This implementation will produce NRU pages more quickly for a process with a high page fault rate. These changes in kernel 2.2 take a much more aggressive approach to make an examined process contribute its NRU pages, attempting to help other interacting processes to establish their working sets to fully utilize the CPU.

We have noted the effort made in Kernel 2.2 to retain CPU utilization by avoiding widely spreading page faults among all the interacting processes. However, such an effort increases the possibility of replacing fresh NRU pages in the process being examined, while some NRU pages in other interacting processes that have not been used for long time continue to be kept in the memory. This approach benefits CPU utilization at the cost of lowering memory utilization. Fortunately, in our experiments, we find that each interacting process is still examined periodically with a reasonable time interval. Although the average time interval in kernel 2.2 is longer than that in kernel 2.0.38, it seems to be sufficiently short to let most interacting processes have a chance to be examined. Thus memory utilization is not a major concern. However, the risk of system instability caused by low CPU utilization remains.

4.2.3 Kernel 2.4

The latest Linux kernel is version 2.4, which makes considerable changes in the paging strategy. Many of these changes target at addressing concerns on memory performance arising in Kernel 2.2. For example, without page aging, NRU replacement in kernel 2.2 can

not accurately distinguish the working set from incidentally accessed pages. Thus Kernel 2.4 has to reintroduce page aging, just as Kernel 2.0 and FreeBSD do. However the page aging could help processes with high page fault rates to keep their working sets, thus cause other processes to have serious page fault rate, and trigger thrashing.

Kernel 2.4 distinguishes the pages with age of zero and those with positive ages by separating them into non-active and active lists, respectively to prevent bad interactions between page aging and page flushing [72]. This change does not help protect the system against thrashing, because the system still has no knowledge on which working sets of particular processes should be protected when frequent page replacement takes place under heavy memory workload. Similar argument can be applied in BSD and FreeBSD, where a system-wide list of pages forces all processes to compete for memory on an equal basis.

To make memory more efficiently utilized, Kernel 2.4 reintroduces the method used in Kernel 2.0 for selecting processes to contribute NRU pages. Going through a process list each time, it walks about 6% of the address space in each process to search NRU pages. Compared with Kernel 2.2, this method increases its possibility of thrashing.

4.2.4 The Impact of Page Replacement on CPU and Memory Utilizations

From the evolution of recent LINUX kernel, we can see that in VM designs and implementations, finding an optimal MPL concerning to thrashing has been translated into considerations of the tradeoff between the CPU and memory utilizations. For the purpose of high CPU utilization, we require that CPU be not idle when there are computing demands from “cycle-demanding” processes. For the purpose of high memory utilization, we require that no idle pages be kept unaccessed when there are memory demands from “memory-

demanding” processes. Our analysis has shown that the conflicting interests between the requirements on CPU and memory utilizations are inherent in a multiprogramming system. Regarding CPU utilization, the page replacement policy should keep at least one process active in the process queue. Regarding memory utilization, the page replacement policy should apply the LRU principle consistently to all the interacting processes. No process should hide its old NRU pages from swapping while other processes contribute their fresh NRU pages. It is difficult for a policy in favor of both CPU and memory utilizations constantly to eliminate the risk of system instability leading to thrashing. The difficulty in the design of the page replacement in multiprogramming environment is general in operating systems. Current systems lack effective mechanisms to integrate the two requirements for the purpose of thrashing protection.

From the perspective of thrashing prevention, page replacement implementations in Kernel 2.2 is more effective than Kernel 2.0 and Kernel 2.4. However, we will show that the critical weakness resulted from the conflicting interests between the requirements on CPU and memory utilizations is inherent in the Kernel 2.2. Our experimental results shown in the next section reveal its serious thrashing . Thus, we implement our TPF in Kernel 2.2 to show its effectiveness, which is not in favor of our performance evaluation.

4.3 Evaluation of Page Replacement in Linux Kernels 2.2

4.3.1 Experimental environment

Our performance evaluation is experimental measurement based. The machine we have used for all experiments is a Pentium II of 400 MHz with physical memory space of 384

MBytes. The operating system is Redhat Linux release 6.1 with the kernel 2.2.14. Program memory space is allocated in units of 4KByte pages. The disk is an IBM Hercules with capacity of 8,450 MBytes.

When memory related activities in program execution occur, such as memory accesses and page faults, the system kernel is heavily involved. To gain insight into VM behavior of application programs, we have monitored program execution at the kernel level and carefully added some simple instrumentation to the system. Our monitor program has two functions: user memory space adjustment and system data collection. In order to flexibly adjust available memory space for user programs in experiments, the monitor program can serve as a memory-adjustment process requesting a memory space of a fixed size, which is excluded from page replacement. The available user memory space can be flexibly adjusted by running the memory-adjustment process with different fixed sizes of memory demand. The difference between the physical memory space for users and the memory demand size of the memory-adjustment process is the available user space in our experiments.

In addition, the monitoring program dynamically collects the following memory system status quanta periodically for every second during execution of programs:

- *Memory Allocation Demand* (MAD): is the total amount of requested memory space reflected in the page table of a process in pages. The memory allocation demand quantum is dynamically recorded in the kernel data structure of *task_struct*, and can be accurately collected without intrusive effect on program execution.
- *Resident Set Size* (RSS): is the total amount of physical memory used by a process in pages, and can be obtained from the kernel data structure of *task_struct*.

- *Number of Page Faults* (NPF): is the number of page faults of a process, and can be obtained from *task_struct* of the kernel. There are two types of page faults for each process: minor page faults and major page faults. A minor page fault will cause an operation to relink the page table to the requested page in physical memory. The timing cost of a minor page fault is trivial in the memory system. A major page fault happens when the requested page is not in memory and has to be fetched from disk. We only consider major page fault events for each process, which can also be obtained from *task_struct*.
- *Number of Accessed Pages* (NAP):¹ is the number of pages accessed by a process within a time interval of one second. This is collected by a simple system instrumentation. During program execution, a system routine is periodically called to examine all the reference bits in the page table of a specified process.

We have selected three memory-intensive application programs from SPEC 2000: *gcc*, *gzip*, and *vortex*. Using the system facilities described above, we first run each of the three programs in a dedicated environment to observe the memory access behavior without major page faults and page replacement (the demanded memory space is smaller than the available user space). Table 4.1 presents the basic experimental results of the three programs, where the “description” column gives the application nature of each program, the “input file” column is the input file names from SPEC2000 benchmarks, the “memory requirement” column gives the maximum memory demand during the execution, and the “lifetime” column is the execution time of each program. The “lifetime” of each program

¹This quantum is only collected for dedicated executions of benchmark programs.

is measured without memory status quanta collection involved. These numbers for each program represent the mean of 5 runs. The variation coefficients calculated by the ratio of the standard deviation to the mean is less than 0.01.

Programs	description	input file	memory requirement (MB)	lifetime (s)
gcc	optimized C compiler	166.i	145.0	218.7
gzip	data compression	input.graphic	197.4	248.7
vortex	database	lendian1.raw	115.0	342.3
vortex	database	lendian3.raw	131.2	398.0

Table 4.1: Execution performance and memory related data of the 3 benchmark programs.

4.3.2 Page Replacement Behavior of Kernel 2.2.14

The memory usage patterns of the three programs are plotted by memory-time graphs. In the memory-time graph, the x axis represents the execution time sequence, and the y axis represents three memory usage curves: the memory allocation demand (MAD), the resident set size (RSS), and the number of accessed pages (NAP). The memory usage curves of the 3 benchmark programs measured by MAD, RSS, and NAP are presented in Figures 4.1 (gcc), 4.2 (gzip), and 4.3 (vortex1, which is vortex with input file of "lendian1.raw"). However, we find that Linux kernel 2.2.14 still provides a high potential for interacting processes to chaotically replace pages among themselves, significantly lowering CPU utilization and causing thrashing if the page replacement continues under heavy load. To show this, we have monitored executions and memory performance of several groups of multiple interacting programs. To make the presentation easily understandable on how memory pages are allocated among processes and their effects on CPU utilization, we only present the results of running two benchmark programs together as a group. We present three program interaction

groups: gzip+vortex3 (vortex3 is vortex with input file of “lendian3.raw”), gcc+vortex3, and vortex1+vortex3. The available user memory space was adjusted by the monitoring program accordingly so that each interacting program had considerable performance degradation due to 27% to 42% memory shortage. (The shortage ratios are calculated based on the maximum memory requirements. In practice, the realistic memory shortage ratios are smaller due to dynamically changing memory requirements of interacting programs.)

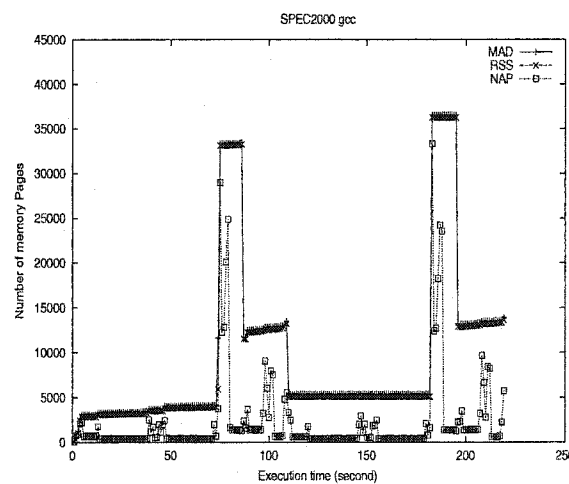


Figure 4.1: The memory performance of gcc in a dedicated environment.

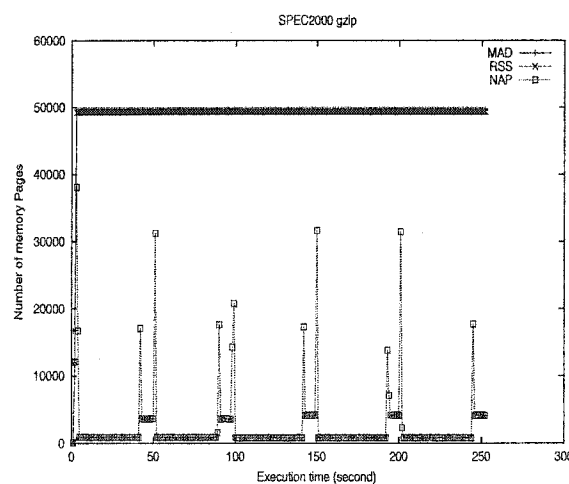


Figure 4.2: The memory performance of gzip in a dedicated environment.

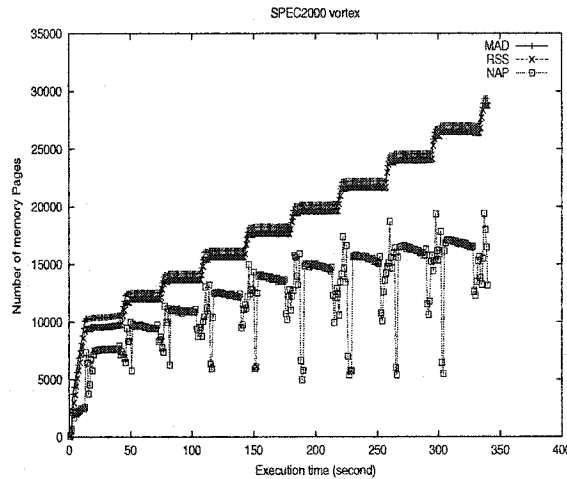


Figure 4.3: The memory performance of vortex1 in a dedicated environment.

Figure 4.4 presents the memory usage behavior measured by MAD and RSS of interacting programs gzip (left figure) and vortex3 (right figure). After we added gzip to interact with vortex3 at the 250th second, we observed that both their RSS curves are up and down in most of the times. CPU utilization is lower than 50% during the interaction because both processes were held in waiting list by page faults for the most time. Adding more processes would worsen the case due to lack of free memory in the system. We found that at around 620th and around 780th second, gzip did get its working set and ran with a small number of page faults. Unfortunately, it went back to chaotic competition after that period. The measurement shows that the slowdown of gzip is 5.23, and is 3.85 for vortex3.

Figure 4.5 presents the memory usage behavior measured by MAD and RSS of interacting programs gcc (left figure) and vortex3 (right figure). For program vortex3, the RSS curve suddenly dropped to about 14,000 pages after it reached to 26,870 pages, which was caused by the memory competition of the partner program gcc. After that, the RSS curve entered a fluctuating stage, causing a large number of page faults in each process to extend

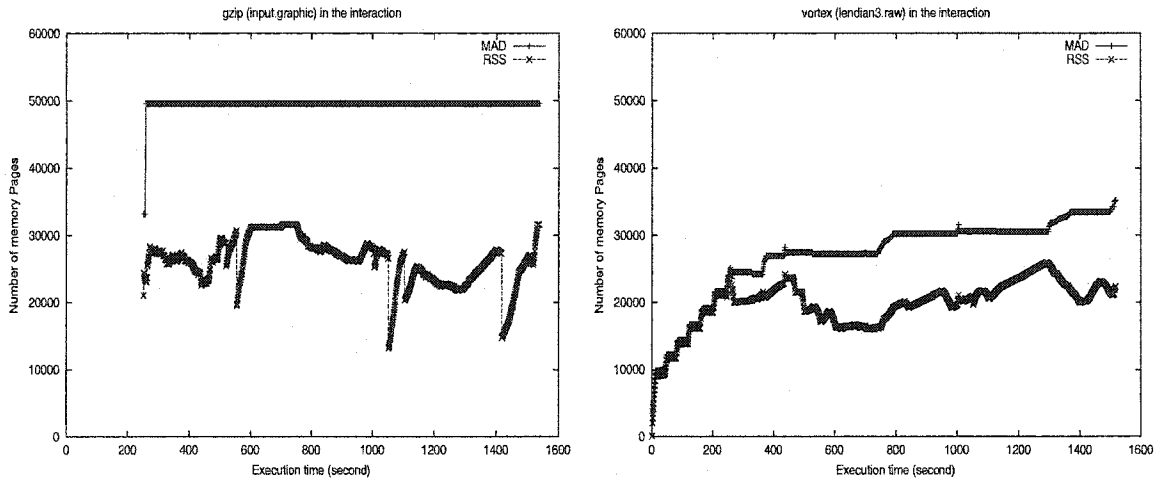


Figure 4.4: The memory performance of gzip (left figure) and vortex3 (right figure) during the interactions.

the first spike of gcc in the MAD and RSS curves to 865 seconds, and to extend a RSS stair in vortex to 563 seconds. In this case the slowdown of program gcc is 5.61, and is 3.37 for vortex.

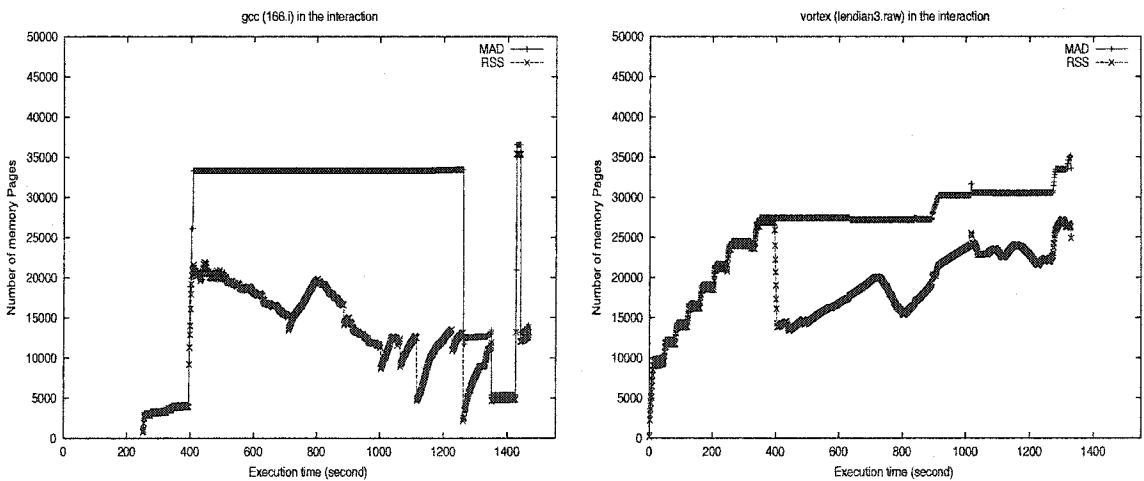


Figure 4.5: The memory performance of gcc (left figure) and vortex3 (right figure) during the interactions.

Figure 4.6 presents the memory usage behavior measured by MAD and RSS of interacting programs vortex1 (left figure) and vortex3 (right figure). Although the input files are

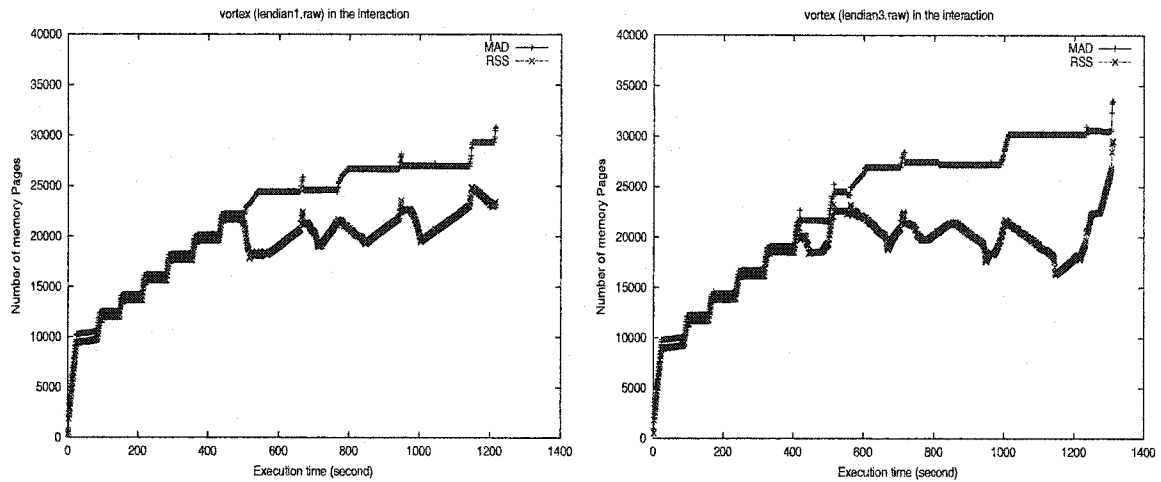


Figure 4.6: The memory performance of vortex1 (left figure) and vortex3 (right figure) during the interactions.

different, the memory access patterns of the two programs are the same. Our experiments show that the RSS curves of both vortex programs changed similarly during the interactions. To favor memory utilization, NRU pages were allocated between the two processes back and forth, causing low CPU utilization and poor system performance. After the RSS curves of both programs reached about 22,000 pages, their MADs could not be reached due to memory shortage. Our experiments again show that the execution times of both programs were significantly increased due to the page faults in the interaction. The slowdown for vortex1 is 3.58, and is 3.33 for vortex3.

Our experiments indicate that although thrashing could be triggered by a brief, random peak in memory demand of a workload, the system may continue thrashing for an unacceptably prolonged time. To make a system more resilient against dynamically changing virtual memory load, a dynamical protection mechanism is desirable instead of a brute-force process stop, such as process suspension or even process removal.

4.4 The Design and Implementation of TPF

We propose to implement TPF as part of the page replacement for thrashing protection in order to improve the system stability under a heavy load. The main idea of TPF is simple and intuitive. Once the system detects high page fault rates and low CPU utilization caused by multiple processes, TPF will identify a process and help it to quickly establish its working set by temporarily granting a privilege to the process for its page replacement. With this action, the CPU utilization quickly increases because at least one process is able to do useful work. In addition, the memory space is expected to be released soon by the process after its completion, so that the memory demands of other processes can be satisfied. We have implemented TPF in the Linux kernel 2.2.14, which consists of two kernel utilities: detection and protection routines.

The detection routine is used to dynamically monitor the page fault rate of each process and the CPU utilization of the system. The protection routine will be awakened to conduct priority-based page replacement when CPU utilization is lower than a predetermined threshold, and when the page fault rates of more than one interacting process exceed a threshold. The protection routine then grants a privilege to an identified process that will only contribute a limited number of NRU pages. The identified process is the one that has the smallest difference between its MAD and its RSS (the least memory demanding process). The detection routine also monitors whether the identified process has lowered its page fault rate to a certain degree. If so, its privilege will be disabled. This action will retain memory utilization by treating each process equally.

4.4.1 The detection routine

There are four predetermined parameters in TPF:

1. CPU_Low: is the lowest CPU utilization the system can tolerate.
2. CPU_High: is the targeted CPU utilization for TPF to achieve.
3. PF_Low: is the targeted page fault rate ² of the identified process for TPF to achieve.
4. PF_High: is the page fault rate threshold of a process to potentially cause thrashing.

We add one global linked list, `high_PF_proc`, in the kernel to record interacting processes with high page fault rates. Once we find the current page fault of a process exceeds `PF_High`, we will enter it in the linked list.

We have also added three new fields in `task_struct` data structure for each process:

1. `num_pf`: the number of page faults detected recently;
2. `start_time`: the system time for the first page fault in the above “`num_pf`” page faults;
and
3. `privilege`: the process is granted the privilege (`=1`) or not (`=0`).

Here are the kernel operations to determine and manage the processes exceeding the threshold page fault rates.

```
if (process p encounters page faults) {  
  
    if (p->num_pf == 0)
```

²In our experiments only those page faults that are revolved by loading pages from the swap files in disk are counted, because they are the most appropriate factors to reflect the effect of memory shortage on processes.

```
p->start_time = current system time;

p->num_pf++;

if (p is not in the "high_PF_proc" list)

    if (p->num_pf > high_PF) {

        if (current system time -

            p->start_time <= 1 second)

            place p in high_PF_proc;

        p->num_pf = 0;

    }

}
```

We check the page fault rate of each process in the high_PF_proc list every second. If a process's page fault rate is lower than low_PF, we will dynamically remove the process from the list by the following operations:

```
if (length(high_PF_proc) >= 1) {

    for each p in the list do {

        if (current system time -

            p->start_time >= 1 second) {

            if (p->num_pf/(current system time

                - p->start_time) < low_PF) {

                if (p->privilege == 1)

                    p->privilege = 0;
```

```
        remove p from the list;

    }

    p->num_pf = 0;

    p->start_time = current system time;

}

}

}
```

The CPU utilization is measured every second, based on the CPU idle time. Specifically, we use $(1 - \text{idle ratio})$ to represent the current CPU utilization, where the idle ratio is the CPU time portion used for the idle processes in the last second. The current CPU utilization is compared with CPU_Low to determine if the the system is experiencing an unacceptably low CPU utilization. The protection routine is triggered when the following three conditions are all true.

```
if ((CPU utilization < CPU_Low) &&
    (length(high_PF_proc) >= 2) &&
    (no process has been protected)) {
    for all processes in high_PF_proc
        select the least memory hungry p;

    p->privilege = 1;

}
```


4.4.2 The protection routine

The privilege granting is implemented in a simple way in the kernel routine “swap_out” presented in Section Chapter 4.3. The function `swap_out_process(pbest, gfp_mask)` will reset its “swap_cnt” to 0 if, and only if, the system fails to get an NRU page from process “pbest”, as we have showed in Chapter 4.3. A small modification in `swap_out_process()` will make the privilege effective; that is, we reset its “swap_cnt” to 0 even if an NRU page is obtained in the protected process. This will cause the protected process to provide at most one NRU page in each examination loop on all swappable processes. Considering that a large number of of NRU pages exist in the rest of the interacting processes, such a change will effectively help the protected process build up its working set and reduce its page fault rate. Once its page fault rate is lowered satisfactorily, the protected process will be removed from the “high_PF_proc” list and loose its privilege.

4.4.3 State transitions in the system

The kernel memory management has the following three states with dynamic transitions:

1. *normal state*: In this state, no monitoring activities are conducted. The system deals with page faults exactly as the original Linux kernel does. The system keeps track of the number of page faults for each process and places the process with high page fault rates in “high_PF_proc”.
2. *monitoring state*: In this state, the detection routine is awakened to start monitoring the CPU utilization and the page fault rates of processes in the linked list. If the protection condition is satisfied, the detection routine will select a qualified process

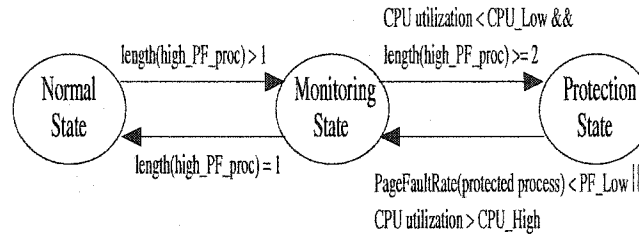


Figure 4.7: Dynamic transitions among normal, monitoring, and protection states in the improved kernel system.

for protection and go to the protection state. The system returns to the normal state when no more than one process's page fault rate is as high as the predetermined threshold.

3. *protection state:* The protection routine will make the selected process quickly establish its working set. In the protection state, the detection routine keeps monitoring the CPU utilization and the page fault rate of each process in the list. The detection routine is deactivated and the protection state transfers to the monitoring state as soon as the protected process becomes stable and/or the CPU utilization has been sufficiently improved.

Figure 4.7 describes the dynamic transitions among the three states, which gives a complete description of TPF facility. When the system is normal (no page faults occur), detection and protection routines are not involved. As we have described in the implementation, the algorithm only adds limited operations for each page fault and checks several system parameters with the interval of one second. So, overhead involved for detection and protection is trivial compared with the CPU overhead to deal with page faults.

4.5 Performance Measurements and Analysis

4.5.1 Observation and measurements of TPF facility

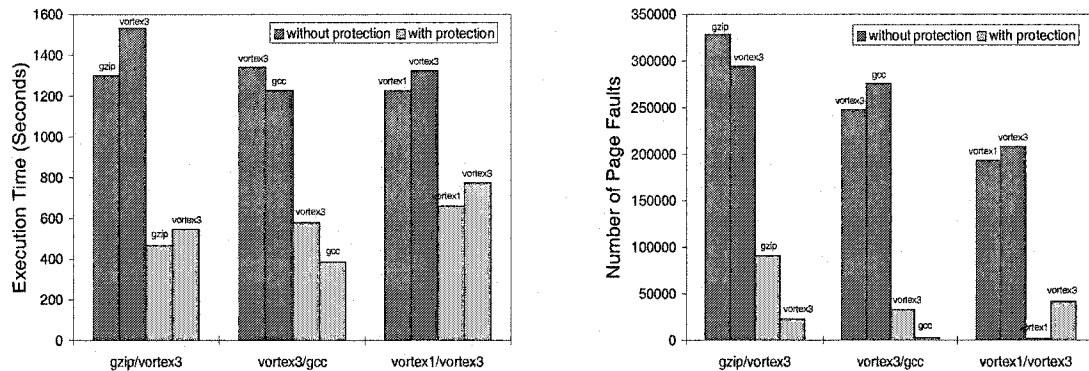


Figure 4.8: The execution time comparisons (left figure) and comparisons of numbers of page faults (right figure) for the three group of program interactions in the Linux without TPF and with TPF.

The predetermined threshold values are set as follows: CPU_{Low} = 40%, CPU_{High} = 80%, PF_{High} = 10 page faults/second, PF_{Low} = 1 page fault/second. The performance of TPF is experimentally evaluated by the three groups of the interacting programs. Each of the experiments has the exactly same setting as its counterpart conducted in Chapter 4.3, except that the TPF is implemented in the kernel.

Figure 4.9 presents the memory usage measured by MAD and RSS of interacting programs gzip (left figure) and vortex3 (right figure) in the Linux with TPF. Figure 4.4 shows that thrashing between processes started as soon as gzip joined the execution at the 250th second without TPF. In contrast, Figure 4.9 shows that TPF quickly detected the problem and went into the protection state. Because the RSS of vortex3 is close to its MAD, it was selected for protection. After the protection, its page fault rate was lowered with the

establishment of its working set. Then the protection was disabled to allow the NRU pages of vortex3 to be fully utilized. This is confirmed by the small gap between MAD and RSS, which does not exist in the dedicated execution (see Figure 4.3). In the experiment we observed that TPF had to come back and forth during the program interaction over 10 times to help vortex3 establish working sets. This is because program vortex is not strong enough to keep its established working set with the competition of gzip. Even for this type of program, TPF demonstrates its effectiveness. The numbers of page faults and execution time of vortex3 are reduced by 72% and 92%, respectively (see Figure 4.8).

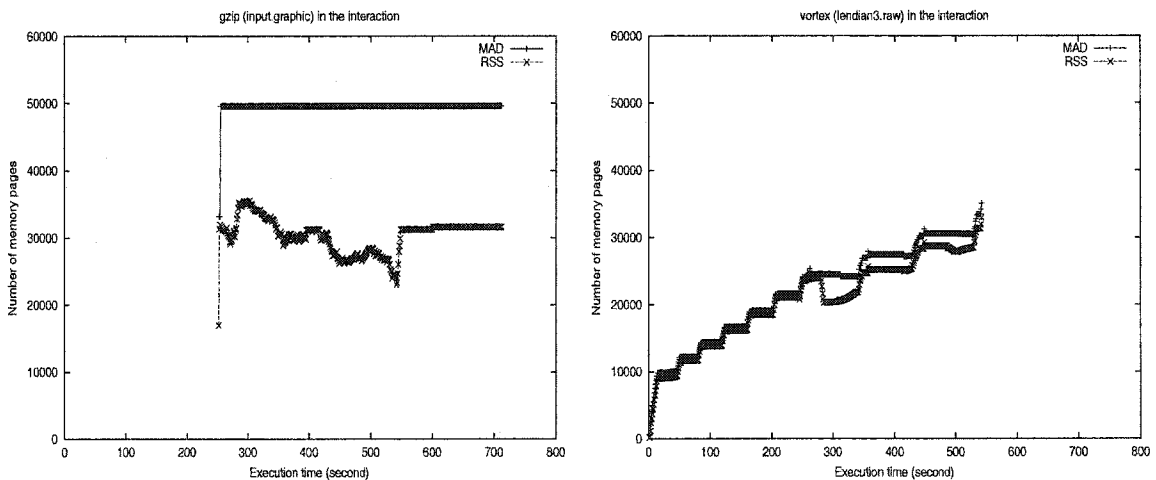


Figure 4.9: The memory performance of gzip (left figure) and vortex3 (right figure) during the interactions in the Linux with TPF.

The performance improvement for gzip is also significant. Its number of page faults and execution time are reduced by 72% and 64%, respectively. Intuitively, its performance should have been degraded because it contributed more memory space to vortex3 for building up its working set enforced by TPF. But this is not the case for two reasons. First, under the protection of TPF, vortex3 had an early completion. Then gzip could run without memory competition and use CPU cycles solely. Second, under the protection of TPF,

vortex3 could greatly reduce its page fault rate, which made gzip utilize most of the I/O bandwidth and reduced page fault penalty.

Figure 4.10 presents the memory usage measured by MAD and RSS of interacting programs gcc (left figure) and vortex3 (right figure) in the Linux with TPF. At the 397th second, memory demand from gcc rapidly rose, both programs started page faults due to memory shortage. The thrashing significantly lowered CPU utilization, which triggered TPF to take actions. Because gcc demanded memory gradually, and kept the gap between MAD and RSS small, gcc was selected for protection on its rising slope of the first MAD spike by TPF. The memory is dynamically allocated between two processes to ensure a reasonable level of CPU utilization. The period for the system to stay in the protection state is very limited, thus memory utilization is maintained. TPF successfully smoothed out the peak in memory load that might otherwise have caused the system to thrash. Compared with the same run in the original Linux kernel, the execution times of programs gcc and vortex3 are reduced by 69%, and 57% respectively; and the numbers of page faults of programs gcc and vortex3 are reduced by 99% and 87% respectively. (see Figure 4.8).

Figure 4.11 presents the memory usage measured by MAD and RSS of interacting programs vortex1 (left figure) and vortex3 (right figure) in the Linux with TPF. During the interactions at the execution time of 433th second, both programs started page faults due to memory shortage. The program vortex1 was then protected by TPF. We observed that vortex1 easily held its working set thereafter and only a small amount times of TPF involvement were needed. This is because vortex1 and vortex3 have similar memory access rates and patterns. Thus once vortex1 was given privilege to establish its working set, it would keep the working set by frequently using it. In contrast to the performance seen in

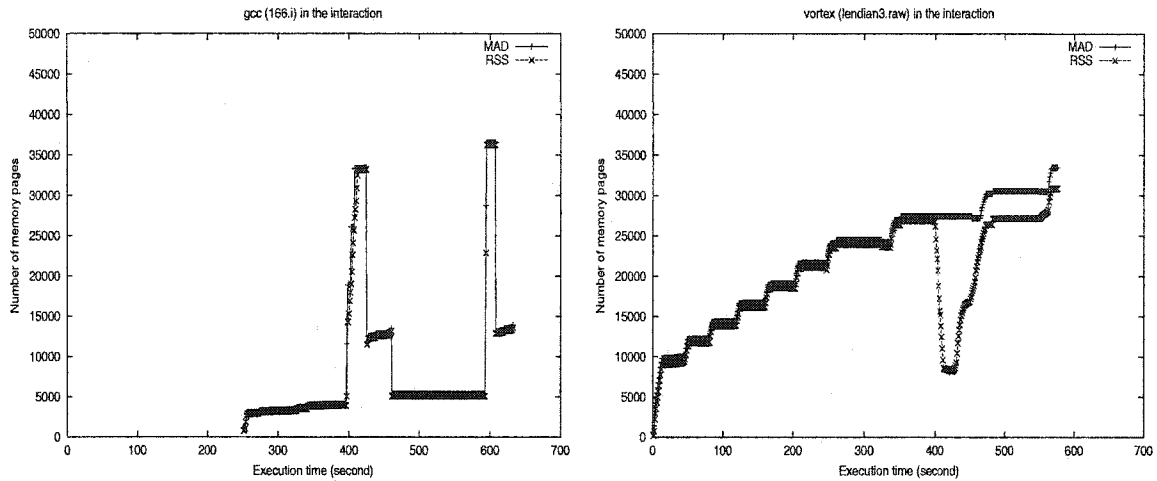


Figure 4.10: The memory performance of gcc (left figure) and vortex3 (right figure) during the interactions in the Linux with TPF.

Figure 4.6, a small correction from TPF could make a big difference in multiprogramming. Compared with the same execution in the original Linux kernel, the execution times of programs vortex1 and vortex3 are reduced by 46% and 42%, respectively; and the numbers of page faults are reduced by 99% and 80% respectively. (see Figure 4.8).

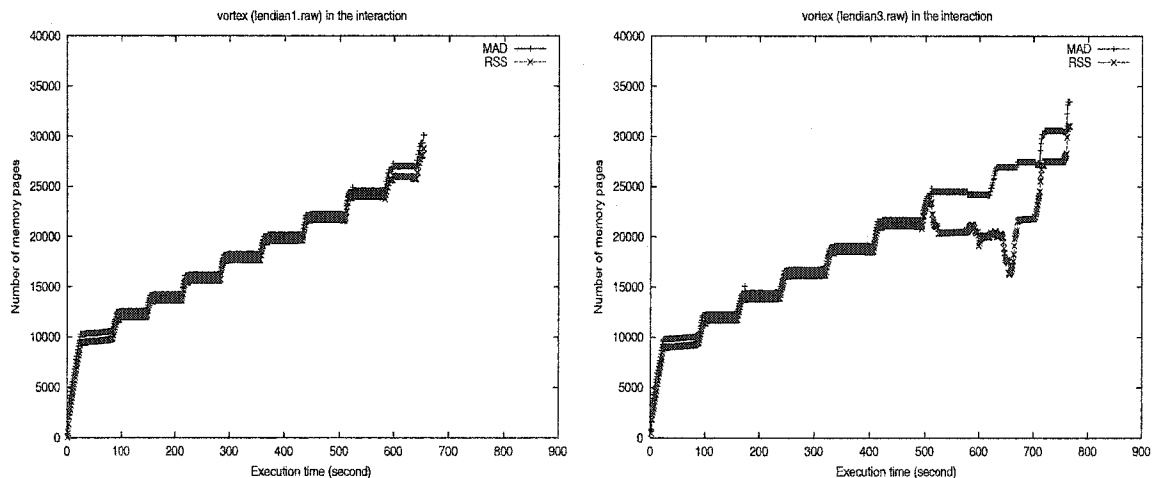


Figure 4.11: The memory performance of vortex1 (left figure) and vortex3 (right figure) in the Linux with TPF.

Figure 4.12 compares the total execution times for the three groups of interacting pro-

grams in Linux with and without TPF. We define the execution times of each pair programs under the same multiprogramming condition with sufficient memory space as the the ideal interaction execution time. Figure 4.12 shows that the total interacting execution times in the Linux with TPF for the three groups are significantly smaller than those in the Linux without TPF, and very close to the ideal execution times. These experiments also indicate that TPF has little runtime overhead.

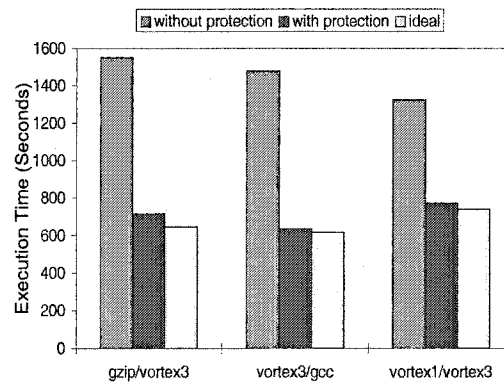


Figure 4.12: Comparison of total interaction execution times for the three group of program interactions in the Linux with TPF, without TPF and the ideal interaction times.

4.5.2 Experiences with TPF in the multiprogramming environment

1. *Under what conditions, does thrashing happen in a multiprogramming environment?*

Our experiments show that VMs in Linux can normally keep a reasonable CPU utilization even under an heavy workload, adapting the variance of memory demands, access patterns and access rates of different processes. A process that can frequently access its working set in execution interactions has a strong position for memory space competitions during interactions. However, under the following three condi-

tions, thrashing can be triggered.

- The memory demands of one interacting program have unexpected “spikes”.

Case studies in Figure 4.4 and Figure 4.5 show such examples.

- The variance of memory demands, access patterns, and access rates of interacting processes are similar. Case studies in Figure 4.6 show such examples.
- Serious memory shortage happens in the system.

2. *For what cases is TPF most effective?*

TPF is most effective in the first two cases discussed above. In other words, TPF is able to quickly resolve the thrashing for interacting programs having dynamically changing memory demands. We have shown that TPF is highly responsive to increase the CPU utilization and to stop thrashing by adapting page replacement to memory allocations. In addition, the scheduling action from TPF has little intervention to the system and multiprogramming environment because the protection period is very short, but is effective to lead the system back to normal.

3. *For what cases is TPF ineffective?*

If the memory shortage problem is too serious in a multiprogramming environment, the selected process may build up and hold its working set in memory at the cost of obtaining most of the memory space of other processes. Although TPF can still cause CPU cycles to be effectively utilized, the CPU overhead serving page faults of other processes will significantly increase, and I/O channels may become heavily loaded due to a large amount of page faults. As a result, the protected process will

not run smoothly. Under such conditions, the load control has to be used to swap a process for releasing memory space. With the support of TPF, load control facility will be used only when it is truly necessary.

4. *How do the threshold parameters affect the performance of TPF?*

We summarize the relationships between the four threshold parameters (see Chapter 4.3.1) and effectiveness of TPF, and the memory performance of interacting programs. Smaller values of parameters CPU_Low and PF_high will make TPF more responsive to system thrashing. On the other hand, larger values of CPU_High and PF_Low will make the identified process stay longer in the protection state after it enters the state. Thus, adjusting these parameters is equivalent to changing the extent of TPF intervention to the system. The parameters are set only based on system requirements, not dependent on application program natures. For example, for systems with high I/O bandwidths (e.g. parallel disk arrays), values of PF_High and PF_Low can be set larger, because page faults can be resolved quickly. In our experiments we found that the performance of TPF was quite stable within a large range of parameter values.

4.6 Related Work

Improvement of CPU and memory utilizations has been a fundamental consideration in the design of operating systems. Extensive research on thrashing had been conducted in the 1960s and 1970s. Among the proposed policies the most influential one, which was able to thoroughly protect against thrashing while keeping high CPU utilization, is working set policy. Working set policy provides a solution at the page replacement level, similar to our

policy.

4.6.1 The Working Set Model and its Implementation Issues

Denning proposes a working set model [24], [26], and [27] to estimate the current memory demand of a running program in the system. A working set of a program is a set of its recently used pages. Specifically, at virtual time t , the program's working set $W_t(\theta)$, is the subset of all pages of the program, which has been referenced in the previous θ virtual time units (working set window). The task's virtual time is a measure of the duration the program has control of the processor and is executing instructions. The working set model ensures that the same program with the same input data would have the same locality measurements, which is independent of the memory size, the multiprogramming level, and the scheduler policy used. A working set policy is used to ensure no pages in the working set of a running program will be replaced. Assume that priorities among the processes exist. Once there is a request for free pages, but they are not available, the processes with the lowest priority has to produce a victim page for replacement. This implies that an active process with the lowest priority may not fully allocate its working set. Since the I/O time caused by page faults is excluded in the working set model, the working set replacement algorithm can theoretically eliminate the thrashing caused by chaotic memory competition. Comparatively, other global policies like LRU approximations (two-handed clock, FIFO with second chance) used in the currently popular UNIX-like operating system, are highly susceptible to thrashing, because a program's resident set depends on many factors besides its own locality. Our experimental observations are consistent with the conclusions in the cited work on working set models.

A major difficulty to implement the working set model in a modern computer system is its implementation overhead scaling with the capacity of CPU and memory. The working set model can be implemented by either hardware or software. The hardware approach requires that each page frame be associated with a counter and an identifier register indicating which process it belongs to. A broadcast clock pulse periodically increments the counter of each page frame whose identifier register matches the memory domain of a running process. When the running process refers to a page, the counter of that page frame is automatically reset. When a counter is incremented over a pre-determined threshold value, the corresponding page frame is no longer a member of the working set.

Compared with the approach of only associating a page-reference bit with each page frame to support LRU related page replacement policies in Linux and Unix systems, an implementation of the working set detector is more expensive. With the increase of CPU speed and memory capacity, and with an increasing amount of memory-intensive workloads in applications, the number of active pages owned by a process has dramatically increased, which has become a major reason to limit such an implementation. In addition, since system thrashing is considered as an exceptional event, it may be difficult to convince computer vendors to provide a hardware support for the working set model. Instead, the computer architects prefer to adopt some brute-force methods as exceptional handlers, such as to release memory space by urgently removing some processes.

Implementing the working set detector by system software, we need routinely update a software counter associated with each page frame. Since monitoring huge amount of page frames is routine operations in memory management, it would affect the system performance when the system functions normally.

Although there exist a number of good approximations, its implementation cost of the working policy model may limit its direct usage in modern computer systems. However, this model has given us several important motivations in memory system designs and implementations. First, being a local memory policy, working set policy has inherent load control and need no special, additional mechanism to deal with thrashing. This certainly save the cost of additional mechanism to stabilize global LRU policy. While built on global replacement policy, TFP can protect the working set of a process, like temporarily under a local memory policy to eliminate the thrashing. Second, working set policy employs “feed-forward control”, rather than “feedback control”, which means working set does not have to react to thrashing, but avoid thrashing in advance. The instability resulted from feedback of load control is greatly reduced by the TPF responsive action on the global policy.

implementations in existing system kernels, and guided by the principle of the working set model, we propose the TPF, which is not part of the routine operations in memory management, but is only triggered in an early stage of thrashing to effectively stop the thrashing or significantly delay the load controls.

4.6.2 Other Related Work

Studies of page replacement policies have a direct impact on memory utilization, which have continued for several decades (e.g. a representative and early work in [1], and recent work in [30, 67]). The goal of an optimal page replacement is to achieve efficient memory usage by only replacing those pages not used in the near future when available memory is not sufficient, reducing the number of page faults. In a single-programming environment, these proposed methods address both concerns of CPU and memory utilization since any

extra page faults due to low memory utilization will make the CPU stall. However system thrashing issue in a multiprogramming environment can not be fully addressed by the cited work due to the conflicting interests between CPU and memory utilization.

In the multiprogramming context, existing systems mainly apply two methods to eliminate thrashing. One is local replacement, another is load control. A local replacement requires that the paging system select pages for a program only from its allocated memory space when no free pages can be found in their memory allotments. Unlike a global replacement policy, a local policy needs a memory allocation scheme to satisfy the need of each program. Two commonly used policies are equal and proportional allocations, which can not capture dynamical changing memory demand of each program [38]. As a result, memory space may not be well utilized. On the other hand, an allocation policy dynamically adapting to the demand of individual programs will shift the scheme to global replacement. VMS [41] is a representative operating system using a local replacement policy. Memory is partitioned into multiple independent areas, each of which is localized to a collection of processes that compete with one another for memory. Unfortunately, this scheme can be difficult to administer [44]. Researchers and system practitioners seem to have agreed that a local policy is not an effective solution for virtual memory management. Our TPF is built on a global replacement policy.

The objective of load control is to lower the MPL by physically reducing the number of interacting processes. A commonly used load control mechanism is to suspend/reactivate processes, even swapping out/in processes to free more memory space, when thrashing is detected. The 4.4 BSD operating system[50], AIX system in the IBM RS/6000[32], HP-UX 10.0 in HP 9000 [31] are examples that adopt this method. In addition, HP-UX

system provides a “serialize()” command to run the processes one at a time when thrashing is detected. In contrast, TPF protects system from thrashing at the page replacement level. Memory allocation scheduling at this level allows us to carefully consider the tradeoff between CPU and memory utilizations.

In [35], we proposed another thrashing prevention mechanism called *Token-ordered LRU*, which attempts to prevent the occurrence of thrashing by eliminating *false LRU pages*. False LRU pages are produced because of I/O penalties of page faults, rather than because of the program access delays. Using a token to set a memory allocation priority, Token-ordered LRU can effectively prevent thrashing and achieve a performance improvement similar to the TPF.

4.7 Summary

We have investigated the risk of system thrashing in page replacement implementations by examining the Linux kernel code of versions 2.0, 2.2, and 2.4, and running interacting SPEC2000 benchmark programs in a Linux system. Our study indicates that this risk is rooted in conflicting interests of requirements on CPU and memory utilizations in a multiprogramming environment. We have experimentally observed several system thrashing cases when processes dynamically and competitively demand memory allocations, which causes low CPU utilization and long execution time delays, and eventually threatens system stability.

We have proposed TPF and implemented it in the Linux kernel to prevent the system from thrashing among interacting processes, and to improve the CPU utilization under

heavy load. TPF will be awakened when the CPU utilization is lower than a predetermined threshold, and when the page fault rates of more than one interacting processes exceed a threshold. TPF then grants privilege to an identified process to limit its contributions of NRU pages. We create a simple kernel monitoring routine in TPF to dynamically identify an interacting process which highly deserves temporary protection. The routine also monitors whether the identified process has satisfactorily lowered its page fault rate after the protection. If so, its privilege will be disabled to let it equally participate in contributing NRU pages with other processes.

Conducting experiments and performance evaluation, we show that the TPF facility can effectively provide thrashing protection without negative effects to overall system performance for three reasons: (1) the privilege is granted only when a thrashing problem is detected; (2) although the protected process could lower the memory usage of the rest of the interacting processes for a short period of time, the system will soon become stable by the protection; and (3) TPF is simple to implement with little overhead in the Linux kernel. Because the conflicting interests between CPU and memory utilization are inherent in global page replacement, and our solution is targeted at regulating the conflicts through tuning page replacement, we believe that the TPF idea is applicable to VMs of other UNIX-like systems.

Chapter 5

Multi-Level Buffer Cache Management

In a large client/server cluster system, file blocks are cached in a multi-level storage hierarchy: client buffer caches, multiple server buffer caches, and built-in caches of disks at the bottom level. More and more applications rely on the hierarchy for their file accesses, so the caching effectiveness of the hierarchy is important to the application performance.

5.1 Background

5.1.1 Hierarchical Caching and its Challenges

With the ever-widening gap between the speeds of processors and hard disks, practitioners try to make a full use of the available buffer caches along a file block retrieving route for the purpose of satisfying the requests before they reach disk surfaces. Besides the buffer caches at clients, the requested blocks can also be cached at server buffer caches and disk built-in caches, which form a multi-level buffer cache hierarchy (see Figure 5.1). For example, modern high-end disk arrays typically have several gigabytes of cache RAM. Though multiple

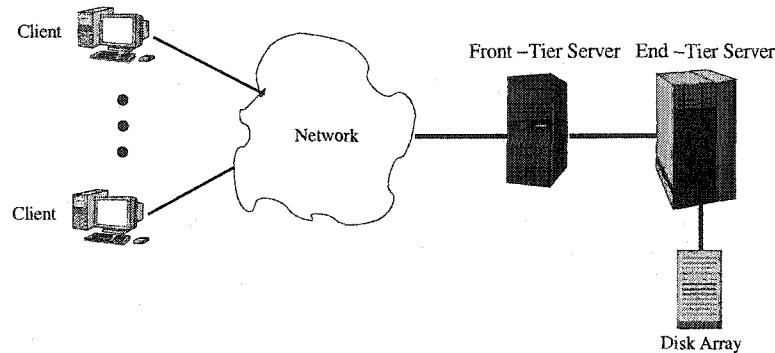


Figure 5.1: Multi-level buffer cache hierarchy. Caches are distributed along the clients, intermediate servers, and disk array, where accessed blocks can be buffered.

buffer resources are lined up and their aggregate size is increasingly large, the issue of how to make them work together effectively to deliver the expected performance commensurate to the aggregate size of the distributed buffer caches is still not well addressed. There are two challenges related to this issue.

The first challenge comes from the weakened locality in the low level buffer caches¹. Caching works because of the existence of locality, which is an inherent property of application workloads. Only the first level buffer cache is exposed with the original locality and has the highest potential to exploit it. Low level caches hold the misses from their upper level buffer caches. In other words, the stream of access requests from applications is filtered by the high level caches before it arrives at the low level ones. Thus the access stream seen by low level caches has weaker locality than those available to the first level cache. The performance of widely used recency-based replacements such as LRU can be significantly degraded once these replacements are employed in the low level buffer caches. Muntz and Honeyman [54] as well as Zhou et al [82] have observed the serious performance degradation in their file server buffer cache studies. In a work to investigate the cost-effectiveness of

¹By low level buffer caches, we customarily refer to the caches not close to the workload running clients. Similarly, high levels of buffer caches are those close to the clients. Thus, the first level buffer cache is the client buffer cache with the highest level.

disk built-in caches for desktop PCs, Zhu and Hu found that the built-in caches contribute little more to the average response time reduction when its size exceeds 512KB with a client cache size of 16MB [84]. The above cited work indicates applying a replacement independently at a low level buffer cache could lose its chance to exploit the original locality. This motivates us to make replacement decisions based on the original access stream, which is only available at the first level cache.

The second challenge comes from the undiscerning redundancy among levels of the buffer caches. Redundancy means a block is cached and duplicated along its retrieving route in more than one caches. Without a proper coordination among the levels, blocks could reside undiscerningly in multiple buffer caches for a long period of time before they become cold enough to be replaced by a local replacement algorithm. The redundancy can cause the buffer cache hierarchy seriously under-utilized. Even if the aggregate size of the multi-level buffer caches could hold the working set, the hierarchy would behave as if it were as big as the single level of cache with the largest size under some access patterns. We propose to use an unified replacement scheme for a multi-level cache hierarchy, which can determine an appropriate place for a block to be cached (if it needs being cached). Thus undiscerning redundancy can be eliminated. The hierarchy can perform as an unified cache with the size equivalent to the aggregate size, so that all the cache spaces are fully utilized.

5.1.2 Possible Solutions: Customized Second-Level Replacement and the Unified LRU

We have seen recent work on each of the two issues. Most of the work attacks the aforementioned challenges separately. Multi-Queue [82, 81] and unified LRU [76] are two repre-

sentative work among them.

Multi-Queue(MQ) is a customized second-level replacement algorithm. To overcome LRU's inability with weak locality at the second level cache, MQ resorts to the frequency, the number of access times of a block, to differentiate the locality of the accessed blocks. For this purpose, they sets up multiple queues and uses access frequencies to determine which queue a block should be in. Whenever the access frequency of a block accumulates to a certain threshold, it moves up to a queue for high frequency blocks. Periodically, blocks that are not accessed for a period of time are demoted into a queue for low frequency blocks until they are finally replaced. By tracking and utilizing a deep access history, MQ can achieve a higher hit ratio than LRU in a second-level cache. However, there are two weakness in MQ when it is used to address the challenges in the multi-level caching hierarchy. First, it inherits the disadvantages of frequency-based replacement algorithms such as Least Frequency Used (LFU), which respond to the access pattern changes slowly, and carry a high overhead. Second, because the clients own the original locality information the lack of hints from clients greatly limits its potential of exploiting locality for high hit ratio greatly limited.

Another solution was proposed by Wong and Wilkes [76] to eliminate the redundancy simply apply an unified LRU scheme in a two-level buffer cache: client and disk array built-in buffer caches. As it shows in Figure 5.2, there is an unified LRU stack. The first portion of the LRU stack corresponds to the client cache, and the second portion of the LRU stack corresponds to the disk array cache. Any blocks moving from the first portion into the second portion due to the increased recency would incur a demotion, an operation that transfers a block from the current level to its next low level cache. Since any recently

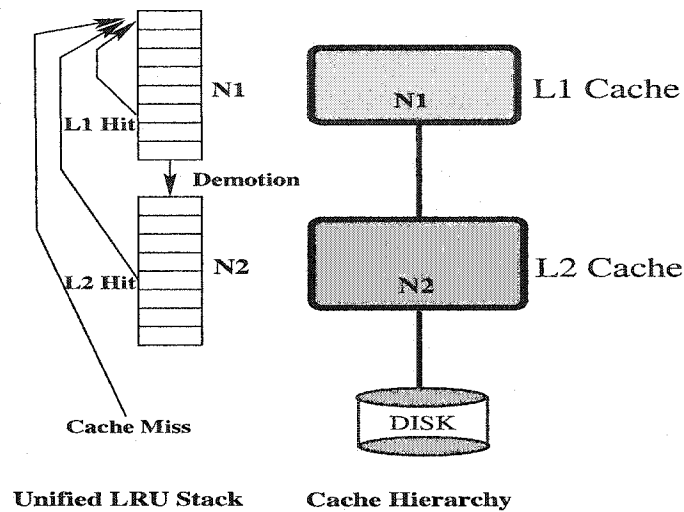


Figure 5.2: In the two-level unified LRU scheme, there is an unified LRU stack corresponding to the two level of caches. The size of each individual LRU stack, $N1$ or $N2$ is equal to its respective cache size in terms of blocks. there are three type of accesses: (1) a hit in the $L1$ cache. (2) a hit in the $L2$ cache. (3) a miss in the two caches. If all the three cases, the accessed blocks are moved to the top of the stack. Except the first case, the block at the bottom of $L1$ LRU stack is demoted onto the top of the $L2$ stack.

referenced blocks are brought into the top of LRU stack, all newly referenced blocks are cached in the first level cache and slipped to the low level caches through demotions if they are re-accessed. Though their scheme has an significant advantage over independent replacements by eliminating redundancy, there are two critical weakness of the unified LRU schemes. First, there is no explicit block placement arrangement adapting to their access pattern. For a block requested by a client, it has be transfered to the client for its use. However, this block is not necessarily to be cached there. For example, the block which is not possible to be re-accessed soon should be quickly evicted from the client cache after its use and may be cached at a low level cache or even not cached. By indiscriminately storing all the accessed blocks, high level caches cannot serve the blocks with strong locality well. Second, it could generate a large number of demotions because any access that is not a hit in clients accompanied with a demotion. It has been shown that the benefits of cache

coordinations can be nullified by the demotion cost once the I/O bandwidth is below a certain threshold [14].

5.1.3 Our Principles to Address the Challenges

Our general approach to address the challenges includes two steps. At first, we propose a new method to quantify locality strength of accessed blocks. Then we develop a mechanism to layout the cached blocks along the cache hierarchy according their quantified locality strength. To serve the purpose of block placement and replacement in multi-level buffer caches, we have two requirements on the locality strength quantification method: (1) distinction of locality strengths; and (2) stability of the distinction, which are also our two principles to address the challenges. Regarding the distinction, if the algorithm can accurately and responsively distinguish blocks with strong locality from those with weak locality², then the stronger the locality of blocks is, the higher level of cache they should be placed in. The distinction of this hierarchical locality will make high levels of caches contribute more to the hit ratios, which reduces the average access time because of their low hit times. Since the arrangement of block caching positions is based on the distinction of locality strengths, we need to re-arrange the blocks once the locality strengths change, which means to transfer blocks among levels. This incurs a communication cost. Thus the stability of the distinctions is critical to keep a low communication cost introduced by an unified caching scheme.

Following these two principles, we propose a client-directed file block placement and

²By a block with strong locality, we mean it is highly likely to be referenced soon, and it contributes more to the hit ratio by being cached than the one with weak locality. The strengths of locality are quantified differently in different replacements, which we will discuss in the next section.

replacement protocol, where non-uniform strengths of locality are dynamically identified at the client level to direct file blocks being placed or replaced at different levels of buffer caches accordingly. The effectiveness of our proposed protocol comes from achieving the following three goals. (1) The multi-level cache retains the same hit ratio as that of a single level cache whose size equals to the aggregate size of multi-level caches. (2) The non-uniform locality strengths of blocks are fully exploited and ranked to fit into the physical multi-level caches. (3) The communication overheads between caches are reduced.

5.2 Quantifying Non-uniform Locality Strengths in Hierarchical Buffer Caching

5.2.1 Methods to Distinguish Locality Strengths

Caching works because of the existence of locality. While spatial locality is mostly exploited in increasing block sizes and prefetching, replacement algorithms usually depend on the temporal locality to make re-accessed blocks hit in the cache. Belady first introduced the concept of locality and recognized its importance in the context of memory systems [7]. With a temporal locality, if a block is referenced, it will tend to be referenced again soon. Although there exists a clear description and an agreed intuitive understanding on the notion, a common quantitative definition on locality is rarely seen in literature. However, for the replacement purpose, each replacement algorithm has its own defined method to quantify locality strengths and to make distinctions among them.

Suppose the block reference stream is $\{R_t, t = 0, 1, 2, \dots\}$, the block accessed at time t is $block(R_t)$, as is shown in Figure 5.3. The distance between two references R_i and R_j

is the number of other distinct blocks accessed between time i and time j . Specifically, if $\text{block}(R_i) = \text{block}(R_j)$, the distance is called re-use distance of $\text{block}(R_i)$. For example, in a segment of reference stream denoted as blocks accessed, $\{\dots a, b, c, b, a, \dots\}$, the re-use distance of blocks a is 2 because there are two other distinct in-between blocks b and c . The distance is also the distance between the positions of two accessed blocks in the LRU stack [20], which is a list in which all accessed blocks are stored in the order of their references, and any newly accessed block is moved to the top of stack. Though LRU stack was initially used for the LRU replacement algorithm, it has been widely used to describe and study various replacement algorithms, such as [37, 33, 67].

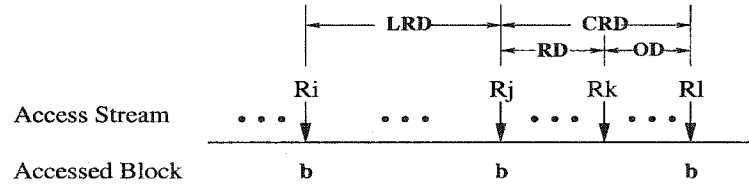


Figure 5.3: In access stream $\{R_t, t = 0, 1, 2, \dots\}$, R_i , R_j , and R_l are three immediately consecutive references to block b . The current time is k . With these timing points, there are various measurements that can be used to quantify the locality strength of block b at time k , including the distance from R_k to R_l , called *OPT Distance* (OD), the distance from R_j to R_k , called *Recency Distance* (RD), the distance from R_j to R_l , called *Current Re-use Distance* (CRD), and the distance from R_i to R_j , called *Last Re-use Distance* (LRD).

Here we do not consider the methods using frequency to estimate locality, because it becomes irrelevant to the current locality when an access took place much earlier than the recent accesses.

As an off-line optimal replacement, OPT, uses the distance between the current time and the next reference to a block, to quantify the locality strength of the block. We call the distance OPT distance (OD). Considering that the OPT replacement maximizes the hit ratio for a given cache by selecting a block with the largest OD for replacement, OD

provides the most accurate distinction of locality strengths among accessed blocks. The LRU replacement takes the assumption that a block accessed recently will be accessed again soon. Using the time of the last reference to a block to predict the time of its next reference, the LRU algorithm uses *Recency Distance* (**RD**), which is the distance between its last reference and the current time, to simulate the *OPT Distance* (**OD**). Both OD and R are measured based on the current time, so they change with every reference to any block. The quantified locality strengths with OD or R could be very dynamic. When the stability of quantified locality strengths is of concern, it is unclear where a block should be cached to reduce the communication cost.

In the unified LRU replacement [76], when a block slips down in the LRU stack with the ongoing references, it may incur demotions once its recencies reach the its local LRU stack size. Had it been known at what recency a block would be re-accessed when the block was requested, we would have cached it directly on the level of cache corresponding to that recency, thus the demotions could be avoided. This motivates us to use the distance between last reference and next reference to a block, called *Current Re-use Distance* (**CRD**) to quantify locality strengths. CRD is also the recency at which the block will be referenced next time. After a block is accessed, its CRD will not change until its next reference. This helps to stabilize the distinction of locality strengths. Because CRD represents a future access timing, it is not collectible on-line. To simulate CRD in an on-line algorithm, we use *Last Re-use Distance* (**LRD**), to simulate CRD (see Figure 5.3). LRD is also the recency at which a block was accessed last time.

However, LRD could miss some most recent access information. The LRD of a block does not count the recent references after the last reference to the block, which is reflected

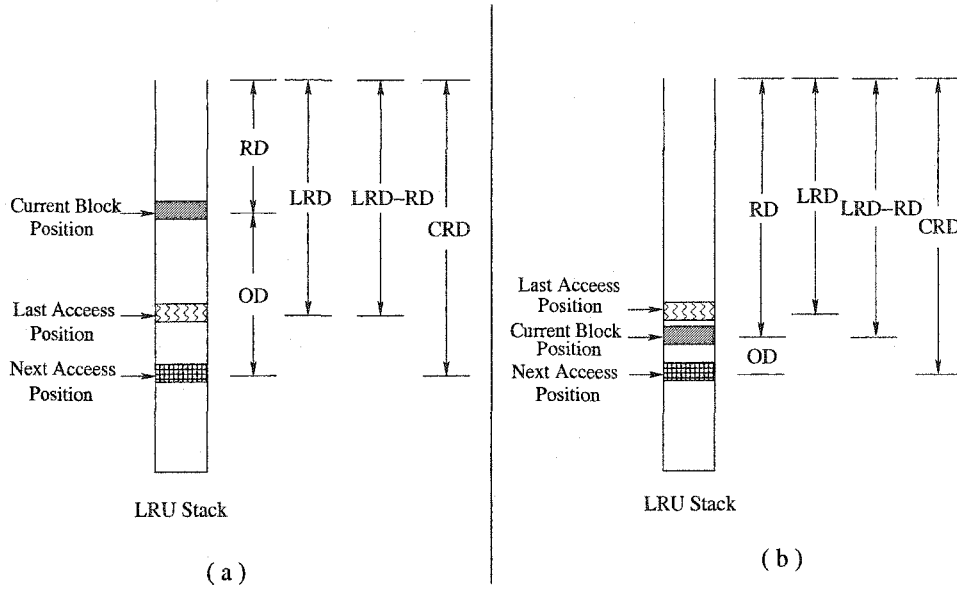


Figure 5.4: In the LRU stack, for a given block, the position for the last access to the block corresponds to its LRD, its current position in the stack corresponds to its RD, and the position for its next access corresponds to its CRD. Before its current position exceeds its last access position (see left figure (a)), LRD-RD is LRD; after that (see right figure (b)), LRD-RD becomes RD. This allows LRD-RD to more accurately simulate CRD. The illustration also shows that RD and OD change with every reference.

in its recency. To responsively capture the changes of locality scope (a hot block becomes cold, or vice versa), we use the recency distance to take place of LRD once recency exceeds LRD. That is, we use the larger of LRD and R to simulate CRD, called LRD-RD. All of aforementioned locality strength measurements can be illustrated in the LRU stack shown in Figure 5.4. We will develop of our caching protocol based on a data structure using the LRU stack as a basis.

5.2.2 Comparisons of Locality Strength Quantification Methods

Each of the four measures, OD, RD, CRD, and LRD-RD, is associated with a replacement algorithm. A replacement algorithm works in the way that it has its accessed blocks ranked

according to a certain measure, and selects the one with the least ranking for replacement once a victim block is needed. For example, the measure used by OPT is OD and the measure used by LRU is RD. How well a measure satisfies the two requirements on its ability — distinction of locality strengths and the stability of the distinction, determines how well the corresponding replacement algorithm serves as an unified replacement algorithm for a multi-level cache hierarchy.

To understand and compare the two abilities of the measures, we use six small-scale workload traces (*cs*, *glimpse*, *zipf*, *random*, *sprite*, and *multi*) with representative access patterns for the evaluation. The traces are briefly described in the following..

1. **cs** is an interactive C source program examination tool trace, which was collected with about 9MB kernel sources as input.
2. **glimpse** is a text information retrieval utility trace. The search was conducted on the text files of about 50MB and their index files of about 5MB.
3. **zipf** is a synthetic trace, in which only a few blocks are frequently accessed. Formally, the probability of a reference to the i th block proportional to $1/i$. The data set it accessed is 39MB.
4. **random** is a synthetic trace with a spatially uniform distribution of references across all the accessed blocks. The data set it accessed is 39MB.
5. **sprite** consists of requests to a file server from client workstations for a two-day period in the Sprite network file system [4], which covers 28MB data set.

6. **multi** is obtained by executing four workloads, *cpp*, *gnuplot*, *glimpse*, and *postgres*, together, which covers up 29MB data set.

Among these traces, *cs*, *glimpse* are used in [18, 19, 33], *sprite* is used in [45, 33], *multi* traces are used in [42, 33], and *zipf*, *random* are used in [76] to evaluate the performance of replacement algorithms. These traces represent the major access patterns common to the I/O requests. Traces *cs* and *glimpse* have a looping access pattern, where all blocks are regularly and repeatedly accessed. Trace *sprite* has a temporally-clustered access pattern, where blocks accessed more recently are the ones more likely to be accessed soon. It is an LRU-friendly pattern. The access pattern of trace *random* is common in database applications. Zipf-like access patterns exhibited in trace *zipf* are typical for file references in Web servers. Trace *multi* has an access pattern mixed with sequential, looping and probabilistic references.

For a given measure, each accessed block has a changing value. When there is a reference to a block, the value of the block, and possibly the values of other blocks are changed. For each measure we maintain an ascendingly ordered block list by their measure values. The list is dynamically updated with each new block reference to maintain the order. In the process there are block movements in the list. We divide the full length of each list into ten segments of equal size. We collect the number of references to each segment to observe the locality strength distinction. We also collect the block movements across each of the segment boundaries to observe the stability of the distinctions when the list is updated with references. For example, if the given measure is RD, the list is actually an LRU stack with its size unbounded. Each of the ten segments represents a range of stack positions

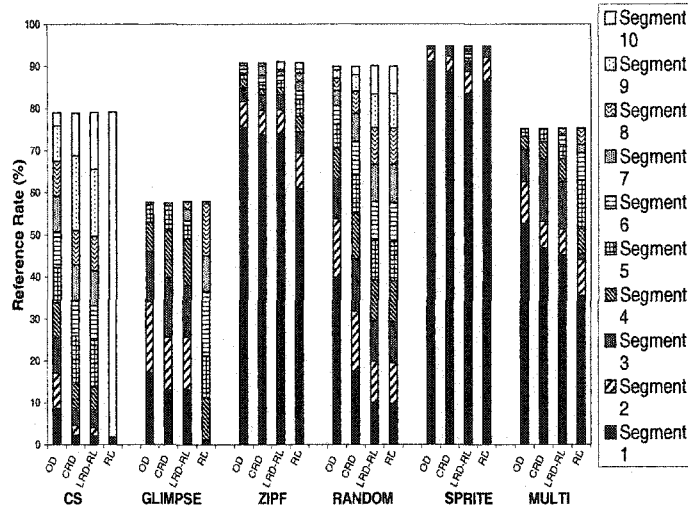


Figure 5.5: Reference ratios to each of the segments (the ratios between the number of references to a segment and the number of all references in a workload). It also shows the accumulative reference ratios for the first N segments in each workload, where N is 1 through 10.

with certain recencies. What we want to investigate is that positions of the stack where references take place and the block movements in the stack for a given workload trace.

Figure 5.5 shows the reference ratio distributions in the list for each measure. Each of the measures orders accessed blocks in its list and places the blocks with small values at the head of the list (in the case of measure RD, it is the top of an LRU stack). A good distinction of locality strengths should generate a reference ratio distribution with more hits appearing in the head portion of the list than those in its tail portion. Assuming each of the segments corresponds to a level of cache, we can observe the hit ratio on each level of cache. From the figure we have the following observations:

- (1) OD provides the best reference ratio distribution. The higher (closer to the list head, and with a *smaller* segment number in Figure 5.5) a segment is, the higher reference

ratio the segment achieves for OD. This reflects the strong ability of OD to accurately make the distinction of locality strengths. Actually, the distribution generated by OD is optimal considering optimality of the OPT algorithm. While high segments are mapped on the high levels of caches, which have small hit times, such a distribution helps reduce the average access time. In contrast, RD provides the worst distribution, though it attempts to simulate (predict) ND. This is specially apparent for the workload with a looping access pattern: *cs* and *glimpse*. Most of their references go to the low segments (after Segment 9 in *cs*, and after Segment 3 for *glimpse*). This indicates that even an unified LRU replacement can hardly achieve high hit ratios until the aggregate cache size can hold all the accessed blocks. RD only performs well on the workloads with an LRU-friendly access pattern, such as *sprite*.

(2) CRD performs well for all the workloads with various access patterns. This reflects its ability to make consistently accurate distinction. Except for trace *random*, LRD-RD performs very closely to CRD, though it does not depend on the future knowledge. Without looking ahead, all the on-line algorithms could perform the same as RANDOM replacement for trace *random* at best, which randomly selects a block for replacement and has a hit ratio proportional to the cache size. Both LRD-RD and RD obtain such a distribution for the trace.

(3) For the two on-line measures, LRD-RD produces significantly better locality distinctions than RD for workloads *cs*, *glimpse*, *zipf*, and *multi*. For LRU-friendly workload *sprite*, both R and LLD-R perform very well, and RD performs a little better than LRD-RD.

Figure 5.6 shows block movement ratios between the number of block movements across each of the segment boundaries and the number of all references for each of the four mea-

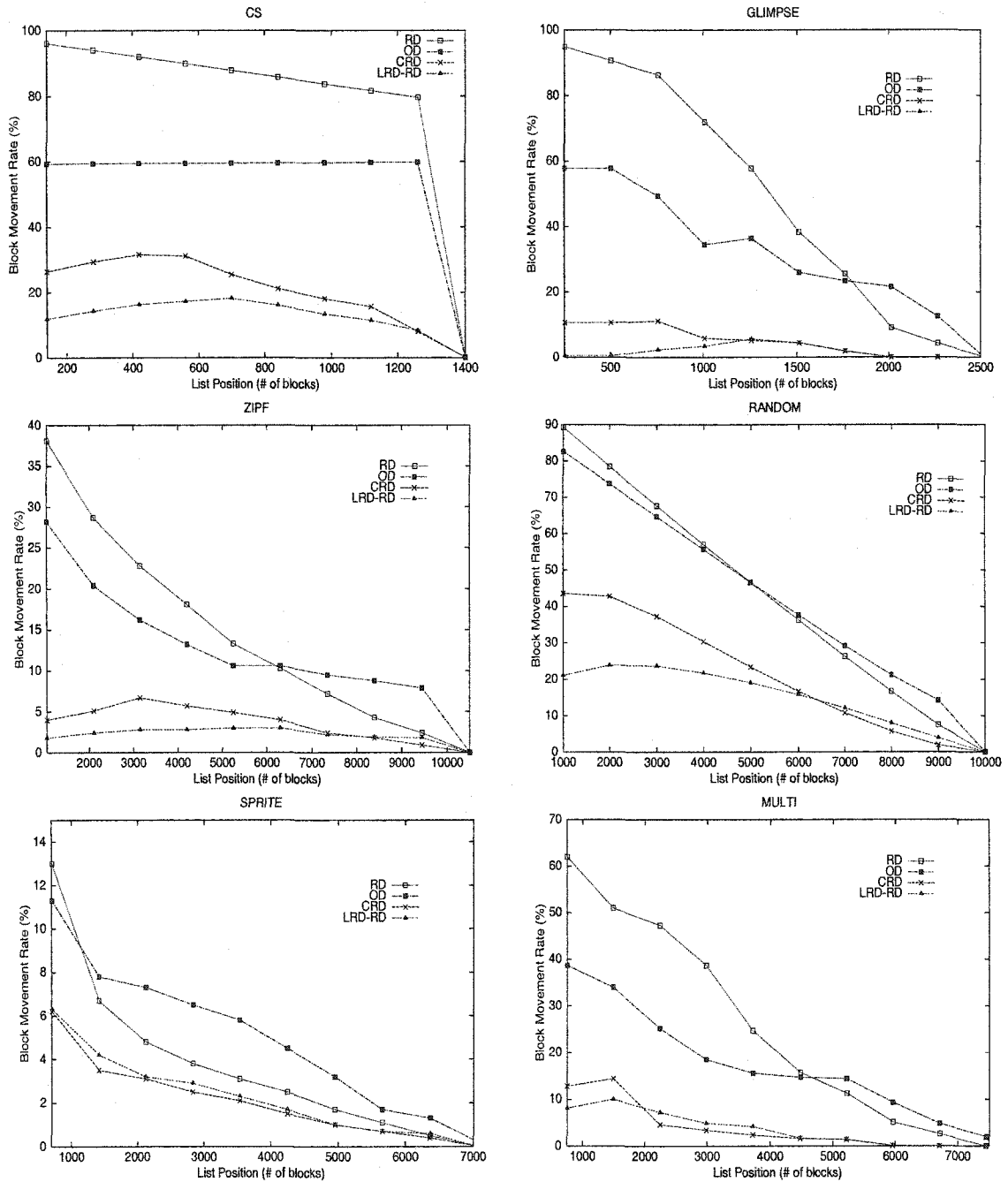


Figure 5.6: Movement ratio curves showing the ratios between the number of block movements across a segment boundary of the ordered lists and the number of total references for the four measures: OD, RD, CRD, and LRD-RD on various workloads. It shows that there are two groups of curves: OD and RD with high movement ratios, NRD and LRD-RD with low movement ratios.

tures. For example, the first point from the left on a curve represents the ratio between the number of times that the blocks cross the boundary between the first and second segments and the number of all references. A small movement ratio means a high stability for the distinction of locality strengths. When the segments are mapped to the levels of caches and a boundary corresponds to the interface of two adjacent levels of caches, a movement ratio determines the communication overhead in an unified caching. We have the following observations in the figure:

(1) OD and RD have the highest movement ratios, which have been expected because of their volatility. Comparatively, CRD and LRD-RD have much lower movement ratios.

(2) The ratio gaps between CRD (resp. LRD-RD) and OD (resp. RD) are especially pronounced with the looping pattern trace *glimpse*. However, even for the LRU-friendly workloads like *sprite* and *zipf*, the gaps are still considerably large. This demonstrates that an on-line unified caching based on LRD-RD promises a much smaller additional communication cost than that based on RD.

(3) The ratios of LRD-RD are smaller than those of NLD in most cases.

	OD	RD	CRD	LRD-RD
Ability to distinguish locality strengths	strong	weak	strong	strong
Stability of distinctions	weak	weak	strong	strong
On-line measures	no	yes	no	yes

Table 5.1: Comparisons of the four measures on locality strengths by comparing their abilities to distinguish locality strengths, the stabilities of the distinctions, and if on-line measurements are possible.

Table 5.1 summarizes the four measures distinguishing locality strengths, showing that using LRD-RD is a desired basis to building an unified caching protocol.

5.3 The Unified and Level-aware Caching (ULC) Protocol

5.3.1 An Executive Summary

We have shown that the position of a block in the list ordered by LRD-RD provides a strong hint for caching the block on a level corresponding to its list position, or not caching it at all³. This also assures us that the block would still stay there with a high probability when the block is accessed next time. Effectively using the hint, we propose a multi-level buffer placement and replacement protocol, called *Unified and Level-aware Caching* (ULC) protocol to exploit hierarchical locality. Based on the access patterns and available cache sizes on each level, ULC running at the first level client dynamically ranks the accessed blocks into levels L_1 , L_2 , ..., and L_{out} according to their LRD-RD positions, thus directing them to be placed (cached) at level L_1 cache, level L_2 cache, ..., or not cached at any levels at the time of the retrieval, respectively. The size of the first level cache determines the number of L_1 blocks, those with the smallest LRD-RD values, and the same correlation holds for other levels of caches. Low level buffer caches are not responsible for extracting locality from the filtered request stream presented to them any more. Every block request from the high level buffer cache carries a level tag, so the low level caches only take their actions accordingly. If the attached level tag matches its level number, this level will cache the retrieved block. Otherwise, the block is discarded after the block is sent to its next upper level cache. When the block positions need adjusting, the client sends block demotion instructions to low level caches, which demand a block originally residing in a cache be

³Those requested blocks that should not be cached in the first level cache are still brought into the client for its use, but will not be cached there. i.e. these blocks will be quickly replaced from the client after the reference.

demoted into its next low level cache. Our client-directed protocol attempts to answer the following questions in designing hierarchy caching algorithms: (1) how to exploit the locality in the entire buffer cache hierarchy thoroughly and consistently; (2) how to make the exploited locality usable by all buffer caches in the hierarchy; and (3) how to minimize the overhead of the protocol.

5.3.2 A Detailed Description

In Chapter 5.2.2 we have shown the LRD-RD measure is a promising basis on which to build a multi-level caching protocol. However, an implementation of an algorithm exactly based on LRD-RD ranking criterion will take at least $O(\log n)$ time, where n is the number of distinct accessed blocks. This is the cost of block ordering. In order to develop an efficient algorithm with the time complexity $O(1)$, we transform the process to determine the position of a block in an LRD-RD ordered list into two separate steps: (1) When a block is accessed, its recency is 0, so its LRD-RD is LRD, which is the recency at which it was just accessed. We use the LRD to determine in which segment the block will be cached at the time of retrieval. (2) Once a block is assigned into a specific segment, we use RD to determine its position in the segment. Each segment corresponds to a level of cache, and the size of the segment is the same as that of the cache.

As is shown in Figure 5.7, the recently accessed blocks are maintained in an unified LRU stack, simplified as *uniLRUstack*. These blocks could be cached in any level of buffer caches, or even not cached⁴. For each level of buffer cache there is a yardstick block in

⁴In a protocol implementation, only some metadata, such as a block identifier and two statuses used in the ULC protocol, are stored in the stack for each block, not the block itself.

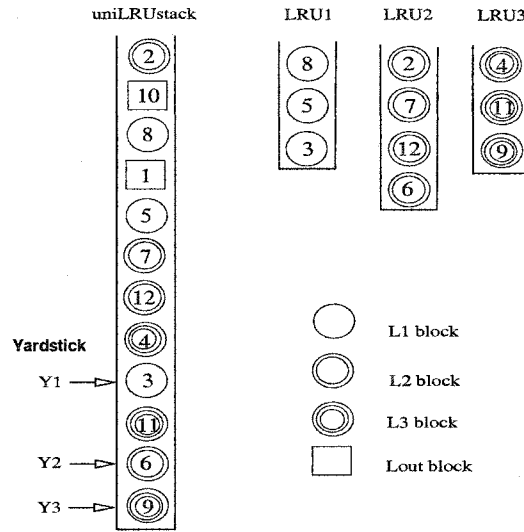


Figure 5.7: An example to show the data structure of ULC for a 3-level hierarchy. The blocks with their recencies less than that of yardstick Y_3 are kept in *uniLRUstack*. The level status (L_1 , L_2 or L_3) of a block is determined by its position between two yardsticks where it was accessed last time. Its recency status (R_1 , R_2 or R_3) is determined by its position between two yardsticks where it sits currently. To decide which block should be replaced in each level, the blocks in the same level can be viewed to be organized in a separate LRU stack (LRU_1 , LRU_2 , or LRU_3), and the bottom block is for replacement.

uniLRUstack, which is the block cached in that level of cache and has the maximal recency among blocks cached there. We call them Y_1 , Y_2 , ..., Y_n for level L_1 , L_2 , ..., L_n cache, respectively. The size of *uniLRUstack* actually is determined by the position of Y_n , the last yardstick, which always sits in the bottom of *uniLRUstack*. Any blocks with recencies larger than that of Y_n will be removed from *uniLRUstack* and become L_{out} blocks, which are not cached in any level of caches. Only when a block gets accessed with the recency between the recencies of Y_{i-1} and Y_i does the block become L_i block, which means it will be cached in the level L_i cache. All of blocks cached on the same level can be viewed as a local LRU stack, called LRU_i , where the order of blocks is determined by their recencies in *uniLRUstack* and its size does not exceed the size of that level of cache. The block to be replaced on level L_i is the bottom block of stack LRU_i . For the requested blocks that

are neither cached in L_1 cache nor going to be cached there because their LRDs are larger than the recency of Y_1 , we set up a small LRU stack called *tempLRU* to temporarily store these blocks, so that they can be quickly replaced from the L_1 cache.

There are two structures for the buffer cache hierarchy. One is the single-client structure, in which there is only one client connected to one server⁵, and another is the multi-client structure, in which more than one clients share the same server, and blocks requested by different clients are shared in the server. There are two additional challenges for the multi-client ULC protocol: (1) How to cache shared blocks in server buffer caches, which could carry different level tags set by different clients. (2) How to allocate server cache buffers to different clients.

5.3.2.1 The Single-client ULC Protocol

The single-client ULC algorithm runs at a client, which holds the first level cache. It has the knowledge of the size of the buffer cache on each level. For each block in *uniLRU stack*, there are two associated statuses: level status and recency status. Level status indicates at which level the block is cached, such as L_1 , L_2 , ..., L_n , or L_{out} . When a block gets accessed, we need to know its recency to determine its level status. The recency is actually its LRD. It takes at least $O(N)$ time to maintain the exact recency information for all blocks, where N is the aggregate size of the buffer caches. Actually we only need to know the recencies of whatever two yardsticks the recency lies in. Thus we maintain a recency status R_i for each block, which means its recency is between the recencies of yardsticks Y_{i-1} and Y_i (or

⁵Here we call the high level buffer cache, client, and low buffer cache, server, when we discuss two adjacent levels.

just less than Y_i if i is 1). The cost to maintain recency statuses is $O(1)$, which will be explained.

Initially, if level L_i is not full and the levels that are higher than it are full, any requested L_{out} blocks get level status L_i and reside in level L_i . If all the caches are full, any blocks accessed when they are not in *uniLRUstack* are given level status L_{out} . There are two circumstances for a block to be outside *uniLRUstack*. One is that the block is accessed for the first time, another is that block has not been accessed for a long period of time so that it leaves *uniLRUstack* from the bottom. For these blocks their level status is L_{out} , and recency status is R_{out} .

We define an operation for yardsticks in *uniLRUstack* called *YardStickAdjustment*, which moves a yardstick from the current yardstick block with level status L_i in the direction towards the stack top to the next block with level status L_i . All the blocks it passes including the current yardstick block change their recency status from R_i to R_{i+1} . When a yardstick block changes its position in *uniLRUstack*, we need to conduct yardstick adjustment to ensure the yardstick is on the block with correct recency status and with the largest recency among the blocks on the level. Demoting a block into a low level cache is equivalent to moving the bottom block of local stack LRU_i into LRU_{i+1} , which is sorted on their recencies in *uniLRUstack*. To place the block at the correct recency position in LRU_{i+1} , we define another operation for a demoted block called *DemotionSearching*, which searches in the direction towards the stack bottom in *uniLRUstack* for next block with a higher level status.

There are two types of requests in ULC, which are sent from the client to the low level caches to coordinate various levels of caches to work under an unified caching algorithm.

1. Retrieve(b, i, j) ($i \geq j$): retrieve block b from level L_i , and cache it on level L_j when it passes level L_j on its way to level L_1 .
2. Demote(b, i, j) ($i < j$): demote block b from level L_i into level L_j .

If there is a reference to block b with level status L_i and recency status R_j , there are only two cases we need to deal with: $i = j$ and $i > j$. The case $i < j$ is not possible because block b is demoted to level L_{i+1} before j is larger than i . When block b is referenced, it is moved to the top of *uniLRUstack* and its recency status becomes R_1 . This also makes it stay in the top of stack LRU_i . If $i > 1$, block b goes to stack *tempLRU* in the client and is going to be replaced soon from the client cache. Then for each of the two cases, we act as follows: (1) $i = j$. Block b remains in its current level of cache with the same level status (Retrieve(b, i, i)). (2) $i > j$. Because block b will be moved from level L_i and cached at level L_j (Retrieve(b, i, j)), a space needs to be freed at level L_j . We demote the yardstick block Y_j to its next low level cache, whose yardstick block may have to be demoted in turn if its status level is higher than L_i . Yardstick adjustment and demotion searching are conducted here.

5.3.2.2 The Multi-client ULC Protocol

When there are multiple clients sharing one server, the cache buffers in the server are no longer solely used by one client. In the single client ULC protocol, the number of the blocks with level status L_i (also called L_i blocks), or the size of stack LRU_i , is determined by the size of level L_i cache. If the buffers at level L_i are shared by multiple clients, an allocation policy is needed on level L_i for the performance of the entire system. To

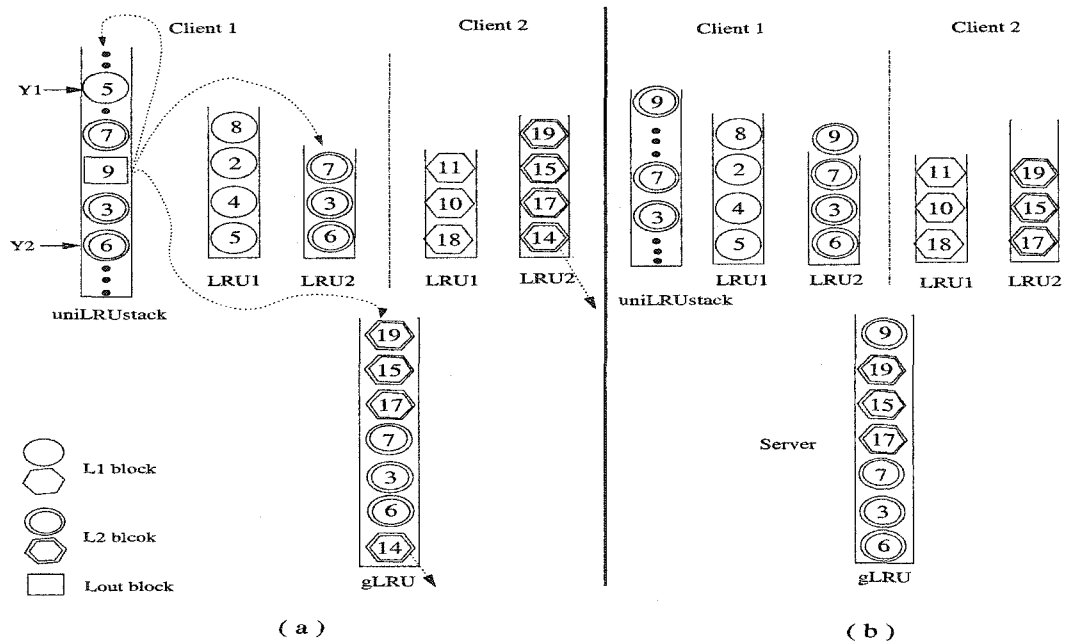


Figure 5.8: An example to explain how a requested block is cached in the server cache, and how the allocation scheme adjusts the size of the server cache used by various clients in a multi-client two-level caching structure. Originally in (a) server stack *gLRU* holds all the L_2 blocks from clients 1 and 2, which are also in their LRU_2 stacks, respectively. Then block 9 is accessed in client 1. Because block 9 is between yardstick Y_1 and Y_2 in its *uniLRUstack*, it turns into L_2 block and needs to be cached in the server. Because the server cache is full, the bottom block of *gLRU*, block 14, is replaced, which will be notified to its owner, client 2, through a piggyback on the next retrieved block going to client 2 (delayed notification). After the server buffers re-allocation (b), the size of server cache for client 1 is increased by 1 and that for client 2 is decreased by 1. So the clients and the server cooperate to make the server cache efficiently allocated with the aim of high performance for the entire system.

obtain best performance, it is known that allocation should follow the dynamic partition principle: each client should be allocated a number of cache blocks that varies dynamically in accordance with its working set size. Experience has shown that global LRU performs well by approximating the dynamic partition principle [11]. Thus we use a global LRU stack called *gLRU* in the server to facilitate the allocation operation. The block order in *gLRU* is determined by the block recencies, which are determined by the timings of requests from clients requiring a block be cached in the server. The bottom block of *gLRU* is the one to be replaced when a free buffer is needed. For each block in *gLRU* we record its owner — the client most recently requesting the block be cached in this server. A block is cached on the highest level among all the clients' direction. If there is only one client, the bottom block of *gLRU* is always the yardstick block Y_i in *uniLRU stack*, and also is the bottom block of stack LRU_i in the client. Because the server cache buffer is shared among the clients, the bottom block of LRU_i could have been replaced in the server. If this is the case, it is equivalent to shrinking the cache size of the server dedicated to the client. So when a block is replaced from *gLRU*, a message is sent to its owner client so that a yardstick adjustment can occur there. Correspondingly, the size of LRU_i is decreased by one. The owner notifications of block replacements can be delayed until the next requested block is sent to its owner client without affecting its correctness. Then they are piggybacked on the next retrieved block, thus saving extra messages. Figure 5.8 shows an example to illustrate the multi-client case. By dynamically adjusting yardsticks of affected clients based on the information provided by the allocation policy, we have a ULC algorithm in clients allowing low level caches to change their sizes dynamically. The changing sizes are the results of the allocation policy with the aim of high performance for the entire system.

5.4 Performance Evaluation

This section presents our trace-driven simulation results. We compare ULC with two other multi-client caching schemes: independent LRU, simplified as *indLRU*, which is a commonly used scheme, and unified LRU, simplified as *uniLRU*, an LRU-based unified caching protocol[76].

5.4.1 Performance Metric

We use average block access time, T_{ave} , to evaluate the performance of various protocols. This metric measures the average time required to access a block perceived by applications. The access time is determined by the hit ratios and miss penalties at different levels of the caching hierarchy, as well as other communication costs. Generally, we can estimate T_{ave} for an n -level cache hierarchy as follows: $T_{ave} = \sum_{i=1}^n h_i T_i + h_{miss} T_m + T_{demotion}$ where h_i is the hit ratio at level L_i cache, T_i is the time it takes to access the cache at level L_i , h_{miss} is the miss ratio for the cache hierarchy (equivalent to $1 - \sum_{i=1}^n h_i$), T_m is the cost for the miss, and $T_{demotion}$ is the demotion cost for block placements required by an unified replacement protocol. If we assume the demotion cost for a block from level L_i to L_{i+1} is T_{di} , and the demotion rate between level L_i and L_{i+1} is h_{di} , then $T_{demotion} = \sum_{i=1}^{n-1} T_{di} h_{di}$. We do not consider the situation where demotions are delayed, thus their costs could be hidden from applications, for two reasons: (1) Demotions are highly possible to occur in a bursting fashion, especially for an LRU-based unified replacement, where 50%, even around 90% of the references incur demotions. A small number of dedicated buffers have difficulty in buffering the delayed blocks, thus its performance is unpredictable. (2) Reserving a large

number of buffers for delayed demotions actually reduces the cache size and would hurt the hit ratios.

Specifically, for a two-level client-server cache hierarchy, the average access time is as follows: $T_{ave} = h_c T_c + h_s T_s + (1 - h_c - h_s) T_m + h_{c-s} T_{c-s}$ where h_c and h_s are the hit ratios for the client and server respectively, T_c and T_s are the costs for a hit in the client and server respectively, and T_m is the cost for a miss in the server. If the disk access time for a block is T_d , T_m can be regarded as $T_s + T_d$, h_{c-s} is the demotion rate between the client and the server. T_{c-s} is the cost for a demotion. We assume $T_c \approx 0$, the demotion cost T_{c-s} is approximated as the server hit time T_s . Then $T_{ave} \approx h_s T_s + (1 - h_c - h_s) T_d + h_{c-s} T_s$.

5.4.2 Simulation Environment

We use trace-driven simulation for the evaluation. Our simulator tracks the statuses of all accessed blocks, monitors the requests and hits seen at each cache level, and the demotions at each level boundary. We assume 8 KB cache block. We use seven large-scale traces to drive the simulator, including two synthetic traces: *random* and *zipf* and five other real-life workload traces. We have described the two synthetic traces in Chapter 5.2. Here we significantly increase the scale of these two traces: *random* accesses 65536 unique blocks with a 512MB data set. It contains about 65M block references. *zipf* accesses 98304 unique blocks with a 768MB data set. It contains about 98M block references. The three real-life traces used for the single-client simulation are described as follows:

1. **httpd** was collected on a 7-node parallel web-server for 24 hours. [71]. The size of the data set served was 524 MB which is stored in 13,457 files. A total of about 1.5M HTTP requests are served, delivering over 36 GB of data. We aggregate the seven

request streams into a single stream in the order of the request times for the single client structure study.

2. **dev1** is an I/O trace collected over 15 consecutive days on a Redhat Linux 6.2 desktop [13]. It contains text editor, compiler, IDE, browser, email, and desktop environment usage. It has around 100K references. The size of the data set it accessed is around 600M.
3. **tpcc1** is also an I/O trace collected while running the TPC-C database benchmark with 20 warehouses on Postgres 7.1.2 with Redhat Linux 7.1[13]. It has around 3.9M references. The data set size is around 256M.

We also select three traces for multi-client simulation. One of them is the original *httpd* trace with seven access streams, each for one client. The other two multi-client traces are as follows:

1. **openmail** was collected on a production e-mail system running the HP OpenMail application for 25,700 users, 9,800 of whom were active during the hour-long trace [76]. The system has 6 HP 9000 K580 servers running HP-UX 10.20. The size of the data set accessed by all six clients is 18.6G.
2. **db2** was collected by an 8 node IBM SP2 system running an IBM DB2 database that performed join, set and aggregation operations for 7,688 seconds [71]. The total data set size is 5.2GB and it is stored in 831 files.

For all the simulation experiments, we use the first one tenth of block references in the traces to warm the system before the measurements were collected.

5.4.3 Comparisons of Multi-level Schemes in a Three-level Structure

To demonstrate the ability of multi-level caching schemes (ULC, indLRU, and uniLRU) to make distinctions of locality strengths as well as the ability to keep their stability, we test them in a three-level caching hierarchy for the five single client traces, simulating a scenario where the block transfer route consists of a client, a server and its disk array containing a large RAM cache. For a common local network environment, we assume the cost to transfer an 8KB block between the client and the server through LAN is 1ms, the cost between the server and the RAM cache in the disk array through SAN is 0.2ms, and the cost of a block from a disk into its cache is 10ms [76]. We assume the cache sizes of the client, the server, and the disk array are 100MB each for traces *random*, *zipf*, *httpd*, and *dev1*, and the cache sizes are 50MB each for trace *tpcc1* due to its comparatively small data set. We report the hit ratios in each of the three levels, demotion rates on each boundary, and average access time for each workload with the three multi-level caching schemes in Figure 5.9.

Confirming the experimental results in [76], we observe that there are significant performance improvements of uniLRU over indLRU for all the five traces, from 17% to 80% reduction on average access time (see the third graph). Actually these are the results of two combined effects of uniLRU: (1) increasing the cache hit ratios; (2) generating additional demotion cost. UniLRU eliminates the redundancy in the hierarchy, making the low levels of caches contribute to the hit ratio just as if they stayed in the first level. For example, in a random access pattern, the contribution of a cache to the hit ratio should be proportional to its size. However, the second and third levels of caches gain much lower hit ratios (1.7% and 0.3% respectively) than that of first level cache (19.5%) for trace *random* in indLRU

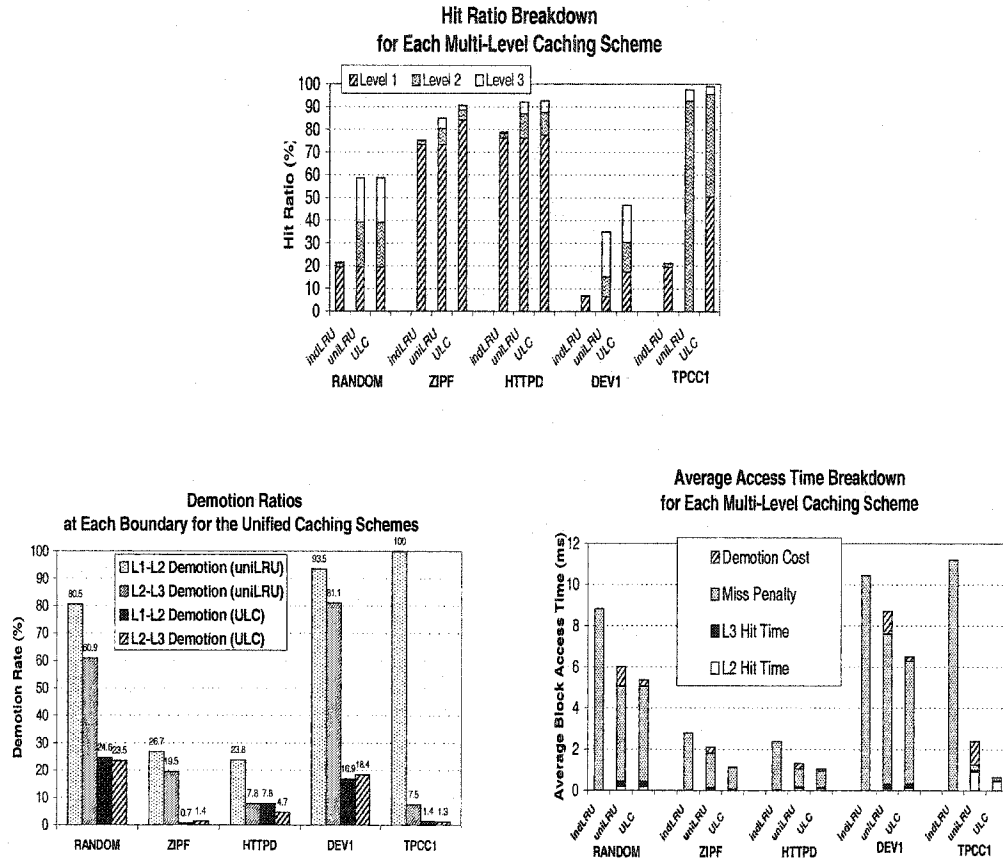


Figure 5.9: hit ratios in each of the three levels, demotion rates at each of two boundaries (between L1 and L2, and between L2 and L3 cache), and average access time for each workload with the multi-level caching schemes indLRU, uniLRU and ULC.

(see the first graph). The unified replacement scheme uniLRU makes the low levels of caches much better utilized. Their hit ratios (19.6% and 19.5% respectively) are almost the same as that of first level cache (19.5%). However this improvement comes with a considerably high price: high demotion rates. For example, in trace *random* uniLRU has a first boundary demotion rate 80.5%, which means 80.5% of block references accompany “write-backs” to the server. Furthermore, it has a 60.9% demotion rate at the second boundary (see the

second graph). The worst case for the demotion rates of uniLRU is trace *tpcc1*, which has a looping access pattern. Its first boundary demotion rate is 100%! This is because uniLRU has little power to predict the level where an accessed block will be accessed. For a looping access pattern, blocks are accessed at a large recency equal to the loop distance, which implies almost all the blocks of *tpcc1* are accessed after they are demoted into the second level of cache. So the hit ratio of the second level cache is very high (92.5%) and 44.7% of the average access time is spent on the demotion. According to the requirement on the ability of distinguishing locality strengths for a multi-level caching scheme, the distribution that the level L_1 hit ratio (0.03%) is much less than the L_2 hit ratio (92.5%) under uniLRU shows a bad case.

Compared with uniLRU, ULC protocol has an access-time-aware hit ratio distribution along the levels of caches: more hits appearing on upper levels. For example, the hit ratios of the level L_1 , L_2 , and L_3 are 50.3%, 45.1%, and 3.4%, respectively for trace *tpcc1*. And such a distribution is achieved without paying high costs of demotions. For example, the two boundary demotion rates of *tpcc1* are 1.4% and 1.3%, respectively (see the second graph). It is also shown that ULC has significant demotion rate reductions over uniLRU for all 5 traces. This explains why the proportion of demotion cost in the average access time for ULC is much smaller (from 1% to 8.3% with an average of 4.1%) than that for uniLRU (from 12.6% to 44.7% with an average of 21.5%) (see the third graph).

The access time breakdowns also show that ULC still performs significantly better than uniLRU except for trace *random*, even if we assume the demotions could be moved off the critical path for response time. Actually this is an unrealistic assumption. The experiments on the client-server system running a TPC-C benchmark show that demotions can signif-

icantly delay the network and lower the system throughput [14]. In summary, our ULC achieves from 11% to 71% reduction on average access time with an average of 34.6% over that of uniLRU.

5.4.4 The Performance Implication of System Parameters

To be widely applicable, a caching scheme should consistently deliver improved performance over existing schemes with a large range of system parameters such as cache size and network bandwidth. For the convenience of observing and comparing performance differences of the schemes in this study, we choose the client-server structure, a two-level cache hierarchy to present our results. For the two-level hierarchy evaluation, we include Multi-Queue (MQ). In a client-server caching hierarchy, the environment that MQ is designed for, we use MQ in the server and use LRU in the client independently. There is a parameter in the MQ replacement, called *lifeTime*, which determines the speed to decay the frequency of an in-accessed block. Because this parameter is workload dependent, we run each trace for multiple sample *lifeTime* values in the range suggested in [79], and report the best results of these runs. For this client-server structure, we set the time to retrieve an 8K block from the server, T_s , as 0.4 ms, and the average disk access time, T_d , for an 8K block is 10 ms. Due to the space constraints we only report the results for one synthetic trace, *zipf*, and two real-life traces, *httpd* and *dev1*. The results for other traces are consistent with those presented here.

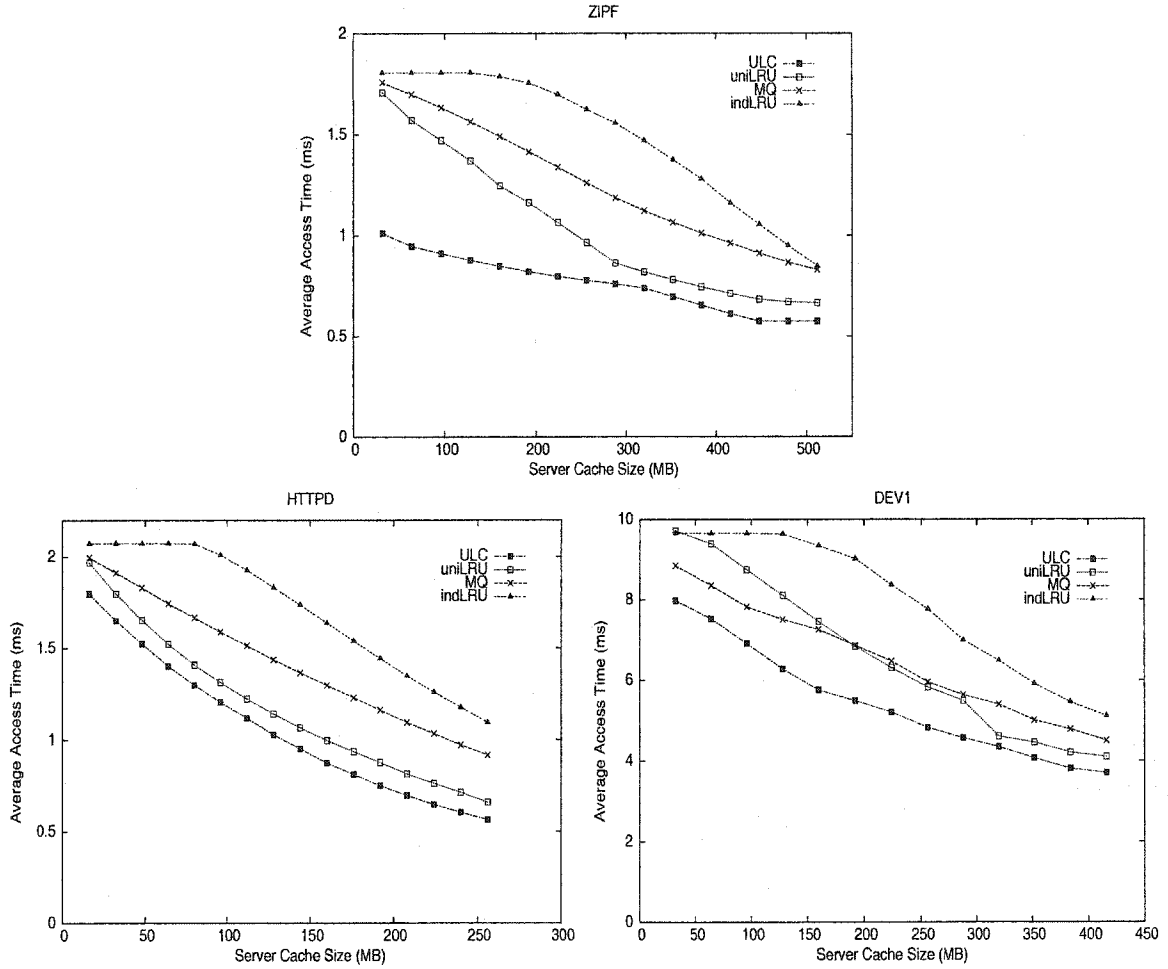


Figure 5.10: The average access times for schemes ULC, uniLRU, MQ and indLRU with various server cache sizes. The client cache size is fixed. It is 256MB for *zipf*, and 128MB for *httpd* and *dev1*.

5.4.4.1 The Impact of Server Cache Size

Figure 5.10 shows the average access time for each workload as the server cache size changes for all the four caching schemes: ULC, uniLRU, MQ, and indLRU. An observation for the indLRU hit ratio curves is that there is a segment of flat curve for each workload with small server cache sizes. These curves start to drop when the server cache sizes approach the client cache size. This demonstrates the serious under-utilization of the server cache under indLRU due to the redundancy and locality filtering effect. That is, under indLRU

a relatively small server cache unfortunately has little contribution to the reduction of the average access time in a system with a large client cache size. This is consistent with the study in [84], which suggests increasingly large built-in disk cache help little with a comparatively large file system buffer cache under two independent LRU replacements. However, such an observation does not exist for all the other three schemes, which achieve better performance than indLRU for all the workloads.

It is shown that for most of the cases, the performance of uniLRU is better than that of MQ, though MQ does not have demotion costs. This reflects the merit of unified caching scheme – elimination of data redundancy. It is also shown that the performance gains of uniLRU over MQ are increased with the increase of server cache size. Our study shows that this is because MQ relies more on the reference frequencies to make replacement decision when the cache size becomes large. Thus MQ becomes less responsive to react to the changing access patterns, and less effective than LRU-based schemes with large server cache sizes. For all the traces ULC achieves the best performance, steadily decreasing the access time with the increase of server cache sizes. Its high hit ratios and low demotion rates are the two major contribution factors.

5.4.4.2 The Impact of Client Cache Size

Figure 5.11 shows the average access time for each workload as the client cache size changes. It is shown that uniLRU benefits much more from the added client cache size than indLRU and MQ. This is because increasing client size has negative effects for indLRU and MQ: more data redundancies in indLRU and weaker locality available for MQ in the server. An unified caching scheme is immune from these effects. However, the performance of uniLRU

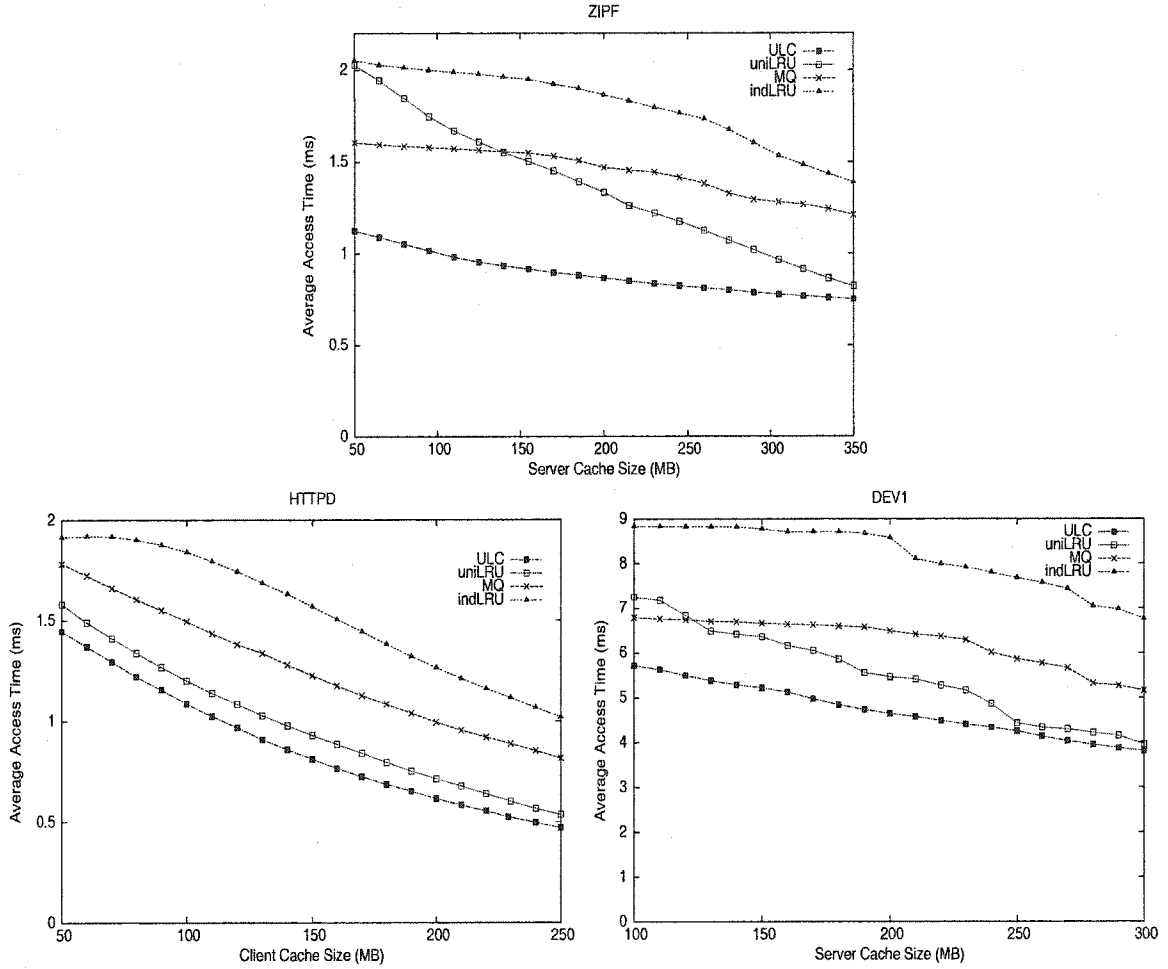


Figure 5.11: The average access times for schemes ULC, uniLRU, MQ and indLRU with various client cache sizes. The server cache size is fixed. It is 200MB for *zipf* and *dev1*, and 150MB for *httpd*.

is worse than that of MQ with small client cache sizes for *zipf* and *dev1*. Here is the explanation. The smaller the client cache size is, the more requested blocks are retrieved from outside of the client. In uniLRU every block brought from outside of the client incurs a demotion. Small client caches cause large demotion costs, which increase the access time in uniLRU. Though ULC is also an unified caching scheme, it maintains its best performance in the whole range of client cache sizes because of its accurate block placement decisions.

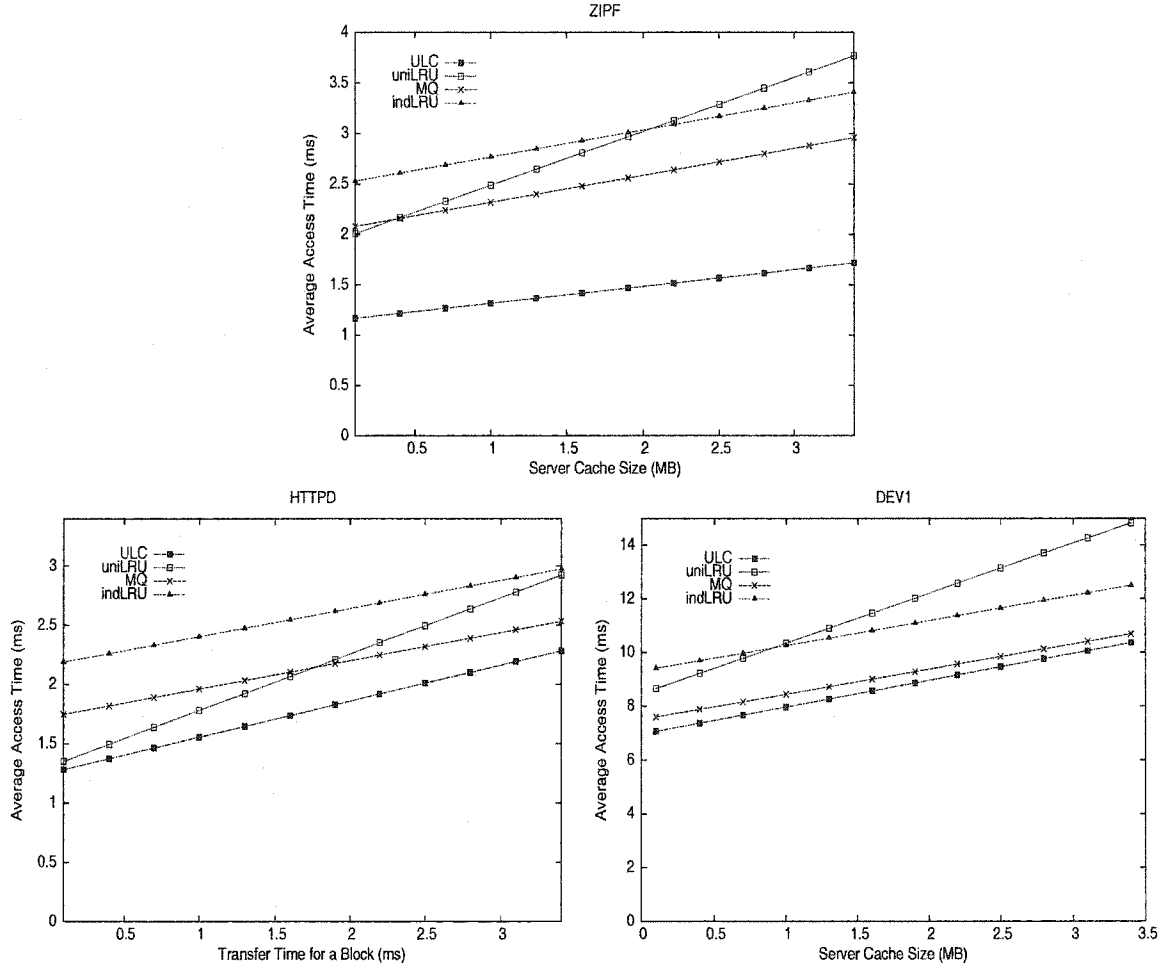


Figure 5.12: The average access times for schemes ULC, uniLRU, MQ and indLRU with various block transfer times. The client and server cache sizes are fixed, and are 100MB each for all the workloads.

5.4.4.3 The Impact of Network Bandwidth

Figure 5.12 shows the average access time for each workload as we change the 8KB block transfer time. It is expected that the increase of transfer time has a more seriously negative effect for unified schemes than for independent schemes, because the former have the additional demotion costs determined by the transfer time. We see the average access time of uniLRU does increase more quickly than those of indLRU and MQ. However, with low demotion rates, ULC have the similar impact from the increase of transfer time as indLRU

and MQ do, even less impact for trace *zipf* because of the contribution of transfer time to the miss penalty and its much reduced miss ratios.

5.4.5 Comparisons of Caching Schemes for Multi-client Workloads

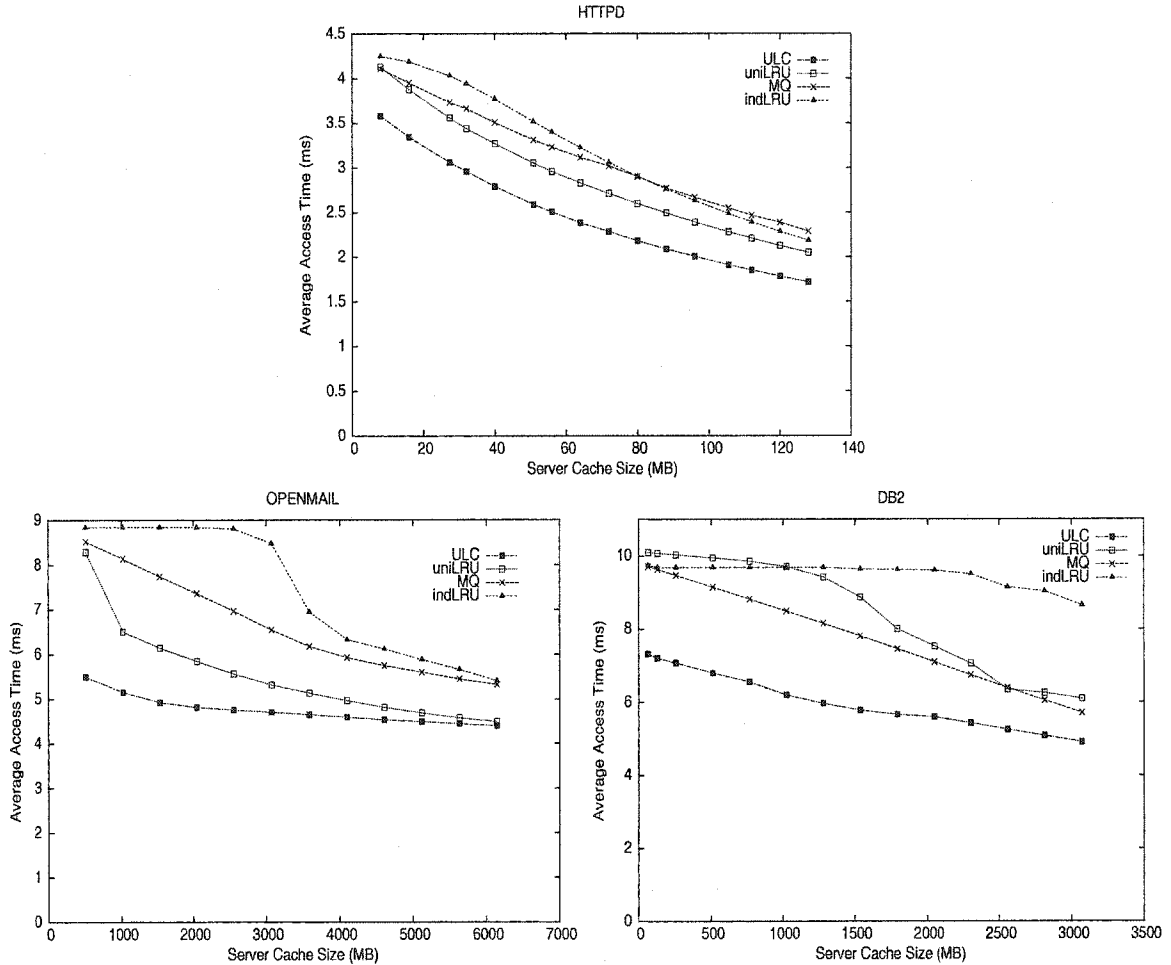


Figure 5.13: The average access times of multi-client traces *httpd*, *openmail*, and *db2* with various server cache sizes. Among them *httpd* is with 7 clients, *openmail* is with 6 clients, and *db2* is with 8 clients. Each client contains 8MB, 1GB, or 256MB respectively.

Because the performance of uniLRU scheme can significantly deteriorate due to buffer competition and data sharing among clients for the multi-client structure, Wong and Wilkes also proposed two adaptive cache insertion policies to supplement their primitive scheme

[76]. Among their three multi-client traces *httpd*, *openmail*, and *db2*, *httpd* is the one with data sharing. While they did not state which version of their unified LRU schemes should be used for a specific workload, we ran all the versions and report the best results for comparisons.

Figure 5.13 shows that for all the workloads ULC achieves the best performance. For most of the time, indLRU has the worst performance. However, there are two cases where indLRU beats uniLRU or MQ. One case is MQ with large server cache sizes for trace *httpd*. When server cache sizes become large enough, LRU's inability of dealing with weak locality becomes less destructive. However, as a frequency-based replacement, MQ's shortcoming of slowness to respond to pattern changes becomes obtrusive. Another case is uniLRU with small cache sizes for trace *db2*. This is because *db2* contains looping access patterns. LRU is not effective on a workload with this pattern until all blocks in the looping scopes can be held in the cache. Carefully examining detailed experiment reports indicates that both indLRU and uniLRU achieve very low hit ratios (6.9% and 7.9%, respectively for the two levels of caches, compared with that of MQ (12.3%) and that of ULC (35.1%). Thus it is the large demotion cost of uniLRU (with an average demotion rate 88.6% for the 8 clients, compared with that of ULC (7.2%)) that makes the difference. With the increase of the cache size, some looping scopes are covered by the combined two-level caches, but not by a single level, which explains why the performance of uniLRU starts exceeding that of indLRU when the server cache size reaches 1GB. However, the performance of uniLRU is worse than that of MQ because of its looping access pattern. For the traces *httpd* and *openmail*, uniLRU beats MQ by eliminating data redundancy.

5.5 Related Work and Discussions

Addressing the challenges of replacements in buffer caching hierarchy, researchers have mainly tried these two approaches: (1) re-designing low level cache replacement; (2) extending existing replacement into an unified hierarchy replacement through coordination. The MQ algorithm [82] is a representative of the first approach. However, without the coordination with clients, the performance potential of MQ is significantly constrained. Since LRU is commonly used in software managed buffer caches due to their simplicity and adaptability, Wong and Wilkes [76] propose a protocol to integrate two-level buffer cache hierarchy into a single, large unified cache based on “demotion” operations, and manage it using LRU. Their goal is to effectively utilize the built-in cache in RAID, so the network they assumed is high speed SAN. To reduce the possible network bottleneck caused by demotions in a database client and storage server system, Chen et al [14] even proposed to re-load evicted blocks from disks rather than from clients. Our technique deals with the reduction of demotions by effectively utilizing history access patterns.

Jiang and Zhang [33] propose the LIRS replacement algorithm to address the performance degradation of LRU on workloads with weak localities. They use a LIRS stack to track the recencies of accessed blocks. The blocks with small recencies at which they get accessed, are kept in the cache. This single-level cache replacement motivates us to investigate if the last locality distance, LLD, can be effectively used to exploit hierarchical locality, so that blocks with different locality strengths can be arranged into correct cache levels.

The work on cooperative caching [23, 66, 74] is to coordinate the buffer caches of many client machines distributed on a LAN to form a fourth level in the network file system’s

cache hierarchy. Besides local cache, server cache, server disk, data can also be cached in another client's cache. Some associated issues include idle cache availability, consistent sharing. Our ULC protocol is intended for the conventional file buffer cache hierarchy, while the characterization of non-uniform locality is expected to enhance the effectiveness of data placements in the cooperative caching.

As far as the cache hierarchy between processor and memory is considered, the interaction of replacements at various levels and its performance implication do not pose a problem. Multi-level inclusivity between L_1 , L_2 , .. L_n cache could be accepted as a principle to simplify the cache coherence protocol [3]. This is because a lower level cache is more than ten times larger than its upper level cache. With this large difference, the size reductions of useful caches due to data redundancy have only limited negative performance impact on the low level caches. In contrast, the sizes of buffer caches in the hierarchy do not follow this regularity: a client buffer cache could even be larger than the second level cache.

We assume ULC works in a trusted environment. Though it is a client-directed protocol, ULC does not increase the vulnerability of servers. This is because even with independent caching schemes, a client still has the opportunity to abuse server buffers by sending extra requests to servers to keep its blocks in the server.

The underlying algorithms on almost all the existing file systems are LRU or its variants. ULC basically inherits their data structure – LRU stack. The operation costs associated with the stacks are $O(1)$ time with each reference request. Regarding space cost used for the stacks, we need 17 bytes (8 bytes for file identifier and block offset, 8 bytes for two pointers in a double linked list, and 1 byte for statuses) for a block in the client, which only represents 0.2% of an 8 Kbytes block. The metadata in the shared server cache needs additional one

or two bytes for recording block owner. The stack sizes on other levels except the first one are determined by their cache sizes. Thus a server with a 1GB cache only uses 2.2MB for its metadata. The first level cache has to hold *uniLRUstack*, whose actual size is determined by the working set size of applications running on the client. The relatively cold blocks (with low level statuses) can be trimmed from the stack without compromising the ULC locality distinction ability if needed to save space cost. For example, an 8.5MB metadata in the client can support a workload working set up to 4GB. This is highly affordable in a system endeavoring for improved file I/O performance through large caches.

5.6 Summary

An effective caching scheme for multi-level cache hierarchy is important to the performance of applications because increasingly more applications rely on the hierarchy for their file accesses. After carefully investigating the non-uniform locality strength quantifications in the representative file access patterns, we propose the ULC caching protocol. Compared with the commonly used independent LRU scheme and the other recently proposed schemes, the ULC protocol shows its distinguished performance merits: (1) It consistently and significantly reduce average block access time perceived by applications; (2) It can be implemented efficiently with $O(1)$ time complexity with only a few stack operations associated with a reference.

Chapter 6

Conclusions and Future Work

This dissertation has provided solutions to address several important memory management issues aiming at reducing disk accesses: general-purpose replacement algorithms, virtual memory replacement policies, thrashing prevention, and multi-level buffer cache management. The proposed solutions are based on the extensive application behavior characterization and accurate access locality quantification. These solutions cover both process virtual memory accesses and file data accesses in program execution, both page replacement for a single program and for multiple running programs, and both buffer cache in a single computer and multi-level buffer caches in a distributed system. Each proposed algorithm or scheme has been extensively evaluated using either driven-driven simulations or system implementations to demonstrate its effectiveness and practical value. Using the techniques together will comprehensively enhance the system performance with memory-intensive and I/O-intensive applications, and make the system more robust in face of dynamical memory accesses.

6.1 General-Purpose Replacement Algorithms

LRU replacement algorithm has been commonly used in file systems, data base systems, storage systems and in numerous other applications. It has been successful in general due to its good performance in most cases and its low cost. However, it is also well known for LRU to be incapable with the access patterns of weak locality such as scan and loop accesses. Because of its important role in today's computing, the negative implication of LRU's performance inability cases would be high. Thus, it is understandable that there are so much research work still focusing on improving LRU performance.

Motivated by the limitations of previous studies, we propose the Low Inter-reference Recency Set (LIRS) replacement policy. LIRS effectively addresses the limitations of LRU by using recency to evaluate Inter-Reference Recency (IRR) for making a replacement decision. This is in contrast to what LRU does: directly using recency to predict next reference timing. LIRS dynamically and responsively distinguishes low IRR (LIR) blocks from high IRR (HIR) blocks, and keep LIR blocks in cache. Compared with LRU, LIRS does not directly use recency to make a replacement decision, but uses it to determine LIR or HIR status of a block. At the same time, LIRS almost retains the same simple assumption of LRU to predict future access behavior of blocks. The only additional assumption of LIRS is there is correlation between consecutive IRRs of a block. It also does not rely on any detectable regularities.

Performance evaluations with a variety of traces and a wide range of cache sizes show that LIRS effectively addresses the limitations of LRU, retains the low-cost merit of LRU, and outperforms those replacement policies relying on the access regularity detections.

In the dissertation, we follow the convention of the replacement algorithm study to use only hit ratio as a performance metric. However, in practical systems the correlation between the hit ratio increase and application performance improvement is more complicated than being linear. For example, a number of misses on a set of pages consecutively residing on the disk would cause a penalty of almost the same of a single miss because of the property of hard disks. On the other hand, the same number of misses on a set of pages scattered on the disk will cost much more than the sequential case. Taking the ultimate application performance into consideration could significantly affect the design and evaluation of replacement algorithms. As a future work, we will use the more performance-relevant metrics such as average block access time in our replacement algorithm study. We expect this will make the research in the area play a more important role in the system design.

6.2 Low Cost Virtual Memory Replacement Algorithms

The low cost requirement of virtual memory (VM) management make the research of low-cost approximations of general-purpose replacement algorithms a necessity. However, this is not easy considering that only very limited history access information can be used to maintain a low cost. This explains why the CLOCK, a replacement policy developed at least 35 years ago, still dominates almost all the today's systems.

While pure LRU has an unaffordable cost in VM, CLOCK simulates LRU replacement algorithm with a low cost acceptable in VM management. Over the last three decades, the inability of LRU as well as CLOCK to handle weak locality accesses is getting serious, and an effective fix on it becomes increasingly demanding. However, almost all the major

improved replacement algorithms are built on LRU and have a cost at least equivalent to the LRU cost.

Inspired by the general-purpose replacement algorithm, LIRS [33], we propose an enhanced CLOCK replacement policy, called CLOCK-Pro. By additionally keeping track of a limited number of replaced pages, CLOCK-Pro works in a similar fashion as CLOCK with a VM-affordable cost. In the meanwhile, it brings all the much-needed performance advantages from LIRS into CLOCK. CLOCK-Pro also eliminates the only tunable parameter in LIRS and makes itself a policy adapting to the changing access locality to serve a broad spectrum of workloads. Extensive simulation experiments on real-life I/O and VM traces show significant and consistent performance improvements. We believe that CLOCK-Pro would be very attractive to the VM system designers in industry.

The potential performance advantages of CLOCK-Pro can only be fully demonstrated through real system implementation. As a future work, we plan to continue our efforts to make CLOCK-Pro practically and efficiently run on Linux systems, where some system-specific issues will arise, such as how to keep track of the replaced pages in memory, how to coordinate replacement decisions with individual process access behaviors in a global memory replacement policy. Because the memory management is a complicated portion in an operating system and the replacement codes are heavily coupled with other parts of the memory management in Linux, there will be a number of technical challenges to be addressed. Our objective is to make CLOCK-Pro be widely used in the main stream operating systems in both commercial and open source communities.

6.3 Thrashing Prevention

Operating system designers attempt to keep high CPU utilization by maintaining an optimal multiprogramming level (MPL). Although running more processes makes it less likely to leave the CPU idle, too many processes adversely incur serious memory competition, and even introduce thrashing, which eventually lowers CPU utilization. A common practice to address the problem is to lower the MPL with the aid of process swapping out/in operations. This approach is expensive and is only used when the system begins serious thrashing. The objective of our study is to provide highly responsive and cost-effective thrashing protection by adaptively conducting priority page replacement in a timely manner.

We have designed a dynamic system Thrashing Protection Facility (TPF) in the system kernel. Once TPF detects system thrashing, one of the active processes will be identified for protection. The identified process will have a short period of privilege in which it does not contribute its LRU pages for removal so that the process can quickly establish its working set, improving the CPU utilization. With the support of TPF, thrashing can be eliminated in its early stage by adaptive page replacement, so that process swapping will be avoided or delayed until it is truly necessary.

We have implemented TPF in a Linux kernel. Compared with the original Linux page replacement, We show that TPF consistently and significantly reduces page faults and the execution time of each individual job in several groups of interacting SPEC2000 programs. We also show that TPF introduces little additional overhead to program executions, and its implementation in Linux (or Unix) systems is easy.

6.4 Multi-Level Buffer Cache Management

In a large client/server cluster system, file blocks are cached in a multi-level storage hierarchy: client buffer caches, multiple server buffer caches, and built-in caches of disks at the bottom level. Existing file block placement and replacement are either conducted at each level of the hierarchy independently, or by applying an LRU policy on more than one levels. One major limitation of these schemes is that hierarchical locality of file blocks with non-uniform strengths is ignored, resulting in many unnecessary block misses, or additional communication overhead, even when the aggregate size of multi-level buffer caches is sufficiently large to hold the working set. To address this limitation, we propose a client-directed, coordinated file block placement and replacement protocol, where the non-uniform strengths of locality are dynamically identified on the client level to direct servers on placing or replacing file blocks accordingly on different levels of the buffer caches. In other words, the locality of block accesses dynamically matches the caching layout of the blocks in the hierarchy. The effectiveness of our proposed protocol comes from achieving the following three goals: (1) The multi-level cache retains the same hit rate as that of a single level cache whose size equals to the aggregate size of multi-level caches. (2) The non-uniform locality strengths of blocks are fully exploited and ranked to fit into the physical multi-level caches. (3) The communication overheads between caches are also reduced.

Conducting simulations with a variety of synthetic and real-life traces, and with a wide range of system parameters, we show our caching protocol significantly and consistently outperforms existing multi-level caching schemes.

In the work of cooperative caching [23, 66, 74], there are schemes to coordinate the

buffer caches of many client machines distributed on a LAN to form a fourth level in the network file system's cache hierarchy. Because of the heterogeneity of the working sets of applications running on each client, possibly as well as the memory at each client, If we allow memory space sharing at the level of clients, we can reduce the workloads on servers and further increase the performance of applications with large working sets. One of the challenges with the design is that how to set the priority of memory allocations on a client to local applications and to applications running on other clients. We plan to look into these technical issues on cooperative caching in distributed environment.

Bibliography

- [1] A. V. AHO, P. J. DENNING, AND J. D. ULLMAN. Principles of optimal page replacement. *Journal of ACM*, 18(1):80–93, 1971.
- [2] G. ALMSI, C. CASCAVAL, AND D. A. PADUA. Calculating stack distances efficiently. In *Proceedings of the workshop on Memory system performance*, pages 37–43, 2004.
- [3] J.-L. BAER AND W.-H. WANG. On the inclusion properties for multi-level cache hierarchies. In *Proceedings of Annual International Symposium on Computer Architecture*, pages 73–80, 1988.
- [4] M. G. BAKER, J. H. HARTMAN, M. D. KUPFER, K.W. SHIRRIFF, AND J. K. OUSTERHOUT. Measurements of a distributed file system. In *Proceedings of Symposium on Operating System Principles*, pages 198–212, 1991.
- [5] R. BALASUBRAMONIAN, D. ALBONESI, A. BUYUKTOS, AND S. DWARKADAS. Dynamic memory hierarchy performance and energy optimization. In *Proceedings of Annual International Symposium on Computer Architecture*, pages 245–257, 2000.
- [6] S. BANSAL AND D. MODHA. CAR: Clock with adaptive replacement. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies*, 2004.
- [7] L. A. BELADY. A study of replacement algorithms for virtual storage. *IBM System Journal*, 5(2):78–101, 1966.
- [8] L. A. BELADY, R. A. NELSON, AND G. S. SHEDLER. An anomaly in space-time characteristics of certain programs running in a paging machine. *Communication of the ACM*, 12(6):349–353, 1969.
- [9] B. T. BENNETT AND V. J. KRUSKAL. Lru stack processing. *IBM Journal of Research and Development*, pages 353–357, 1975.
- [10] K. BEYLS AND E. H. D’HOLLANDER. Reuse distance-based cache hint selection. In *Proceedings of INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING*, pages 265–274, 2002.
- [11] P. CAO, E. W. FELTEN, AND K. LI. Application-controlled file caching policies. In *Proceedings of USENIX Summer Technical Conference*, pages 171–182, 1994.
- [12] R. W. CARR. *Virtual Memory Management*. UMI Research Press, 1984.

- [13] TRACE DISTRIBUTION CENTER. <http://tds.cs.byu.edu>. Brigham Young University.
- [14] Z. CHEN, Y. ZHOU, AND K. LI. Eviction-based placement for storage caches. In *Proceedings of Annual USENIX Technical Conference*, 2003.
- [15] T. M. CHILIMBI. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 191–202, 2001.
- [16] T. M. CHILIMBI. On the stability of temporal data reference profiles. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 2001.
- [17] T. M. CHILIMBI AND M. HIRZEL. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 199–209, 2002.
- [18] J. CHOI, S. NOH, S. MIN, AND Y. CHO. Towards application/file-level characterization of block references: A case for fine-grained buffer management. In *Proceedings of Annual USENIX Technical Conference*, pages 286–295, 2000.
- [19] J. CHOI, S. MIN S. NOH, AND Y. CHO. An implementation study of a detection-based adaptive block replacement scheme. In *Proceedings of Annual USENIX Technical Conference*, pages 239–252, 1999.
- [20] E. G. COFFMAN AND P. J. DENNING. *Operating Systems Theory*. Prentice-Hall, 1973.
- [21] F. J. CORBATOJ. A paging experiment with the Multics system. In *In Honor of Philip Morse, H. Feschbach, and U. Ingard*, pages 217–228. MIT Press, 1969.
- [22] STORAGE PERFORMANCE COUNCIL. *I/O Traces from a Popular Search Engine*. <http://www.storageperformance.org>.
- [23] M. D. DAHLIN, R. Y. WANG, T. E. ANDERSON, AND D. A. PATTERSON. Cooperative caching: Using remote client memory to improve file system performance. In *Symposium on Operating System Design and Implementation*, pages 267–280, 1994.
- [24] P. J. DENNING. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.
- [25] P. J. DENNING. Virtual memory. *Computer Survey*, 2(3):153–189, 1970.
- [26] P. J. DENNING. Working sets past and present. *IEEE Transactions Software Engineering*, 6(1):64–84, 1980.
- [27] P. J. DENNING. Before memory was virtual. In *The Beginning: Personal Recollections of Software Pioneers*. IEEE Press, 1997.
- [28] C. DING AND Y. ZHONG. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 245–257, 2003.

- [29] G. GLASS. Adaptive page replacement. Master's thesis, University of Wisconsin, 1997.
- [30] G. GLASS AND P. CAO. Adaptive page replacement based on memory reference behavior. In *Proceedings of ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, pages 115–126, 1997.
- [31] HP Corporation. *HP-UX 10.0 Memory Management White Paper*, 1995.
- [32] IBM Corporation. *AIX Versions 3.2 and 4 Performance Tuning Guide*, 1996.
- [33] S. JIANG AND X. ZHANG. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, pages 31–42, 2002.
- [34] S. JIANG AND X. ZHANG. TPF: a system thrashing protection facility. *Software - Practice and Experience*, 32(3):295–318, 2002.
- [35] S. JIANG AND X. ZHANG. Token-ordered LRU: An efficient page replacement policy and implementation for program interactions. In *Special Issue on Performance Modeling and Evaluation of High-Performance Parallel and Distributed Systems in Performance Evaluation: An International Journal*, 2004.
- [36] S. JIANG AND X. ZHANG. ULC: A file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches. In *Proceedings of International Conference on Distributed Computing Systems (ICDCS)*, pages 168–177, 2004.
- [37] T. JOHNSON AND D. SHASHA. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the International Conference on VLDB Surveys*, pages 439–450, 1994.
- [38] E. G. COFFMAN JR. AND T. A. RYAN. A study of storage partitioning using a mathematical model of locality. *Communications of the ACM*, 15(3):185–190, 1972.
- [39] S. F. KAPLAN, L. A. MCGEOCH, AND M. F. COLE. Adaptive caching for demand prepaging. In *Proceedings of the third International Symposium on Memory Management*, pages 114–126, 2002.
- [40] S. F. KAPLAN, Y. SMARAGDAKIS, AND P. R. WILSON. Trace reduction for virtual memory simulations. In *Proceedings of ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, pages 47–58, 1999.
- [41] L. J. KENAH AND S. F. BATE. *VAX/VMS Internals and Data Structures*. Digital Press, 1984.
- [42] J. KIM, J. CHOI, J. KIM, S. NOH, S. MIN, Y. CHO, AND C. KIM. A low-overhead, high-performance unified buffer management scheme that exploits sequential and looping references. In *Symposium on Operating System Design and Implementation*, 2000.

- [43] Y. H. KIM, M. D. HILL, AND D. A. WOOD. Implementing stack simulation for highly-associative memories. In *Proceedings of ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, pages 212–213, 1991.
- [44] E. D. LAZOWSKA AND J. M. KELSEY. Notes on tuning VAX/VMS. Technical report, Univ. of Washington, Dept. of Computer Science, 1978.
- [45] D. LEE, J. CHOI, J. KIM, S. NOH, S. MIN, Y. CHO, AND C. KIM. On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies. In *Proceedings of ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, pages 134–143, 1999.
- [46] R. L. MATTSON, J. GECSEI, D. R. SLUTZ, AND I. L. TRAIGER. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.
- [47] S. MAXWELL. *Linux Core Kernel Commentary*. CoriolisOpen Press, 1999.
- [48] S. MCFARLING. Cache replacement with dynamic exclusion. In *Proceedings of Annual International Symposium on Computer Architecture*, pages 191–200, 1992.
- [49] K. S. MCKINLEY AND O. TEMAM. Quantifying loop nest locality using spec’95 and the perfect benchmarks. In *ACM Transactions on Computer Systems*, pages 288–336, 1999.
- [50] M. K. MCKUSICK, K. BOSTIC, M. J. KARELS, AND J. S. QUARTERMAN. *The Design and Implementation of the 4.4 BSD Operating System*. Addison Wesley, 1996.
- [51] N. MEGIDDO AND D. MODHA. ARC: a self-tuning, low overhead replacement cache. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies*, 2003.
- [52] R. T. MILLS, A. STATHOPOULOS, AND D. NIKOLOPOULOS. Adapting to memory pressure from within scientific applications on multiprogrammed COWs. In *Proceedings of International Parallel and Distributed Processing Symposium*, 2004.
- [53] T. C. MOWRY, A. K. DEMKE, AND O. KRIEGER. Automatic compiler-inserted i/o prefetching for out-of-core application. In *Symposium on Operating System Design and Implementation*, pages 297–306, 1993.
- [54] D. MUNTZ AND P. HONEYMAN. Multi-level caching in distributed file system – or – your caching ain’t nuthin’ but trash. In *Proceeding of the USENIX Winter Technical Conference*, 1992.
- [55] V. F. NICOLA, A. DAN, AND D. M. DIAS. Analysis of the generalized Clock buffer replacement scheme for database transaction processing. In *Proceedings of ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, pages 35–46, 1992.
- [56] D. NIKOLOPOULOS. Malleable memory mapping: User-level control of memory bounds for effective program adaptation. dimitrios.s.nikolopoulos. In *Proceedings of International Parallel and Distributed Processing Symposium*, 2003.

- [57] E. J. O'NEIL, P. E. O'NEIL, AND G. WEIKUM. The lru-k page replacement algorithm for database disk buffering. In *Proceedings of ACM SIGMOD Conference*, pages 297–306, 1993.
- [58] C. N. PARKINSON. *Parkinson's Law or the Pursuit of Progress*. World Scientific Publishing Co, 1994.
- [59] R. H. PATTERSON, G. A. GIBSON, E. GINTING, D. STODOLSKY, AND J. ZELENKA. Informed prefetching and caching. In *Proceedings of Symposium on Operating System Principles*, pages 1–16, 1995.
- [60] J. L. PETERSON AND A. SILBERSCHATZ. *Operating System Concepts*. Addison Wesley, 1985.
- [61] F. PETRINI, D. KERBYSON, AND S. PAKIN. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ascii q. In *Proceedings of international Supercomputing 2003 Conference*, 2003.
- [62] V. PHALKE AND B. GOPINATH. An inter-reference gap model for temporal locality in program behavior. In *Proceedings of ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, pages 291–300, 1995.
- [63] B. G. PRIEVE AND R.S. FABRY. Min – an optimal variable-space page replacement algorithm. *ACM Press*, 19(5):295–297, 1976.
- [64] J. A. RIVERS, E. S. TAM, G. S. TYSON, E. S. DAVIDSON, AND M. FARRENS. Utilizing reuse information in data cache management. In *Proceedings of the ACM International Conference on Supercomputing*, 1998.
- [65] J. T. ROBINSON AND N. V. DEVARAKONDA. Data cache management using frequency-based replacement. pages 134–142. *Proceedings of ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, 1990.
- [66] P. SARKAR AND J. HARTMAN. Efficient cooperative caching using hints. In *Symposium on Operating System Design and Implementation*, 1996.
- [67] Y. SMARAGDAKIS, S. KAPLAN, AND P. WILSON. EELRU: simple and effective adaptive page replacement. In *Proceedings of ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, pages 122–133, 1999.
- [68] E. SMIRNI AND D. A. REED. Lessons from characterizing the input/output behavior of parallel scientific applications performance evaluation. In *Performance Evaluation*, pages 27–44, 1998.
- [69] A. J. SMITH. Sequentiality and prefetching in database systems. *ACM Trans. on Database Systems*, 3(3):223–247, 1978.
- [70] A. S. TANENBAUM AND A. S. WOODHULL. *Operating Systems, Design and Implementation*. Prentice Hall, 1997.

- [71] M. UYSAL, A. ACHARYA, AND J. SALTS. Requirements of I/O systems for parallel machines: An application-driven study. CS-TR-3802, Dept. of Computer Science, 1997.
- [72] R. VAN RIEL. Page replacement in linux 2.4 memory management. In *Proceeding of USENIX Annual Technical Conference (FREENIX track)*, 2001.
- [73] R. VAN RIEL. Towards an O(1) VM: Making linux virtual memory management scale towards large amounts of physical memory. In *Proceedings of the Linux Symposium*, 2003.
- [74] G. VOELKER, E. ANDERSON, T. KIMBREL, M. FEELEY, J. CHASE, A. KARLIN, AND H. LEVY. Implementing cooperative prefetching and caching in a globally managed memory system. In *Proceedings of ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, 1998.
- [75] P. R. WILSON, S. F. KAPLAN, AND Y. SMARAGDAKIS. The case for compressed caching in virtual memory systems. In *Proceedings of Annual USENIX Technical Conference*, 1999.
- [76] T. M. WONG AND J. WILKES. My cache or yours? making storage more exclusive. In *Proceedings of Annual USENIX Technical Conference*, 2002.
- [77] Y. ZHONG, C. DING, AND K. KENNEDY. Reuse distance analysis for scientific programs. In *Proceedings of 6th Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, 2000.
- [78] Y. ZHONG, M. ORLOVICH, X. SHEN, AND C. DING. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, 2004.
- [79] Y. ZHOU. Memory management for networked servers. In *Ph.D Dissertation, Computer Science Department, Princeton University*, 2000.
- [80] Y. ZHOU, A. BILAS, S. JAGANNATHAN, C. DUBNICKI, J. F. PHILBIN, AND K. LI. Experiences with vi communication for database storage. In *Proceedings of Annual International Symposium on Computer Architecture*, pages 257–268, 2002.
- [81] Y. ZHOU, Z. CHEN, AND K. LI. Second-level buffer cache management. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):505–519, 2004.
- [82] Y. ZHOU, J. F. PHILBIN, AND K. LI. The multi-queue replacement algorithm for second level buffe. In *Proceedings of Annual USENIX Technical Conference*, pages 91–104, 2001.
- [83] Q. ZHU, F. M. DAVID, C. F. DEVARAJ, Z. LI, Y. ZHOU, AND P. CAO. Reducing energy consumption of disk storage using power-aware cache management. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 118–129, 2004.

- [84] Y. ZHU AND Y. HU. Can large disk built-in caches really improve system performance? In *Proceedings of ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, pages 284–285, 2002.

VITA

Song Jiang

Song Jiang was born in Hefei, Anhui, China on October of 1969. He graduated from Hefei No. 8 High School in July of 1988. Song Jiang received his B.S. at University of Science and Technology of China (USTC) in 1993 with a degree in Computer Science, where he also received his M.E. in 1996 with a degree in Computer Science. After that he worked as a lecturer in the Department of Computer and Technology of the university for another three years.

In August of 1999, he entered the College of William and Mary as a Ph.D student in the Computer Science Department.