A NEW HYBRID APPROACH TO EXPLOIT LOCALITIES:

LRFU WITH ADAPTIVE PREFETCHING

By

JIKE CUI

Bachelor of Engineering

Nanjing Agricultural University

Nanjing, P. R. China

1994

Master of Science

Zhengzhou Institute of Technology

Zhengzhou, P. R. China

1997

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
In partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
August 2003

Oklahoma State University Library

A NEW HYBRID APPROACH TO EXPLOIT LOCALITIES:

LRFU WITH ADAPTIVE PREFETCHING

Thesis Approved:

Mansur H. Samadzadeh
Thesis Advisor

G. E. Hedrick

H. Lu

Timothy J. Pettibone
Dean of the Graduate College

PREFACE

This thesis reviewed a number of existing methods to exploit the spatial and temporal locality commonly existing in programs. This thesis provided detailed analysis and testing of adaptive prefetching (a method designed to utilize spatial locality) and the least recently and frequently used (LRFU) method (a method designed to utilize temporal locality). The two methods were combined in this thesis work in terms of their exploitation of locality. The comparative studies of the methods were done using real traces, and hit rate was used as an evaluation measure.

Adaptive prefetching dynamically detects spatial localities in the references and accordingly prefetches various numbers of blocks into cache to reduce CPU misses. Results showed that by using adaptive prefetching, the hit rate improved significantly by an average of 11.7% over the hit rate of LRU in the traces and cache configurations used. LRFU quantifies the recency and frequency and uses the combined recency and frequency value (CRF) to make replacement decisions. Results showed that LRFU consistently gives higher hit rates than LRU, but not by much in the trace files and cache configurations tested. And the λ value (a controllable parameter which determines the weights given to recency and frequency) has to be in a certain range, which is usually narrow, in order to get the best performance for hit rate. Compared to adaptive prefetching and LRU, the hybrid approach of combining adaptive prefetching and LRFU

gave a consistently higher hit rate also. But, affected by the performance of LRFU, the improvement in the hit rate by the combination was low.

ACKNOWLEDGEMENTS

First and foremost I would like to thank my advisor, Dr. Mansur H. Samadzadeh, for his intelligent supervision, essential guidance, and inspiration. I am especially appreciative of his patience and considerateness. Without his understanding and help I would not have finished my thesis smoothly.

I am grateful to my other committee members Dr. George E. Hedrick and Dr. H. K. Dai, whose guidance, assistance, encouragement, and friendship were also invaluable. Also, I would like to express my sincere appreciation to Dr. Nohpill Park and Mr. Terry Wright who provided much good advice and assistance for this study.

Moreover, hearty thanks go to my family for their endless love and my friends for their support throughout this whole process.

Finally, I would like to express my gratitude to the Computer Science Department at OSU. Without the financial support the Department has been giving to me for two and a half years, it would have been impossible for me to finish my graduate studies here.

TABLE OF CONTENTS

Chapter | Page

I. INTRODUCTION 1

II. LITERATURE REVIEW 3

2.1 Exploiting Spatial Locality 3
2.1.1 Hardware Prefetching 4
2.1.2 Software Prefetching 6
2.1.3 Integrating Software and Hardware Prefetching 6
2.2 Exploiting Temporal Locality 7

III. ADAPTIVE PREFETCHING 9

3.1 Algorithm 9
3.2 Implementation 10
3.3 Evaluation of Adaptive Prefetching 15

IV. LRFU – LEAST RECENTLY AND FREQUENTLY USED 17

4.1 Algorithm 17
4.2 Implementation 19
4.3 Evaluation of the LRFU Algorithm 22
4.3.1 Effect of λ on Hit Rate 22
4.3.2 Effectiveness and Limitations 27

V. COMBINING LRFU WITH ADAPTIVE PREFETCHING 30

5.1 Implementation 30
5.2 Evaluation of the Hybrid Approach 31
5.2.1 Effect of λ on Hit Rate 31
5.2.2 Effectiveness and Limitations 33

VI. SUMMARY AND FUTURE WORK 34

REFERENCE 37

APPENDICES 40

APPENDIX A: Glossary 41
APPENDIX B: Program Listing 43

LIST OF TABLES

Table Page

I The Effect of Threshold Values on PCC Distribution and Hit Rate 12

II PCC Distribution and Hit Rate with 0 as the Minimum Value of PCC 14

III The Hit Rate at Different Prefetched Cache Size 15

IV The Comparison of Hit Rate with and without Prefetch 15

V The Comparison of Hit Rate between LRU and LRFU Algorithm 27

VI The Hit Rate of Different Algorithms 33

LIST OF FIGURES

Figure | Page

1 Hit Rate of LRFU Implemented with and without History Memorized 21

2.a Hit Rate of LRFU for Different Traces at 0 < lambda < 1 22

2.b Hit Rate of LRFU for Different Traces at 0 < lambda < 0.01 23

2.c Hit Rate of LRFU for Trace File 022.li.din 23

3.a Hit Rate of LRFU for Cache Size of 128 and 256 Byte 24

3.b Hit Rate of LRFU for Cache Size of 512 to 2048 Byte at 0 < lambda < 1 24

3.c Hit Rate of LRFU for Cache Size of 512 to 2048 Byte at 0 < lambda < 0.02 25

3.d Hit Rate of LRFU for Cache Size of 512 Byte 25

4.a Hit Rate of LRFU with Adaptive Prefetching for Different Trace Files
at 0 < lambda < 1 32

4.b Hit Rate of LRFU with Adaptive Prefetching for Different Trace Files
at 0 < lambda < 0.01 32

CHAPTER I

INTRODUCTION

As the gap between processor speed and memory access time increases, the performance of cache becomes increasingly critical in mitigating the high latency of memory access and thus improving the overall system performance.

The potential benefit of caching lies in two kinds of localities in a program, spatial locality, i.e., the tendency for the neighboring items of a referenced item to be referenced again soon, and temporal locality, i.e., the tendency for a referenced item to be referenced soon again [Smith 82] [Johnson et al. 97].

Exploitation of spatial locality can result in cache hits for subsequent accesses to multiple nearby items, and exploitation of temporal locality can result in cache hits for subsequent accesses to a particular item [Johnson et al. 97]. Therefore to take advantage of caching fully, both localities must be exploited. In either case, the cache can be assumed to be fully associative, which means each block in main memory can be placed in any of the available block frames in cache, and therefore offering the greatest flexibility in implementing block replacement algorithms for a higher hit rate. The other mapping methods are direct mapping, where a block in memory can be placed only in a unique block frame in cache, set associative, where block frames in cache are divided into sets and a block in memory can be placed anywhere in a unique set, and sector

mapping, where blocks in both cache and memory are partitioned into sectors and any sector in memory can be placed in any sector frame in cache [Hwang 92].

The rest of this thesis is organized as follows. Chapter II provides a review of the literature on exploiting spatial and temporal localities. Chapter III discusses adaptive prefetching. Chapter IV discusses the least recently and frequently used (LRFU) algorithm. Chapter V describes the hybrid approach by combining adaptive prefetching and LRFU. Finally, Chapter VI gives the summary and future work.

CHAPTER II

LITERATURE REVIEW

2.1 Exploiting Spatial Locality

Traditionally, exploitation of spatial locality is achieved through fetching either larger block or prefetching additional blocks [Johnson et al. 97] [Pendse and Bhagavathula 98].

Prefetching anticipates cache misses and fetches data from the memory system before the processor actually needs the data. So prefetching attempts to overlay processor computation time and memory access time, and avoid stall cycles [Smith 78].

A larger block size has the natural effect of prefetching, but it has a number of negative consequences [Pendse and Bhagavathula 98] such as increasing the miss rate by fetching too many useless data items into cache, wasting bus bandwidth and thus increasing miss penalty, and poor cache utilization due to fragmentation and underutilized cache blocks. Many of these problems can be avoided by employing prefetching with a small block size.

There are three popular prefetching techniques: hardware prefetching, software prefetching, and the combination of them [Vanderwiel and Lilja 00]. These techniques are explained in the followed sections. The advantages offered by using different

prefetching methods depends on the type of localities exhibited [Milutinovic et al. 96] [Gonzales et al. 95].

2.1.1 Hardware Prefetching

Hardware prefetching includes sequential prefetching and prefetching with arbitrary strides. These two technologies are discussed below.

Sequential Prefetching: the simplest method of sequential prefetching is the One Block Lookahead (OBL) approach, that is, prefetch block m+1 when block m is accessed [Vanderwiel and Lilja 00]. The implementations differ depending on what type of access to a block initiates the prefetch of the succeeding block. Three approaches were summarized by Smith [Smith 82]: always prefetch, prefetch-on-miss, and tagged prefetch. A brief discussion of the three approaches follows.

Always prefetch means that on every memory reference an access to block m implies the prefetch of block m+1. Always prefetch can reduce the miss rate by 75 to 80 percent for large cache sizes, but it can also increase the transfer rate between memory and cache by 20 to 80 percent [Smith 78].

The *prefetch-on-miss* algorithm simply initiates a prefetch for block m+1 if access to block m causes a miss and block m+1 is not in cache. It does not increase the memory traffic much, but is not quite effective either, because it results in a cache miss for every other cache block in a strictly sequential access pattern [Vanderwiel and Lilja 00].

In *tagged prefetch*, a bit, called a tag, is associated with each block. The initial value of the tag is 0, but changed to 1 if the block is accessed by a program. And the tag is reset to 0 if the block is removed from cache. Whenever the tag of a block changes from 0 to 1, the next sequential block is fetched [Gindele 77]. Tagged prefetch is equally effective with the always prefetch algorithm and has the benefit of much lower access ratio, which is defined as the ratio of total accesses to the cache (actual plus prefetch lookup) to the number of actual references [Smith 82].

In access streams such as a tight loop, OBL (One Block Lookahead) may not avoid cache miss because the prefetch may not be initiated far enough in advance [Vanderwiel and Lilja 00]. To overcome this problem, more than one block may be fetched, that is, the degree of prefetching can be increased. However, in a program phase that exhibits little spatial locality, this aggressive prefetching can cause cache pollution, where a prematurely prefetched block may displace the data in the cache that is currently in use by the processor [Casmira and Kaeli 98], and also increase the traffic between cache and memory, which could increase the miss penalty.

In general, the benefit of sequential prefetching is that no change to the existing executables is required, and that it can be implemented with relatively simple hardware [Vanderwiel and Lilja 00].

<u>Prefetching with Arbitrary Strides</u>: for program phases with nonsequential memory access patterns such as scalar references or array accesses with large strides, sequential prefetching performs poorly. To solve that problem, Fu and Chen [Fu et al. 92] [Chen and Baer 95] proposed techniques that employ special logic to monitor a processor's address

referencing pattern to detect constant stride array references originating from looping structures. However, Dahlgren and Stenstrom [Dahlgren and Stenstrom 95] showed that those approaches only have limited performance benefits over tagged prefetching in a distributed shared memory multiprocessor.

2.1.2 Software Prefetching

The fetch instruction, which is supported by most microprocessors, can be used to implement prefetching [Bernstein et al. 95] [Santhanam et al. 97]. Software prefetching is the insertion of fetch instructions by a programmer or the compiler during an optimization pass. It is most often used within loops responsible for large array calculations. Those loops, which are common in scientific code, often have predictable array referencing patterns and exhibit poor cache utilization [Vanderwiel and Lilja 00]. Studies indicate that adding just a few prefetch directives to a program can substantially improve performance [Mowry et al. 92].

2.1.3 Integration of Hardware and Software Prefetching

Hardware prefetching initiates prefetching at run time. In contrast, software prefetching relies exclusively on compile time analysis to schedule fetch instructions. Some researches have tried to take advantage of both prefetching policies by making use of compile time program information without too much instruction overhead as incurred by pure software prefetching [Gornish and Veidenbaum 94] [Zhang and Torrellas 95]

[Chen 95]. Simulation results have showed that these approaches achieve better performance than pure hardware and pure software prefetch.

2.2 Exploiting Temporal Locality

Temporal locality can be exploited by a cache replacement algorithm [Pendse and Bhagavathula 99]. A good replacement algorithm evicts data that have no immediate need to reside in the cache, and thus retains the data that is more likely to be accessed for as small a cache size as possible.

The LRU and the LFU algorithms exploit the localities of user program and are two of the most popular replacement policies [Karedla et al. 94]. However, they both have disadvantages. LRU does not account for the possibility of references to a page in the near future, so a page that is referenced only once may replace a page that may be referenced multiple times in the near future [Wong and Baer 00]. LFU uses historical frequency counts to select a victim page or block for replacement, but it is expensive to implement and may also cause cache pollution, where inactive data tend to increase the miss rate and hence reduce the performance of the cache [Karedla et al. 94].

A number of efforts have been done to find good cache replacement algorithms based on LRU and LFU. O'Neil and his colleagues [O'Neil et al. 93] provided the LRU-K page algorithm on database page buffering. This algorithm maintains a history table that contains the counts of the last K references to popular database pages, and statistically estimates the inter-arrival times of the references on a page by page basis. Similarly, Pomerene and his colleagues [Pomerene et al. 89] suggested the use of a

shadow directory in order to look at a relatively longer history when making decisions with LRU. The problem with those two algorithms is that the size of the history table or the shadow directory limits the length of the history consulted. Some other algorithms such as 2Q [Johnson and Shasha 94] and S-LRU [Karedla et al. 94], have also been proposed to try to partially take account of the frequencies while making the LRU decisions and keeping the overhead low.

Many researchers have also tried to find other ways to reduce the miss rate. Pendse and Walterscheitdt [Pendse and Walterscheidt 96] proposed a fuzzy replacement algorithm that is based on the application of nine fuzzy rules. It utilizes the age and the reference frequency of a block to make replacement decisions. The performance of the fuzzy replacement algorithm is better than that of LRU for small cache sizes, and comparable with that of LRU for large cache sizes. Compared to traditional approaches that are mostly based on LRU, LFU, etc., this approach opens a new direction in disk caching and page replacements in virtual memory management system.

CHAPTER III

ADAPTIVE PREFETCHING

3.1 Algorithm

The advantage of sequential prefetching is that the implementation is relatively simple. However OBL (One Block Lookahead) has the shortcoming that prefetch may not be initiated early enough to avoid a memory processor stall (for details see Section 2.1.1). To solve the problem of memory processor stall, an adaptive prefetching technique has been proposed [Pendse and Bhagavathula 98]. In this scheme, the degree of prefetching is matched to prefetch efficiency, the percentage of hit in the total number of prefetched blocks. Prefetch efficiency is an indication of the degree of spatial locality of a program and is calculated periodically. The adaptive prefetching algorithm performs well [Pendse and Bhagavathula 98] and, compared to other algorithms, it is relatively simple in terms of the complexity of the algorithm and the ease of implementation. The algorithm designed by Pendse and Bhagavathula [Pendse and Bhagavathula 98] is discussed below with some modifications and supplements. Those modifications and supplements have to do with block placement and the effect on hit rate by the change of some parameter values as listed in Section 3.2.

Assume the cache is fully associative. It is divided into two parts, the main cache and the prefetched cache, and both are managed by the LRU replacement algorithm. On each reference, both caches will be searched. Two variables, the prefetch counter that is preset to 0 and the prefetch coefficient counter that is preset to 1, are used to control the number of blocks to be prefetched.

On a cache hit the prefetch counter is incremented. The prefetch counter is sampled at a regular interval of 16 references and the sampled value is compared against two preset threshold levels of 6 and 12. If it is higher than 12, the prefetch coefficient counter is incremented. If it is lower than 6, the prefetch coefficient counter is decremented. Otherwise, the prefetch coefficient counter remains unchanged. After the comparison, the prefetch counter is reset to 0. The numbers mentioned above were used in the original algorithm without explanation. The next section discusses the validity of those numbers.

On a cache miss, the missed block is brought into the main cache and the victim block is placed in the prefetched cache. Prefetched blocks, whose numbers are given by the prefetch coefficient counter, are also brought into the prefetched cache if they are not in the cache.

The upper limit of the prefetch coefficient counter is the size of the prefetched cache, and the lower limit is 1, because 0 means no prefetch.

3.2 Implementation

The efficient implementation of the LRU replacement algorithm includes several ways, such as the second chance algorithm or the additional reference bit algorithm

[Deitel 90] [Silberschatz and Galvin 94]. Since the efficiency in implementing LRU is not the goal of this thesis report, LRU is implemented using a stack which is implemented by a linked list.

Four trace files, 015.doduc.din, 008.espresso.din, 013.spice2g6.din, and 022.li.din, were used for the evaluation and experimentation. Each trace contains one million references from a MIPS R3000 machine and was downloaded from http://tracebase.nmsu.edu.

The following are discussions on some details about the block placement in adaptive prefetching implementation that are important but have not been addressed as well as discussions on the validity and effect of some parameters that are important but have not been fully discussed in the original algorithm for adaptive prefetching [Pendse and Bhagavathula 98]:

1) How to deal with blocks expelled from the main cache? Because according to temporal locality, those blocks may still be needed. Therefore in the implementation used for this thesis work, those blocks are placed at the head of the block list in the prefetched cache to give them more life time.

2) How to deal with the blocks that are hit in prefetched cache? In the implementation used for this thesis work they are placed at the head of the block list in the main cache.

3) The choice of the threshold values used to increment or decrement the prefetch coefficient counter, which is a variable that decides the number of blocks to be prefetched. In the original algorithm [Pendse and Bhagavathula 98], a sample rate of 16 references is used, and the two threshold levels are 6 and 12 (sample rate is the number of references that are sampled and used to increment the prefetch counter, and prefetch

counter is a variable that stands for the level of spatial locality in the current piece of the workload, it has an initial value of zero and is incremented by every hit in the references sampled at a certain rate). However the actual running results given below raise questions on the effectiveness of the choice of those two values in detecting spatial locality quickly and at the same time maintaining low bus traffic.

Table I is the result from trace file 022.li.din. The size of prefetched cache is 8 blocks, so 1 to 8 blocks can be prefetched each time. The prefetch coefficient counter distribution gives the percentages of the number of prefetched blocks each time in all the prefetches. For example, "8: 99%" means that in 99% of prefetches 8 blocks are prefetched. In the prefetch coefficient counter distribution in Table I, only significant percentages are listed.

TABLE I: THE EFFECT OF THRESHOLD VALUES ON PCC DISTRIBUTION AND HIT RATE
(PCC: Prefetch Coefficient Counter)

Cache Size: 512 Byte, Block Size: 8 Byte, Size of Prefetched Cache: 8 blocks				
Thresholds	6, 12	10, 12	10, 14	12, 14
PCC Distribution	8: 99%	8: 76%, 7: 14%	8: 49%, 7: 13%	8: 37%, 1: 32%
Hit Rate	82.8%	83.0%	83.7%	82.4%
Cache Size: 256 Byte, Block Size: 8 Byte, Size of Prefetched Cache: 4 blocks				
Thresholds	6, 12	10, 12	10, 14	12, 14
PCC Distribution	4: 98%	4: 41%, 1: 33%	4: 31%, 1: 51%	4: 22%, 1: 60%
Hit Rate	74.3%	73.7%	73.3%	73.5%

Table I shows that while the threshold values are low, the prefetch coefficient counter distribution concentrates on the maximum value of the prefetch coefficient counter. So in each fetch, very likely the maximum number of blocks are prefetched. However those prefetches do not increase the hit rate significantly. So it is better to use higher threshold values to reduce the bus traffic. Therefore, the values 6 and 12 as used in the original algorithm are not good for trace files 022.li.din, and may not be optimal for other workloads either. If the spatial locality is low, low threshold values may create more prefetches but, if that is the case, more prefetches may not be effective at all because of the low spatial locality, and may not be worthwhile with respect to the increase in the bus traffic.

With the increase in cache size, the hit rate increases too. Therefore the threshold values should increase accordingly to reduce the bus traffic.

4) The minimum value of prefetch coefficient counter, 1 or 0. If set to 1, questions may arise that if the spatial locality is not good, we should not allow the system to always prefetch. In Table I, the minimum value is set to 1. Table II gives the result from trace file 022.li.din, with the minimum value of prefetch coefficient counter as 0.

Table II shows that if the minimum value of the prefetch coefficient counter is 0, then in many cases there will be no prefetch at all, especially when the threshold values are high and cache size is small, as a result, the hit rate is reduced significantly. Therefore it is safer to keep 1 as the minimum value of the prefetch coefficient counter. Since the prefetched blocks are only added to the prefetched cache that takes a small part of the whole cache, there is no cache pollution to the main cache.

TABLE II: PCC DISTRIBUTION AND HIT RATE WITH 0 AS THE MINIMUM VALUE OF PCC
(PCC: Prefetch Coefficient Counter)

Cache Size: 512 Byte, Block Size: 8 Byte, Size of Prefetched Cache: 8 blocks				
Thresholds	6, 12	10, 12	10, 14	12, 14
PCC Distribution	8: 99%	8: 76%, 7: 14%	8: 49%, 7: 13%	8: 36%, 0: 36%
Hit Rate	82.8%	83.0%	83.5%	76.2%
Cache Size: 256 Byte, Block Size: 8 Byte, Size of Prefetched Cache: 4 blocks				
Thresholds	6, 12	10, 12	10, 14	12, 14
PCC Distribution	4: 98%	4: 37%, 0: 35%	4: 29%, 0: 53%	4: 19%, 0: 63%
Hit Rate	74.3%	67.8%	65.1%	63.3%

5) Prefetched cache size. It cannot be too small, otherwise a good spatial locality may not be satisfied. In the original algorithm [Pendse and Bhagavathula 98], a prefetched cache size of 1/3 of the total cache size was used. However Table III shows that the prefetched cache size does not have a significant effect on the hit rate. So a small prefetched cache size is preferred to avoid possible heavy bus traffic caused by a large prefetched cache.

TABLE III: THE HIT RATE AT DIFFERENT PREFETCHED CACHE SIZE (TOTAL CACHE SIZE: 1 Kilobyte; BLOCK SIZE: 8 Byte)

Trace File	Prefetched Cache Size / Total Cache Size				
	1/4	1/6	1/8	1/10	1/12
022.li.din	88.3%	89.7%	89.8%	89.8%	89.8%
013.spice2g6.din	88.9%	89.2%	89.5%	89.4%	89.4%

3.3 Evaluation of Adaptive Prefetching

This section discusses the evalution of adaptive prefetching in terms of improvement in the hit rate. Table IV shows the effect of LRU with adaptive prefetching as compared to LRU with no prefetch. The hit rate is improved significantly by an average of 11.7%. The reason seems to be that adaptive prefetching dynamically detects the spatial locality in the references and accordingly prefetches various numbers of blocks into cache to avoid CPU miss.

TABLE IV: COMPARISON OF HIT RATE WITH AND WITHOUT PREFETCH (CACHE SIZE: 512 Byte; BLOCK SIZE: 8 Byte; RATIO OF PREFETCHED CACHE SIZE: 1/8)

Prefetch	Trace File			
	078.swm256.din	013.spice26.din	015.doduc.din	022.li.din
No Prefetch	88.3%	66.0%	75.6%	69.9%
Adaptive Prefetch	92.8%	82.7%	87.2%	83.7%

There are also a number of shortcomings in using adaptive prefetching. They are briefly explained below.

1) The performance of adaptive prefetching depends on two important parameters: the threshold values used to decrement or increment the prefetch coefficient counter, and the size of the prefetched cache that although may not affect the hit rate significantly, but could affect the bus traffic very much. Those parameters have to be well tuned for maximum performance output.

2) Adaptive Prefetching should be used for instruction cache. It may not work well on data cache that has very limited spatial locality.

3) Since adaptive prefetching splits the cache into two parts, it makes the cache design more complicated.

Besides the hit rate, the miss penalty is also an important parameter to evaluate a cache system. Miss penalty includes access time and transfer time. Since the prefetch is on consecutive blocks with the missed one, it does not increase access time. Obviously the prefetch increases the bus traffic and thus may increase the transfer time. However, transfer time is usually a small fraction of the access time. So the miss penalty will not be increased too much as a result of prefetching multiple blocks for each cache miss.

CHAPTER IV

LRFU – LEAST RECENTLY AND FREQUENTLY USED

4.1 Algorithm

LRFU is a relatively new cache replacement algorithm proposed by Lee and his colleagues [Lee et al. 99]. It has the following features that distinguish it from other algorithms.

- It quantifies both recency and frequency and their combined value: Combined Recency and Frequency (CRF). In a cache miss, the block with the smallest CRF is replaced.
- Unlike many previous policies such as LRU-K, 2Q, and S-LRU, which use limited history to make block replacement decisions, the LRFU policy uses the complete reference history by using only a few words for each block.
- LRFU can be implemented efficiently with a time complexity ranging from O(1) to $O(\log_2 n)$, corresponding to the native implementations of the LRU and the LFU policies [Lee et al. 99].

The CRF (Combined Recency and Frequency) value is calculated as follows.

The CRF value of a block b at time t, denoted by $C_t(b)$, is defined as

$$C_t(b) = \sum_{i=1}^{k} F(t - t_{b_i})$$

where t_{b_i} is the time of the ith reference of block b, and F is a weighting function, $F(x) = \left(\frac{1}{2}\right)^{\lambda x}$, for $0 \leq \lambda \leq 1$. Here λ is a controllable parameter and x is the time span from the reference in the past to the current time. The λ value determines the weights given to recent and old history, and therefore makes it possible to find an optimal combination of the effects of the recency and frequency factors of the past references on the probability of the future re-references.

The LRFU policy has five properties as stated in the paper by Lee and his colleagues [Lee et al. 99]. The following are mostly verbatim quotes with a minor addition in property 2 on the upper bound of k.

1) If $\lambda = 0$, $F(x) = 1$, then LRFU replaces the same block as LFU.

2) $\forall i,\ F(i) > \sum_{j=i+1}^{k} F(j)$, where $k \geq i + 1$ and $k \leq \log_{2^{-\lambda}}(2 - 2^{\lambda})$

   when $\lambda \rightarrow 1$, the upper bound of k becomes ∞, then LRFU replaces the same block as LRU

3) Because $F(x+y) = F(x)\,F(y)$, $C_{t_c}(b) = F(\delta) C_{t_{b_k}}(b)$ for $\delta = t_c - t_{b_k}$, and

   $C_{t_{b_k}}(b) = F(0) + F(\delta) C_{t_{b_{k-1}}}(b)$ for $\delta = t_{b_k} - t_{b_{k-1}}$

   where x and y are nonnegative integer, t_c is the current time, and t_{b_k} is the time of the kth reference of block b.

This shows that at any time the CRF value of a block can be computed using only two variables, the last CRF value and the last reference time, for each block, and these are all the history the block needs to maintain in order to calculate the current CRF value.

4) If $C_t(a) > C_t(b)$ and neither a nor b has been referenced after t, then $C_{t'}(a) > C_{t'}(b)$ for all $t' > t$. Therefore the relative ordering between two blocks does not change until either of them is referenced. Hence reordering of blocks is needed only upon a block reference.

5) $\forall d \geq d_{threshold}, F(0) > \sum_{i=d}^{\infty} F(i), where\ d_{threshold} \geq \left\lceil \left(\log_{1/2}\left(1-\left(1/2\right)^{\lambda}\right)\right)/\lambda \right\rceil$

This states that the number of blocks that have CRF values larger than F(0) is bounded by $d_{threshold}$. Hence $d_{threshold}$ blocks can be maintained in a heap and the remaining blocks in a linked list, such that any block in the heap has a larger CRF value than that of any block in the linked list. With this setting, the CRF value of the blocks in the linked list cannot be larger than F(0), since the number of blocks that can have CRF values larger than F(0) is bounded by $d_{threshold}$ and the number of blocks maintained in the heap is $d_{threshold}$.

4.2 Implementation

This section discusses the optimized implementation of the LRFU policy. For the materials discussed in this section and Section 4.3, four trace files, 015.doduc.din, 008.espresso.din, 013.spice2g6.din, and 022.li.din, were used in the experimentation. Please see Section 3.2 for information on those trace files.

Because of property 5, the cache was implemented as a heap and a linked list. The root of the heap has the smallest CRF value in the heap, the blocks in the linked list are ordered by their CRF values with the largest at the head and the smallest at the tail.

When $\lambda = 1$, $d_{threshold} = 1$ and hence only one block needs to be maintained in the heap. This implies all other blocks are in the linked list. This corresponds to the native LRU implementation and its time complexity is O(1).

When $\lambda = 0$, $d_{threshold} = \infty$. Therefore all blocks are maintained in the heap and the time complexity is $O(\log_2 n)$. This also corresponds to the native LFU implementation and its time complexity.

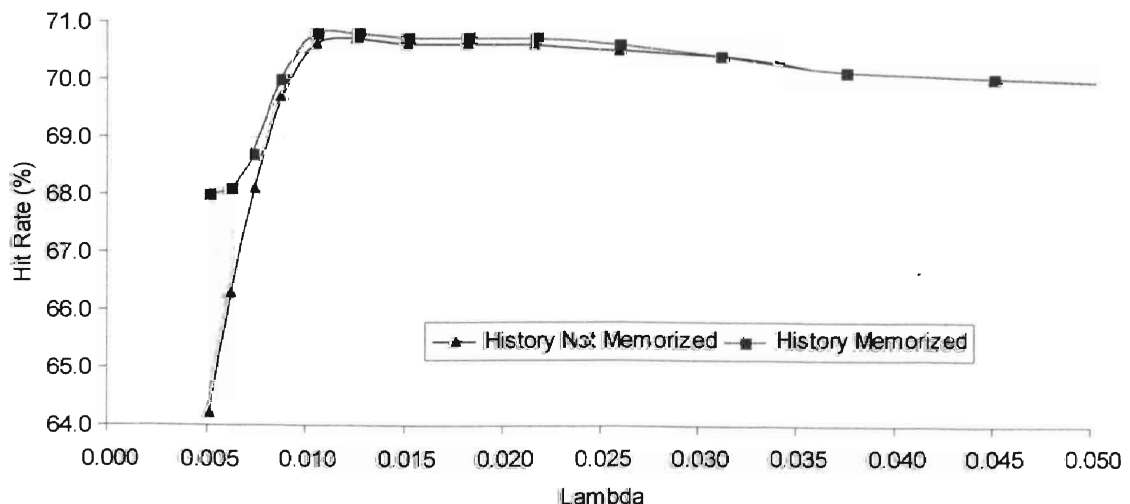
The algorithm of the program is given below:

```
A reference to block b is issued by CPU
If the cache is not full    //cool start
    if b is not in the cache    //a cache miss. b: the requested block
        if size of the heap equals d_threshold
            remove the root of the heap and make it the head of the linked list
        fi
        update b's CRF and time stamp
        insert b into the root of heap H, reorder the heap
    else    // b  is in the cache, a cache hit
        if b is in the heap
            update b's CRF and time stamp
            reorder the subheap rooted by b
        else    //b is in the linked list, which means the heap is full
            remove the root of the heap and make it the head of the linked list
            remove b from the linked list
            update b's CRF and time stamp
            insert b into heap H, reorder the heap
        fi
    fi
else    //cache is full
    if b is not in the cache    //a cache miss. b: the requested block
        remove the tail of the linked list
        remove the root of the heap and make it the head of the linked list
        update b's CRF and time stamp
        insert b into heap H, reorder the heap
    else    // b  is in the cache, a cache hit
        if b is in the heap
            update b's CRF and time stamp
            reorder the subheap rooted by b
        else    //b is in the linked list
            remove the root of the heap and make it the head of the linked list
            remove b from the linked list
            update b's CRF and time stamp
            insert b into heap H, reorder the heap
```

```
        fi
    fi
fi
```

In the implementation in this thesis work, after a block is replaced, it memorizes its last CRF value and its last reference time. So, when it is placed in cache again, its previous hits are counted. Figure 1 compares the hit rate of LRFU with and without history memorized from trace file 022.li.din. It indicates that, when frequency is given more weight, this implementation produces better results than the implementation where a block does not memorize its history after it is replaced.

When $\lambda > 0.03$, both the original and the modified algorithms have the same hit rate, therefore the hit rates for $\lambda > 0.05$ are not shown in Figure 1. When $\lambda < 0.03$, which means when frequency is giving more and more weight in calculating CRF, the implementation with history memorized quickly outperforms the one without history memorized.



4.3 Evaluation of the LRFU Algorithm

This section describes the evaluation of the LRFU replacement policy based on the hit rate. The evaluation focuses on the effect of λ and the effectiveness and limitations of LRFU.

4.3.1 Effect of λ on Hit Rate

λ balances the weight between frequency and recency, and is the main variable in the LRFU algorithm. Figures 2.a, 2.b, 2.c, 3.a, 3.b, 3.c, and 3.d illustrate that λ is important to the hit rate. Figures 2.a, 2.b, and 2.c show the hit rate of LRFU for four trace files, Figures 3.a, 3.b, 3.c, and 3.d show the hit rate of LRFU for six cache sizes for trace file 022.li.din. In some of the figures, the individual experimentation points are too crowded and hence are not shown.

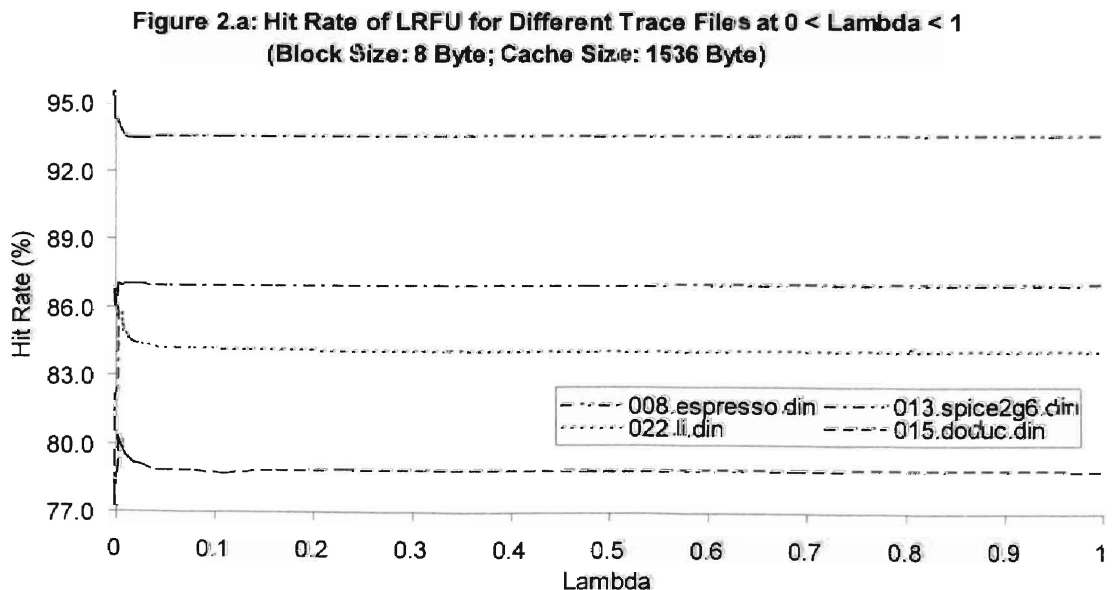


Figure 2.a: Hit Rate of LRFU for Different Trace Files at 0 < Lambda < 1 (Block Size: 8 Byte; Cache Size: 1536 Byte)

Figure 2.b: Hit Rate of LRFU for Different Trace Files at 0 < Lambda < 0.01
(Block Size: 8 Byte; Cache Size: 1536 Byte)

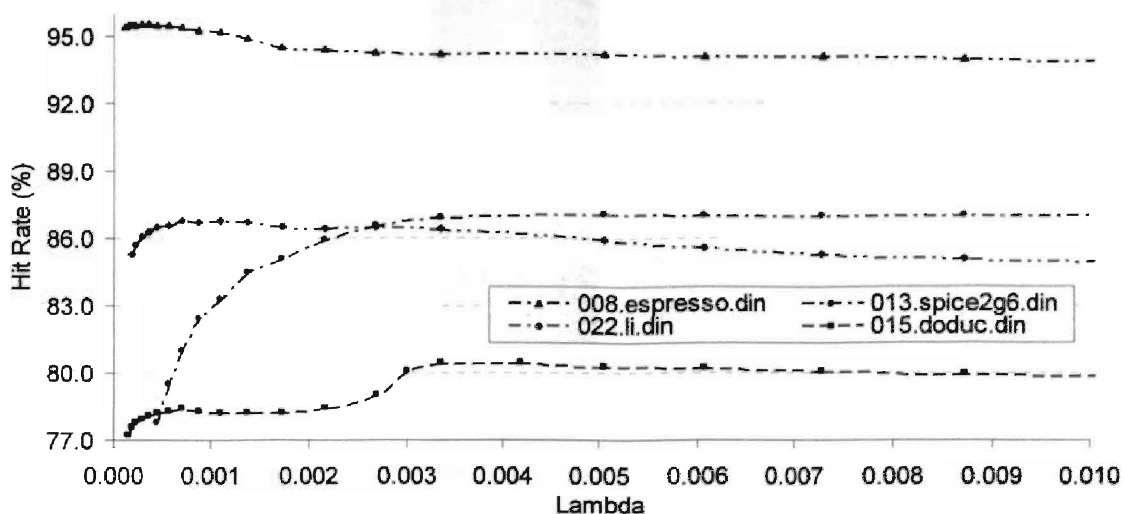


Figure 2.c: Hit Rate of LRFU for Trace File 022.li.din
(Block Size: 8 Byte; Cache Size: 1536 Byte)

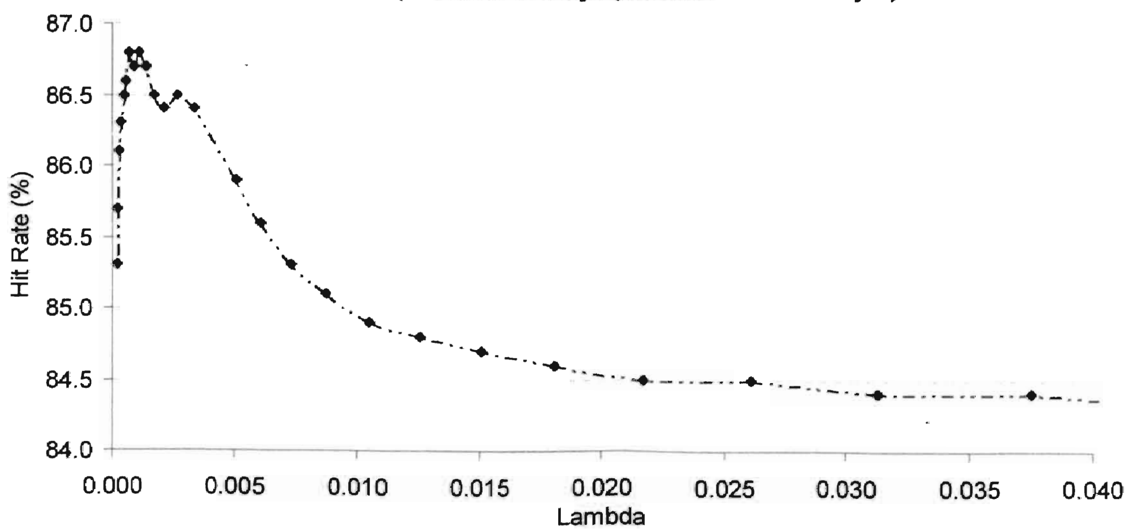


Figure 3.a: Hit Rate of LRFU for Cache Size of 128 and 256 Byte (Block Size: 8 Byte)

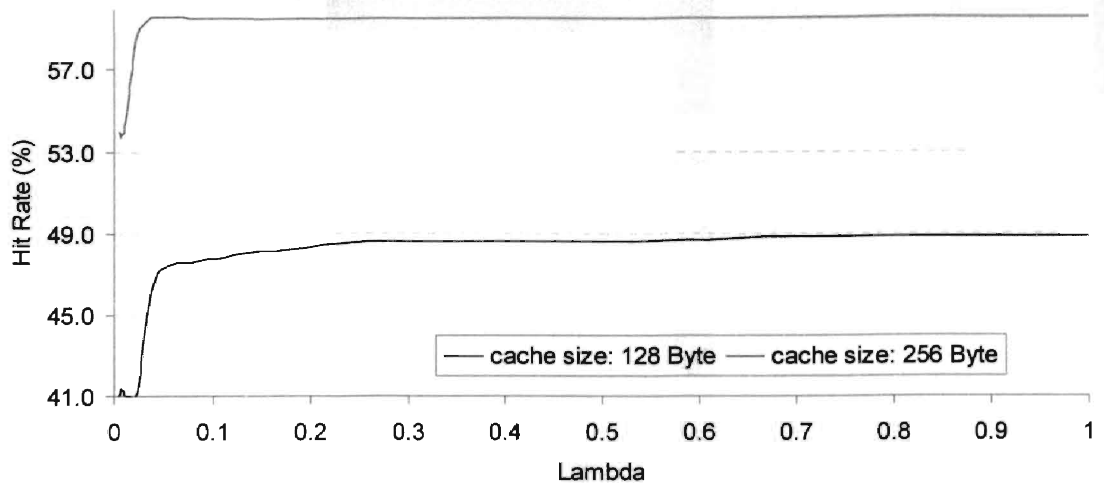


Figure 3.b: Hit Rate of LRFU for Cache Size of 512 to 2048 Byte at 0 < Lambda < 1 (Block Size: 8 Byte)

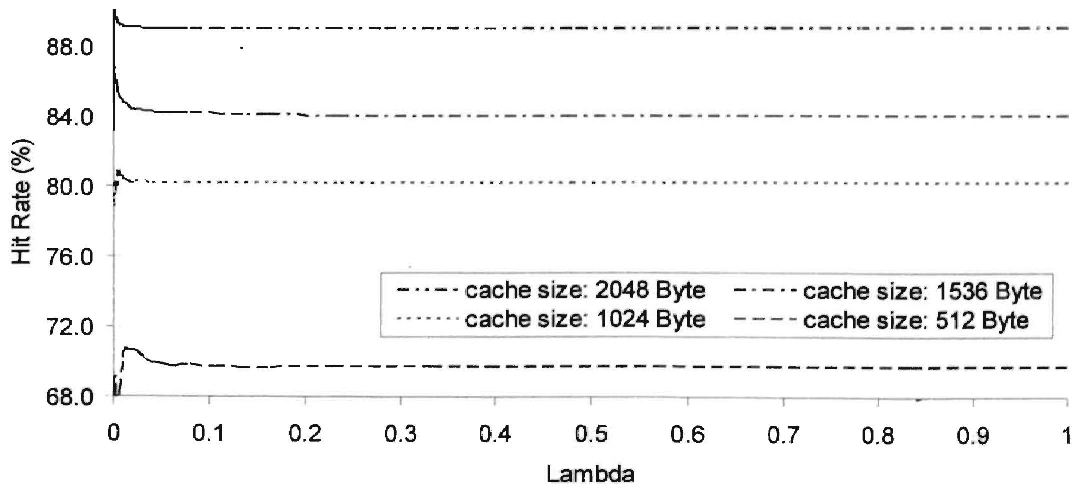


Figure 3.c: Hit Rate of LRFU for Cache Size of 512 to 2048 Byte at 0 < Lambda < 0.02 (Block Size: 8 Byte)

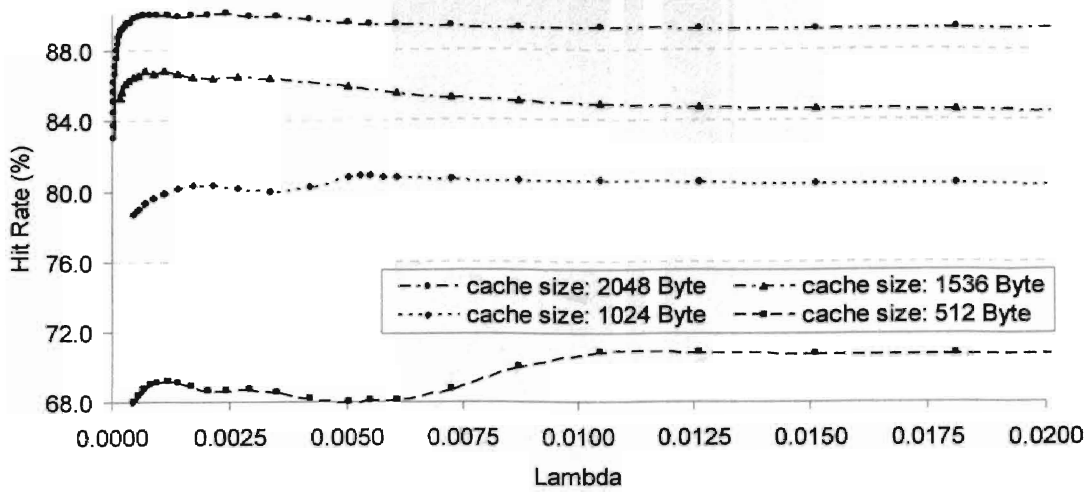
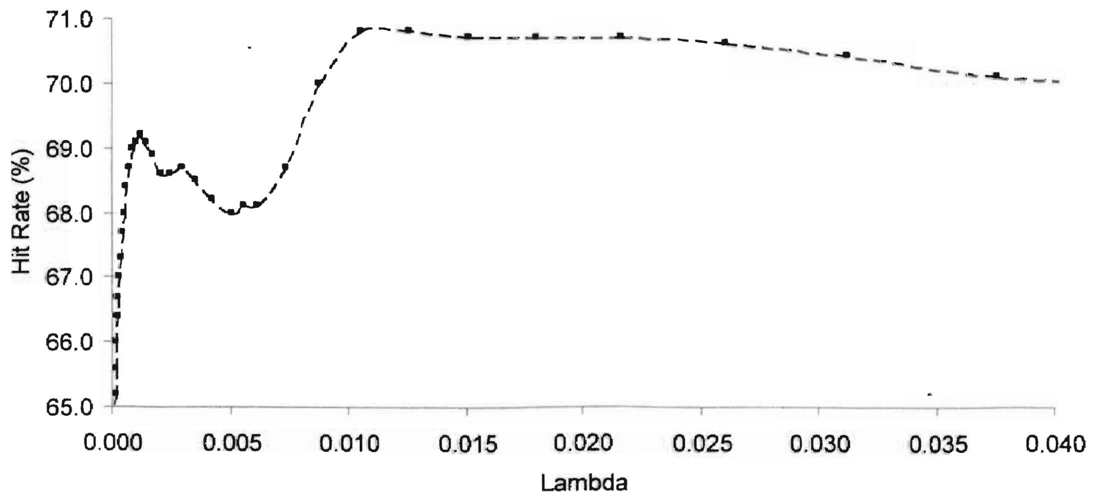


Figure 3.d: Hit Rate of LRFU for Cache Size of 512 Byte (Block Size: 8 Byte)



Figures 2.a, 2.b, 2.c, 3.a, 3.b, 3.c, and 3.d indicate that the improvement of hit rate by LRFU compared to LRU depends on two factors:

1) The temporal locality of the trace file, especially the weight of frequency. Figures 2.a, 2.b, and 2.c illustrate that when λ is 1, LRFU is on the LRU side and has the same hit rate as LRU. With the decrease of λ, frequency is giving more weight and the hit rate increases very slowly until a certain point that depends on the trace files and the cache sizes. After that point, if frequency weighes more than recency in the trace files, the hit rate increases more quickly and reaches a peak. But after the peak, the hit rate decreases rapidly as exhibited by trace file 022.li.din. When λ is very close to zero, which means when LRFU is actually LFU, the hit rate may be lower than that of LRU. After that point, if frequency does not weigh more than recency, the hit rate just begins decreasing and goes below the hit rate of LRU as exhibited by trace file 013.spice2g6.din.

2) Cache size. Figure 3.a shows that with the decrease of λ, LRFU cannot improve the hit rate. More decrease on λ makes the hit rate go down finally. It indicates that when the cache size is too small, even if there is temporal locality in the traces, it cannot be exploited because blocks are replaced too quickly. With the increase of the cache size, the lambda value that starts to give bad hit rate in Figure 3.a and that gives the maximum hit rate in Figure 3.b becomes smaller. That means that with bigger cache temporal locality can be exploited more, thus frequency is getting more weight in improving the hit rate. Figure 3.b shows that with the increase of the cache size, LRFU can improve the hit rate over LRU. However Figure 3.b also shows that it is not true that the larger the cache size is, the bigger the improvement will be. The reason might be that a larger cache size

naturally produces more chance to utilize temporal locality, regardless of the replacement algorithm used.

4.3.2 Effectiveness and Limitations

A competitive analysis of LRFU [Cohen et. al 02] shows that although LRFU is not optimally competitive, it may behave well in practice because it combines the benefits of both LRU and LFU. However, for the trace files and the test conditions that we used, LRFU does not perform as good as expected as listed in Table V. The scale of λ in Table V is those λ values where the highest hit rate is produced. Since in different trace files the correlation between the hit rate and λ are different, the border numbers of those scales have different significant digits.

Table V: THE COMPARISON OF HIT RATE BETWEEN LRU AND LRFU ALGORITHM (CACHE SIZE: 1536 Byte; BLOCK SIZE: 8 Byte)

Trace File	LRU	LRFU
022.li.din	84.1%	86.8% ($0.00056 < \lambda < 0.00138$)
008.espresso.din	93.6%	95.6% ($0.00023 < \lambda < 0.00045$)
013.spice2g6.din	87.0%	87.1% ($0.010 < \lambda < 0.037$)
015.doduc.din	78.9%	80.4% ($0.0030 < \lambda < 0.0050$)

Based on the analysis in Section 4.3.1, the reason that the performance improvement of LRFU over LRU is small is that not much temporal locality can be

utilized, which is because of either the lack of temporal locality in the workload or the small cache size.

The results expose some problems with the LRFU algorithm. They are listed below.

1) It is more complicated and the time complexity is higher than that of LRU. The optimized LRFU implementation has a time complexity of $O(\log_2 d_{threshold})$, where $d_{threshold} \geq \left\lceil \left(\log_{\frac{1}{2}}\left(1-\left(\frac{1}{2}\right)^{\lambda}\right)\right)/\lambda \right\rceil$, whereas the time complexity of LRU is just O(1).

2) The benefit of LRFU is closely related to the type of workload, and may be very limited for data lacking temporal locality, thus it may not be worth imposing the increased complexity on the system.

3) More importantly, λ has to be well tuned for a specific system and workload in order to maximize the performance, otherwise the hit rate of LRFU may be even lower than that of LRU. And the scale of the λ values where LRFU gives the highest hit rate is very narrow. Furthermore. it is difficult to predetermine the exact optimal scale based on the cache size and the type of the trace files.

But LRFU is still a promising algorithm. Firstly, for traces that have good temporal locality, the hit rate of LRFU is always higher than LRU for any λ in [0,1). Secondly, the range of λ values where LRFU produces bad hit rate is usually very narrow. Thirdly, although it is difficult to predetermine the exact optimal scale of λ values, by using statistical methods, obtaining a rough estimate of the optimal scale is always possible based on a large amount of running results. Finally, a self-adjusting algorithm that dynamically adjusts λ to its optimal values by detecting the local temporal

locality is possible, and LRFU with such an algorithm can produce a better hit rate than the non-adjusting LRFU.

CHAPTER V

COMBINING LRFU WITH ADAPTIVE PREFETCHING

To better exploit both spatial and temporal locality, it makes sense to combine adaptive prefetching and LRFU. See Chapter IV for an introduction and discussion of the LRFU (Least Recently and Fequently Used) algorithm.

5.1 Implementation

This section describes the implementation of LRFU with adaptive prefetching. The following paragraph outlines the selection of some important parameters in implementing adaptive prefetching.

The threshold values used to increment or decrement the prefetch coefficient counter in adaptive prefetching: Figures 3.a and 3.b indicate that if the cache size is too small, LRFU may not give better a hit rate than LRU. But, on the other hand, if the cache size is too large, the hit rate will be too high and there will not be much room for LRFU to improve the hit rate further. Therefore a cache size of 1536 Byte was chosen. At this cache size, the hit rate will be relatively high. Therefore the threshold values were set to 10 and 14 to get a high hit rate and at the same time maintain a low bus traffic.

The prefetched cache size: Since it is shown in Section 3.2 that the prefetched cache size does not have a significant effect on the hit rate, a small prefetched cache size of 1/12 of the total cache size was used.

LRFU is only used to manage the main cache: The prefetched cache is too small, and there is not a significant difference between LRFU and LRU for it. So the prefetched cache is still managed by LRU.

5.2 Evaluation of the Hybrid Approach

This section discusses the performance of the combination of LRFU with adaptive prefetching in terms of the hit rate. The evaluation focuses on the effect of λ in LRFU and the effectiveness and limitations of the combination of LRFU and adaptive prefetching. Four trace files, 015.doduc.din, 008.espresso.din, 013.spice2g6.din, and 022.li.din, were used in the experimentation. Please see Section 3.2 for information on those trace files.

5.2.1 Effect of λ on Hit Rate

Figures 4.a and 4.b illustrate the effect of changes in λ on the hit rate. Compared to Figures 2.a and 2.b, all the curves change in shape to some degree except for trace file 013.spice2g6.din. The overall trend of those curves does not change, but the improvements in the hit rate become smaller, and the ranges of the optimal λ are also

different, which means that the distribution of temporal locality in those trace files is also affected by adaptive prefetching.

Figure 4.a: Hit Rate of LRFU with Adaptive Prefetching for Different Trace Files at 0 < Lambda < 1 (Block Size: 8 Byte; Cache Size: 1536 Byte)

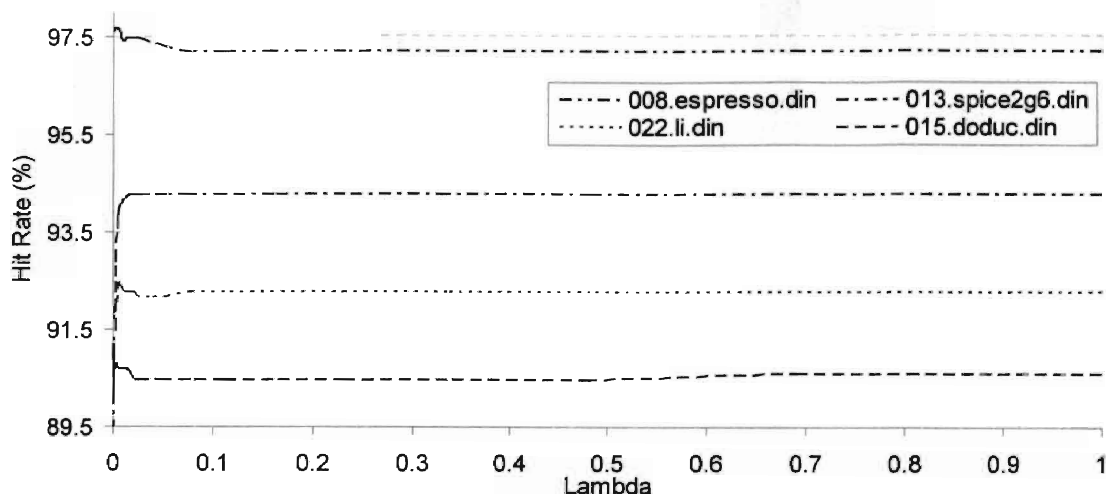
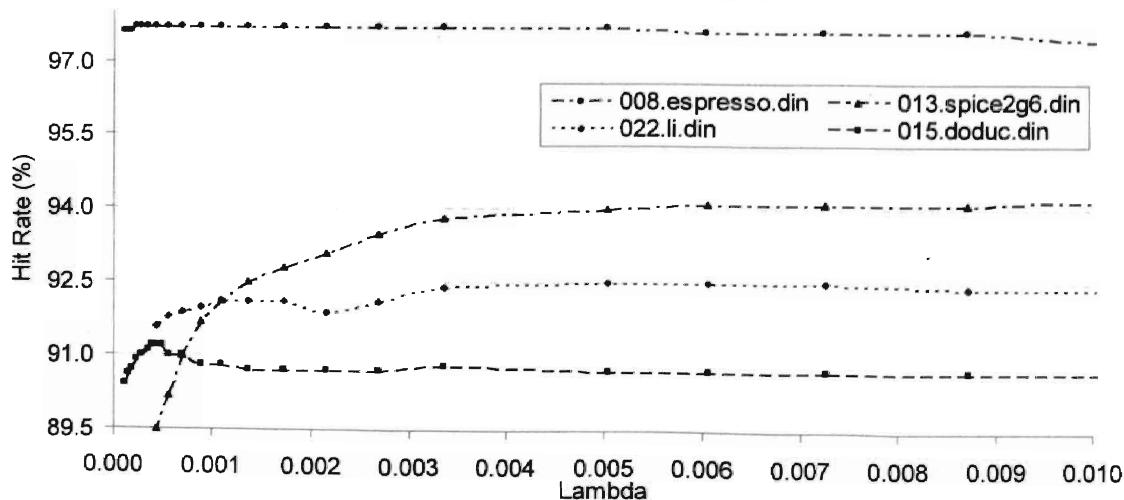


Figure 4.b: Hit Rate of LRFU with Adaptive Prefetching for Different Trace Files at 0 < Lambda < 0.01 (Block Size: 8 Byte; Cache Size: 1536 Byte)



5.2.2 Effectiveness and Limitations

Table VI gives the hit rate of LRFU with adaptive prefetching. It shows that the hybrid approach produces higher hit rates than LRU by an average improvement of 8% for the tested cases. And it consistently gives a better hit rate, although only slightly in those tests, than the combination of adaptive prefetching and LRU. The reason that the hit rate improvement is small is mainly that the improvement of hit rate by LRFU over LRU is small, secondly the prefetching improves the hit rate to above 90%, thus leaveing a small space for the hit rate to be improved further.

The limitations are the problems related to each algorithm, as discussed in Section 3.3 and 4.3.2.

TABLE VI: THE HIT RATE OF DIFFERENT ALGORITHMS
(CACHE SIZE: 1536 Byte; BLOCK SIZE: 8 Byte)

Trace File	LRU	LRFU	Adaptive Prefetching	LRFU + Adaptive Prefetching
022.li.din	84.1%	86.8%	92.3%	92.5% ($0.0033 < \lambda < 0.0087$)
008.espresso.din	93.6%	95.6%	97.2%	97.7% ($0.00062 < \lambda < 0.00070$)
013.spice2g6.din	87.0%	87.1%	94.3%	94.3% ($0.013 < \lambda < 1$)
015.doduc.din	78.9%	80.4%	90.6%	91.2% ($0.00037 < \lambda < 0.00056$)

CHAPTER VI

SUMMARY AND FUTURE WORK

In this thesis report, methods of exploiting localities in programs were discussed. Adaptive prefetching, LRFU, and a hybrid approach combining those two methods, were examined.

Chapter I gives a brief definition of spatial locality, temporal locality, and cache mapping methods, and introduces the significance of exploiting those localities. Chapter II provides a review of literature on exploiting spatial and temporal localities. Chapter III discusses adaptive prefetching and evaluates its effectiveness. Chapter IV gives an introduction to the least recently and frequently used (LRFU) algorithm, discusses the implementation issues, and evaluates the effect of λ on the hit rate and the performance of LRFU. Chapter V explores and evaluates the combination of adaptive prefetching and LRFU.

Testing results reveal that adaptive prefetching, which exploits spatial locality, works well by improving the hit rate by an average amount of 11.6% in the four trace files and cache configurations used. However, testing results also show that the threshold values used to decrement or increment the prefetch coefficient counter and the size of prefetched cache need to be well tuned to maximize performance.

LRFU is a relatively new and interesting technique to exploit temporal locality. Results show that it does consistently give higher hit rates than LRU, but not much higher in the trace files tested. And the λ value has to be in a certain range, which is usually narrow, to yield the best performance.

Compared to adaptive prefetching and LRU, the hybrid approach, which combines adaptive prefetching and LRFU, gives consistently higher hit rates as well. But similar to the performance improvement of LRFU over LRU, the improvement of hit rate by the combined method is low.

The future work can focus on the following aspects.

1) Making adaptive prefetching self-adjusting: The performance of adaptive prefetching depends on two important parameters: the threshold values used to decrement or increment the prefetch coefficient counter and the size of prefetched cache. Those parameters affect not only the hit rate, but also the bus traffic. The prefetched cache size does not have a big impact on the hit rate and can be fixed for a given system. But the optimal threshold values are closely related to the spatial locality in the workload, therefore an algorithm that can dynamically detect the spatial locality and consecutively adjust the threshold values should produce the highest hit rate and lowest bus traffic at the same time.

2) Making LRFU self-adjusting: The scale of λ values usually is in a narrow range to achieve the best performance. Thus a self-adjusting λ, which can adjust its value by detecting temporal locality in the workload, is very important for processing different type of workloads.

3) Integrating adaptive prefetching and LRFU into a real system such as FreeBSD to verify their performance: In a real system, LRFU may work better than in our tests. Firstly, in our test the traces may not have had enough temporal localities either because they are just not the types that exhibit good temporal localities, or because they are too short compared to real workloads. The different types and different lengths of workloads in a real system can give more accurate test results for LRFU. Secondly, in our test the cache size was very small, which may not allow temporal locality to be exploited much because blocks are replaced in a short time.

REFERENCES

[Bernstein et al. 95] D. Bernstein, D. Cohen, and A. Freund, "Compiler Techniques for Data Prefetching on the PowerPC," *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, pp. 19-26, Limassol, Cyprus, June 1995.

[Casmira and Kaeli 98] J. P. Casmira and D. R Kaeli, "Modeling Cache Pollution," *International Journal of Modeling and Simulation*, Vol. 19, No. 2, pp. 132-138, May 1998.

[Chen 95] T. F. Chen, "An Effective Programmable Prefetch Engine for On-chip Caches," *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pp. 237-242, Ann Arbor, MI, Nov. 29-Dec. 1, 1995.

[Chen and Baer 95] T. F. Chen and J. L. Baer, "Effective Hardware-Based Data Prefetching for High-Performance Processors," *IEEE Transactions on Computers*, Vol. 44, No. 5, pp. 609-623, May 1995.

[Cohen et. al 02] E. Cohen, H. Kaplan and U. Zwick. "Competitive Analysis of the LRFU Paging Algorithm", *Algorithmica* Vol. 33, No. 4, pp. 511-516, August 2002.

[Dahlgren and Stenstrom 95] F. Dahlgren and P. Stenstrom, "Effectiveness of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors," *Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, pp. 68-77, Raleigh, NC, January 1995.

[Deitel 90] H. M. Deitel, *An Introduction to Operating Systems*, Second Edition, Addison-Wesley Publishing Company, Inc., Reading, MA, February 1990.

[Fu et al. 92] J. W. C Fu, J. H. Patel, and B. L. Janssens, "Stride Directed Prefetching in Scalar Processors," *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 102-110, Portland, OR, December 1992.

[Gindele 77] J. D. Gindele, "Buffer Block Prefetching Method," *IBM Tech Disclosure Bulletin*, Vol. 20, No. 2, pp. 696-697, July 1977.

[Gonzales et al. 95] A. Gonzales, C. Aliagas, and M. Valero, "A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality," *Proceedings*

of the 9th ACM International Conference on Supercomputing, pp. 338-347, Barcelona, Spain, July 1995.

[Gornish and Veidenbaum 94] E. H. Gornish and A. V. Veidenbaum, "An Integrated Hardware/Software Scheme for Shared-Memory Multiprocessors," *Proceedings of the International Conference on Parallel Processing*, pp. 281-284, St. Charles, IL, June 1994.

[Johnson and Shasha 94] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," *Proceedings of the 20th International Conference on Very Large Data Bases*, pp. 439-450, Santiago de Chile, Chile, September 1994.

[Johnson et al. 97] T. L. Johnson, M. C. Merten, and W. W. Hwu, "Run-Time Spatial Locality Detection and Optimization," *Proceedings of the Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 57-64, Research Triangle Park, NC, December 1997.

[Hwang 92] K. Hwang, *Advanced Computer Architecture* (p. 231), McGraw-Hill, Inc, Columbus, OH, 1992.

[Karedla et al. 94] R. Karedla, J. S. Love, and B. G. Wherry, "Caching Strategies to Improve Disk System Performance," *IEEE Computer*, Vol. 27, No. 3, pp. 38-46, March 1994.

[Lee et al. 99] D. H. Lee, J. M. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. K. Cho, and C. S. Kim, "On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies," *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pp. 134-143, Atlanta, GA, May 1999.

[Milutinovic et al. 96] V. Milutinovic, M. Tomasevic, B. Markovi, and M. Tremblay, "A New Cache Architecture Concept: The Split Temporal/Spatial Cache," *Proceedings of 8th Mediterranean Electrotechnical Conference*, Vol. 2, pp. 1108-1111, Bari, Italy, May 1996.

[Mowry et al. 92] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 62-73, Boston, MA, October 1992.

[O'Neil et al. 93] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering," *Proceedings of the ACM SIGMOD Conference*, pp. 297-306, Washington D. C., May 1993.

[Pendse and Walterscheidt 96] R. Pendse and U. Walterscheidt, "A Fuzzy Block Replacement Algorithm for Disk Caches," *Proceedings of 1996 Midwest Symposium on Circuits and Systems,* Vol.3, pp. 995-998, Ames, Iowa, August 1996.

[Pendse and Bhagavathula 98] R. Pendse and R. Bhagavathula, "Performance of LRU Block Replacement Algorithm with Prefetching," *Proceedings of 1998 Midwest Symposium on Circuits and Systems,* pp. 86-89, South Bend, Indiana, August 1998.

[Pendse and Bhagavathula 99] R. Pendse and R. Bhagavathula, "Prefetch with the Segmented LRU Algorithm," *Proceeding of the 42nd Midwest Symposium on Circuits and Systems,* Vol. 2, pp. 862-865, Las Cruces, NM, August 1999.

[Pomerene et al. 89] J. Pomerene, T. R. Puzak, R. Rechtschaffen, and F. Sporacio, "Prefetching System for a Cache Having a Second Directory for Sequentially Accessed Blocks," *US Patent*, Patent Number: 4807110, Feburary 21, 1989.

[Silberschatz and Galvin 94] A. Silberschatz and P. B. Galvin, *Operating System Concepts*, Fourth Edition, Addison-Welsey Publishing Company, Reading, MA, 1994.

[Smith 78] A. J. Smith, "Sequential Program Prefetching in Memory Hierarchies," *IEEE Computer,* Vol. 11, No. 12, pp. 7-21, December 1978.

[Smith 82] A. J. Smith, "Cache Memories," *ACM Computing Surveys,* Vol. 14, No. 3, pp. 473-530, Septembre 1982.

[Santhanam et al. 97] V. Santhanam, E. H. Gornish, and W. C. Hsu, "Data Prefetching on the HP PA-8000," *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 264-273, Denver, CO, June 1997.

[Vanderwiel and Lilja 00] S. P. Vanderwiel and D. J. Lilja, "Data Prefetch Mechanisms," *ACM Computing Surveys*, Vol. 32, No. 2, pp. 174-199, June 2000.

[Wong and Baer 00] W. A. Wong and J. L. Baer, "Modified LRU Policies for Improving Second-Level Cache Behavior," *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pp. 49-60, Toulouse, France, January 2000.

[Zhang and Torrellas 95] Z. Zhang and J. Torrellas, "Speeding Up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching," *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 188-199, Santa Margherita Ligure, Italy, June 1995.

APPENDICES

APPENDIX A:

GLOSSARY

2Q: A frequency-based variation of LRU. Blocks in cache locate in two queues, a FIFO queue and a LRU queue. A new page is placed in the head of FIFO queue, and any re-referenced page is placed at the head of the LRU queue. At a miss, the tail of FIFO queue or LRU queue is removed.

Adaptive Prefetch: The method of adjusting the number of blocks to be prefetched into cache according to the level of spatial locality in the workload at run time.

Direct Mapping Cache: A cache placement scheme where each block in main memory can be placed in only a unique block frame in cache.

Fully Associative Cache: A cache placement scheme where each block in main memory can be placed in any of the available block frames in cache, and therefore offering the greatest flexibility in implementing block replacement algorithms for a higher hit rate.

LFU: Least Frequently Used, a memory replacement algorithm that replaces the page or block with the smallest frequency count.

LRFU: Least Recently and Frequently Used, a memory replacement algorithm that replaces the page or block with the smallest CRF value (combined recency and frequency).

LRU: Least Recently Used, a memory replacement algorithm that replaces the page or block that has not been used for the longest period of time.

LRU-K: LRU-K is to keep track of the times of the last K references to popular database pages, using this information to statistically estimate the inter-arrival times of references on a page by page basis. At a miss, the page that has the maximum Backward K-distance is replaced.

OBL:	One Block Lookahead, a method in sequential prefetching where block m+1 is prefetched when block m is accessed.
Prefetch:	The fetching of blocks into cache before processors request them.
Prefetch Coefficient Counter:	A variable in an adaptive prefetching algorithm [Pendse and Bhagavathula 98]. It is used to control the number of blocks to be prefetched.
Prefetch Counter:	A variable in an adaptive prefetching algorithm [Pendse and Bhagavathula 98]. It stands for the level of spatial locality in the current piece of the workload. It has an initial value of zero and is incremented by every hit in the references sampled at a certain rate.
Prefetch Efficiency:	The percentage of hit in the total number of prefetched blocks.
Sample Rate:	A variable in an adaptive prefetching algorithm [Pendse and Bhagavathula 98]. It is the number of references that are sampled and used to increment the prefetch counter.
Setor Mapping Cache:	A cache placement scheme where blocks in both cache and memory are partitioned into sectors, and any sector in memory can be placed in any sector frame in cache.
Set Associative Cache:	A cache placement scheme where block frames in cache are divided into sets, and a block in memory can be placed anywhere in a unique set.
S-LRU:	Segmented LRU, a frequency-based variation of LRU. The cache is divided into two segments, a probationary segment and a protected segment. Both are managed by LRU. Misses are added to the head of the probationary segment, and hits are removed from the current segment and added to the head of the protected segment. Lines in the protected segment have thus been accessed at least twice. S-LRU protects cache against access patterns that can flood an LRU cache with data that will not be reused.
Spatial Locality:	The tendency for the neighboring items of a referenced item in a program to be referenced soon.
Stall Cycle:	The clock cycle in CPU that is wasted by waiting for requested data.
Temporal Locality:	The tendency for a referenced item in a program to be referenced again soon.

APPENDIX B:

PROGRAM LISTING

```
/*******************************************************************************
 * This is a one-level cache simulator managed by the LRU replacement
 * algorithm.
 * Inputs are arrays of block size and cache size, and trace file. The hit
 * rate and number of misses of different combinations of block size and cache
 * size are given.
 ******************************************************************************/

package lru;

import java.io.*;
import java.util.StringTokenizer;

public class testLRU {
    private static int cacheNo;
    private static int blockNo;
    private static int[] blockSize = new int[5];
    private static int[] cacheSize = new int[5];
    private static double[][] hitRate = new double[5][5];
    private static int[][] numOfMiss = new int[5][5];

    public static void main(String args[]) {
        cacheNo = 0;  //number of cache sizes
        blockNo = 0;     //number of block sizes

        Caching caching = null;
        BufferedReader inFile = null;
        String address = "";

        String line = "";    // a single line in the input file
        int fileLines = 0;  //number of lines in the input file
        int usedLines = 0;  //number of processed lines in the input file

        //0: number of read in the trace
        //1: number of write in the trace
        //2: number of instruction fetch in the trace
        int[] numOfInstruction = new int[3];

        //identify the type of the instruction:
        //0: read; 1: write; 2:instruction fetch
        int label = -1;
        String tempStr = "";

        String fileName = "";

        System.out.print("This is LRU without prefetching." +
        "\n========================================" +
```

```
"\nplease input (separate input by space):" +
"\nArray of Block Size(Byte): ");
BufferedReader stdin = new BufferedReader(new
                          InputStreamReader(System.in));

try {
    line = stdin.readLine();
    System.out.println(line);
    StringTokenizer blockTokens = new StringTokenizer(line);
    for(; blockTokens.hasMoreTokens(); blockNo++)
        blockSize[blockNo] = Integer.parseInt(blockTokens.nextToken());

    System.out.print("Cache Size(Byte): ");
    line = stdin.readLine();
    System.out.println(line);
    StringTokenizer cacheTokens = new StringTokenizer(line);
    for(; cacheTokens.hasMoreTokens(); cacheNo++)
        cacheSize[cacheNo] = Integer.parseInt(cacheTokens.nextToken());

    System.out.print("File Name: ");
    fileName = "C:\\thesisCode\\myCode\\MIPS R3000\\" +
               stdin.readLine();
    System.out.println(fileName);

    for(int c=0; c<cacheNo; c++) {
        for(int b=0; b<blockNo; b++) {
            caching = new Caching(cacheSize[c], new
                          Block(blockSize[b]));

            fileLines = 0;
            usedLines = 0;
            inFile = new BufferedReader(new FileReader(fileName));

            line = inFile.readLine();
            while(line !=null) {
                fileLines++;

                StringTokenizer tokens = new StringTokenizer(line);
                if(tokens.countTokens() >= 2) {
                    usedLines++;

                    tempStr = tokens.nextToken();
                    if(label != -2) {

                        //record the number of read,write and fetch
                        label = Integer.parseInt(tempStr);
                        numOfInstruction[label]++;
                    }

                    address = tokens.nextToken();
                    caching.process(address);
                }

                line = inFile.readLine();
            } //one combination of cacheSize and blockSize is done

            inFile.close();
            label = -2; //to stop record
            hitRate[c][b] = caching.getHitRate();
            numOfMiss[c][b] = caching.getNumOfMiss();
```

```
            }  //block array is done
        } //cache array is done
    }//try
    catch(Exception e) {
        System.out.println(e);
    }

    System.out.println("Processing Result(Y is cacheSize, X is blockSize: "
    + "\nTotal Number of Lines in the File: " + fileLines +
    "\nProcessed Number of Lines in the File: " + usedLines +
    "\nPercent of read instruction: " + numOfInstruction[0]*100/usedLines +
    "\nPercent of write instruction: " + numOfInstruction[1]*100/usedLines
    + "\nPercent of fetch instruction: " +
    numOfInstruction[2]*100/usedLines );
    printHitRate();
    System.out.println();
    printNumOfMiss();
}  //end of main


//print hit rate
private static void printHitRate() {
    System.out.println("Hit Rate:\n------------------");

    //print blockSize
    System.out.print("       ");      //7 spaces
    for(int i=0; i<blockNo;i++)
        System.out.print(formatDoubleStr(Integer.toString(blockSize[i])));
    System.out.println();

    //print cacheSize and hitRate
    for(int i=0; i<cacheNo; i++){
        System.out.print(formatDoubleStr(Integer.toString(cacheSize[i])));
        for(int j=0; j<blockNo;j++)
            System.out.print(formatDoubleStr(Double.toString(hitRate[i][j])
            ));
        System.out.println();
    }
}


//print number of misses
private static void printNumOfMiss() {
    System.out.println("Number of Miss:\n------------------");

    //print blockSize
    System.out.print("       ");      //7 spaces
    for(int i=0; i<blockNo;i++)
        System.out.print(formatDoubleStr(Integer.toString(blockSize[i])));
    System.out.println();

    //print cacheSize and number of miss
    for(int i=0; i<cacheNo; i++){
        System.out.print(formatDoubleStr(Integer.toString(cacheSize[i])));
        for(int j=0; j<blockNo;j++)
            System.out.print(setStringLength(Integer.toString(numOfMiss[i][
            j]),7));
        System.out.println();
    }
}
```

```
    //format double number string to "##.#    "
    private static String formatDoubleStr(String str) {
        str += "     ";
        return str.substring(0,4) + "    ";
    }


    //set a string to a designated length len
    private static String setStringLength(String str, int len) {

        String tempStr = str;
        while(tempStr.length() < len)
            tempStr += " ";

        return tempStr;
    }

}
```

```
/*******************************************************************************
 * The class Caching used in package lru. Implement a one-level cache managed
 * by LRU.
*******************************************************************************/

package lru;

import java.util.LinkedList;

public class Caching {
    private int numBlockOffsetBits;  //number of bits in block offset
    private int size;   //byte
    private int blockSize;
    private LinkedList blocks;
    private int numOfBlock;

    private int numOfRef;
    private int numOfHit;

    /** Creates new Cache */
    public Caching(int size, Block block) {
        this.size = size;
        blockSize = block.getSize();
        blocks = new LinkedList();
        numOfRef = 0;
        numOfHit = 0;
        numOfBlock = size/blockSize;

        int temp = blockSize;
        while(temp > 1) {
            numBlockOffsetBits++;
            temp = temp >> 1;
        }

        for(int i=1; i<=numOfBlock; i++)
```

```
            blocks.add(block);
    }

    //process a CPU address request
    //address16: the address in hexdecimal
    public void process(String address16) {

        Block tempBlock = null;
        boolean cacheMiss = true;

        numOfRef++;

        int address10 = Integer.parseInt(address16,16);

        int virtualPageNo = address10 >> numBlockOffsetBits;

        for(int i=0; i<numOfBlock; i++) {
            tempBlock = (Block)(blocks.get(i));

            if(tempBlock.getTag() == virtualPageNo ) {     //hit
                cacheMiss = false;
                numOfHit++;
                blocks.remove(i);
                blocks.addFirst(tempBlock);
            }
        }

        if(cacheMiss) {
            tempBlock = fetchBlock(virtualPageNo);
            blocks.addFirst(tempBlock);
            blocks.removeLast();
        }
    }


    //fetch a block with tag as the tag number
    private Block fetchBlock(int tag) {
        return  new Block(tag, blockSize);
    }


    //get hit rate
    public double getHitRate()
    {   return ((double)(numOfHit*100))/numOfRef;          }


    //get number of hit
    public int getNumOfHit()
    {   return numOfHit;    }


    //get number of misses
    public int getNumOfMiss()
    {   return numOfRef - numOfHit; }

}
```

```
/*****************************************************************************************
 * The class Block used in package lru. Implement a block in a cache.
 *****************************************************************************************/

package lru;

public class Block {

    private int tag;
    private int size;

    /** Creates new Block */
    public Block(int size) {
        tag = -1;
        this.size = size;
    }


     /** Creates new Block */
    public Block(int tag, int size) {
        this.tag = tag;
        this.size = size;
    }


    //get tag of the block
    public int getTag()
    {   return tag;     }


    //set tag of the block
    public void setTag(int tag)
    {   this.tag = tag;  }


    //get size of the block
    public int getSize()
    {   return size;    };

}
```

```
/*******************************************************************************
 * This is a one-level cache simulator managed by adaptive prefetching and the
 * LRU replacement algorithm.
 * Inputs are arrays of block size, cache size, ratio of prefetch, and trace
 * file. The hit rate and number of misses of different combinations of block
 * size, cache size, and ratio of prefetch are given.
*******************************************************************************/

package adaptivePrefetch;

import java.io.*;
import java.util.StringTokenizer;

public class testAdaptivePrefetch {
    private static int ratioNo; //number of ratios of prefetch cache size
    private static int cacheNo; //number of cache sizes
    private static int blockNo; //number of block sizes

    //the ratio of prefetch cache, should be 2, 4, 8
    //that means 1/2, 1/4, 1/8
    private static int[] ratio = new int[5];
    private static int[] blockSize = new int[5];
    //size of the whole cache size
    private static int[] cacheSize = new int[5];

    private static double[][][] hitRate = new double[5][5][5];
    private static int[][][] numOfMiss = new int[5][5][5];

    public static void main(String args[]) {
        cacheNo = 0;
        blockNo = 0;
        ratioNo = 0;

        Caching caching = null;
        BufferedReader inFile = null;
        String address = "";

        String line = "";    // a single line in the input file
        int fileLines = 0;  //number of lines in the input file
        int usedLines = 0;  //number of processed lines in the input file

        //0: number of read in the trace
        //1: number of write in the trace
        //2: number of instruction fetch in the trace
        int[] numOfInstruction = new int[3];

        //identify the type of the instruction:
        //0: read; 1: write; 2:instruction fetch
        int label = -1;
        String tempStr = "";

        String fileName = "";

        System.out.print("This is the adaptive prefetching." +
        "\n================================" +
        "\nplease input (separate input by space):" +
        "\nArray of Block Size(Byte)(Max: 5 sizes): ");
        BufferedReader stdin = new BufferedReader(new
                                InputStreamReader(System.in));
```

```
try {
    line = stdin.readLine();
    System.out.println(line);
    StringTokenizer strTokenizer = new StringTokenizer(line);
    for(; strTokenizer.hasMoreTokens(); blockNo++)
        blockSize[blockNo] =Integer.parseInt(strTokenizer.nextToken());

    System.out.print("Cache Size(Byte)(Max: 5 sizes): ");
    line = stdin.readLine();
    System.out.println(line);
    strTokenizer = new StringTokenizer(line);
    for(; strTokenizer.hasMoreTokens(); cacheNo++)
        cacheSize[cacheNo] =Integer.parseInt(strTokenizer.nextToken());

    System.out.print("Ratio of prefetch cache(Max: 5 ratios): ");
    line = stdin.readLine();
    System.out.println(line);
    strTokenizer = new StringTokenizer(line);
    for(; strTokenizer.hasMoreTokens(); ratioNo++)
        ratio[ratioNo] = Integer.parseInt(strTokenizer.nextToken());

    System.out.print("File Name: ");
    fileName = "C:\\thesisCode\\myCode\\MIPS R3000\\" +
               stdin.readLine();
    System.out.println(fileName + "\n");

    for(int d=0; d<ratioNo; d++) {
        for(int c=0; c<cacheNo; c++) {
            for(int b=0; b<blockNo; b++) {
                caching = new Caching(ratio[d], cacheSize[c], new
                          Block(blockSize[b]));

                fileLines = 0;
                usedLines = 0;
                inFile = new BufferedReader(new FileReader(fileName));

                line = inFile.readLine();
                while(line !=null) {
                    fileLines++;

                    strTokenizer = new StringTokenizer(line);
                    if(strTokenizer.countTokens() >= 2) {
                        usedLines++;

                        tempStr = strTokenizer.nextToken();
                        if(label != -2) {
                            //record the number of read,write and fetch
                            label = Integer.parseInt(tempStr);
                            numOfInstruction[label]++;
                        }

                        address = strTokenizer.nextToken();
                        caching.process(address);
                    }

                    line = inFile.readLine();
                } //one combination of cacheSize and blockSize is done

                inFile.close();
                label = -2; //to stop record
```

```
                    hitRate[d][c][b] = caching.getHitRate();
                    numOfMiss[d][c][b] = caching.getNumOfMiss();
                    caching.printCC_Distribution();
                } //block array done
            }   //cache array done
        }   //cache ratio done
    }//try
    catch(Exception e) {
        System.out.println(e);
    }

    System.out.println("\nProcessing Result(Y is cacheSize, X is
                        blockSize:" +
    "\nTotal Number of Lines in the File: " + fileLines +
    "\nProcessed Number of Lines in the File: " + usedLines +
    "\nPercent of read instruction: " + numOfInstruction[0]*100/usedLines +
    "\nPercent of write instruction: " + numOfInstruction[1]*100/usedLines
    + "\nPercent of fetch instruction: " +
    numOfInstruction[2]*100/usedLines );
    printHitRate();
    System.out.println();
    printNumOfMiss();
}  //end of main


//print hit rate
private static void printHitRate() {
    System.out.println("Hit Rate:\n-----------------");

    for(int k=0; k<ratioNo; k++) {
        System.out.println("Ratio of prefetch cache: 1/" + ratio[k]);
        //print blockSize
        System.out.print("       ");      //7 spaces
        for(int i=0; i<blockNo;i++)
            System.out.print(formatDoubleStr(Integer.toString(blockSize[i])
            ));
        System.out.println();

        //print cacheSize and hitRate
        for(int i=0; i<cacheNo; i++){
            System.out.print(formatDoubleStr(Integer.toString(cacheSize[i])
            ));
            for(int j=0; j<blockNo;j++)
                System.out.print(formatDoubleStr(Double.toString(hitRate[k]
                [i][j])));
            System.out.println();
        }
    }
}


//print number of misses
private static void printNumOfMiss() {
    System.out.println("Number of Miss:\n-----------------");

    for(int k=0; k<ratioNo; k++) {
        System.out.println("Ratio of prefetch cache: 1/" + ratio[k]);
        //print blockSize
        System.out.print("       ");      //7 spaces
```

```
            for(int i=0; i<blockNo;i++)
                System.out.print(formatDoubleStr(Integer.toString(blockSize[i])
                ));
            System.out.println();

            //print cacheSize and number of miss
            for(int i=0; i<cacheNo; i++){
                System.out.print(formatDoubleStr(Integer.toString(cacheSize[i])
                ));
                for(int j=0; j<blockNo;j++)
                    System.out.print(setStringLength(Integer.toString(numOfMiss
                    [k][i][j]),7));
                System.out.println();
            }
        }
    }

    //format double number string to "##.#    "
    private static String formatDoubleStr(String str) {
        str += "     ";
        return str.substring(0,4) + "    ";
    }

    //set string length to designated length len
    public static String setStringLength(String str, int len) {

        String tempStr = str;
        while(tempStr.length() < len)
            tempStr += " ";

        return tempStr;
    }

}
```

```
/***************************************************************************
 * The class Caching used in package adaptivePrefetch. Implement a one-level
 * cache managed by adaptive prefetching and LRU.
 ***************************************************************************/

package adaptivePrefetch;

import java.util.LinkedList;

public class Caching {
    private int ratio;  //ratio of prefetch cache size
    private int size;   //byte, the whole size of cache
    private int blockSize;
    private int numBlockOffsetBits;  //number of bits in block offset

    private LinkedList mainBlocks;
    private int numOfMainBlocks;
    private LinkedList prefetchedBlocks;
    private int numOfPrefetchedBlocks;
```

```
private int prefetchCounter;
private int coefficientCounter;

private int numOfRef;
private int numOfHit;

//coefficientCounter distribution
//0: times of >= 5
//i: times of == i
private int[] ccDistribution;



/** Creates new Cache */
public Caching(int ratio, int size, Block block) {
    this.ratio = ratio;
    this.size = size;
    blockSize = block.getSize();

    numBlockOffsetBits = 0;
    int temp = blockSize;
    while(temp > 1) {
        numBlockOffsetBits++;
        temp = temp >> 1;
    }

    prefetchedBlocks = new LinkedList();
    numOfPrefetchedBlocks = (size/blockSize)/ratio;
    for(int i=1; i<=numOfPrefetchedBlocks; i++)
        prefetchedBlocks.add(block);

    mainBlocks = new LinkedList();
    numOfMainBlocks = (size/blockSize) - numOfPrefetchedBlocks;

    for(int i=1; i<=numOfMainBlocks; i++)
        mainBlocks.add(block);

    prefetchCounter = 0;
    coefficientCounter = 1;

    numOfRef = 0;
    numOfHit = 0;

    ccDistribution = new int[numOfPrefetchedBlocks+1];
    for(int i=0; i<numOfPrefetchedBlocks+1; i++)
        ccDistribution[i] = 0;
}


//process a CPU address request
//address16: the address in hexdecimal
public void process(String address16) {

    Block tempBlock = null;
    boolean cacheMiss = true;

    int address10 = Integer.parseInt(address16,16);

    int virtualPageNo = address10 >> numBlockOffsetBits;
```

```
//firstly search the main cache
for(int i=0; i<numOfMainBlocks; i++) {
    tempBlock = (Block)(mainBlocks.get(i));

    if(tempBlock.getTag() == virtualPageNo ) {     //hit
        cacheMiss = false;
        numOfHit++;
        mainBlocks.remove(i);
        mainBlocks.addFirst(tempBlock);

        prefetchCounter++;

        break;
    }
}

if(cacheMiss) { //cache miss in main cache
    //then search the prefetched cache
    for(int i=0; i<numOfPrefetchedBlocks; i++) {
        tempBlock = (Block)(prefetchedBlocks.get(i));

        if(tempBlock.getTag() == virtualPageNo ) {     //hit
            cacheMiss = false;
            numOfHit++;

            //remove block hit in prefetched cache
            prefetchedBlocks.remove(i);

            //put the block at the head of the main cache
            mainBlocks.addFirst(tempBlock);

            //remove the last block of the main cache
            tempBlock = (Block)mainBlocks.removeLast();

            //add the last block of the main cache into the head
            //of prefetched cache
            prefetchedBlocks.addFirst(tempBlock);

            prefetchCounter++;
            break;
        }
    }

    if(cacheMiss) { //cache miss in prefetched cache too
        LinkedList fetchedBlocks = fetchBlocks(virtualPageNo,
                                               coefficientCounter);

        //the head of fetchedBlock is the one CPU needs
        //remove it and add to the head of mainCache
        tempBlock = (Block)fetchedBlocks.removeFirst();
        mainBlocks.addFirst(tempBlock);

        tempBlock = (Block)mainBlocks.removeLast();
        fetchedBlocks.addLast(tempBlock);

        while(fetchedBlocks.size() != 0) {
            tempBlock = (Block)fetchedBlocks.removeLast();
            prefetchedBlocks.addFirst(tempBlock);
            prefetchedBlocks.removeLast();
```

```
                }
            }
        }

        numOfRef++;
        adjustCoefficientCounter();
    }


    //adjust prefetch coefficient counter
    private void adjustCoefficientCounter()
    {
        if(numOfRef % 16 == 0)  {   //interval
            if(prefetchCounter < 10 && coefficientCounter > 1)
                coefficientCounter--;
            else if(prefetchCounter > 14 && coefficientCounter <
                 numOfPrefetchedBlocks)
                coefficientCounter++;

            ccDistribution[coefficientCounter]++;

            prefetchCounter = 0;
        }
    }



    //print prefetch coefficient counter distribution
    public void printCC_Distribution()
    {
        if(numOfPrefetchedBlocks == 0) {
            System.out.println("No Prefetch!");
            return;
        }

        int sum = 0;
        for(int i=0; i<numOfPrefetchedBlocks+1; i++)
            sum += ccDistribution[i];

        System.out.println("Ratio: " + ratio + "     Cache Size: " + size +
        "\nPrefetched Cache Size (blocks): " + numOfPrefetchedBlocks +
        "\nBlock Size: " + blockSize + "     CC Sum: " + sum );

        for(int i=0; i<numOfPrefetchedBlocks+1; i++)
            System.out.print(testAdaptivePrefetch.setStringLength(Integer.toStri
            ng(i),4));
        System.out.println();

        for(int i=0; i<numOfPrefetchedBlocks+1; i++)
            System.out.print(testAdaptivePrefetch.setStringLength(
            Integer.toString(ccDistribution[i]*100/sum), 4));
        System.out.println("\n");
    }


    //fetch a block with tag as the tag number,
    //and prefetch the next num blocks
    private LinkedList fetchBlocks(int tag, int num)
    {
        LinkedList fetchs = new LinkedList();
```

```
        for(int i=0; i<=num; i++)
            fetchs.addLast( new Block(tag+i, blockSize));

        return fetchs;
    }


    //get hit rate
    public double getHitRate()
    {   return ((double)(numOfHit*100))/numOfRef;          }


    //get number of hit
    public int getNumOfHit()
    {   return numOfHit;     }


    //get number of misses
    public int getNumOfMiss()
    {   return numOfRef - numOfHit; }

}
```

```
/*******************************************************************************
 * The class Block used in package adaptivePrefetch. Implement a block in
 * cache.
 *******************************************************************************/

package adaptivePrefetch;

public class Block {

    private int tag;
    private int size;

    /** Creates new Block */
    public Block(int size) {
        tag = -1;
        this.size = size;
    }


     /** Creates new Block */
    public Block(int tag, int size) {
        this.tag = tag;
        this.size = size;
    }


    //get tag
    public int getTag()
    {   return tag;      }


    //set tag
    public void setTag(int tag)
```

```
    {    this.tag = tag;   }


    //get size
    public int getSize()
    {    return size;     };

}
```

```
/*******************************************************************************
 * This is a one-level cache simulator managed by the LRFU replacement
 * algorithm.
 * History of each block is memorized to increase the performance of LRFU.
 * Inputs are arrays of block size, cache size and lambda values, and trace
 * file. The hit rate of different combinations of block size, cache size and
 * lambda values are given.
*******************************************************************************/

package LRFU_History;

import java.io.*;
import java.util.StringTokenizer;

public class testLRFU_Heap {
    private static int lambdaNo; //number of lambda
    private static int cacheNo; //number of cache sizes
    private static int blockNo; //number of block sizes

    private static double[] lambda = new double[30];
    private static int[] blockSize = new int[5];
    //size of the whole cache size
    private static int[] cacheSize = new int[5];

    private static double[][][] hitRate = new double[30][5][5];
    private static int[][][] numOfMiss = new int[30][5][5];

    public static void main(String args[]) {
        cacheNo = 0;
        blockNo = 0;
        lambdaNo = 0;

        Caching caching = null;
        BufferedReader inFile = null;
        String address = "";

        String line = "";    // a single line in the input file
        int fileLines = 0;   //number of lines in the input file
        int usedLines = 0;   //number of processed lines in the input file

        //0: number of read in the trace
        //1: number of write in the trace
        //2: number of instruction fetch in the trace
        int[] numOfInstruction = new int[3];

        //identify the type of the instruction:
        //0: read; 1: write; 2:instruction fetch
        int label = -1;
        String tempStr = "";

        String fileName = "";

        System.out.print("This is LRFU_Heap." +
        "\n==================================" +
        "\nplease input (separate input by space):" +
        "\nArray of Block Size(Byte)(Max: 5 sizes): ");
        BufferedReader stdin = new BufferedReader(new
                                InputStreamReader(System.in));

        try {
```

```
line = stdin.readLine();
System.out.println(line);
StringTokenizer strTokenizer = new StringTokenizer(line);
for(; strTokenizer.hasMoreTokens(); blockNo++)
    blockSize[blockNo] =Integer.parseInt(strTokenizer.nextToken());

System.out.print("Cache Size(Byte)(Max: 5 sizes): ");
line = stdin.readLine();
System.out.println(line);
strTokenizer = new StringTokenizer(line);
for(; strTokenizer.hasMoreTokens(); cacheNo++)
    cacheSize[cacheNo] =Integer.parseInt(strTokenizer.nextToken());

System.out.print("lambda (Max: 30 lambdas): ");
line = stdin.readLine();
System.out.println(line);
strTokenizer = new StringTokenizer(line);
for(; strTokenizer.hasMoreTokens(); lambdaNo++)
    lambda[lambdaNo] =
          Double.parseDouble(strTokenizer.nextToken());

System.out.print("File Name: ");
fileName = "C:\\thesisCode\\myCode\\MIPS R3000\\" +
            stdin.readLine();
System.out.println(fileName + "\n");

for(int d=0; d<lambdaNo; d++) {
    for(int c=0; c<cacheNo; c++) {
        for(int b=0; b<blockNo; b++) {
            caching = new Caching(lambda[d], cacheSize[c], new
                      Block(blockSize[b]));

            fileLines = 0;
            usedLines = 0;
            inFile = new BufferedReader(new FileReader(fileName));

            line = inFile.readLine();
            while(line !=null) {
                fileLines++;

                strTokenizer = new StringTokenizer(line);
                if(strTokenizer.countTokens() >= 2) {
                    usedLines++;

                    tempStr = strTokenizer.nextToken();
                    if(label != -2) {
                        //record the number of read,write and fetch
                        label = Integer.parseInt(tempStr);
                        numOfInstruction[label]++;
                    }

                    address = strTokenizer.nextToken();
                    caching.process(address);
                }

                line = inFile.readLine();
            } //one combination of cacheSize and blockSize is done

            inFile.close();
            label = -2; //to stop record
```

```
                    hitRate[d][c][b] = caching.getHitRate();
                    numOfMiss[d][c][b] = caching.getNumOfMiss();
                    System.out.println("\nHashtable Size: " +
                    caching.getHashtableSize());
                }  //block array done
            }   //cache array done
        }   //lambda array done
    }//try
    catch(Exception e) {
        System.out.println(e);
    }

    System.out.println("\nProcessing Result(Y is cacheSize, X is
    blockSize:" +
    "\nTotal Number of Lines in the File: " + fileLines +
    "\nProcessed Number of Lines in the File: " + usedLines +
    "\nPercent of read instruction: " + numOfInstruction[0]*100/usedLines +
    "\nPercent of write instruction: " + numOfInstruction[1]*100/usedLines
    + "\nPercent of fetch instruction: " +
    numOfInstruction[2]*100/usedLines );
    printLambdaAndHitRate();

}  //end of main


//print lambda and hit rate
private static void printLambdaAndHitRate() {
    System.out.print("Hit Rate: \n----------------------\n" +
    setStringLength("lambda",15));
    for(int m=0;m<cacheNo;m++)

    System.out.print(setStringLength(Integer.toString(cacheSize[m]),7));
    System.out.println();

    for(int d=0;d<lambdaNo;d++) {
        System.out.print(setStringLength(Double.toString(lambda[d]),15));
        for(int c=0;c<cacheNo;c++)
            System.out.print(setStringLength(formatDoubleStr(Double.toStrin
            g(hitRate[d][c][0])),7));
        System.out.println();
    }
}


//format double number string to "##.#    "
private static String formatDoubleStr(String str)
{
    str += "     ";
    return str.substring(0,4) + "    ";
}


//set string length to designated length
private static String setStringLength(String str, int len) {

   String tempStr = str;
    while(tempStr.length() < len)
        tempStr += " ";

    return tempStr;
```

```
    }

}
```

```
/*******************************************************************************
 * The class Caching used in package LRFU_History. Implement a one-level cache
 * managed by the LRFU replacement algorithm with history memorized.
 ******************************************************************************/

package LRFU_History;

import java.util.LinkedList;
import java.util.Hashtable;

public class Caching {
    private int numBlockOffsetBits;  //number of bits in block offset
    private int size;   //cache size in byte
    private int blockSize;

    private LinkedList heap;    //simulate a heap
    private int heapSize;

    private int numOfRef;
    private int numOfHit;

    public static long clock;
    public static double lambda;

    private Hashtable refHistory;


    /** Creates new Cache */
    public Caching(double lambda ,int size, Block block) {

        this.lambda = lambda;
        this.size = size;
        blockSize = block.getSize();

        clock = 0;
        numOfRef = 0;
        numOfHit = 0;

        heapSize = size/blockSize;
        heap = new LinkedList();
        for(int i=0; i<heapSize; i++)
            heap.add(block);

        int temp = blockSize;
        while(temp > 1) {
            numBlockOffsetBits++;
            temp = temp >> 1;
        }

        refHistory = new Hashtable(20000);
    }
```

```
//process a CPU address request
//address16: the address in hexdecimal
public void process(String address16) {

    Block tempBlock = null;
    boolean cacheMiss = true;

    numOfRef++;
    clock++;

    int address10 = Integer.parseInt(address16,16);

    int virtualPageNo = address10 >> numBlockOffsetBits;

    Block replaced = null;

    //first search the heap
    for(int i=0; i<heapSize; i++) {
        tempBlock = (Block)(heap.get(i));

        if(tempBlock.getTag() == virtualPageNo ) {    //hit in heap
            cacheMiss = false;
            numOfHit++;

            heap.remove(i);
            tempBlock.updateCRF();
            tempBlock.updateLastRefTime();
            insertIntoHeap(heap, tempBlock);

            break;
        }
    }

    if(cacheMiss) { //cache miss in both list and heap
        tempBlock = fetchBlock(virtualPageNo);

        replaced = (Block)heap.removeFirst();
        if(replaced.getTag() != -1)
            refHistory.put(new Integer(replaced.getTag()),
            new CRF_RefTime(replaced.getCRF(), replaced.getLastRefTime()));

        //undateCRF and updapeLastRefTime are done in fetchBlock
        insertIntoHeap(heap, tempBlock);
    }
}


//insert a block into a heap
private void insertIntoHeap(LinkedList orderedList, Block newBlock) {
    int tempSize = orderedList.size();
    if( tempSize + 1 != heapSize) {
        System.out.println("Error: orderedList size != heapSize");
        System.exit(0);
    }

    boolean find = false;
    Block tempBlock = null;

    for(int j=0; j<tempSize; j++) {
```

```
            tempBlock = (Block)orderedList.get(j);
            if(newBlock.isSmallerThan(tempBlock)) {
                orderedList.add(j,newBlock);
                find = true;
                break;
            }
        }

        if(find == false)
            orderedList.addLast(newBlock);
    }


    //fetch a block with tag as the tag number
    private Block fetchBlock(int tag) {
        CRF_RefTime crfRefTime = (CRF_RefTime)refHistory.get(new Integer(tag));

        if(crfRefTime != null)
            return new Block(tag, blockSize, crfRefTime);
        else
            return new Block(tag, blockSize);
    }


    //get hit rate
    public double getHitRate()
    {   return ((double)(numOfHit*100))/numOfRef;          }


    //get number of hit
    public int getNumOfHit()
    {   return numOfHit;     }


    //get number of misses
    public int getNumOfMiss()
    {   return numOfRef - numOfHit; }


    //get hash table size
    public int getHashtableSize()
    {   return refHistory.size();    }

}




/*****************************************************************************
 * The class Block used in package LRFU_History. Implement a block in cache.
 * Additional properties lastCRF and lastRefTime are added.
 *****************************************************************************/

package LRFU_History;

public class Block {

    private int tag;
    private int size;
```

```
// CRF of last reference
private double lastCRF;
private long lastRefTime;


/** Creates new Block */
public Block(int size) {
    tag = -1;
    this.size = size;
    lastCRF = -1;
    lastRefTime = -1;
}


 /** Creates new Block */
public Block(int tag, int size) {
    this.tag = tag;
    this.size = size;
    lastCRF = F(0);
    lastRefTime = Caching.clock;
}


//a constructor using class CRF_RefTime
 public Block(int tag, int size, CRF_RefTime crt) {
    this.tag = tag;
    this.size = size;
    lastCRF = crt.getLastCRF();
    lastRefTime = crt.getLastRefTime();
    updateCRF();
    updateLastRefTime();
}


//update last reference time
public void updateLastRefTime()
{
    lastRefTime = Caching.clock;
}


//update CRF value
public void updateCRF()
{
     lastCRF = F(0) + F(Caching.clock - lastRefTime) * lastCRF;
}


//get last reference time
public long getLastRefTime()
{   return lastRefTime;      }


//get CRF value
public double getCRF()
{   return lastCRF; }


//get current CRF value
```

```
    public double getCurrentCRF()
    {   return F(0) + F(Caching.clock - lastRefTime) * lastCRF;    }


    //compare two blocks
    public boolean isSmallerThan(Block aBlock)
    {
        double left = F(Caching.clock - lastRefTime) * lastCRF;
        double right = F(Caching.clock - aBlock.getLastRefTime()) *
                       aBlock.getCRF();

        if( left < right )
            return true;

        if(left == right && lastRefTime < aBlock.getLastRefTime())
            return true;

        return false;
    }


    //weighing fuction F
    private double F( long x)
    {
        return Math.pow(0.5, Caching.lambda*x);
    }


    //get tag
    public int getTag()
    {   return tag;     }


    //set tag
    public void setTag(int tag)
    {   this.tag = tag;  }


    //get size
    public int getSize()
    {   return size;    };

}




/*****************************************************************************
 * The class CRF_RefTime used in package LRFU_History. It is used to save the
 * history information into the hash table. Compared to using class Block, it
 * saves memory space for the hash table.
 ****************************************************************************/

package LRFU_History;

public class CRF_RefTime {
    // CRF of last reference
    private double lastCRF;
    private long lastRefTime;
```

```
    //constructor
    public CRF_RefTime(double crf, long time) {
        lastCRF = crf;
        lastRefTime = time;
    }


    //get last CRF valuse
    public double getLastCRF()
    {   return lastCRF;     }


    //get last reference time
    public long getLastRefTime()
    {   return lastRefTime;     }

}
```

```
/*****************************************************************************
 * This is a one-level cache simulator managed by adaptive prefetching and the
 * LRFU replacement algorithm. History of each block is memorized to increase
 * the performance of LRFU.
 * Inputs are arrays of block size, cache size and lambda values, and trace
 * file. The hit rate and number of misses of different combinations of block
 * size, cache size and lambda values are given.
 *****************************************************************************/

package combination;

import java.io.*;
import java.util.StringTokenizer;

public class testCombination {
    private static int lambdaNo; //number of lambda
    private static int cacheNo; //number of cache sizes
    private static int blockNo; //number of block sizes

    private static double[] lambda = new double[40];
    private static int[] blockSize = new int[5];
    //size of the whole cache size
    private static int[] cacheSize = new int[5];

    private static double[][][] hitRate = new double[40][5][5];
    private static int[][][] numOfMiss = new int[40][5][5];

    public static void main(String args[]) {
        cacheNo = 0;
        blockNo = 0;
        lambdaNo = 0;

        Caching caching = null;
        BufferedReader inFile = null;
        String address = "";

        String line = "";    // a single line in the input file
        int fileLines = 0;   //number of lines in the input file
        int usedLines = 0;   //number of processed lines in the input file

        //0: number of read in the trace
        //1: number of write in the trace
        //2: number of instruction fetch in the trace
        int[] numOfInstruction = new int[3];

        //identify the type of the instruction:
        //0: read; 1: write; 2:instruction fetch
        int label = -1;
        String tempStr = "";

        String fileName = "";

        System.out.print("This is the adaptive prefetching with LRFU Heap with
        history" +
        "\n==================================" +
        "\nplease input (separate input by space):" +
        "\nArray of Block Size(Byte)(Max: 5 sizes): ");
        BufferedReader stdin = new BufferedReader(new
                                InputStreamReader(System.in));
```

```
try {
    line = stdin.readLine();
    System.out.println(line);
    StringTokenizer strTokenizer = new StringTokenizer(line);
    for(; strTokenizer.hasMoreTokens(); blockNo++)
        blockSize[blockNo] =Integer.parseInt(strTokenizer.nextToken());

    System.out.print("Cache Size(Byte)(Max: 5 sizes): ");
    line = stdin.readLine();
    System.out.println(line);
    strTokenizer = new StringTokenizer(line);
    for(; strTokenizer.hasMoreTokens(); cacheNo++)
        cacheSize[cacheNo] =Integer.parseInt(strTokenizer.nextToken());

    System.out.print("lambda (Max: 5 lambdas): ");
    line = stdin.readLine();
    System.out.println(line);
    strTokenizer = new StringTokenizer(line);
    for(; strTokenizer.hasMoreTokens(); lambdaNo++)
        lambda[lambdaNo] =
              Double.parseDouble(strTokenizer.nextToken());

    System.out.print("File Name: ");
    fileName = "C:\\thesisCode\\myCode\\MIPS R3000\\" +
                stdin.readLine();
    System.out.println(fileName + "\n");

    for(int d=0; d<lambdaNo; d++) {
        for(int c=0; c<cacheNo; c++) {
            for(int b=0; b<blockNo; b++) {
                caching = new Caching(lambda[d], 12, 10, 14,
                             cacheSize[c], new Block(blockSize[b]));

                fileLines = 0;
                usedLines = 0;
                inFile = new BufferedReader(new FileReader(fileName));

                line = inFile.readLine();
                while(line !=null) {
                    fileLines++;

                    strTokenizer = new StringTokenizer(line);
                    if(strTokenizer.countTokens() >= 2) {
                        usedLines++;

                        tempStr = strTokenizer.nextToken();
                        if(label != -2) {
                            //record the number of read,write and fetch
                            label = Integer.parseInt(tempStr);
                            numOfInstruction[label]++;
                        }

                        address = strTokenizer.nextToken();
                        caching.process(address);
                    }

                    line = inFile.readLine();
                } //one combination of cacheSize and blockSize is done

                inFile.close();
```

```
                    label = -2; //to stop record
                    hitRate[d][c][b] = caching.getHitRate();
                    numOfMiss[d][c][b] = caching.getNumOfMiss();
                    System.out.println("\nHashtable Size: " +
                    caching.getHashtableSize());
                    caching.printCC_Distribution();
                }  //block array done
            }  //cache array done
        }  //lambda array done
    }//try
    catch(Exception e) {
        System.out.println(e);
    }

    System.out.println("\nProcessing Result(Y is cacheSize, X is
    blockSize:" +
    "\nTotal Number of Lines in the File: " + fileLines +
    "\nProcessed Number of Lines in the File: " + usedLines +
    "\nPercent of read instruction: " + numOfInstruction[0]*100/usedLines +
    "\nPercent of write instruction: " + numOfInstruction[1]*100/usedLines
    + "\nPercent of fetch instruction: " +
    numOfInstruction[2]*100/usedLines );
    printLambdaHitRate();
    printLambdaNumOfMiss();
}  //end of main


//print lambda and hit rate
private static void printLambdaHitRate() {
    System.out.print("Hit Rate: \n---------------------\n" +
    setStringLength("lambda",15));
    for(int m=0;m<cacheNo;m++)
        System.out.print(setStringLength(Integer.toString(cacheSize[m]),7));
    System.out.println();

    for(int d=0;d<lambdaNo;d++) {
        System.out.print(setStringLength(Double.toString(lambda[d]),15));
        for(int c=0;c<cacheNo;c++)
            System.out.print(setStringLength(formatDoubleStr(Double.toString
            (hitRate[d][c][0])),7));
        System.out.println();
    }
}


//print lambda and number of misses
private static void printLambdaNumOfMiss() {
    System.out.print("Hit Rate: \n---------------------\n" +
    setStringLength("lambda",15));
    for(int m=0;m<cacheNo;m++)
        System.out.print(setStringLength(Integer.toString(cacheSize[m]),7));
    System.out.println();

    for(int d=0;d<lambdaNo;d++) {
        System.out.print(setStringLength(Double.toString(lambda[d]),15));
        for(int c=0;c<cacheNo;c++)
            System.out.print(setStringLength(Integer.toString(
                numOfMiss[d][c][0]),13));
        System.out.println();
    }
```

```
    }

    //format double number string to "##.#    "
    private static String formatDoubleStr(String str) {
        str += "     ";
        return str.substring(0,4) + "    ";
    }

    //set a string to designated length len
    public static String setStringLength(String str, int len) {

        String tempStr = str;
        while(tempStr.length() < len-2)
            tempStr += " ";

        return tempStr + "  ";
    }

}
```

```
/****************************************************************************
 * The class Caching used in package combination. Implement a one-level cache
 * managed by adaptive prefetching and the LRFU replacement algorithm with
 * history memorized.
 ***************************************************************************/

package combination;

import java.util.LinkedList;
import java.util.Hashtable;

public class Caching {
    private int ratio;  //ratio of prefetch cache size
    private int size;   //byte, the whole size of cache
    private int blockSize;
    private int numBlockOffsetBits;  //number of bits in block offset

    private LinkedList heap;
    private int heapSize;
    private LinkedList prefetchedBlocks;
    private int numOfPrefetchedBlocks;

    private int prefetchCounter;
    private int low;
    private int high;
    private int coefficientCounter;

    private int numOfRef;
    private int numOfHit;

    //coefficientCounter distribution
    //0: times of >= 5
    //i: times of == i
    private int[] ccDistribution;
```

```
public static long clock;
public static double lambda;
private Hashtable refHistory;



/** Creates new Cache */
public Caching(double lambda, int ratio, int low, int high, int size, Block
              block) {
    this.lambda = lambda;
    this.ratio = ratio;
    this.size = size;
    blockSize = block.getSize();

    numBlockOffsetBits = 0;
    int temp = blockSize;
    while(temp > 1) {
        numBlockOffsetBits++;
        temp = temp >> 1;
    }

    prefetchedBlocks = new LinkedList();
    numOfPrefetchedBlocks = (size/blockSize)/ratio;
    for(int i=1; i<=numOfPrefetchedBlocks; i++)
        prefetchedBlocks.add(block);

    heap = new LinkedList();
    heapSize = (size/blockSize) - numOfPrefetchedBlocks;

    for(int i=1; i<=heapSize; i++)
        heap.add(block);

    prefetchCounter = 0;
    this.low = low;
    this.high = high;
    coefficientCounter = 1;

    numOfRef = 0;
    numOfHit = 0;

    ccDistribution = new int[numOfPrefetchedBlocks+1];
    for(int i=0; i<numOfPrefetchedBlocks+1; i++)
        ccDistribution[i] = 0;

    refHistory = new Hashtable(2000);
}


//process a CPU address request
//address16: the address in hexdecimal
public void process(String address16) {

    Block tempBlock = null;
    Block replaced = null;
    boolean cacheMiss = true;

    int address10 = Integer.parseInt(address16,16);

    int virtualPageNo = address10 >> numBlockOffsetBits;
```

```
//firstly search the main cache
for(int i=0; i<heapSize; i++) {
    tempBlock = (Block)(heap.get(i));

    if(tempBlock.getTag() == virtualPageNo ) {     //hit
        cacheMiss = false;
        numOfHit++;
        heap.remove(i);
        tempBlock.updateCRF();
        tempBlock.updateLastRefTime();
        insertIntoHeap(heap, tempBlock);

        prefetchCounter++;

        break;
    }
}

if(cacheMiss) { //cache miss in main cache
    //then search the prefetched cache
    for(int i=0; i<numOfPrefetchedBlocks; i++) {
        tempBlock = (Block)(prefetchedBlocks.get(i));

        if(tempBlock.getTag() == virtualPageNo ) {     //hit
            cacheMiss = false;
            numOfHit++;
            //remove block hit in prefetched cache
            prefetchedBlocks.remove(i);
            replaced = (Block)heap.removeFirst();

            tempBlock.updateCRF();
            tempBlock.updateLastRefTime();
            insertIntoHeap(heap, tempBlock);

            //use LRU in prefetched block
            prefetchedBlocks.addFirst(replaced);

            prefetchCounter++;
            break;
        }
    }

    if(cacheMiss) { //cache miss in prefetched cache too
        //update CRF and RefTime is done in fetchBlocks
        LinkedList fetchedBlocks = fetchBlocks(virtualPageNo,
                                               coefficientCounter);

        tempBlock = (Block)heap.removeFirst();
        fetchedBlocks.addLast(tempBlock);

        //the head of fetchedBlock is the one CPU needs
        //remove it and add to the head of mainCache
        tempBlock = (Block)fetchedBlocks.removeFirst();

        //update is done in fetchBlocks
        insertIntoHeap(heap, tempBlock);

        while(fetchedBlocks.size() != 0) {
            tempBlock = (Block)fetchedBlocks.removeLast();
```

```
                prefetchedBlocks.addFirst(tempBlock);
                replaced = (Block)prefetchedBlocks.removeLast();

                if(replaced.getLastRefTime() > (-1)) {
                    refHistory.put(new Integer(replaced.getTag()),
                    new CRF_RefTime(replaced.getCRF(),
                     replaced.getLastRefTime()));
                }
            }
        }
    }

    clock++;
    numOfRef++;
    adjustCoefficientCounter();
}


// insert a block into a heap
private void insertIntoHeap(LinkedList orderedList, Block newBlock) {
    int tempSize = orderedList.size();
    if( tempSize + 1 != heapSize) {
        System.out.println("Error: orderedList size != heapSize");
        System.exit(0);
    }

    boolean find = false;
    Block tempBlock = null;

    for(int j=0; j<tempSize; j++) {
        tempBlock = (Block)orderedList.get(j);
        if(newBlock.isSmallerThan(tempBlock)) {
            orderedList.add(j,newBlock);
            find = true;
            break;
        }
    }

    if(find == false)
        orderedList.addLast(newBlock);
}


//adjust prefetch coefficient counter
private void adjustCoefficientCounter() {
    if(numOfRef % 16 == 0)   {    //interval
        if(prefetchCounter < low && coefficientCounter > 1)
            coefficientCounter--;
        else if(prefetchCounter > high && coefficientCounter <
             numOfPrefetchedBlocks)
            coefficientCounter++;

        ccDistribution[coefficientCounter]++;

        prefetchCounter = 0;
    }
}
```

```
//print prefetch coefficient counter distribution
public void printCC_Distribution() {
    if(numOfPrefetchedBlocks == 0) {
        System.out.println("No Prefetch!");
        return;
    }

    int sum = 0;
    for(int i=0; i<numOfPrefetchedBlocks+1; i++)
        sum += ccDistribution[i];

    System.out.println("Ratio: " + ratio + "    Cache Size: " + size +
    "\nPrefetched Cache Size (blocks): " + numOfPrefetchedBlocks +
    "\nBlock Size: " + blockSize + "    CC Sum: " + sum +
    "\nPrefetchCounter: low: " + low + "; high: " + high);

    for(int i=0; i<numOfPrefetchedBlocks+1; i++)
        System.out.print(testCombination.setStringLength(Integer.toString(i)
        ,4));
    System.out.println();

    for(int i=0; i<numOfPrefetchedBlocks+1; i++)
        System.out.print(testCombination.setStringLength(
        Integer.toString(ccDistribution[i]*100/sum), 4));
    System.out.println("\n");
}


// fetch a block with tag as the tag number
// and prefetch the next num blocks
private LinkedList fetchBlocks(int tag, int num) {
    LinkedList fetchs = new LinkedList();
    CRF_RefTime crfRefTime = null;

    for(int i=0; i<=num; i++) {
        crfRefTime = (CRF_RefTime)refHistory.get(new Integer(tag + i));

        if(i == 0) {
            if(crfRefTime != null)
                fetchs.addLast(new Block(tag, blockSize, crfRefTime,
                true));
            else
                fetchs.addLast(new Block(tag, blockSize, 1, clock));
        }
        else {
            if(crfRefTime != null)
                fetchs.addLast(new Block(tag + i, blockSize,
                crfRefTime,false));
            else
                fetchs.addLast(new Block(tag + i, blockSize, 0, -1));
        }
    }

    return fetchs;
}


//get hit rate
public double getHitRate()
{   return ((double)(numOfHit*100))/numOfRef;        }
```

```
    //get number of hit
    public int getNumOfHit()
    {   return numOfHit;    }


    //get number of miss
    public int getNumOfMiss()
    {   return numOfRef - numOfHit; }


    //get hast table size
     public int getHashtableSize()
    {   return refHistory.size();   }

}




/*****************************************************************************
 * The class Block used in package combination. Implement a block in cache.
 * Additional properties lastCRF and lastRefTime are added.
 ****************************************************************************/

package combination;

public class Block {

    private int tag;
    private int size;

    // CRF of last reference
    private double lastCRF;
    private long lastRefTime;


    /** Creates new Block */
    public Block(int size) {
        tag = -1;
        this.size = size;
        lastCRF = 0;
        lastRefTime = -1;
    }


    //constructor
    public Block(int tag, int size, double lastCRF, long lastRefTime) {
        this.tag = tag;
        this.size = size;
        this.lastCRF = lastCRF;
        this.lastRefTime = lastRefTime;
    }


    //constructor to use class CRF_RefTime
    public Block(int tag, int size, CRF_RefTime crt, boolean update) {
        this.tag = tag;
```

```
        this.size = size;
        lastCRF = crt.getLastCRF();
        lastRefTime = crt.getLastRefTime();
        if(update == true) {
            updateCRF();
            updateLastRefTime();
        }
    }


    //update last Reference Time
    public void updateLastRefTime() {
        lastRefTime = Caching.clock;
    }


    //update CRF
    public void updateCRF() {
        lastCRF = F(0) + F(Caching.clock - lastRefTime) * lastCRF;
    }


    //get last reference time
    public long getLastRefTime()
    {   return lastRefTime;     }


    //get last CRF
    public double getCRF()
    {   return lastCRF; }


    //get Current CRF
    public double getCurrentCRF()
    {   return F(0) + F(Caching.clock - lastRefTime) * lastCRF;    }


    //compare two blocks
    public boolean isSmallerThan(Block aBlock) {
        double left = F(Caching.clock - lastRefTime) * lastCRF;
        double right = F(Caching.clock - aBlock.getLastRefTime()) *
                       aBlock.getCRF();

        if( left < right )
            return true;

        if(left == right && lastRefTime < aBlock.getLastRefTime())
            return true;

        return false;
    }


    //weighing function F
    private double F( long x) {
        return Math.pow(0.5, Caching.lambda*x);
    }


    //get tag
```

```
    public int getTag()
    {   return tag;     }


    //set tag
    public void setTag(int tag)
    {   this.tag = tag;  }


    //get size
    public int getSize()
    {   return size;    };

}
```

```
/*****************************************************************************
 * The class CRF_RefTime used in package combination. It is used to save the
 * history information into the hash table. It saves memory space for the hash
 * table, compared to use class Block.
 *****************************************************************************/

package combination;

public class CRF_RefTime {
    // CRF of last reference
    private double lastCRF;
    private long lastRefTime;

    public CRF_RefTime(double crf, long time) {
        lastCRF = crf;
        lastRefTime = time;
    }

    public double getLastCRF()
    {   return lastCRF;     }


    public long getLastRefTime()
    {   return lastRefTime;     }
}
```

2.

VITA

Jike Cui

Candidate for the Degree of

Master of Science

Thesis: A NEW HYBRID APPROACH TO EXPLOIT LOCALITIES: LRFU WITH ADAPTIVE PREFETCHING

Major Field: Computer Science

Biographical:

Personal Data: Born in Linying, June 16, 1971, son of Jiaan Cui and Xiuzhi Lu.

Education: Graduated from Henan Experimental High School, Zhengzhou, Henan, P. R. China in July 1990. Received Bachelor of Engineering degree in Food Science from Nanjing Agricultural University, Nanjing, Jiansu, P. R. China in July 1994. Received Master of Science degree from Zhengzhou Institute of Technology, Zhengzhou, Henan, P. R. China in December 1997. Completed the requirements for the Master of Science degree in Computer Science at the Computer Science Department of Oklahoma State University in August 2003.

Experience: Teaching and Research Assistant for the Compute Science Department at Oklahoma State University from August 2000 to December 2002.

Prefessional Membership: Member of the local chapter of the ACM at OSU.