Design of Disk Cache for High Performance Computing

By

Vincent, Kwan Chi Wai

June 1995

Supervised By

Dr. Chi-Hung Chi

A Thesis

submitted in partial fulfillment of the requirements

for the Degree of Master of Philosophy

Department of Computer Science

The Chinese University of Hong Hong

QA
76.88
K83
1995
ult

Abstract

The investigation of reducing disk I/O is an important issue today because large volume of disk data are stored and retrieved frequently. The access time of a disk is usually much slower than that of the memory system. Traditional method using in the CPU cache has been applied to the disk cache and got an acceptable result. The main focus in this thesis is to design an effective caching strategy that can maximize the cache performance in *limited cache size*. The difference between CPU cache and disk cache is discussed. We also introduce new algorithms to further reduce the disk access time. Besides, a more accurate method to measure the performance of disk cache is provided.

If the disk access can *overlap* the program execution, the cache system will have time to get some sectors 'on the fly'. Traditional cache design stores all requested sectors in the cache and cannot make use of this overlapping advantage. A cache partitioning model is proposed to achieve this advantage. The cache is divided into *Branch Target Cache* and *Prefetch Buffer*. With the assist of the proposed algorithms, *Alternative Storing Sectors Technique* and *Storing Enough Sectors Technique*, the performances of our models outperform that of the standard, unified cache with prefetch on miss, by 15%-30%.

Acknowledgement

I gratefully acknowledge the support and encouragement from Dr. Chi-Hung Chi, without whom this thesis could not have been completed. I also thank my committee, Dr. Wei-Min Zheng, Dr. Gilbert Young and Dr. Ada Fu for their efforts in making thoughtful comments on this thesis.

I must thank Kevin Chan, Vico Chong and Sunny Lee for their kind help on proofreading this thesis. Thanks also go to all the friends in the department : Yung Chan, Chi-Sum Ho, Chi-Kwun Kan, Chong-Meng Lee, Keith Mak, Alywin Yu and Jeffrey Cheung. Keith and Meng provide many technical supports on the computing environment. Yung, Sum, Kan and Jeffrey have told me many interesting stories in the leisure time. Alywin has helped me to setup the computer on my desk.

Besides, I must express my sincere gratitude to my father, mother, two brothers and my girlfriend, Grace, for their support. I must give a special thank to Grace for her nice dinners and for her staying with me to work on this thesis.

Contents

Abstract i

Acknowledgement ii

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 I/O System 2

1.2 Disk Cache 4

1.3 Dissertation Outline 5

2 Related Work 7

2.1 Prefetching 7

2.2 Cache Partitioning 9

2.2.1 Hardware Assisted Mechanism 9

2.2.2 Software Assisted Mechanism 10

2.3 Replacement Policy 12

2.4 Caching Write Operation 13

2.5 Others 14

2.6 Summary 15

3 Methodology and Models 17

3.1 Performance Measurement 17

3.1.1 Partial Hit 17

3.1.2 Time Model . . . 17
3.2 Terminology . . . 19
3.2.1 Transfer Block . . . 19
3.2.2 Multiple-sector Request . . . 19
3.2.3 Dynamic Block, Heading Sectors and Content Sectors . . . 20
3.2.4 Heading Reuse and Non-heading Reuse . . . 22
3.3 New Models . . . 23
3.3.1 Unified Cache with Always Prefetch . . . 24
3.3.2 Partitioned Cache: Branch Target Cache and Prefetch Buffer . . . 25
3.3.3 BTC + PB with Alternative Storing Sector Technique . . . 29
3.3.4 BTC + PB with ASST Applying to Dynamic Block . . . 34
3.3.5 BTC + PB with Storing Enough Head Technique . . . 35
3.4 Impact of Block Size . . . 38

4 Trace Driven Simulation **41**
4.1 Simulation Environment . . . 41
4.2 Two Kinds Of Disk . . . 43
4.3 Control Models . . . 43
4.3.1 Model 1: No Cache . . . 43
4.3.2 Model 2: Unified Cache without Prefetch . . . 44
4.3.3 Model 3: Unified Cache with Prefetch on Miss . . . 44
4.4 Two Comparison Standards . . . 45
4.5 Trace Properties . . . 46

5 Performance Evaluation of Common Disk **54**
5.1 The Effect Of Cache Size . . . 54
5.1.1 Trends of Absolute Reduction in Time . . . 55
5.1.2 Trends of Relative Reduction in Time . . . 55
5.2 The Effect Of Block Size . . . 68
5.2.1 Trends of Absolute Reduction in Time . . . 68
5.2.2 Trends of Relative Reduction in Time . . . 73
5.3 The Effect Of Set Associativity . . . 77

5.3.1 Trends of Absolute Reduction in Time 77
5.4 The Effect Of Start-up Time C1 79
5.4.1 Trends of Absolute Reduction in Time 80
5.4.2 Trends of Relative Reduction in Time 80
5.5 The Effect Of Transfer Time C2 83
5.5.1 Trends of Absolute Reduction in Time 83
5.5.2 Trends of Relative Reduction in Time 83
5.5.3 Impact of C2=0.5 on Cache Size 86
5.5.4 Impact of C2=0.5 on Block Size 87
5.6 The Effect Of Prefetch Buffer Size 90
5.7 Others 93
5.7.1 In The Case of Very Small Cache with Large Block Size 93
5.7.2 Comparing Performance of Model 6 and Model 7 94
5.8 Conclusion 95
5.8.1 The Number of Actual Sectors Transferred between Disk and Cache . 95
5.8.2 The Efficiency of Our Models on Common Disk 96

6 Performance Evaluation of High Performance Disk 98
6.1 Difference Between Common Disk And High Performance Disk 98
6.2 The Effect Of Cache Size 99
6.2.1 Trends of Absolute Reduction in Time 99
6.2.2 Trends of Relative Reduction in Time 99
6.3 The Effect Of Block Size 103
6.3.1 Trends of Absolute Reduction in Time 105
6.3.2 Trends of Relative Reduction in Time 105
6.4 The Effect Of Start-up Time C1 110
6.4.1 Trends of Relative Reduction in Time 110
6.5 The Effect Of Transfer Time C2 110
6.5.1 Trends of Relative Reduction in Time 112
6.5.2 Impact of C2=0.5 on Cache Size 112
6.5.3 Impact of C2=0.5 on Block Size 116

6.6 Conclusion 117

7 Conclusions and Future Work **119**

7.1 Conclusions 119

7.2 Future Work 122

Bibliography **123**

List of Tables

3.1 The Un-stored Sectors for C2=2 and C2=1.5 32

4.1 Number of Requests for the Four Traces 47

4.2 Number of Sectors per Request 48

4.3 Frequency of Displacement 48

4.4 Frequency of Dynamic Block Size for Access 49

4.5 Frequency of 1-sector Dynamic Block Size and Request 50

4.6 Frequency of the Largest and Most Frequent Dynamic Block 51

4.7 Ten Topmost Largest I/O Percentage of Dynamic Blocks 52

5.1 Maximum and Minimum Relative Performance of Model 4 (C2 = 1.5) 58

5.2 Maximum and Minimum Relative Performances of Model 7 (C2 = 1.5) 62

5.3 Maximum and Minimum Relative Performances of Model 7 (Prefetch Time = 0) 64

5.4 Maximum Relative Performance of Model 7 65

5.5 Hit Ratio and Disk Access Time Ratio for Model 7 72

5.6 Actual Number of Sectors Transferred for Dbase when Varying Block Size . . 73

5.7 Behavior of Model 2 in 1M Cache Size for Excel Trace 94

5.8 Absolute Performance of Varying Cache Size of Model 6 and Model 7 95

5.9 Actual Number of Sectors Transferred between Disk and Cache 96

6.1 Relative Performance of Model 7 for two kinds of disks 102

List of Figures

1.1 I/O System Model 2

2.1 Logical flow of SLRU cache lines 13

3.1 Multiple-sector Request 19

3.2 Formation of Dynamic Block 20

3.3 Heading Sectors and Content Sectors 22

3.4 Two Different Kinds of Reuse 23

3.5 Flow Chart of Model 4 25

3.6 Partitioned Cache: BTC + PB 26

3.7 Flow Chart of Model 5 27

3.8 Problem of Storing First Block in BTC 28

3.9 Alternative Storing Sector Technique (ASST), C1=3, C2=2 29

3.10 Flow Chart of Model 6 and Model 7 33

3.11 Applying ASST to Multiple-sector Request and Dynamic Block (C1=5, C2=2) 34

3.12 Storing Some Heading Sectors for Each Request 35

3.13 Problem of Storing Starting Head of Each Request to 1-sector Requests . . . 36

3.14 Storing Enough Head Technique (SEHT), C1=10, C2=2 37

3.15 Flow Chart of Model 8 39

4.1 Flow Chart of Model 1 44

4.2 Flow Chart of Model 2 45

4.3 Flow Chart of Model 3 46

5.1 Absolute Performance of Varying Cache Size 56

5.2 Relative Performance of Varying Cache Size (without Model 5) 57

5.3 Trend of Relative Performance of Model 7 and Model 8 59
5.4 Killing of Correct Prefetch 61
5.5 Relative Performance of Varying Cache Size with Prefetch Time=0 63
5.6 Relative Performance of Varying Cache Size 67
5.7 Absolute Performance of Varying Block Size 69
5.8 A 4-sector Request in Block Size of 4 sectors 70
5.9 A 8-sector Request in Block Size of 8 sectors 71
5.10 Relative Performance of Varying Block Size (without Model 5) 74
5.11 Relative Performance of Varying Block Size 76
5.12 Absolute Performance of Varying Set Associativity 78
5.13 Absolute Performance of Varying Start-up Time C1 81
5.14 Relative Performance of Varying Start-up Time C1 82
5.15 Absolute Performance of Varying Transfer Time C2 84
5.16 Relative Performance of Varying Transfer Time C2 85
5.17 Absolute Performance of Varying Cache Size when C2=0.5 87
5.18 Relative Performance of Varying Cache Size when C2=0.5 88
5.19 Absolute Performance of Varying Block Size when C2=0.5 91
5.20 Relative Performance of Varying Block Size when C2=0.5 92
5.21 Absolute Reduction in Time of Varying Prefetch Buffer Size 93

6.1 Absolute Performance of Varying Cache Size 100
6.2 Relative Performance of Varying Cache Size (without Model 5) 101
6.3 Relative Performance of Varying Cache Size 104
6.4 Absolute Performance of Varying Block Size 106
6.5 Relative Performance of Varying Block Size (without Model 5) 107
6.6 Relative Performance of Varying Block Size 109
6.7 Relative Performance of Varying Start-up Time C1 111
6.8 Relative Performance of Varying Transfer Time C2 113
6.9 Absolute Performance of Varying Cache Size when C2=0.5 114
6.10 Relative Performance of Varying Cache Size when C2=0.5 115
6.11 Absolute Performance of Varying Block Size when C2=0.5 116

6.12 Relative Performance of Varying Block Size when C2=0.5 117

Chapter 1

Introduction

Relative speed gap between the disk and the main memory is increasing. With increasing computational power of the CPU and with the trend for parallel processing, the rate of data consumption is getting higher. The size of the data set that a program operates on is also increasing. For example, in query applications, hundreds of gigabytes of data are scanned to determine the answer. Even worse, sometimes slower disks are used because of the cost. All these are reflected by the following facts:

- For high-end servers and high performance computers, the increasing rate for MIPS is about 10%-20% while the increasing rate for DASD (direct address device, e.g. disk) is about 40%-60%.
- For high end data servers and enterprise systems, about 60%-70% of the system cost is spent on DASD.
- Disk caching is attracting more and more attentions.

It is expected that the system performance bottleneck will be on I/O instead of the computing power of processors. This is because microprocessor speed has been increasing at an extremely fast rate. For example, over 100 MIPS processors are very common. It is expected that processors with throughput of 300-500 MIPS will be available in the coming two years. On the other hand, although the size of disk storage is increasing greatly in this few years, the disk access time does not have any great

breakthrough. Therefore, the speed of the disk cannot catch up the fastly increasing CPU speed, and makes disk I/O become a performance bottleneck.

1.1 I/O System

Magnetic disk has its advantages to store information. It has a large capacity. It is significantly cheaper than higher performance alternatives and provides permanent storage. On the other hand, access time of disk is slow when compared with the rest of the computer system. Disk is slower than DRAM (Dynamic Random Access Memory) in both access time and transfer time. Disk is a result of economic consideration and limited current technology. If fast, reliable and cheap storage were available, slow disk access would have disappeared.

Figure 1.1: I/O System Model

Figure 1.1 shows the general relationship between a running process in memory, the disk cache and the disk storage. The number of disk I/O requests from the process is r and w, and it depends on the nature of the application. The r' and w' are the actual disk I/O requests between the disk cache and the disk. Reducing the values of r' and w' means that fewer data are needed to be transferred between the disk cache and the disk storage. Since the disk is a slow device, reducing the traffic can increase

the overall performance.

There are two parameters that determine the impact of disk I/O on overall system performance. One is the amount of time waiting for each I/O to complete, and the other is the number of disk I/O operations performed. If a process performs a large number of disk I/O and the time to complete an I/O is long, the performance of the overall system must be poor no matter how good the performance of other parts of the system is. The role of the disk cache is to reduce the number of disk I/O, so the average disk access time decreases.

The disk access time can be estimated as

$$Disk\ access\ time = C1 + C2 * n$$

where *C1* is the start-up time of the disk which includes the seek time and rotational delay; *C2* is the disk transfer time, i.e. the time to transfer one sector; and n is the number of sectors in a request. This equation is, in fact, not accurate for an individual request. However, over a large sample, this equation can be considered as average approximation of the time of each request.

Using more arms and disk platter can reduce the start-up time and the transfer time. Studies [Ng91, MK89] show that using two arms can reduce the average rotational latency to one-fourth of a revolution, instead of a half revolution. Synchronously interleaving of data across several disks can also reduce the transfer time [Kim86, GMS88]. The technology of Redundant Arrays of Inexpensive Disks (RAID) provides a high performance and very reliable way to stripe data across multiple disks [PGK88, CGK+88, KOP+89]. It can also increase the overall reliability because of the cost-effective redundant feature.

Many techniques have been developed to improve the disk access time, but they cannot overcome the fact that disk is still slower than the rest of the computer system. Disk I/O is still the bottleneck of the overall system performance.

1.2 Disk Cache

Disk cache serves as a buffer between the process in memory and the physical device. The earliest study about disk cache was reported in 1968 [SV68]. All the disk requests go to the cache system first and the cache system will decide how to process them. If the requested sectors are found in cache, no physical I/O operation is performed. This also means that the average access time of I/O is reduced. Disk cache also reduces the amount of time that a process spends on disk accessing.

All modern operating systems use disk cache to reduce disk accesses. For instance, Unix has an inode buffer to cache the (inode and directory) information of files and has a buffer cache to cache data in files [Bac86]. Cache can be located in three possible locations: a cache in host, a cache in the device storage controller, or a cache in the drive itself.

Current disk caching techniques are not as good as we want. Most of current research efforts on caching are on on-chip caching and memory chip design. Old techniques such as hardware oriented one block/sector look ahead are still being used in controlling disk cache. However, in on-chip cache design, new techniques have already been developed to improve cache performance by a significant amount, e.g. cache partitioning, software assisted data prefetching, smart hardware oriented data prefetching, etc. Furthermore, traditional disk cache design often emphasizes on cache hit rate for one level in the memory hierarchy. Very little effort has been spent to study the performance of disk cache in terms of the execution time on a multi-level memory hierarchy system.

The hit ratio of disk cache is generally 70%-90%. It is very low when compared with the hit ratio (>90%) in memory cache. This difference is due to the poor disk cache management to capture the locality of disk access. In this thesis, we will examine new caching algorithms which are expected to produce a better performance and to use the disk cache in a more efficient way.

Our project aims to design a general disk cache strategy that is simple and cheap

enough to be incorporated in hardware, and can fully utilize the limited cache size. The main idea is that by proper overlapping data fetching and program execution, some sectors can be discarded and so they are not stored in the cache. When these un-stored sectors are requested, they can be retrieved during program execution. So, this method can effectively enlarge the cache size. To achieve this propose, we have incorporated a cache partitioning and newly designed algorithms to build a disk cache architecture. The basic contributions of our research are summarized as follows:

- Discover that a highly sequential property has been exhibited in the traced data.
- Discover that always prefetch performs much better than currently used prefetching mechanisms in disk cache design such as prefetch on miss.
- Design a disk cache architecture based on CPU cache partitioning technique and new algorithms to control the disk cache in order to further improve the performance of always prefetch.
- Build a simulator to investigate the performance characteristics of the designed disk cache models in various kinds of disks.
- Compare the pros and cons of using the designed disk cache architecture and algorithms to enhance the performance of disk I/O.
- The designed models perform much better than the traditional ones in a wide range of cache size.

1.3 Dissertation Outline

The outline of the rest of the thesis is as follows:

- Chapter 2 describes the previous work on cache design.
- Chapter 3 gives a detail description of the newly proposed models, including the disk cache architecture and the control algorithms.

- Chapter 4 describes the trace driven simulation environment and analyzes the trace data in detail. Highly sequential property of the traced data is shown in the analysis.

- Chapter 5 evaluates the performances of our proposed models for common disks. Various simulation results are presented with detail discussion of the pros and cons of different models.

- Chapter 6 evaluates the performances of our proposed models for high performance disks.

- Chapter 7 gives the conclusion and suggests future extension of our work.

Chapter 2

Related Work

Since cache performance depends on a wide range of design and implementation parameters, many efforts have been paid to study the impact of these parameters on processor caches [Smi82, Prz90]. Disk cache is generally much larger, in volume, than CPU cache and often has a lower hit rate but many of the same design principles apply.

A comprehensive study in disk cache was discussed in [Smi85]. Trace driven simulation is used to show, among other results, that cache sizes on the order of 8 Mbytes can service 80%-90% of all disk requests [Smi85]. A simple prefetching strategy is also explored to load block i+1 into the cache when block i is referenced, but concludes that it is not uniformly effective for all types of files. Besides, it proposed to perform intelligent prefetching based on the user types (system, interactive, batch) and the file types (temporary, system, paging). However, it is difficult to incorporate Smith's suggestion in many cache designs due to the requirement of either analyzing the program or accepting user advices in advance.

2.1 Prefetching

Prefetching is a very common method that incorporates into a cache design. Prefetching means fetch before actual reference. This method fetches some sectors before the sectors are actually referenced. Therefore, if the prefetched sectors are actually referenced, it will reduce the time of a process to wait for accessing the disk, and so will increase the execution speed of a process. However, if the prefetched sectors are not be

referenced, they will still occupy the cache space. Then cache is *polluted* because they kick out some useful data. Therefore, the usefulness of prefetching highly depends on whether the mechanism can capture the access property of the requested data. Much work has been done to improve the hit rate of prefetching. *Prefetch on miss*, *prefetch on hit* and *always prefetch* are three commonly known prefetching methods [Smi82]. Prefetch on miss triggers prefetching action when there is cache miss. Prefetch on hit triggers prefetching action when there is cache hit. Always prefetch triggers prefetching action when there is a request. For currently used disk caches, they only incorporate prefetch on miss as their prefetching mechanism. They sometimes do not incorporate any explicit prefetching mechanisms but they rely on the implicit prefetching property of a large block size. For instance of using a large block size, a request of one sector will let a whole disk track to be loaded in the cache.

The traditional prefetching mechanism is One Block Look-ahead (OBL). It loads sector i+1 into the cache when the sector i is referenced [Smi85]. This prefetching algorithm is based on spatial locality. However, in modern file system design, the data may not be continuous on disk, i.e. fragmentation may occur. Therefore, OBL may not be a good mechanism to capture spatial locality, i.e. sector i+1 may not contain data that continue from sector i. This is the problem of the logical continuity not matching the physical continuity. To solve this problem, maintaining an adaptive table of most probable successors for each disk block was proposed [GAN93]. Each successor is tagged with a weight which indicates the likelihood that it will be referenced given that its parent is referenced. This table and associated weights are used to control the prefetch mechanism. Unlike sequential prefetching, this algorithm functions well when logically successive disk blocks are not physically adjacent on the disk. However, the overhead of the large size of the adaptive table may make the algorithm impractical.

An adaptive prefetch design based on the run-time caching statistics of files was proposed [SLO90]. Cache hit histories that are produced by prefetching are used as a measure of the file access sequentialities and are used to determine the dynamic prefetching length. More disk blocks are prefetched for transition that has a tendency

to produce additional cache hits. Prefetch lengths are reduced for transition that generates poor cache hit histories. This method can improve the cache utilization and prefetch efficiency. However, to implement this function, a prefetch id is needed to attach each data block to represent the prefetch type, demand or prefetch, of the block. This may involves design overheads for maintaining the prefetch id and screening various cache hits.

2.2 Cache Partitioning

Prefetching is a very useful technique because it can reduce the disk access time. On the other hand, it has a tradeoff that the prefetched data flush out the original data, i.e. the previously captured access pattern. Besides, some prefetched data may be useless but they still flush out the data in the cache. For instance, after a huge sequential access like playing an animation, whole cache will be occupied by the prefetched data of the animation and the prefetched data are usually useless afterward. Then the cache will be like in a cold start situation. To eliminate the problem of previously captured access pattern being flushed out, cache partitioning technique has been proposed. The basic idea of cache partitioning is that the prefetched data should be placed in a separated buffer so that they cannot affect the previously captured access pattern.

2.2.1 Hardware Assisted Mechanism

CPU cache also has this kind of prefetch problem when there is a loop accessing a large array. To eliminate this problem in CPU cache in hardware and to preserve the advantage of prefetching, a small fully-associative cache [Jou90], was presented to improve the CPU cache performance. There are three methods to fill the small cache: miss caching, victim caching and stream buffers. For miss caching, the data store in both the original cache and the small cache on cache miss. If data are replaced in the original cache but can find in the small cache, it can still provide a faster response to the requests. For victim caching, it stores the data which are flushed out from the

original cache. Simulation shows that the small cache needs only 1 to 5 entries to effectively remove conflict misses. The stream buffer stores the *prefetched* data on a cache miss. It is operated in a FIFO way. Simulation shows that the stream buffer can reduce 72% of instruction cache misses and 25% of data cache misses.

Branch target cache/buffer is another product of CPU cache partitioning. A branch target cache/buffer can reduce the performance penalty of branches in pipelined processors by predicting the path of the branch and caching information used by the branch. Two issues are needed to be solved: a branch resolution scheme to decide the direction and target of a branch early in the pipeline, thus allowing target instruction fetch to start, and mechanisms to minimize the impact of unpredictable branches. Many efforts have been paid to study the branch target cache [DA95, CG94, Gon94, PS93, PSR92, BF91].

Branch target cache/buffer has been widely used in CPU cache design. For instance, Intel Pentium and Am29000 CPU have already incorporated this technique for instruction reference only. Branch target cache is used to store the first block of the non-sequential reference. Therefore, for each branch, the next instruction is probably in the branch target cache. The execution time is then reduced. This technique has been really incorporated in the CPU cache design but it has rarely been considered to be used in disk cache design.

2.2.2 Software Assisted Mechanism

Most caches are controlled by hardware technology. However, hardware has very little, or even no, information about a running process. This may lead to inefficiently utilize the cache. If more information is required, expensive and complicated hardware is needed. Therefore, software-assisted mechanism becomes another aspect in cache design. Since software has more information about the program execution, it is very suitable to act as a guide to control the cache.

A small prefetch buffer to support the software-assisted prefetching was proposed in CPU cache [KL91, CMCH91]. Simulation shows that this approach greatly improves

the cache performance. Therefore, placing the data in a small buffer does not affect the effectiveness of prefetching. Using a small buffer to store the prefetched data is commonly used in CPU cache. However, it has rarely been considered to be used in disk cache.

For currently used disk caches, they generally use an unified cache approach. However, analog to the concept of software-assisted mechanism in CPU cache, disk cache can also use the *operating system* to provide more information for cache control. The most typical information is the file access statistics. Many cache partitioning algorithms have been proposed according to the file usage and they usually give satisfactory results.

An adaptive algorithm was proposed to partition a fully associative disk cache that were shared by several identifiable processes [TSW92]. The partition algorithm alters the cache size dynamically in response to changes in the access pattern — the miss rate of each process. The partitioning model is evaluated on a trace of 1.6 million disk I/O accesses directed to 13 physical disks sharing one cache and its associated cache controller. The partitioned cache performs slightly better than the unified cache by 1% to 2% increase in hit ratio. A queueing network model is set up and shows that such the 1% to 2% increase in hit ratio can provide a significant decrease in disk response time in a system with a heavy throughput of I/O requests.

An attribute cache is another kind of cache partitioning [RF93, Ric94]. The attribute cache uses the workload characteristics (the file access pattern) to determine the appropriate cache configuration for a given cache size. It captures the statistically distinct behavior of the workload. The attribute cache is divided into various parts that is efficiently tailored to different types of files such as inodes, directories, executable and data. The trace driven simulation shows that Inodes and Directories occupy 80% of all file requests. They are small and have a highly temporal access pattern. A significant amount of space should be allocated to capture these requests. This portion of the cache should have small blocks to effectively capture the temporal

nature. Small executables and datafiles have temporal access patterns but large executables and datafiles have sequential access patterns. A temporal subcache which has small block size is assigned to small executables and datafiles. A sequential subcache which has large block size is assigned to large files. The attribute cache can reduce the miss ratio by 25%-60% depending on the cache size when comparing with a UNIX style cache.

2.3 Replacement Policy

Replacement policy is another important issue in cache design. The most commonly used policy is least recently used (LRU) algorithm. LRU algorithm replaces the entry that has not been used in the longest time. LRU algorithm is simple and easy to implement in general. Although it is a commonly used policy, it may not be suitable in all situations. For example, LRU treats fetched data and prefetched data as the same weight so that *many useless prefetched data flush out useful data.* Therefore, variations of LRU algorithm are proposed to tackle this problem.

LRU algorithm has a problem that the cache can be occupied by lines that are accessed only once, flushing out lines that have a higher probability of being reused. Segmented Least Recently Used (SLRU) is proposed to eliminate this problem [KLW94]. SLRU cache is divided into two segments: a probationary segment and a protect segment. Probationary segment holds the sectors that cause cache misses. When a sector in the probationary segment is referenced again, it will be transferred to the protect segment. Therefore, the protect segment holds only the sectors that were referenced twice or more times. This can prevent those sectors, that were referenced once, flush out all the data in the cache. The LRU line of protect segment will transfer to the most resently used (MRU) space of the probationary segment if the protect segment needs space to store new data. The structure of SLRU cache is shown in Figure 2.1 from [KLW94].

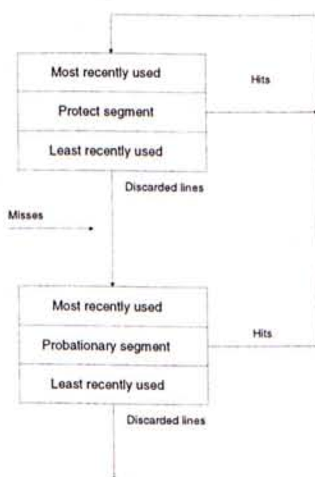


Figure 2.1: Logical flow of SLRU cache lines

2.4 Caching Write Operation

Write operation is different from read operation because improper treatment of write may cause inconsistency and loss of data. However, caching write operations can let the process to run more smoothly because it does not need to wait the I/O to complete. There are many kinds of write operations such as write back and write through. Write back means that the written data are placed in the cache and will be written to the disk later. Write through means that the written data are immediately written to disk. Write back usually gives a better performance than write through. However, write back cache may cause data loss in system failure. Therefore, protection mechanism such as using SRAM (Static RAM) instead of DRAM to store the data was proposed to prevent data loss.

Write-only disk cache was proposed to optimize the write operation [SO90, OS92]. A single surface write-only disk cache model [SO90] was developed to approximate a system with small amounts of disk cache or a system of parallel transfer disks where all read/write heads can simultaneously transfer data. A multi-surface model [OS92] was developed to study large disk cache systems and investigate the interference between conflicting sectors on different disk surfaces during disk transfers. Write-only disk cache

has an advantage that writes can be performed at little or no cost by piggybacking them on reads.

Periodic update write policy [CS92] is widely used in existing computer systems. It writes dirty cache blocks to disk on a periodic basis. The periodic update concept arises from a need to balance the generation of I/O traffic with the potential loss of data due to system failure. The response time for read operations under the periodic update policy was compared with that using write through policy in [CS92]. It concluded that the performance of periodic update write policy is better only if the cache system has achieved a sufficiently high write hit ratio.

Redundant disk arrays are gaining increased attention as a feasible I/O organization because it is cheap and reliable. In these I/O systems, reads and writes have different performance impacts to the systems [Red92]. When any data are written to the disk system, the corresponding parity information needs to be updated on the disks. To update the parity information, we need to read the old version of the data, XOR this with the old parity information on the disk and the new data, and store the new parity onto the disk. So, one write request results in four I/O operations. Hence the writes to the system cause significant overload on the system. An deeply analysis of read/write characteristics of the I/O workloads was presented in [Red92].

2.5 Others

Hit ratio is a commonly used indicator to the performance of a disk cache but it may be contrasted against the cache overhead. Cache overhead may reduce the performance gain, or may even lead to a poor performance. Therefore, using cache overhead to calculate the lower bound of hit ratio was proposed [Hos92]. Besides, analyzing an I/O tracing could also calculate the average hit ratio [Hos92].

The performance of disk cache was studied in fileserver based distributed computer systems [ME90]. Cache in distributed systems involves additional design decisions due

to the presence of both workstation and fileserver caches. Disk cache replacement policies for network fileservers were also studied [WEB93] and showed that the common least recently used (LRU) policy, which is known to work well on standalone disked workstations and at client workstations in distributed systems, is inappropriate at a fileserver. Simple frequency based approaches, e.g. least frequently used (LFU) algorithm, do better. If the frequency based policy takes file type into account, it can offer additional improvements.

2.6 Summary

Prefetching is an attractive method to improve the cache performance. Many kinds of prefetching, such as prefetch on miss, prefetch on hit, always prefetch and adaptive prefetch by access pattern, have been proposed. However, prefetching may bring some undesirable data into the cache and flush out useful data. In CPU cache, cache partitioning technique has been used to solve this problem. In disk cache, an unified cache approach is generally used and there is very little consideration in cache partitioning. Besides, cache partitioning can be used to capture different access pattern. For instance, we can partition the disk cache according to different types of file, such as directory, data, executable,..., etc.

Cache replacement policy mainly uses Least Recently Used (LRU) algorithm. Other algorithms, such as Segmented Least Recently Used (SLRU) and frequency variation of LRU, have been proposed to increase the efficiency. Those algorithms are mainly based on the ordinary LRU algorithm.

Write policies of disk cache have been examined. Some common ones are periodic write, write back and write through. Redundant disk arrays have been investigated to provide a more reliable and cheap storage environment. Moreover, disk cache has also been studied in different platforms, such as distributed system, fileserver and workstation.

Old techniques, such as unified cache approach and prefetch on miss mechanism,

have been applied to the currently used disk caches. Some disk caches even do not have any prefetching mechanisms but use the implicit prefetching property of large block to act as a substitute. In CPU cache, many new techniques have been developed to improve the cache performance on those old techniques.

In this project, always prefetch is chosen as the basis of the new models. Cache partitioning technique, similar to that in CPU cache, will be used to overcome the problem of prefetching. Prefetched data have less chance to flush out useful stored data in cache now. This aspect has seldom been considered in previous studies. Since algorithms to control the partitioned disk cache have rarely been discussed, new policies will also be designed to control the partitioned disk cache.

Chapter 3

Methodology and Models

3.1 Performance Measurement

3.1.1 Partial Hit

Hit/Miss model is commonly used to justify the effectiveness of a cache design. A request to cache is a hit if the referenced sectors are in cache. Otherwise, it is a miss. However, due to *cache prefetching*, the definition of miss becomes ambiguous. There are situations where the demanded sectors are being prefetched from the disk to the cache but the transfer is not finished yet. We cannot count this situation as cache hit because it needs to pay time penalty. Also, we cannot count this as cache miss because the time penalty needed to pay is less than the cache miss penalty. Therefore, the concept of partial hit is introduced to describe this kind of situation.

Partial hit means that the requested sectors are being prefetched from the disk, i.e. the sectors are coming on the way. The occurrence of partial hit is due to the slow data bus speed and the limited bandwidth. Since the penalty of a partial hit is not constant, we need a time model to accurately measure the disk performance.

3.1.2 Time Model

Accurate modeling of the disk access is a key point to analyze result in simulation. Using time model can eliminate the fuzziness of the concept of partial hit and can provide a clear and accurate way to show the performance.

Assumption:

When a request of multiple sectors is issued, the process can use the transferred sectors while the system can transfer the next sectors asynchronously. This assumes the overlapping between program execution and data fetching.

Let

the time to consume one sector be *Tu*; the time between sending out a request and transferring the first sector from the disk be *C1*; the transfer time per sector be *C2*.

For one request of *N* consecutive sectors, the transfer time of these all sectors from disk to memory is $C1 + C2 * N$.

The *total time to transfer and use up all sectors* in one request without any disk cache is

$$Total\ Time\ per\ Request_{no\ disk\ cache} = C1 + C2 * N + Tu * N$$

where the *Time* is the total access time including transferring and consuming all the requested sectors.

However, when there is disk cache, the time can potentially be much smaller. Consider the case when there is cache hit, *C1* and *C2* can be eliminated. Since the time to transfer data from disk cache to memory is so small that can be neglected, we have

$$Total\ Time\ per\ Request_{cache\ hit} = Tu * N$$

Using a time counter, we can accurately get the time to indicate the situation of partial hit. The total time of a partial hit is between $Tu * N$ and $C1 + C2 * N + Tu * N$.

For convenience, we set *Tu* to 1 and all values of *C1* and *C2* are normalized with the actual consumption time *Tu*. This normalization is only used to simplify the simulation and without any loss of generality. In our simulation, we use the *disk access time* to measure the performance of different models.

3.2 Terminology

3.2.1 Transfer Block

A *transfer block* is the basic unit of data transfer between disk and disk cache. It can contain multiple sectors. For example, a disk is designed to transfer 4-sector block at one time. If there is a request just for one sector, the whole 4-sector transfer block will be transferred. If there is a request of 2 sectors and these 2 sectors map to *one* transfer block, only 1 transfer block (4 sectors) will be transferred. However, if these 2 sectors map to *two* continuous transfer blocks, the two transfer blocks (8 sectors) all need to be transferred. For disk I/O, a request may contain many *transfer blocks*. In the following discussion, we generally set the (transfer) block size to 1 for convenience. All the following discussions also apply to block size larger than 1 sector. We only need to map the sectors to the corresponding transfer blocks. Then adjust the time to transfer one block to be C2*N, instead of the transfer time C2 of 1 sector, where N is number of sectors per transfer block.

3.2.2 Multiple-sector Request

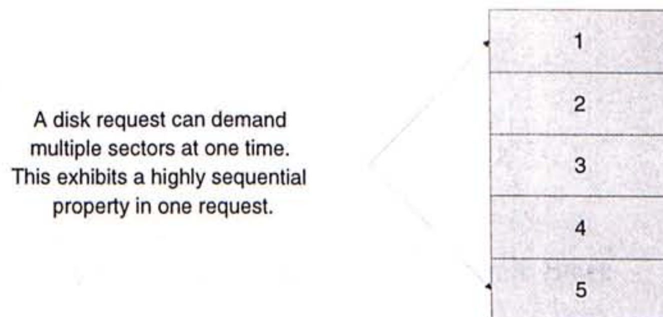


Figure 3.1: Multiple-sector Request

Unlike CPU memory request, a disk request can request more than one datum/sector as shown in Figure 3.1. A multiple-sector request exhibits highly sequential property in just one request. We can make use of this property to improve the performance of disk cache. If the program execution can overlap the fetching of requested sectors,

some sectors do not need to be stored in the cache now. Those un-stored sectors can be fetched by the cache system during program execution.

Due to the slow data bus, large number of beginning sectors must be stored in the cache first. While the process uses those cached sectors, the remaining sectors can be prefetched. For one multiple-sector request, its size is N_r. Some beginning sectors, say its size is N_s, need to be stored in the cache and the rest can be fetched from disk directly during program execution. In practical, the number of sectors, N_s, that need to be stored first is generally larger than number of requested sectors, N_r, in one request, i.e. $N_r < N_s$. So, the idea of un-storing some sectors cannot be applied to this case. This idea needs a larger block of continuous sectors to operate.

3.2.3 Dynamic Block, Heading Sectors and Content Sectors

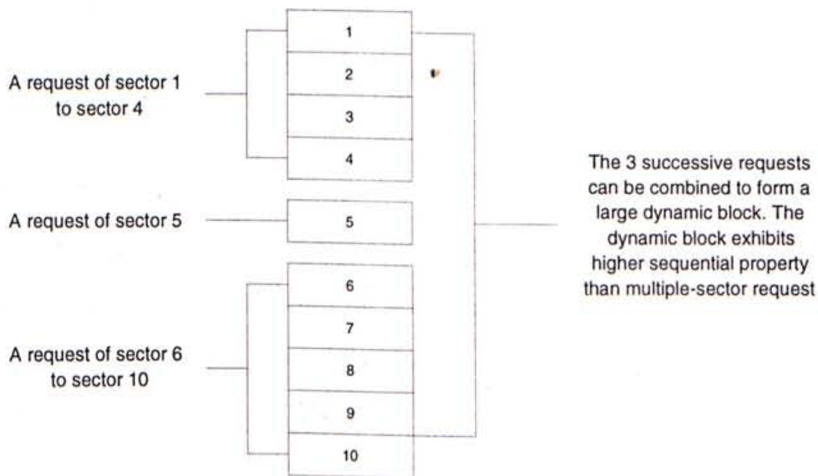


Figure 3.2: Formation of Dynamic Block

When we analyze the disk traces to see whether there is relationship between 2 successive requests, we have found that they might be sequential. Therefore, we define all blocks between 2 *non-sequential* requests as a *Dynamic Basic Block*, or simply *Dynamic Block* as shown in Figure 3.2. Detail discussion on Dynamic block and its property on the traced data can be found in Section 4.5. For instance, many successive 1-sector requests can be combined to form a large dynamic block. From Table 4.5 in

Chapter 4, the percentages of 1-sector requests which cannot be combined as dynamic block are 1.62% for Access trace, 19.06% for Dbase trace, 13.07% for Excel trace and 37.53% for Word trace. The small percentages illustrate that many successive requests can be combined to form a large dynamic block.

We can utilize this highly sequential property to improve the performance of a cache design. Since the next request is highly predictable, it might not need to be stored in cache if the prefetching is fast enough to get it. However, due to the slow data bus, the cache system still needs to store *some* sectors in order to provide enough time to prefetch other sectors in the same dynamic block. Therefore, the importance of each sector in a dynamic block is *different* according to this point of view. If the cache stores enough sectors so that the cache system has enough time to get the next sectors, the next sector does not need to be stored in cache.

This is similar to the case of a multiple-sector request. We can reduce disk access time by overlapping the program execution and the fetching of sectors from disk. A block can be roughly divided into two parts: *heading sectors* and *content sectors* as shown in Figure 3.3. Heading sectors are the first few sectors that are stored in cache. When a process is using the heading sectors, some content sectors will be fetched from disk simultaneously. The size of heading sectors depends on the chosen algorithm. This is an important idea of the newly proposed algorithms and it will be discussed more detail in Section 3.3.

The difference between a dynamic block and a multiple-sector request is that the size of a dynamic block, N_d, can be much larger than that of a multiple-sector request, N_r. Therefore, we can treat a dynamic block to be a very large multiple-sector request. For the case of multiple-sector request, N_r is less than the number of heading sectors N_s that need to be stored in cache, and the idea of overlapping data fetching and program execution may not apply. However, for dynamic block, N_d is larger than N_s because of the large size of a dynamic block. The idea of overlapping data fetching and program execution can apply more efficiently.

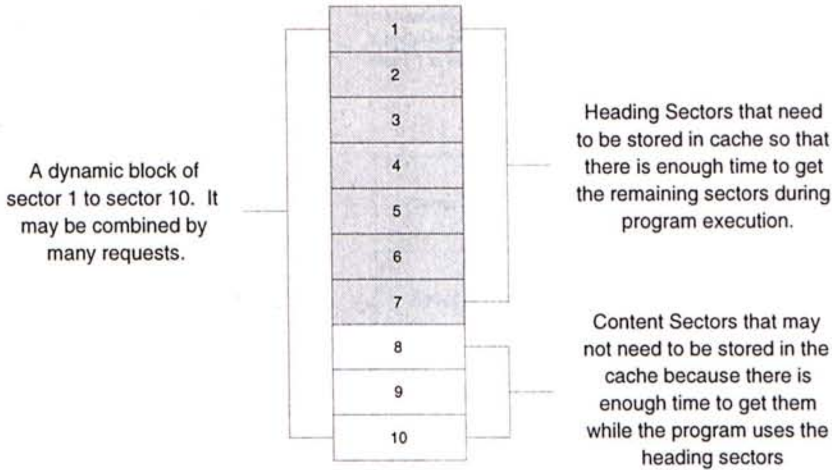


Figure 3.3: Heading Sectors and Content Sectors

3.2.4 Heading Reuse and Non-heading Reuse

Before we discuss our new models of disk cache, we have to introduce a new concept about the *reuse* of data. The reuse pattern of a dynamic block can be divided into two types: *heading reuse* and *non-heading reuse* as shown in Figure 3.4.

Heading reuse of a dynamic block is defined as that the *first heading block* of the current dynamic block is *equal* to the *first heading block* of some previously formed dynamic block in the cache. The 'previous' dynamic block may not be exactly the same as the current one. In Figure 3.4, there were some requests forming a dynamic block of sector 1 to sector 5. Now there are some requests forming a dynamic block of sector 1 to sector 5 again, or sector 1 to sector 4,..., etc. All these requests are said to be *heading reuse.*

Non-heading reuse is defined as that the *first heading block* of the current dynamic block is *not equal* to the *first heading block* of all previously formed dynamic blocks in the cache. In Figure 3.4, there were some requests forming a dynamic block of sector 1 to sector 5. Now there are some requests forming a dynamic block of sector 3 to sector 5, or sector 2 to sector 3,..., etc. Then these requests are said to be *non-heading reuse.*

For traditional algorithms of disk cache, they do not consider whether the request

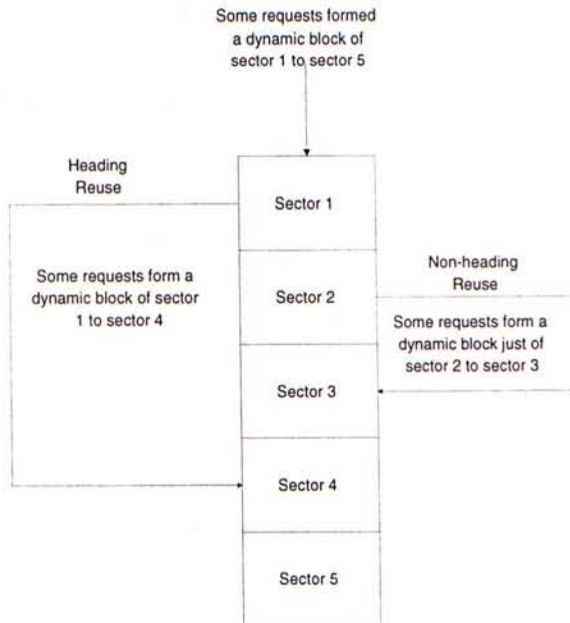


Figure 3.4: Two Different Kinds of Reuse

is heading reuse. However, the basis of our proposed algorithms is that the fetching of sectors can be overlapped the program execution. It divides requests/dynamic blocks into *heading* part and *content* part. The heading part will be stored in the cache in order to provide enough time to fetch the rest. Depending on the algorithms, the content part may not be stored in the cache. As a result of a non-heading request occurs, the requested sectors may not be in cache and our algorithms need to pay time penalty to get them. The frequency of heading reuse and non-heading reuse will affect the performance of the proposed algorithms.

3.3 New Models

The traced disk access pattern shows a strong sequential property. Hence prefetching should be useful to reduce the average disk access time. In fact, we have chosen *always prefetch* technique to incorporate into our proposed models. Besides, by proper overlapping the program execution with the prefetching of data, we expect that the cache performance can be improved. *To prevent flushing out useful data by prefetching,*

we adopt the technique of cache partitioning. It divides the cache into two parts: one part is similar to an ordinary cache and another part is a small buffer. The small buffer is used to store the prefetched data in order to reduce the cache pollution due to inaccurate prefetching. The following models are based on the CPU cache partitioning technique with newly designed algorithms to control the flow of data into these two parts of the cache.

Model names of the newly designed models are given in here for convenience and for consistency to the simulation. Model 4 is set to unified cache with always prefetch. Model 5 is set to the basic partitioned cache model as discussed in Section 3.3.2. Model 6 is set to the partitioned cache with ASST applying to each request as discussed in Section 3.3.3. Model 7 is set to the partitioned cache with ASST applying to each dynamic block as discussed in Section 3.3.4. Model 8 is set to the partitioned cache with SEHT as discussed in Section 3.3.5.

3.3.1 Unified Cache with Always Prefetch

This is not a newly designed model. However, it is different from currently used disk caches because it uses always prefetch to take the following blocks/sectors. For traditional disk caches, they only use prefetch on miss or large block size to take the following blocks/sectors. From the analysis of the traced data, we discover that there is highly sequential property in the I/O requests as discussed in Section 4.5. Therefore, we expect that the performance of always prefetch must be better than that of prefetch on miss, and this has been verified by our simulation.

This model triggers the prefetching mechanism by each block reference. This model always prefetches data from disk after or during each block reference. This is a very aggressive method. If the next requested blocks/sectors matches the prefetching blocks/sectors, always prefetch can *further* reduce the disk access time. On the other hand, always prefetch has the higher chance to increase cache pollution. This model is named as *Model 4* in the simulation.

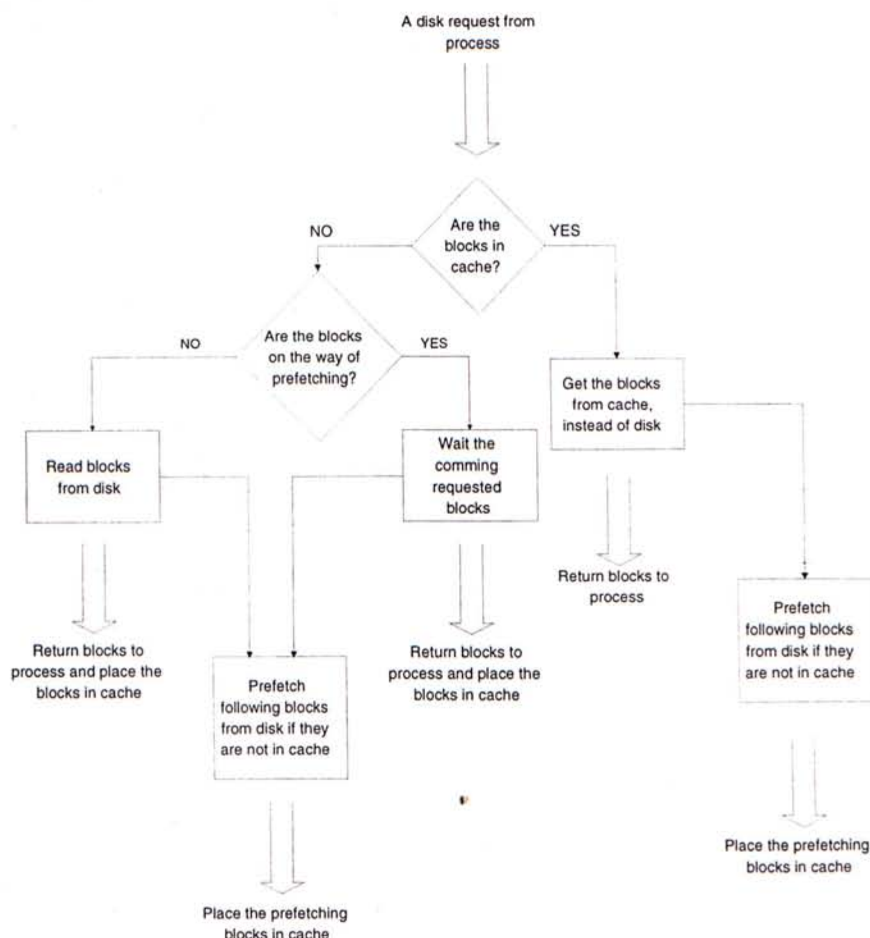


Figure 3.5: Flow Chart of Model 4

3.3.2 Partitioned Cache: Branch Target Cache and Prefetch Buffer

In the analysis of the disk traces, we have observed that there is a highly sequential property in the disk access pattern. Detail discussion can be found in Section 4.5. In order to overlap the program execution with the prefetching of data, it might be possible that the cache system can just store the heading blocks/sectors of a dynamic block and let the prefetching system get the remaining ones. The prefetched ones will be stored in a small buffer and will be discarded after they are used. This basic model partitions the cache into a Branch Target Cache (BTC) and a Prefetch Buffer (PB). BTC is a cache with its size like ordinary cache while PB is a small buffer. PB is exactly like the *fetch buffer* proposed in CPU cache [Jou90, KL91, CMCH91]. And our

models are built on this partitioned cache architecture. Figure 3.6 shows the BTC and PB.

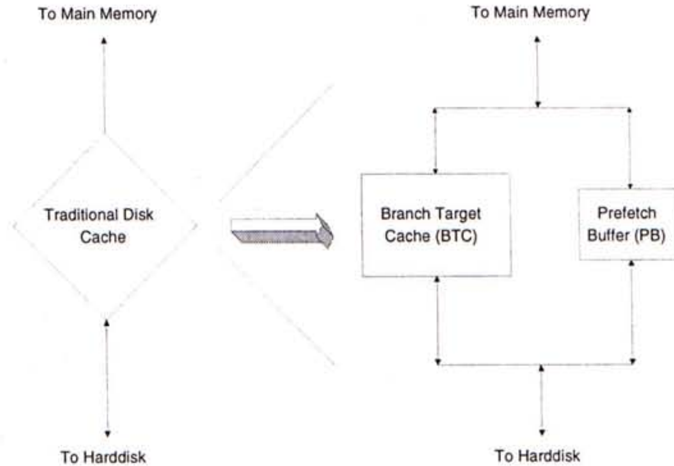


Figure 3.6: Partitioned Cache: BTC + PB

Branch target cache stores only those sectors which cause cache misses. Due to the incorporation of always prefetch technique, *only* the *first block/sector* of a dynamic block causes cache miss. Others will just cause partial hits. So the branch target cache stores only the first block of each dynamic block. The main purpose of BTC is that the cache tries to contain the requested data at each branch reference.

Prefetch buffer is a small buffer. It is used to store the prefetched data. Because of its small size, the data inside it will be replaced very quickly. Since data fetching can overlap the program execution, storing prefetched sectors in PB can eliminate the problem that the prefetched data flush out useful data in the ordinary cache. The replacement policy of prefetch buffer is LRU algorithm.

The operation of this basic model is as shown in Figure 3.7. When there is cache hit, the demanded blocks will return to the process. The cache system starts to prefetching the following sectors. All these sectors will be stored in PB. When there is partial hit, all the prefetched sectors are also stored in PB. When there is cache miss, the first block of the demanded blocks is checked whether it is the starting of a dynamic block. If it is the starting of a dynamic block, it will be stored in BTC. Otherwise, it will be

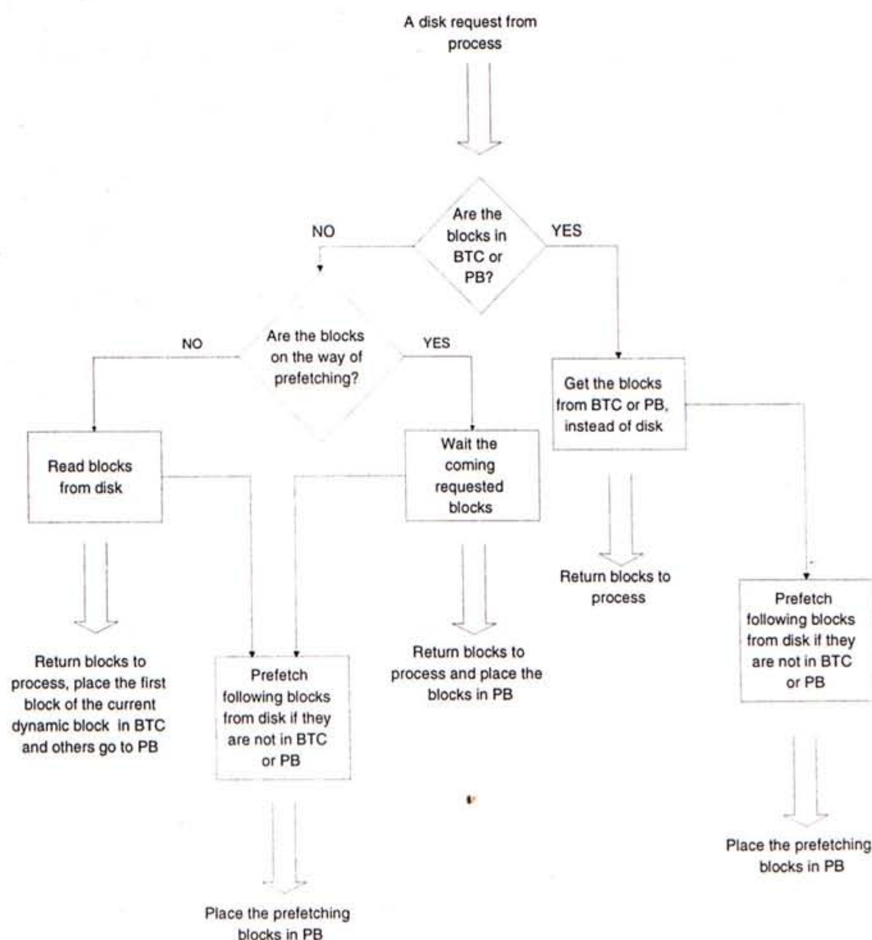


Figure 3.7: Flow Chart of Model 5

stored in the PB. The cache system will then prefetch the following blocks which will be stored in PB.

The large BTC stores only the first block of each non-sequential reference and all prefetched data are captured by the PB. This combination can eliminate the problem of flushing out useful data for inaccurate prefetching, and may provide good performance at each branch reference. Since the BTC stores more first blocks than the ordinary unified cache, the *effective cache size* of the proposed model is greatly enlarged when comparing with the case of unified cache.

On the other hand, owing to the slow data bus and the limited bandwidth, the disk cache system actually does not have enough time to prefetch the sequential referenced

sectors. Even though the next sector has very high chance to be referenced, the first block of each non-sequential reference in BTC cannot provide enough time for the prefetching to finish. This is why partial hit occurs so often. The prefetched data can only be on the way and the cache system still pays a *time penalty* to prefetch requested data. Besides, if the branch target cache just stores the first block of each non-sequential reference, situation of non-heading reuse might occur. Branches might jump directly to the content block of previous dynamic blocks. These branches cannot be handled by BTC because the branched data are not stored in the BTC during previous request.

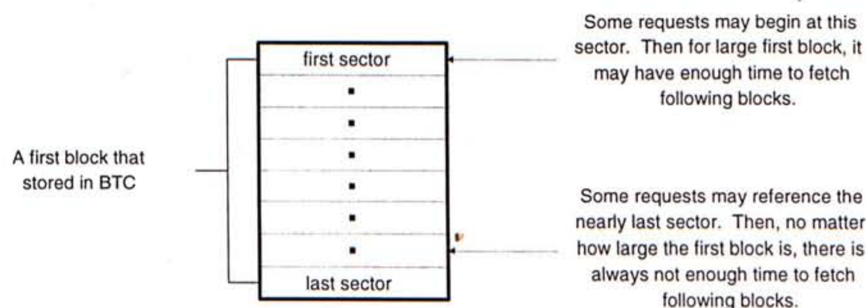


Figure 3.8: Problem of Storing First Block in BTC

The problem of not enough time for prefetching cannot be solved with large block size. This is because a reference may jump to any part of the block, e.g. the near end of the block. As a result, only small portion of the first block can be used to provide time for prefetching in this case. The situation is shown in Figure 3.8. Besides, large block size may lead to fewer heading blocks to be stored in cache, and it makes a problem of inefficiently utilizing the cache.

To solve this problem of insufficient time for prefetching, BTC should not only store the first block, but also some of the following blocks in order to ensure enough time for data prefetching. To make the cache partitioning design suitable for disk cache to use, and to eliminate the problems of traditional BTC and PB, we have invented new algorithms to control the flow of data/sectors into BTC and PB.

This *basic model* is named as *Model 5* in the simulation. It acts as a control experiment to our newly proposed models because our models are built on it. Model 1 to Model 4 are the models of unified cache for performance comparison in the simulation and will be discussed in Section 4.3. Model 6 to Model 8 are the newly proposed models and will be discussed in the following sections.

3.3.3 BTC + PB with Alternative Storing Sector Technique

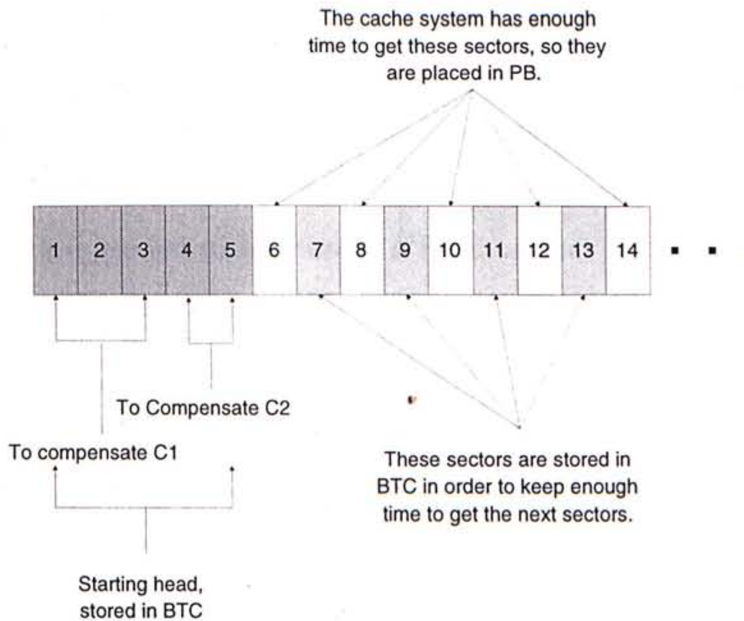


Figure 3.9: Alternative Storing Sector Technique (ASST), C1=3, C2=2

The problem of the basic model is that the time penalty of getting the next sector is large due to the slow data bus and the limited bandwidth. Therefore, a new method is introduced to arrange the sectors going into BTC or PB. This new method, called the *Alternative Storing Sector Technique (ASST)*, is introduced to rearrange sectors going into BTC or PB. Since the ASST applies to a block of sequential sectors, it can apply either to multiple-sector requests, or dynamic blocks. In this section, the ASST will be applied to multiple-sector requests, i.e. per request block basis.

ASST *not only* stores the first heading block/sector that causes the cache miss into the BTC, *but also* stores some content blocks/sectors of each dynamic block in BTC

in order to allow the cache system to fetch the remaining sectors without paying time penalty. In ASST, whether a sector needs to be stored in BTC is determined by the *fixed* parameters – start-up time C1 and transfer time C2. The sectors that do not need to be stored in BTC is termed as *'un-stored'*, but they will still be stored in PB.

To illustrate the operation of the ASST, let us consider a case shown in Figure 3.9. C1 is now equal to 3. C2 is equal to 2 and Tu are equal to 1. Since Tu is equal to 1, the cache system must store 3 (value of C1) sectors in the cache in order to compensate for C1. However, after the first three sectors are used, the cache system has just started to transfer the forth sector. So, the cache system needs to store extra 2 sectors (value of C2) in order to *compensate* the time of transferring the first un-stored (sixth) sector. Therefore, the size of *'starting head'* of a block of request is equal to 5 (C1+C2) sectors. The five sectors are needed to be stored in BTC so that it can compensate the time needed to transfer the first un-stored (sixth) sector.

After the first un-stored (sixth) sector arrives, the process starts to use it. Note that the sixth sector is stored in PB because there is enough time to get it in each reuse. If the seventh sector has already been stored in BTC, the cache system can also pass it to the process. During the processing of the sixth and seventh sectors, it has enough time to get the eighth sector. Therefore, the eighth sector does not need to be stored in BTC, i.e. the eighth sector is the second un-stored sector. Similarly, if the ninth has also already been stored in BTC, during the time of the processing the eighth and ninth sectors, the cache system has enough time to get the tenth sector. Therefore, following the same argument, the un-stored sectors are 6th, 8th, 10th, 12th,..., and so on.

Therefore, which sectors needed to be stored in BTC can be determined from the hardware parameters, i.e. C1 and C2. The size of starting head is $\lceil C1+C2 \rceil$ where $\lceil C1+C2 \rceil$ is the ceiling of C1+C2. Then which of the next sectors needed to be stored in BTC depends on the value of C2−Tu, and follows the rule that *the cache system should store enough previous sectors in order to prefetch the next un-stored sector. In processing of the prefetched sector will also contribute the time (i.e. the use-up time)*

to get the next un-stored sector. Therefore, the operation of ASST can be summarized by the following procedure. Let there be a time counter TC; $\lceil x \rceil$ be the ceiling of x.

```
Procedure ASST
BEGIN
    reset the counter to zero
    storing the starting head, ⌈C1+C2⌉ sectors, in BTC
    TC = ⌈C1+C2⌉ - (C1+C2)
    DO until no more sectors in the block
        IF TC > C2
            TC = TC - C2 + Tu
            current sector does not need to be stored in BTC
        ELSE
            TC = TC + Tu
            current sector has to be stored in BTC
        ENDIF
    ENDDO
END
```

When $TC > C2$, there is enough accumulated time to get the next sectors and so the next sector is not needed to be stored in BTC. The equation $TC = TC - C2 + Tu$ means that the cache system has to pay time to get the un-stored sectors, i.e. $TC - C2$. However, after getting this sector, it will also contribute a use-up time Tu to get the following sectors. Therefore, the total time changes from TC to $TC - C2 + Tu$. When $TC < C2$, the sector must be stored in BTC in order to accumulate time to fetch the next one. The stored sector will contribute a use-up time Tu. So, the total time changes from TC to $TC + Tu$.

To incorporate the ASST algorithm into hardware, a time counter is needed to count the time as shown in the ASST procedure. For each arriving sector, we have to update the counter and check whether the sector needs to be stored in BTC. Then for each new request, the counter is reset to zero in this model. The operations are simple, so the time of calculations can be ignored when comparing with the slow disk access time. As an example of applying the ASST algorithms, Table 3.1 shows un-stored sectors for C2=2 and C2=1.5 when C1=3.

ASST can fully utilize the idle data bus. It uses always prefetch as its basis. BTC

	Un-stored sectors
C2=2	6th, 8th, 10th, 12th, 14th,...
C2=1.5	6th, 7th, 10th, 11th, 14th, 15th,...

Table 3.1: The Un-stored Sectors for C2=2 and C2=1.5

stores some heading and content sectors, the prefetching system can have enough time to get the rest before they are needed. Therefore, this 'un-stored' ones do *not* need to be stored in the BTC. They are only stored in the prefetch buffer. When the process requests them, a hit in the prefetch buffer will occur. Cache pollution for BTC due to prefetching will be greatly reduced because the prefetched data will go to the PB instead. Useful blocks of data can remain in the BTC. Besides, BTC stores blocks from more dynamic blocks now and the problem of not enough time to do prefetching is minimized.

The operation of this model is shown in Figure 3.10. When there is cache hit, the demanded blocks will be returned to the process. The cache system starts to prefetch the following sectors. All these sectors will be stored in BTC or PB according to applying ASST to each request separately. When there is cache miss, the first block is fetched from disk and all the following sectors are prefetched from disk due to always prefetch. All the fetched and prefetched sectors are stored in BTC or PB according to applying ASST to each request. When there is partial hit, the sectors are also stored in BTC or PB according to applying ASST to each request.

In this model, the Alternative Storing Sector Technique applies to *each request* separately. Since C1 is usually very large, the starting head calculated from ASST contains many sectors. Consequently, requests of a small number of sectors are completely stored in the BTC and the cache system cannot gain the advantage of PB. In applying ASST, the size of starting head is fixed and which content sectors needed to be stored in BTC are also predefined. If the block which ASST operates is large, more sectors will not need to be stored in BTC, and the cache space to store a request is reduced. In other words, the cache size is effectively enlarged. However, if the request

block is *not large*, ASST *cannot* show its effectiveness. In this model, ASST is applied on the request block which may *not* fully utilize the power of ASST as shown in Figure 3.11. We will discuss this problem in detail in the next Section 3.3.4. This model is named as *Model 6* in the simulation.

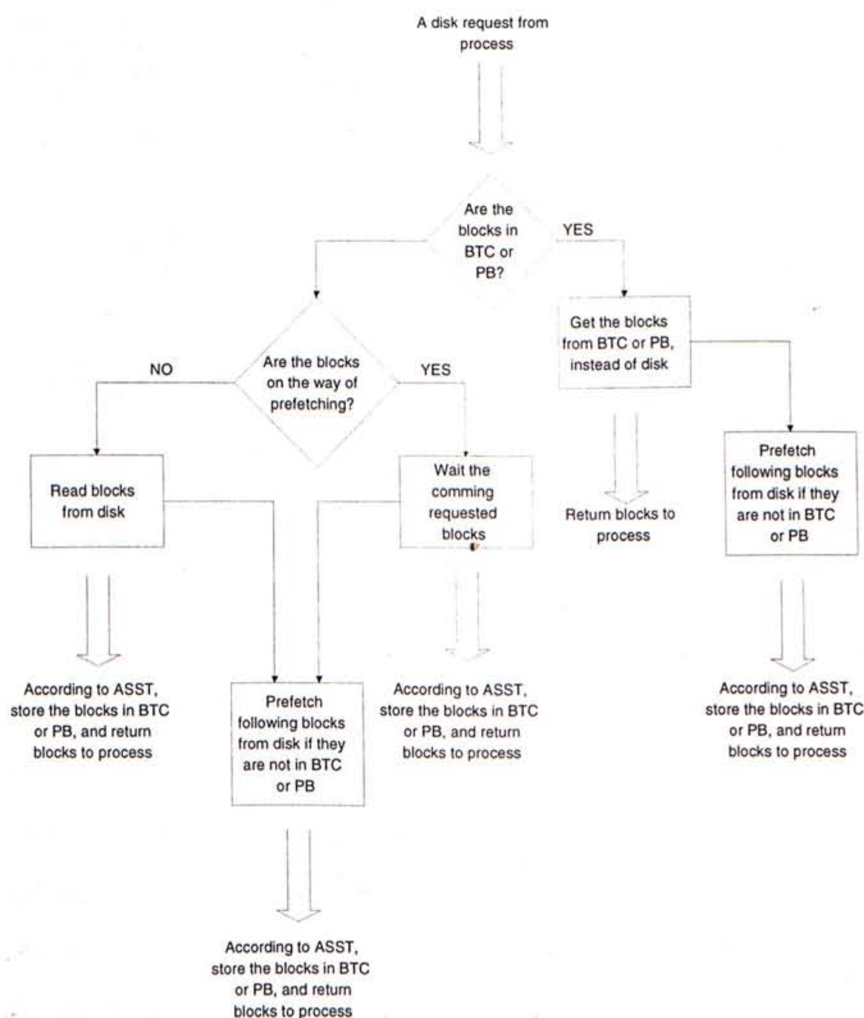


Figure 3.10: Flow Chart of Model 6 and Model 7

3.3.4 BTC + PB with ASST Applying to Dynamic Block

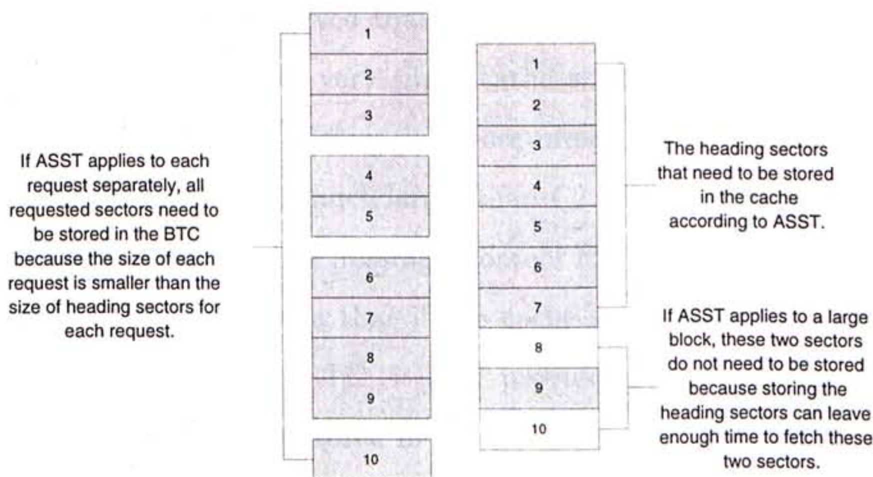


Figure 3.11: Applying ASST to Multiple-sector Request and Dynamic Block (C1=5, C2=2)

There are many small requests in the traced disk I/O. The most frequent one is the 1-sector request. Therefore, applying ASST to every request causes all small requests to be stored in the BTC completely. In the previous Section 3.2.3, we have mentioned that there are high interrelationship between successive requests. Many successive requests, including 1-sector requests, can be combined to form a larger block of sequential sectors, i.e. dynamic block. Therefore, we propose to apply ASST on a dynamic block basis because the block that ASST can apply on will be large. Figure 3.11 shows the situation of ASST applying to multiple-sector request and dynamic block for C1=5 and C2=2.

This model is very similar to the previous model except that the ASST applies to *each dynamic block*. So, the flowchart of this model is the same as the previous model as shown in Figure 3.10. The operation is also very similar to the previous case. The procedure of ASST is nearly the same except that the time counter, TC, will only be reset at each starting of a dynamic block, i.e. at each non-sequential reference. This is different from the case of previous model that the time counter TC will be reset for each request. This difference can improve the cache performance greatly. The comparison of performance of the previous model, Model 6, and this model can be found in Section 5.7.2. This model is named as *Model 7* in the simulation.

3.3.5 BTC + PB with Storing Enough Head Technique

In the simulation, we have observed that the performance of the basic model, Model 5, is quite good in the case of very small cache size. This better performance of Model 5 reveals that heading sectors are more important than the content sectors. This is because C1 is generally much larger than C2. Missing the first heading block causes C1+C2 time penalty while missing a content block causes C2−Tu time penalty. Therefore, there is a simple idea that if the cache system *just stores some heading sectors of each request* in the BTC, it may provide a good performance. No any content sectors of a request are stored in the BTC, i.e. all content sectors of a request are stored in PB.

If the cache system stores only some heading sectors of each request, it will produce a time penalty in each reuse because the BTC has not stored enough sectors to provide enough time to prefetch the remaining sectors. On the other hand, BTC can store heading sectors from more requests, i.e. the effective cache size is enlarged more. Therefore, there is a competition between storing heading sectors from more requests and the time penalty paid for getting remaining sectors. The situation is shown in Figure 3.12.

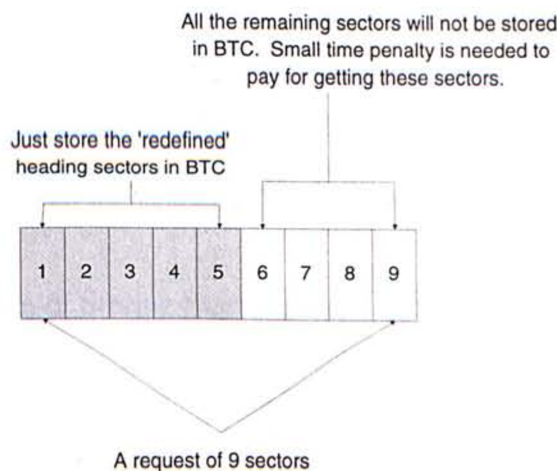


Figure 3.12: Storing Some Heading Sectors for Each Request

Since the idea is applied to *each request*, the size of starting head $\lceil C1+C2 \rceil$ for ASST is too large and nearly all requests will be completely stored in the BTC. This is because the size of each request is generally less than $\lceil C1+C2 \rceil$ sectors. Therefore, the size of the starting head should be reduced. The number of starting sectors of each request that will be stored in BTC is redefined:

$$Size\ of\ starting\ head = \frac{Start\ up\ time\ C1}{Transfer\ time\ C2}\ sectors$$

This equation is a compromise between minimizing the size of starting head and the time penalty for each reuse. The equation defines that the total transfer time of the stored starting head (N sectors) when there is no cache, i.e. C2*N, is equal to the start-up time C1. By this equation, the size of the starting head is reduced and starting heads from more requests can be stored in BTC.

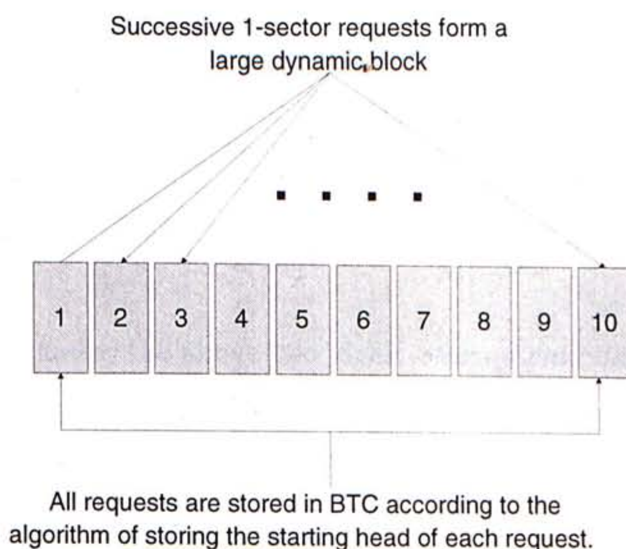


Figure 3.13: Problem of Storing Starting Head of Each Request to 1-sector Requests

However, the above idea has a problem that all 1-sector requests go to BTC as shown in Figure 3.13. Since the most frequent request size is 1 sector, the above idea is nearly useless. To solve this problem and make the idea useful, we incorporate the method of *ASST applying to dynamic block* in this idea. Note that the basic idea is still to store only the starting head of *each request*. For instance, there are many successive 1-sector

requests and they can be combined to a larger dynamic block. When ASST applies to this dynamic block, some 1-sector requests need not be stored in BTC. Therefore, this method can reduce the number of 1-sector requests stored in BTC. Applying ASST to dynamic block can also reduce the storing of many small size requests.

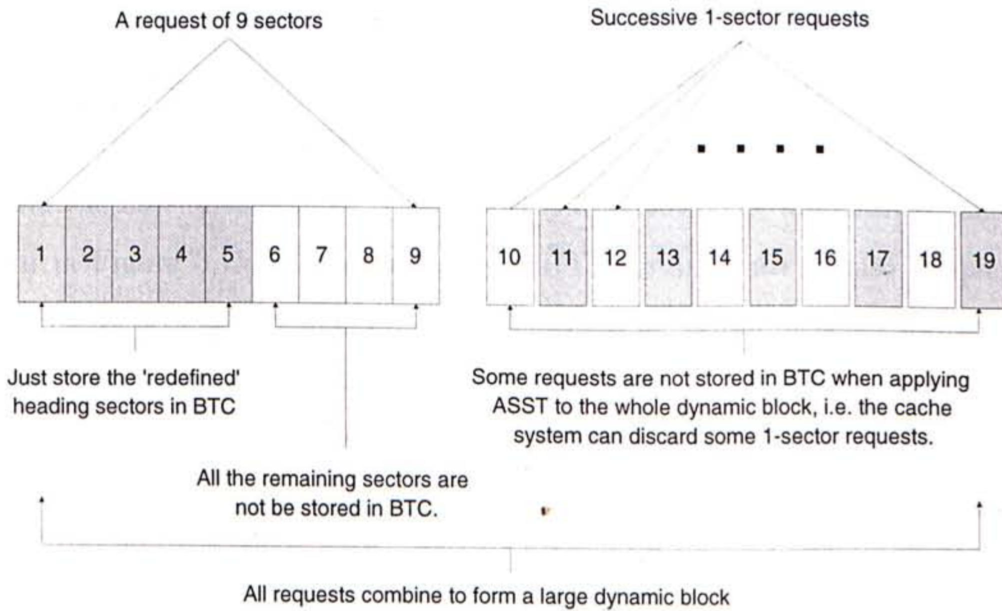


Figure 3.14: Storing Enough Head Technique (SEHT), C1=10, C2=2

Therefore, by *combining* the above two ideas: *storing only the starting head of each request* and *applying ASST to each dynamic block*, we can introduce a new algorithm, called *Storing Enough Heads Technique (SEHT)*. The two criteria of SEHT are

1. The sector is a heading sector (the redefined one) of a request.

2. The sector is needed to be stored according to ASST applying to the current dynamic block.

A sector will be stored in BTC according to SEHT *only if* the sector satisfies the above two criteria simultaneously. Storing Enough Head Technique (SEHT) stores the redefined heading sectors of each request, except that those un-stored sectors predicted by applying ASST to the current dynamic block. Consider an example of application

of SEHT in Figure 3.14. By applying the first criteria to each request, there is a group of sectors (Group 1) that should be stored in BTC. Then apply the second criteria to the current dynamic block and get another group of sectors (Group 2) that should be stored in BTC. Intercepting the two groups, Group 1 and Group 2, we will get the sectors that need to be stored in BTC according to the SEHT. That means a sector must satisfies the two criteria simultaneously in order to be stored in BTC. All other sectors are stored in PB.

The operation of this model is similar to the Model 7 except that the algorithm to control the flow of sectors to BTC or PB is different. The flowchart of this model is shown in Figure 3.15. To incorporate SEHT into hardware is also simple. Only the heading sectors of each request have chance to be stored in BTC and the size of heading sectors is fixed. In addition, for each request, apply ASST to the currently formed dynamic block and check whether the current sector is needed to be stored. If the sector fulfills both requirements, it will be stored in BTC. Otherwise, the sector will be stored in PB.

This method introduces time penalty even for a cache hit because it stores fixed/limited amount of heading blocks for each request, neglecting how large the block of a request is. Therefore, SEHT can effectively increase more cache space to store more data from different branches than that of ASST. On the other hand, its tradeoff is the small time penalty for each request. This model is named as *Model 8* in the simulation.

3.4 Impact of Block Size

In previous discussion, block size is generally set to 1 sector. The new algorithms can also suit to the cases of block size larger than 1 sector with simple modification. The basic modification is to change the basic unit of transfer to a N-sector block, instead of 1-sector block. All sectors are mapped to their corresponding transfer block. Besides, the transfer time of the basic block changes to C2*N, instead of the original C2 for 1-sector block. Furthermore, the use-up time of the basic block changes to Tu*N, instead

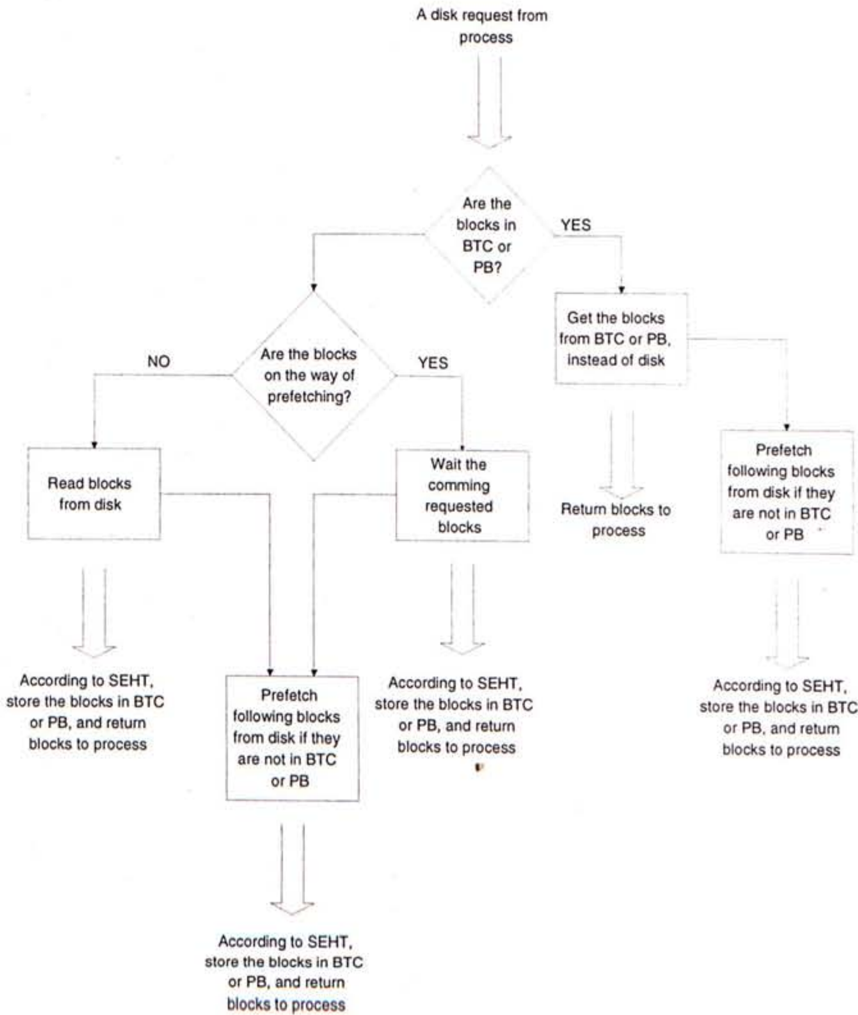


Figure 3.15: Flow Chart of Model 8

of the original Tu for 1-sector block. For instance, the size of starting head of SEHT is changed as follow:

$$Size\ of\ starting\ head = \frac{Start\ up\ time\ C1}{Transfer\ time\ C2 * Block\ Size}\ sectors$$

Following similar argument, the starting head of ASST should contain $\lceil C1/N + C2 \rceil$ blocks where N is the block size. However, there is a problem as shown in Figure 3.8 for the first starting block when the block size is larger than 1 sector. The requested sectors may locate at the end of the first starting block. Therefore, in order to maintain enough time to prefetch the un-stored blocks, we may need to conservatively define the

size of starting head to be $\lceil C1/N+C2 \rceil+1$ blocks. If a more accurate timing for the ASST is needed, the cache system should store enough number of blocks so that the time to use up the requested sectors in those block is equal to, or *just* greater than, $\lceil C1+C2*N \rceil$. The time $\lceil C1+C2*N \rceil$ means that the total use-up time of the requested sectors in the stored heading blocks should compensate the startup time of the disk and the transfer time of the first block. Below is a simple modification of the ASST algorithm to suit the case of block size larger than 1 sector.

```
Procedure ASST
BEGIN
    reset the counter to zero
    storing enough heading blocks to cover ⌈C1+C2*N⌉.
    TC = the use up time provided by the stored heading block − ⌈C1+C2*N⌉
    DO until no more blocks
        IF TC > C2*N
            TC = TC - C2*N + Tu*N
            current blockdoes not need to be stored in BTC
        ELSE
            TC = TC + Tu*N
            current block has to be stored in BTC
        ENDIF
    ENDDO
END
```

Chapter 4

Trace Driven Simulation

4.1 Simulation Environment

In the simulation, we have traced the disk I/O of four applications under Microsoft Windows environment. The applications that we have used to get the traces are Microsoft Access, Microsoft Excel, Microsoft Word and Dbase for Windows. We have used a 486 personal computer with 4M RAM to obtain the traces. Millions of requests have been collected.

Before discussing the result, we should know the assumptions of the simulation:

1. By proper overlapping the program execution and the data fetching, when the process is using some sectors, the cache system can transfer the remaining sectors asynchronously. Therefore, the computing environment must support asynchronously I/O operations.

2. The simulator treats write operation same as read operation and ignores the actual writing back of data.

3. The use-up time Tu is normalized to 1. All other timing values, e.g. start-up time C1 and transfer time C2, are the ratios to actual value of the use-up time.

4. Branch Target Cache is a n-way set associative cache. Within each set, the replacement policy follows LRU (Least Recently Used) algorithm. Prefetch buffer is a fully associative cache. Its replacement policy also follows LRU.

5. The cache line size is set to the transfer block size.

6. Time for searching the cache and time for killing a prefetch is negligible.

The parameters of the simulation are as follow:

- The data format of a record in the trace is shown as below:

  <action, cylinder, sector, head, drive, number of sectors>

  action equals to 0 and 1 means read and write operations respectively.

- Cache size takes the values of 1M, 2M, 4M and 8M.

- PB size = 0.1M, therefore, BTC size = Cache size - 0.1M.

- Block size takes the values of 1 sector, 2 sectors, 4 sectors and 8 sectors.

- Set associativity takes the values of 1-way, 2-way and 4-way.

- Start-up time C1 takes the values of 5, 10, 15 and 20 for the case of common disk and it takes the values of 1, 2, 3 and 4 for the case of high performance disk.

- Transfer time C2 takes the values of 0.5, 1 and 1.5.

- Eight models have been simulated. Model 1 to Model 4 are the models for performance comparison and will be discussed in Section 4.3. Model 5 is the control model of cache partitioning technique. Model 6 to Model 8 are the newly proposed models. Model 5 to Model 8 have been discussed in Section 3.3. The eight models are as follows:

  - Model 1: No Cache

  - Model 2: Unified Cache without Prefetch

  - Model 3: Unified Cache with Prefetch on Miss

  - Model 4: Unified Cache with Always Prefetch

- Model 5: Partitioned Cache: BTC + PB
- Model 6: BTC + PB with ASST Applying on Request
- Model 7: BTC + PB with ASST Applying on Dynamic Block
- Model 8: BTC + PB with SEHT

4.2 Two Kinds Of Disk

In the simulation, we have simulated two kinds of disk. They are *Common Disk* and *High Performance Disk*. We categorize different kinds of disks by the values of the start-up time C1, the transfer time C2 and the use-up time Tu. For common disk and high performance disk, the transfer time C2 is set to near the use-up time Tu because it is on a high performance computing environment.

For common disk, the start-up time C1 is much larger than the transfer C2. For high performance disk, the start-up time C2 is near the transfer time C2. C1 and C2 play an important role in our newly proposed algorithms, ASST and SEHT. Their values control the size of the starting head and how many sectors have to be stored in BTC. Therefore, we will examine the effect of our cache models on common disk and high performance disk.

4.3 Control Models

To compare the performance of the new models, we have also simulated four models for comparison: no cache (Model 1), unified cache without prefetch (Model 2), unified cache with prefetch on miss (Model 3) and unified cache with always prefetch (Model 4). They are the common models used in current disk cache programs.

4.3.1 Model 1: No Cache

This model acts as a boundary model because having cache should performs better than no cache. Therefore, it gives the upper bound of the time. The timing can be

calculated very easily. Each request causes the sectors to be read from the disk and transferred to the process. Therefore, the disk access time per request is equal to C1 + C2 * N, where C1 is the start-up time of the disk; C2 is the transfer time for one sector from disk to memory; N is the number of the requested sectors.

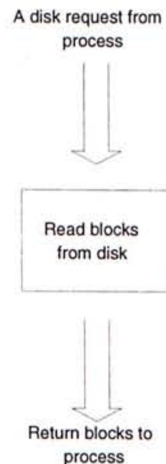


Figure 4.1: Flow Chart of Model 1

4.3.2 Model 2: Unified Cache without Prefetch

This model is commonly used in current disk cache design. For cache hit, it passes the blocks/sectors from cache to process. For cache miss, it gets the blocks/sectors from disk to cache and then from the cache to process. When the cache is full, the LRU replacement strategy is used. Note that it only has cache hit and cache miss, but does not have any partial hit.

4.3.3 Model 3: Unified Cache with Prefetch on Miss

This model is also commonly used in current disk cache design. Prefetching mechanism is only triggered by cache miss. When a cache miss occurs, requested blocks will be fetched from disk. Then some following blocks will be prefetched to the cache. For cache hit, this model passes the blocks from cache to the process and does *not* trigger any disk action.

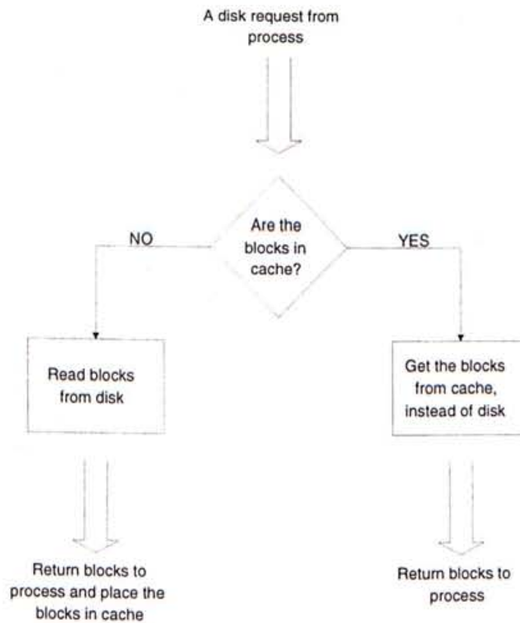


Figure 4.2: Flow Chart of Model 2

4.4 Two Comparison Standards

One of the comparison standard is to compare with *the model of no cache*. We use an *absolute reduction in time* to show the performance of all the models. The higher the absolute reduction in time is, the more the *absolute performance* of a model is. The absolute reduction in time is defined as

$$Absolute\ reduction\ in\ time = \frac{Time\ of\ no\ cache - Time\ of\ our\ model}{Time\ of\ no\ cache} * 100\%$$

where the *time* is the *total process stall time* due to the disk access.

The another baseline is to compare with *Model 3*, unified cache with prefetch on miss. This is a common method used in current disk cache design. The relative performance to Model 3 can give us insight to the effectiveness of our models. The indicator of this relative performance is *relative reduction in time* which is defined as

$$Relative\ reduction\ in\ time\ = \frac{Time\ of\ Model\ 3 - Time\ of\ our\ model}{Time\ of\ Model\ 3} * 100\%$$

The higher the relative reduction in time is, the more the *relative performance* of a model over that of Model 3 is.

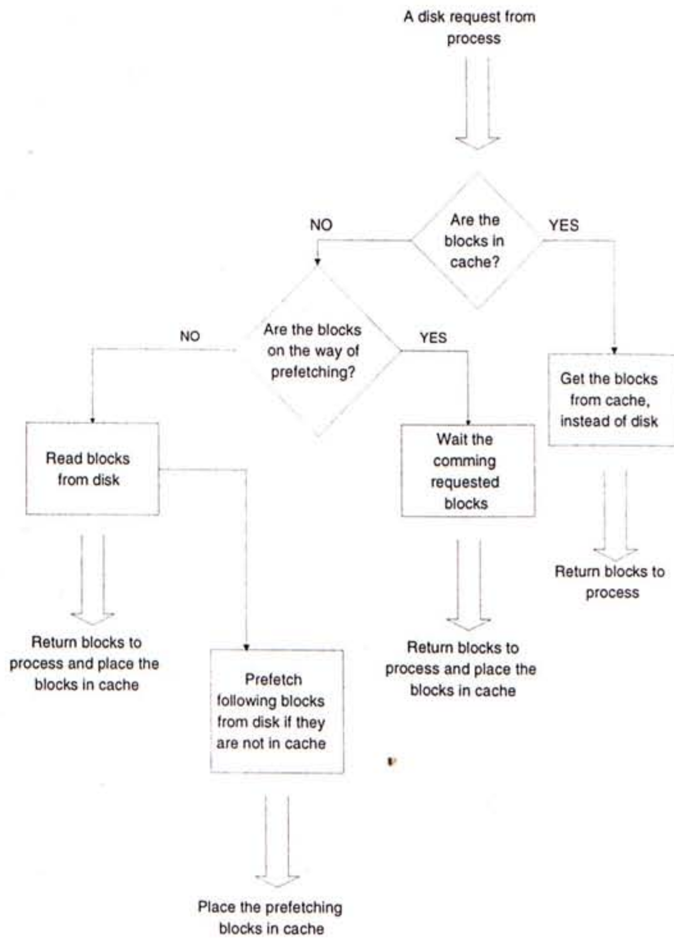


Figure 4.3: Flow Chart of Model 3

The simulation results are generally represented by the values of absolute reduction in time and relative reduction in time.

4.5 Trace Properties

Before we go into the detail of the simulation result, we should have a throughout understanding of the traced data first. The total number of requests for the four traces is shown in Table 4.1. The *distinct number of sectors requested* is the distinct number of sectors that the trace has been touched. It *ignores* how many times of those sectors were touched. On the other hand, the total number of sectors requested is the *total*

	Access	Dbase	Excel	Word
Total number of requests	6,051,491	1,939,119	1,027,301	1,797,561
Total number of sectors requested	10,518,494	10,195,308	5,652,077	2,547,843
Distinct number of sectors requested	105,124	63,606	36,209	80,659
Average reuse ratio	100.06	160.29	156.10	31.59

Table 4.1: Number of Requests for the Four Traces

number of sectors that transferred between disk and memory if the cache system *does not exist*. The *average reuse ratio* is the *ratio* of total number of sectors requested over distinct number of sectors requested. The *average reuse ratio* gives some idea on the reuse property of a sector.

By analyzing the requests in these four traces, we have found that there are many multiple-sector requests and the number of sectors per request is *not* small. The number of sectors per request can be as large as 127 as shown in Table 4.2. It is very different from the CPU reference request because CPU does *not* issue a request for a large block of data from the memory. This also implies that disk accesses exhibit a highly sequential property in one request. This is very important in disk cache design, and we would expect that always prefetch will give better result than prefetch on miss.

Although there are many requests for a large number of sectors, the *most frequent* number of sectors per request is still *1 sector*. On the other hand, from the analysis of the *displacement* between each request, we have found that many requests are *continuous* from the previous one. Displacement between two requests is defined as the starting address of the current request minus the ending address of the previous request.

$$displacement = starting\,address\,of\,current\,request - ending\,address\,of\,pervious\,request$$

Table 4.3 is the summary of the displacements from -1 to 3 for the four traces.

The *continuous percentage* is the percentage of the frequency for *displacement=1* over the total frequency for all displacements. The continuous percentages range from

No. of sector / request	Access	Dbase	Excel	Word
1	4,809,434	1,112,830	513,061	1,660,561
2	277	23,946	14,202	2,234
3	303	28,930	10,489	4,357
4	1,173,568	23,738	10,424	83,103
5	205	13,453	11,192	750
6	172	17,885	5,850	608
7	198	9,484	8,181	761
8	7,722	360,927	266,360	21,127
•	•	•	•	•
79	•	•	2	•
93	2	11		•
127				28

Table 4.2: Number of Sectors per Request

Displacement	Access	Dbase	Excel	Word
-1	359	1,027	786	4,372
0	158	4,099	562	2,921
1	4,145,545	878,829	565,827	959,655
2	89,416	49,089	435	6,606
3	83,422	2,263	266	7,534
Continuous Percentage	68.50%	45.32%	55.08%	53.39%

Table 4.3: Frequency of Displacement

45% for Excel to about 69% for Access. This shows that many requests will follow the previous requests, and the accuracy of always prefetch will be very high.

Since there is high inter-relation between requests from the above analysis, we expect that requests can be virtually coalesced to form *a request for larger block of sectors*. This coalescing block will further reflect the sequential property of a trace. The coalescing block is named as *Dynamic Block* since its size varies. Although the most frequent size of a request is 1 sector, the most frequent dynamic block size is not equal to 1 sector. The frequencies of dynamic block sizes for Access trace is partial shown in Table 4.4. The frequencies of dynamic block sizes for other traces are similar to this and thus are not shown here.

Dynamic Block Size	Frequency	Combined Frequency
1	78,031	(1,77955) (2,47) (3,8) (4,11) (5,4) (6,3) (8,2)
2	75,915	(1,23) (2,75887) (3,3) (4,1) (5,1)
3	274,358	(1,12) (2,57) (3,274289)
4	1,208,044	(1,1000213) (2,63) (3,102) (4,207666)
5	22,456	(1,25) (2,42) (3,91) (4,36) (5,22262)
6	13,454	(1,4) (2,49) (3,78) (4,19) (6,13304)
7	11,382	(1,15) (2,56) (3,55) (4,30) (5,1) (7,11225)
8	26,892	(1,7342) (2,6494) (3,32) (4,28) (8,12996)
9	11,001	(1,3) (2,12) (3,64) (4,35) (5,1) (6,1) (7,1) (9,10884)
10	10,951	(1,2) (2,28) (3,44) (4,31) (7,1) (10,10845)
•	•	•
•	•	•
•	•	•
63	9	(4,9)
64	1929	(4,1918) (5,11)
93	2	(1,2)

Table 4.4: Frequency of Dynamic Block Size for Access

The number pair in the third column of Table 4.4 is the combined frequency. It shows the dynamic block size is combined from how many requests and its frequency.

(Number of requests coalesced to form a dynamic block, Frequency)

For instance, a dynamic block of 5 sectors can be combined from 5 one-sector requests and this kind of combination occurs 22262 times, i.e. (5,22262). It can also be just 1 request of five sectors and this kind of combination occurs 25 times, i.e. (1,25). Beside, a dynamic block of 5 sectors can be formed by 3 requests that may be 1 request of one sector and 2 requests of two sectors, or 2 requests of one sector and 1 request of three sectors, or ..., etc. This kind of combination of dynamic block of 5 sectors from 3 requests occurs 91 times, i.e. (3,91). Therefore, same dynamic block size can be combined from various numbers of requests. For another example, a dynamic block of 10 sectors can be formed from 4 requests that may be 2 requests of three sectors and 2 requests of two sectors, or 1 request of five sectors, 1 request of three sectors and 2 requests of one sector, or ..., etc. This combination of a 10-sector dynamic block from 4 requests occurs 31 times, i.e. (4,31).

We have observed that many requested blocks combine together to form a larger dynamic block. Table 4.5 shows the number of 1-sector dynamic block and the number of 1-sector request.

	Access	Dbase	Excel	Word
1-sector Dynamic Block	78,031	212,072	67,053	623,146
1-sector Request	4,809,434	1,112,830	513,061	1,660,561
Uncombined Percentage	1.62%	19.06%	13.07%	37.53%

Table 4.5: Frequency of 1-sector Dynamic Block Size and Request

The uncombined percentage is defined as

$$Uncombined\ Precentage\ = \frac{Frequency\ of\ 1\text{-}sector\ dynamic\ block}{Frequency\ of\ 1\text{-}sector\ request} * 100\%$$

The uncombined percentage means the percentage of 1-sector requests, that do not participate in forming a dynamic block, over the total number of 1-sector request. Therefore, it can illustrate the interrelationship between successive requests. If more successive requests can be combined to form a dynamic block, the uncombined percentage will be smaller. Since the uncombined percentage is very small, from 1.62%

for Access to 37.53% for Word. Therefore, for each 1-sector request, it has very high chance to have relation with the previous or the next one. So always prefetch should give a very good performance in this case because the sequential property of the data is very strong.

There is an interesting property in Table 4.4. We observed that even for dynamic block of 1 sector, it can be combined from more than 1 request. In our analysis, we have treated the case that $displacement = 0$, i.e. $start\ address\ of\ current\ request = end\ address\ of\ previous\ request$, can also be coalesced to a single dynamic block. Therefore, if two or more successive requests refer to the same sector, they will combine to form a single dynamic block of 1 sector. The successive requests for the same sector are due to read-then-immediately-write and write-then-immediately-read properties.

The most frequent and the largest dynamic block size (in sectors) for the four traces are shown in Table 4.6.

	Access	Dbase	Excel	Word
Largest dynamic block size	93	1,024	398	127
Frequency	2	51	1	28
Most frequent dynamic block size	4	8	8	1
Frequency	1,208,044	310,636	230,621	623,416

Table 4.6: Frequency of the Largest and Most Frequent Dynamic Block

In general, the frequency of the dynamic block size initially increases as the dynamic block size increases from 1 sector. The frequency reaches its maximum rapidly and then decreases. The decreasing rate of the frequency of the dynamic block size is different for the four traces. The decreasing rates for Access and Word are fast but the decreasing rates for Dbase and Excel are slow. For Dbase and Excel, since their decreasing rates are slow, we would expect that the performance of our algorithms on them are better than that on Access and Word. This is because more large dynamic blocks are available.

Table 4.7 shows the ten topmost largest I/O percentage dynamic blocks for the four

traces. The I/O percentage of a dynamic block is defined as

$$I/O\ percentage = \frac{dynamic\ block\ size * frequency}{total\ number\ of\ sectors\ requested} * 100\%$$

which shows the percentage of the total I/O time that various dynamic block sizes occupy when there is no cache. This measurement can more accurately show the sequential property of a trace than just look at the frequency of the dynamic block size. From the I/O percentage, we have observed that larger blocks utilize more I/O although their requested frequencies are not higher than those of small blocks.

Access			Dbase		
DB.	Freq.	I/O perc.	DB.	Freq.	I/O perc.
4	1,208,044	45.94%	8	310,636	24.37%
23	52,773	11.54%	16	40,355	6.33%
3	274,358	7.83%	14	30,201	4.15%
48	12,616	5.76%	26	16,226	4.14%
32	8,417	2.56%	17	19,264	3.21%
8	26,892	2.05%	75	4,342	3.19%
16	12,134	1.85%	60	4,657	2.74%
12	13,108	1.50%	15	17,822	2.62%
2	75,915	1.44%	2	124,306	2.44%
18	7,577	1.30%	1	212,072	2.08%
Excel			**Word**		
DB.	Freq.	I/O perc.	DB.	Freq.	I/O perc.
8	230,621	32.64%	1	623,146	24.46%
16	10,754	3.04%	8	65,687	20.63%
32	4,719	2.67%	28	4,957	5.45%
33	4,478	2.61%	6	22,816	5.37%
57	2,055	2.07%	12	8,867	4.18%
39	2,937	2.03%	4	20,812	3.27%
49	2,222	1.93%	5	16,615	3.26%
31	3,444	1.89%	16	4,484	2.82%
54	1,858	1.78%	52	1,340	2.73%
26	3,822	1.76%	7	9,582	2.63%

Table 4.7: Ten Topmost Largest I/O Percentage of Dynamic Blocks

By the concept of dynamic block, we can visualize the highly sequential property of disk access because many requests coalesce to form a larger block. If we take dynamic

block size into account, it can help to optimize our algorithms because we use the highly sequential property of the I/O requests to provide more large blocks that our algorithms operate on. This kind of combining several successive requests into a larger block has rarely considered by the traditional disk cache design.

Chapter 5

Performance Evaluation of Common Disk

For common disk, *the start-up time C1 is much larger than the transfer time C2.* In the following discussion, we generally choose C1=10 and C2=1.5. Tu is always set to 1 in order to act as the reference point. The values 10 and 1.5 are the ratios of the actual values of C1 and C2 to the actual value of Tu. All other timing values are also ratios to Tu.

We focus mainly on the absolute and relative performance of 4 different models: unified cache with always prefetch (Model 4), the basic model of partitioned cache (Model 5), partitioned cache with ASST applying to dynamic block (Model 7) and partitioned cache with SEHT (Model 8). The model of partitioned cache with ASST applying to each request, Model 6, is generally omitted in the discussion because its performance is generally poorer than that of Model 7. The comparison of performance of Model 6 and Model 7 will be discussed separately in Section 5.7.2.

5.1 The Effect Of Cache Size

As varying the cache size, we choose a fixed reference point for other cache parameters.

Block Size = 1 sector
Set Associativity = 1 way
Start-up Time C1 = 10
Transfer Time C2 = 1.5

This set of parameters will generally be fixed on the above values throughout the discussion of the effect of cache size.

In the simulation, we have examined cache sizes of 1M, 2M, 4M and 8M .

5.1.1 Trends of Absolute Reduction in Time

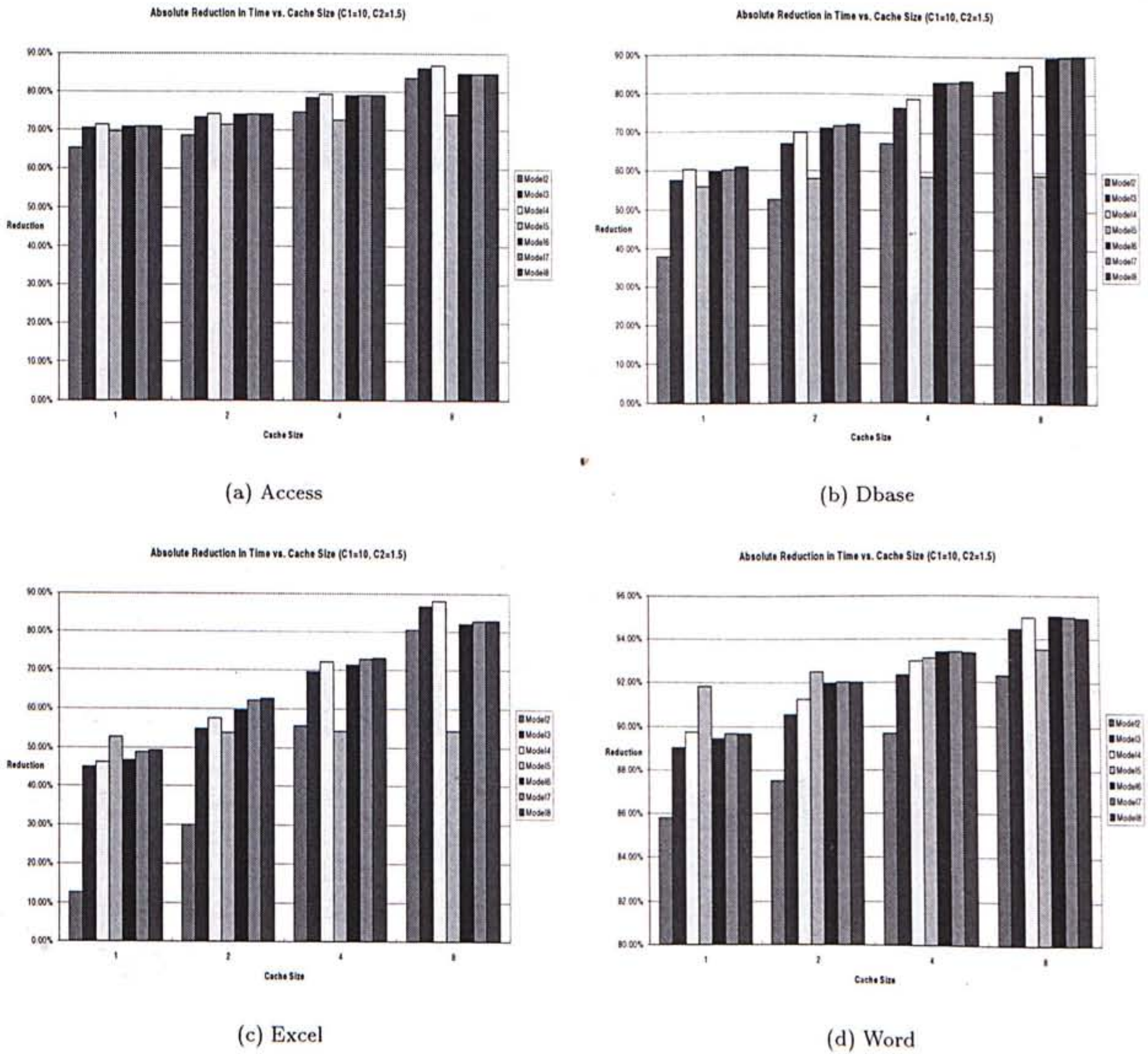
We have observed that *the absolute reduction increases for all models as the cache size increases* which can be illustrated from Figure 5.1. The absolute performance increases because large cache size implies that more data can be stored in cache.

5.1.2 Trends of Relative Reduction in Time

For clarity of graph, we omit the Model 5 in Figure 5.2. This is because the performance of Model 5 generally has a large gap with other models and we omit it to manifest the performances of other models.

5.1.2.1 Performance Of Model 4

Figure 5.2 shows that the relative performance of Model 4 gradually increases when the cache size increases. However, the increase is not large when the cache size increases from 1M to 8M. This is because the main difference between Model 4 and Model 3 is the method of getting the next block/sector. This difference will mainly reduce C2 rather than the large C1. Besides, since C2 is equal to 1.5, there is not enough time to get the next one, so each correct prefetch will reduce the transfer time. However, the time penalty C1 of cache miss for fresh reference usually dominates the access time. Therefore, the increase of relative performance of Model 4 is little. The relative performance of Model 4 is shown in the Table 5.1.



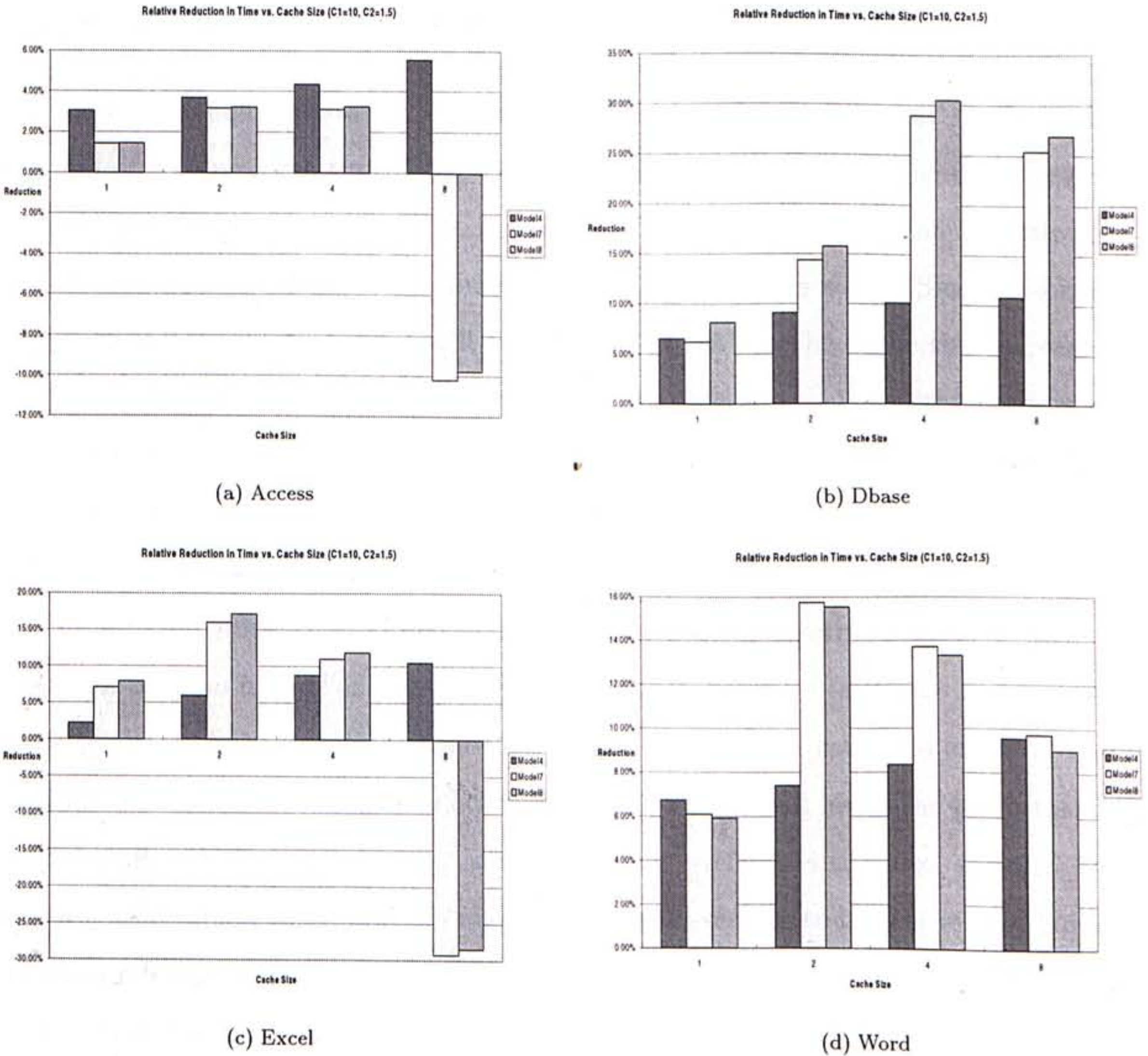
(a) Access

(b) Dbase

(c) Excel

(d) Word

Figure 5.1: Absolute Performance of Varying Cache Size



(a) Access

(b) Dbase

(c) Excel

(d) Word

Figure 5.2: Relative Performance of Varying Cache Size (without Model 5)

	Access		Dbase		Excel		Word	
Performance	Max.	Min.	Max.	Min.	Max.	Min.	Max.	Min.
Cache size	8M	1M	8M	1M	8M	1M	8M	1M
Percentage reduction in time	6%	3%	11%	7%	11%	2%	10%	7%
δpercentage	3%		4%		9%		3%	

Table 5.1: Maximum and Minimum Relative Performance of Model 4 (C2 = 1.5)

If there is enough time to get the next one, the situation is different. As shown in the graph of zero prefetch time, Figure 5.5, the relative performance slightly increases first and then slightly decrease. The increase is due to the above reason that always prefetch reduces more the transfer time when the cache size is small. Since it is zero prefetch time now, all partial hit will transfer to hit. When the cache size increases, more useful data stores in the cache. Therefore, the difference between the cache miss of Model 4 and that of Model 3 decreases. The performance of Model 4 approaches that of Model 3. So, the relative performance drops.

The positive relative performance of Model 4 shows that always prefetch is *better* than prefetch on miss. For Model 3, it is a conservative algorithm to prefetch the next sector only on miss. For non-sequential reference, Model 3 takes less useless next sectors to the cache. For highly sequential reference, Model 3 just takes one next and then wait to another miss to get another next one, so this will lower the performance of the system. On the other hand, Model 4 always prefetches the next sectors. For non-sequential reference, it puts too many useless next sectors to the cache. For highly sequential reference, Model 4 gets the correct next one, so the reduction of I/O time is greater. Now, for all traces, they exhibit a highly sequential property. The prefetched sector is very likely to be referenced soon. This can be shown from the formation of dynamic block in Section 4.5. Therefore, always prefetch can perform well.

5.1.2.2 Performance Of Model 7 And Model 8

We have observed that the trends of Model 7 and Model 8 are similar in Figure 5.2. When the cache size increases, their relative performances increase. Up to a certain limit, about 8M cache size, their relative performances drop. Figure 5.3 shows the general trend.

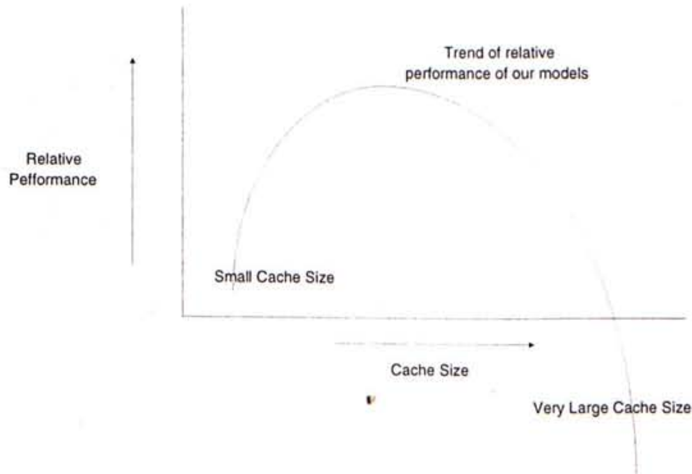


Figure 5.3: Trend of Relative Performance of Model 7 and Model 8

From the graph of absolute performance, Figure 5.1, we note that there is *actual reduction* in time when the cache size increases. Therefore, the dropped relative performances mean that the *increase* in absolute reductions in time of Model 7 and Model 8 is just less than that of Model 3. This is the property of Model 7 and Model 8 because *they are built for small/limited cache*. It is the situation where the cache is not large enough to hold the current working data set. To maximize the performance under this situation, we discard some content blocks in order to store more distinct starting heads. Then by proper overlapping the program execution and the data fetching, the cache system can have enough time to get the un-stored sectors. In other words, Model 7 and Model 8 effectively shift the performance of a small cache to look like a larger ordinary cache. This can be verified by the fact that there is an increase in relative performance when the cache size is increased from 1M to 4M.

When the cache size increases to 8M, the relative performances of Model 7 and Model 8 drop because *the cache becomes large enough to hold much more useful data, i.e. not only the heading sectors but also the content sectors.* The contribution of the enlarged cache size by Model 7 and Model 8 becomes less effective. That means the extra stored blocks cannot obtain advantage but the compulsorily discarded sectors may provide bad effect on the performance. Therefore, there is a fall in the relative performance.

There are five factors affecting the performances of Model 7 and Model 8 when compared with the performance of Model 3:

1. *the increase in the number of distinct starting heads stored in the cache*
2. *how many heading reuses*
3. *how many correct prefetches are killed due to the slow data bus, i.e. there is not enough time to get those prefetched sectors before a demand fetch arrives*
4. *the size of the cache when comparing with the working set of a trace*
5. *the time difference between reuses*

For the first factor, the more starting heads are stored in the BTC, the more chances are for cache hits. In ideal case, i.e. all reuses are in heading base, the performance of Model 7 and Model 8 must be better than the others. However, there are actually some non-heading reuses, so the stored blocks in BTC cannot provide enough time to prefetch the remaining sectors. Therefore, there is time penalty for each non-heading reuse. The performances of Model 7 and Model 8 will drop when the effect of non-whole block reuse *accumulates* to a certain level. The third factor is another tradeoff of Model 7 and Model 8. The two models discard some contents of a dynamic block and rely on the I/O bus to get the un-stored parts from the disk. There will be a situation that an un-stored part is being prefetched from the disk but another request comes to get other sectors. The prefetch must be killed in order to serve the demand fetch.

However, if the killed prefetched part will be used in very soon, the killing behavior will make the performance poor.

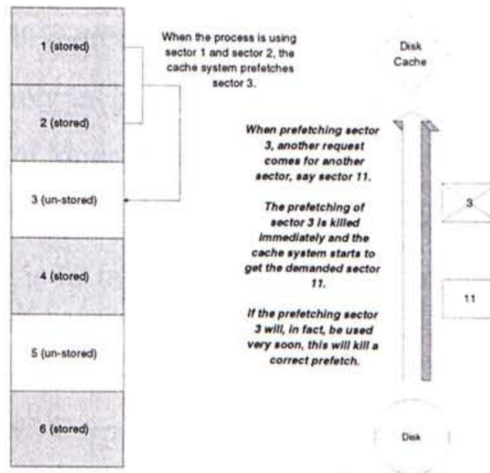


Figure 5.4: Killing of Correct Prefetch

The killing of correct prefetches has the consequence that it will downgrade the performances of Model 7 and Model 8. The killing effect will also accumulate as the cache size increases. Therefore, when the cache size becomes large enough and the accumulated effect of killing correct prefetches becomes dominant, the relative performances of Model 7 and Model 8 drop. However, theoretically, the killing of correct prefetches can be prevented because it is due to the fact that *prefetching is not fast enough*, i.e. the data bus is slow and has limited bandwidth. This is the limitation of current bus speed and bandwidth. If the bus speed becomes faster and faster, this factor will be greatly reduced. In an extreme case, if the *prefetch* were *infinitely fast*, this factor would be completely eliminated.

The cache size is a very important factor for Model 7 and Model 8. When the cache size is *too small*, the extra stored starting heads do not have time to be reused before they are replaced by other sectors, i.e. the time between reuses cannot co-operate with the cache size. Besides, in this time, the disadvantages of the models still exist. Therefore, the relative performances of Model 7 and Model 8 are poor in very small cache size. A *threshold* must exist so that the extra stored starting heads become

useful and then the relative performances increase greatly. However, the threshold varies greatly because it highly depends on the properties of the traces. We can only observe its effect from the result of simulation.

When the cache size becomes very large, the accumulated disadvantages become dominant and the effectively increased cache size becomes less important. Therefore, the relative performances of Model 7 and Model 8 drop dramatically. They may even be poorer than Model 3 although all models have an increase in absolute reduction in time. However, when the cache is in *intermediate size*, the increase in relative performance can be as high as 29%.

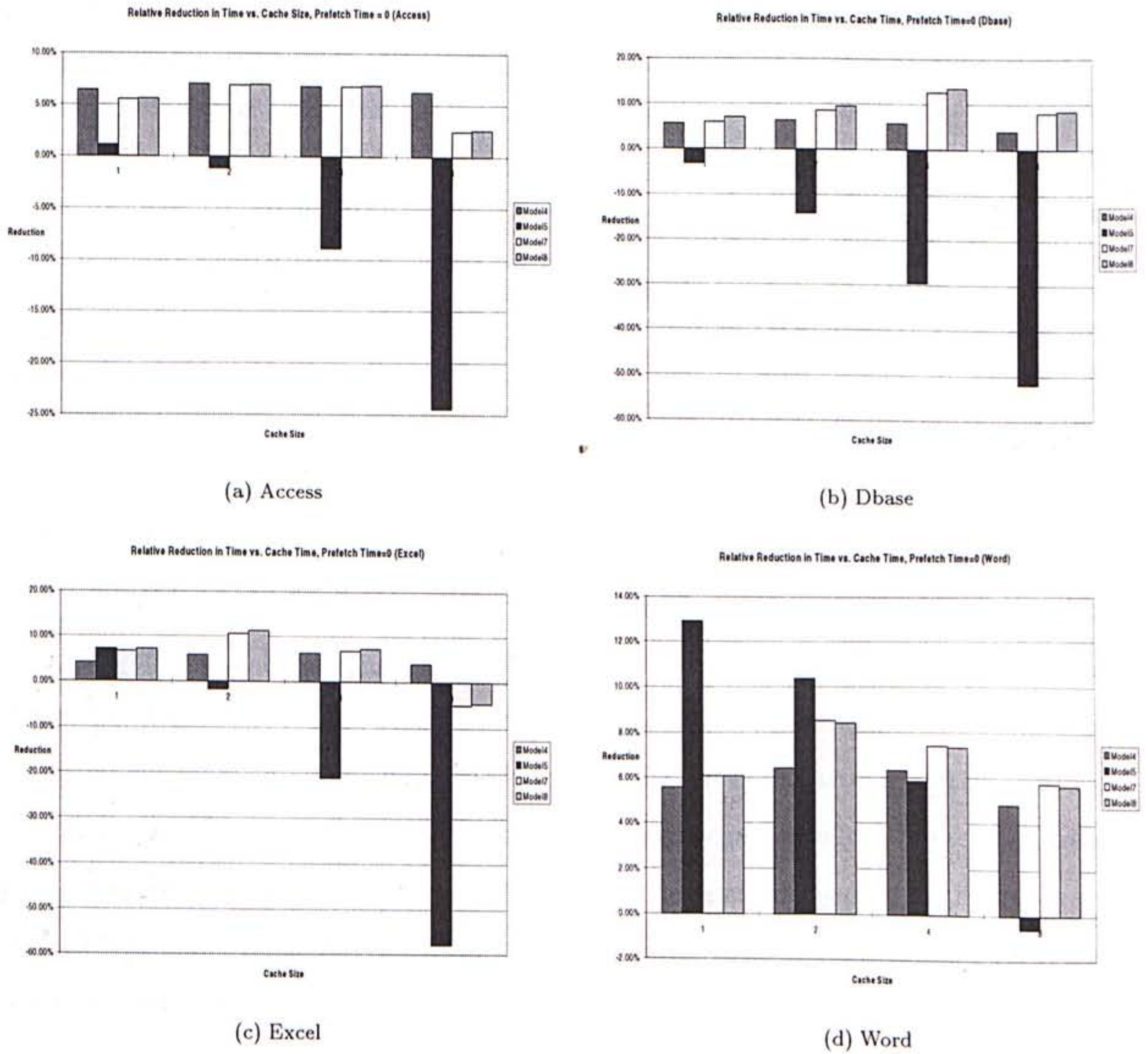
	Access		Dbase		Excel		Word	
Performance	Max.	Min.	Max.	Min.	Max.	Min.	Max.	Min.
Cache size	2M, 4M	8M	4M	1M	2M	8M	2M	1M
Percentage reduction in time	3%	-10%	29%	6%	16%	-29%	16%	6%

Table 5.2: Maximum and Minimum Relative Performances of Model 7 (C2 = 1.5)

The maximum and minimum relative performances of Model 7 for different traces are shown in Table 5.2. The relative performance of Model 8 is similar. From Table 5.2, the maximum relative performance usually occurs in 2M or 4M cache size. The minimum usually occurs in 1M or 8M cache size. The minimum value can drop to negative, e.g. -10% for Access and -29% for Excel. The negative values are due to the accumulated effect of the second and third factors that have been discussed before.

However, for the third factor of killing correct prefetch, it can be eliminated by the *zero prefetch time*, i.e. infinitely fast prefetching, because it is due to the slow bus speed and the limited bandwidth. Therefore, we have simulated the effect of *zero prefetch time* in order to investigate how large the effect of the third factor plays in the cache system. Note that the *fetch time* is still equal to 1.5. Figure 5.5 shows the relative performances of different models for zero prefetch time.

The simulation of the zero prefetch time shows that there are actual occurrences of



(a) Access

(b) Dbase

(c) Excel

(d) Word

Figure 5.5: Relative Performance of Varying Cache Size with Prefetch Time=0

killing correct prefetch. Table 5.3 shows the maximum and minimum relative performances of Model 7 when the prefetch time is equal to zero.

	Access		Dbase		Excel		Word	
Performance	Max.	Min.	Max.	Min.	Max.	Min.	Max.	Min.
Cache size	2M	8M	4M	1M	2M	8M	2M	8M
Percentage reduction in time	7%	2%	13%	6%	11%	-5%	9%	6%

Table 5.3: Maximum and Minimum Relative Performances of Model 7 (Prefetch Time = 0)

The negative relative performance for Access has been eliminated, so the effect of killing correct prefetch is quite important in the Access trace. On the other hand, the negative relative performance for Excel still exists even in zero prefetch time although the relative performance increases from -27% to -5%. So the reason is not just in the factor of killing correct prefetch. Another reason is that the entire working set of Excel has gone into the 8M cache. This can be illustrated from the hit ratio of Model 2, unified cache without any prefetching technique, which is 95% in a 4-way set associative 8M cache. Model 7 and Model 8 discard some contents of each dynamic block *compulsorily*. For very large cache size that can capture almost all reuses, an enlarged cache size does not mean anything. The accumulated effect of non-heading references becomes dominant and this factor cannot be eliminated in our algorithms. Therefore, the relative performance for Excel drops to negative value. In fact, the trend for the drop of relative performance to negative value is also expected in the other two traces, Dbase and Word. However, the cache size is not large enough to show this effect for Dbase and Word.

5.1.2.3 Comparing Performance Of Model 7 And Model 8 With Model 4

The relative performance of Model 4 is sometimes *much poorer* than that of Model 7 and Model 8, especially when the cache size is about 2M to 4M. *At the range of intermediate cache size, Model 7 and Model 8 have the advantage of always prefetch which can capture the highly sequential property of the traces. The extra stored blocks in*

BTC by ASST or SEHT can be reused effectively. Besides, PB can reduce the pollution due to prefetching. Therefore, the relative performances of Model 7 and Model 8 can be boosted up greatly. Table 5.4 shows the maximum relative performances of Model 7. The performances of Model 8 are similar to that of Model 7.

	Cache size	Maximum relative performance of Model 7	Maximum relative performance of Model 4	Ratio: $\frac{Model7}{Model4}$
Access	2M	3%	4%	0.75
Dbase	4M	30%	10%	3
Excel	2M	17%	6%	3
Word	2M	16%	7%	2

Table 5.4: Maximum Relative Performance of Model 7

At the maximum throughput, the relative performances of Model 7 and Model 8 can *double*, or even *triple* that of Model 4. This reveals the fact that the new algorithms, ASST and SEHT, are very useful in the intermediate cache size. However, the case is different for Access. Even in the maximum relative performance, the performances of Model 7 and Model 8 are still worst than that of Model 4. This can be explained by the fact that there are *too many occurrences* of killing correct prefetch and *too many* non-heading reuses for Access. The un-stored parts of a block cannot be accurately prefetched. Therefore, the performances of Model 7 and Model 8 become poor for Access.

Besides, for 1M cache size, the relative performances of Model 7 and Model 8 are usually lower than that of Model 4. This is due to the fact that the threshold of the models have not been exceeded in 1M cache size. Therefore, the extra stored starting blocks cannot be effectively used and the factors of disadvantages lower the performance.

In conclusion, always prefetch is a useful technique to disk cache. Model 7 and Model 8 are designed for the limited cache size when compared with the data size. From the simulation results, Model 7 and Model 8 can perform very good in the intermediate cache size, about 2M to 4M, because the effectively enlarged cache size can store more

starting blocks. Those blocks can be used effectively by ASST and SEHT and this gain can cover the disadvantages of the models. The relative performances of Model 7 and Model 8 can even double/triple that of Model 4.

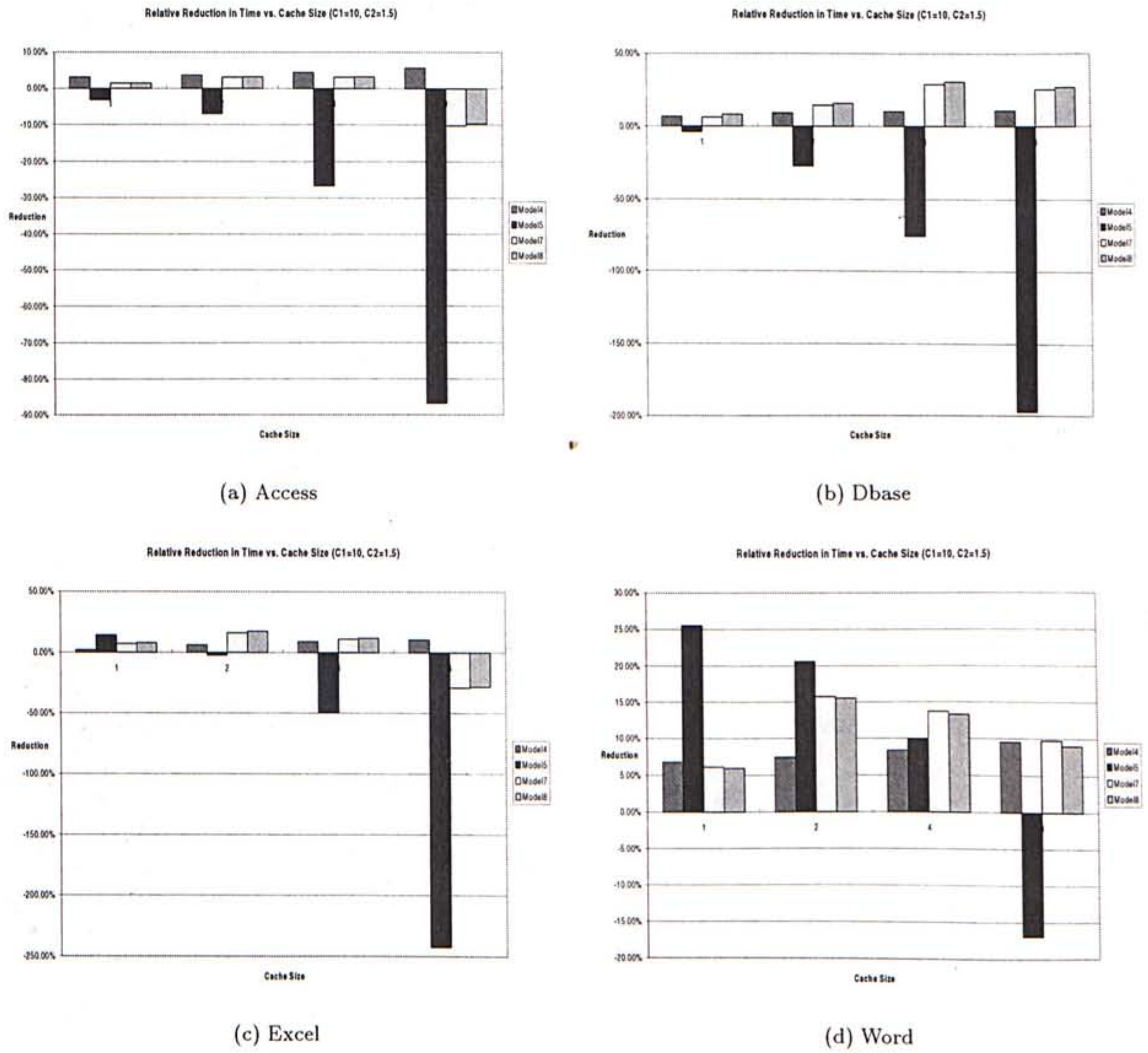
5.1.2.4 Performance Of Model 5

We have omitted Model 5 to get a clearer discussion in before. Now, let us focus on the relative performance of Model 5. Figure 5.6 shows the relative performances for the four models, including Model 5 for different traces. We have observed that the performance of Model 5 is worse than that of Model 3, i.e. negative values, except in small cache size for Excel trace and Word trace.

The performance of Model 5 is poor because it stores only the *first* heading block of each dynamic block in the BTC and lets the cache system get the following blocks. This produces *a large extra time penalty needed to pay for each reuse.* For large cache, Model 5 *underuses* the cache because it *compulsorily* discards the *all remaining blocks* although there are enough spaces to hold them. Therefore, the extra time penalty paid is greater. Model 5 is a control model and it directly used the CPU cache partitioning technique without any modification. The disadvantages of this model have been discussed in Section 3.3.2. This indirectly shows that there are some differences with disk cache and CPU cache. The techniques in CPU cache may need modify before they are applied to disk cache design.

For Excel trace, the hit ratio for 1M cache size is only 10% for unified cache, e.g. Model 2. This is a very low value and shows that ordinary cache store *very little* useful data. Almost all the time, the cache system needs to take the requested data from disk. The algorithm of Model 5 *dramatically enlarges* the cache size, i.e. data from more dynamic blocks can be stored in the cache by only storing the first block of each non-sequential reference. Therefore, more cache hits can occur. The time saving due to more cache hits compensates the extra time penalty paid to take the remaining blocks from disk.

For Word trace, the most frequent dynamic block size is *1 sector*. Model 5 can



(a) Access (b) Dbase

(c) Excel (d) Word

Figure 5.6: Relative Performance of Varying Cache Size

do a good job because it does not pick any useless next sectors to the BTC, and the next sectors all go to the PB. For other models, the control algorithms place some next sectors in the BTC so cause cache pollution in this case. Therefore, Model 5 shows an extra-ordinarily good performance.

5.2 The Effect Of Block Size

As varying the block size, we choose a fixed reference point for other cache parameters.

Cache Size = 4M
Set Associativity = 1 way
Start-up Time C1 = 10
Transfer Time C2 = 1.5

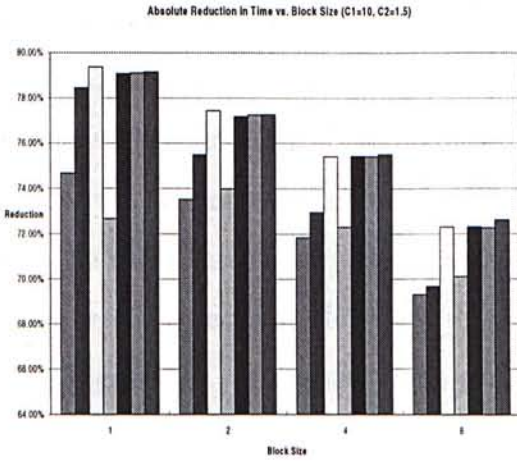
This set of parameters will be fixed on the above values throughout the discussion of the effect of block size. In the simulation, we have examined block sizes of 1 sector, 2 sectors, 4 sectors and 8 sectors.

5.2.1 Trends of Absolute Reduction in Time

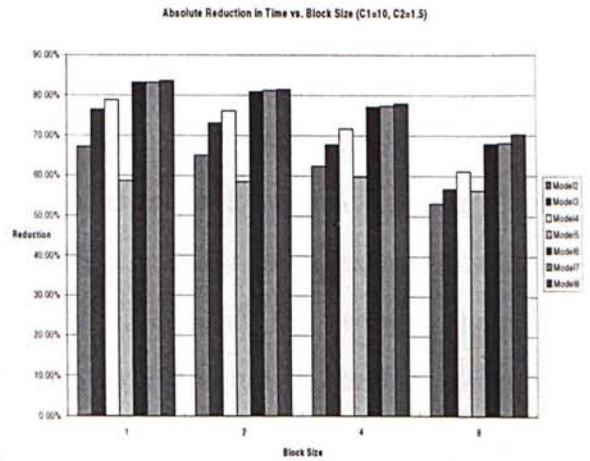
We have observed that *in general, the absolute reduction decreases as the block size increases* which is shown in the Figure 5.7.

The decrease in the absolute reduction means that the absolute performance is poorer in larger block size. Besides, there is an obvious drop in absolute performance when the block size increases from 4 sectors to 8 sectors for all traces.

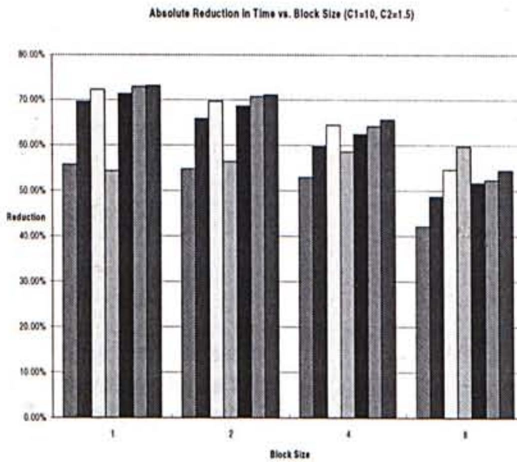
Block size has an effect of *implicit prefetching*. Many current designs of disk cache using very large block size since they have not incorporated the ability of explicit prefetching technique. Large block gets many adjacent sectors of the requested one to the cache. These sectors are hoped to be referenced later. For instance, requesting a sector will let the whole disk track to be fetched to the cache. Large block size is proved to be very useful in current design due to its implicit prefetching property.



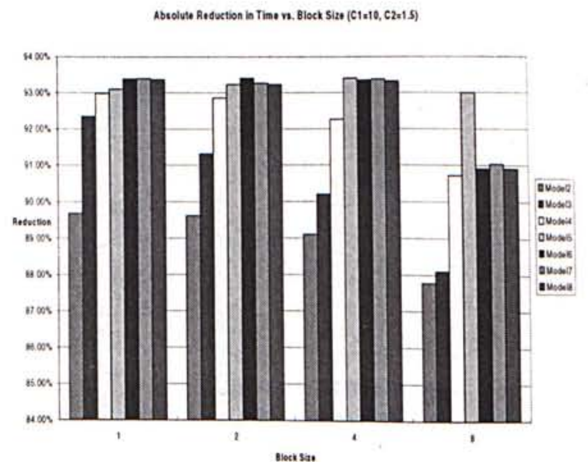
(a) Access



(b) Dbase



(c) Excel



(d) Word

Figure 5.7: Absolute Performance of Varying Block Size

However, *our models have already incorporated the technique of always prefetch.* If the block size is too large, it may cause serious cache pollution, i.e. there are too many useless data placed in the cache so useful data are kicked out. On the other hand, if the adjacent data are useful and will be referred soon, increasing block size causes more cache hits. Therefore, when always prefetching technique combines with the small block size, it can improve the performance. However, when the block size is too large, the combined technique takes too many other sectors to the cache and does not know whether those sectors are useful.

Now, let us consider a case for a request of 4 sectors and using a 4-sector block size. The most satisfactory result is that the data in a transfer block exactly matches the requested 4 sectors. However, this is not the case in general. On average, the case is like Figure 5.8. A request for n sectors, where n is also equal to the block size, is usually across two transfer blocks.

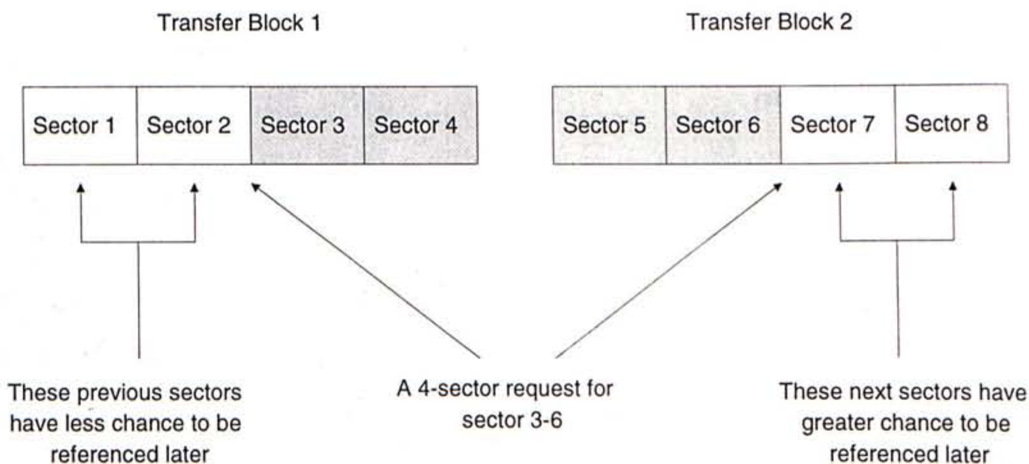


Figure 5.8: A 4-sector Request in Block Size of 4 sectors

In Figure 5.8, the previous sectors, sector 1 and sector 2, have less chance to be referenced later. Therefore, these two sectors pollute the cache. The next sectors, sector 7 and sector 8, have higher chance to be referenced later. Therefore, these two sectors may reduce the disk access time. As the block size increases, more and more previous and next sectors go into the cache. To determine the usefulness of these

sectors, it highly depends on the reference pattern of a particular application.

From analyzing each trace, the most frequent dynamic block sizes are usually not greater than 8 sectors. In the above example, the next sectors, sector 7 and sector 8, have higher chance to be referenced soon because the total number of sector is not greater than 8 after combined with the requested sectors. However, for an 8-sector block size, the situation is different. Consider a case of a request of 8 sectors and the block size is also equal to 8 sectors. As discussed before, the situation is like Figure 5.9.

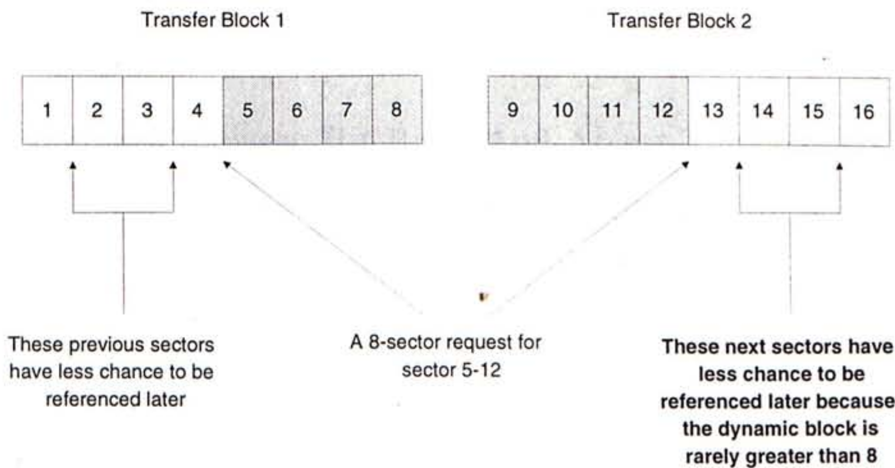


Figure 5.9: A 8-sector Request in Block Size of 8 sectors

The previous sectors are useless as before. The next sectors are also useless in this situation because the combined effect gives a dynamic block size larger than 8 which rarely appears in the four traces. Therefore, when two blocks are placed in the cache, almost a whole block is useless. This causes the poorer performance for 8-sector block size.

From the above result, since we use always prefetch as our basis, 1-sector block size gives the most satisfactory result. 1-sector block size can also give a *full control* of the cache block in the cache system. While the block size increases, more useless sectors will be fetched into the cache as discussed above. Those useless sectors stick with the useful sectors and occupy entries in the cache. In turn, less space is left for useful data. So a large fetched block will also have chance to pollute the cache. Therefore, for

1-sector block size, it can ensure that the fetched/prefetched blocks have higher chance to be useful.

Moreover, we have observed that Model 5 is not like other models, the performance of Model 5 may *not* decrease as the block size increases. This is due to the fact that Model 5 stores only the *first starting block* of each non-sequential reference, i.e. the first block of each dynamic block. It needs to pay very high time penalty *even for a cache hit.* However, as the block size increases, the first starting block becomes larger. The penalty of each cache hit is reduced. So, the effect of bringing undesired sectors into the cache can be compensated.

5.2.1.1 Difference Between Hit Ratio And Access Time

We have chosen the *absolute reduction in time* and the *relative reduction in time* to indicate the performance of a model. The reason of choosing access time rather than hit ratio is that it can provide a better insight of the performance of the model. This can be illustrated from Table 5.5 that shows the hit ratio and disk access time ratio of Model 7 for different block size.

	Access		Dbase		Excel		Word	
Block size	Hit ratio	Time ratio	Hit ratio	Time ratio	Hit ratio	Time ratio	Hit ratio	Time ratio
1 sector	77.93%	0.2089	80.31%	0.1673	63.48%	0.2713	84.46%	0.0661
2 sectors	84.35%	0.2276	80.13	0.1883	64.94%	0.2933	90.31%	0.0673
4 sectors	89.20%	0.2459	81.64%	0.2259	66.90%	0.3579	93.34%	0.0663
8 sectors	91.83%	0.2774	82.28%	0.3183	70.04%	0.4769	96.08%	0.0896

Table 5.5: Hit Ratio and Disk Access Time Ratio for Model 7

The *Time ratio* is defined as

$$Time\ ratio = \frac{Total\ disk\ access\ time\ of\ a\ model}{Total\ disk\ access\ time\ of\ no\ cache}$$

Therefore, the larger the time ratio is, the poorer the performance is. In Table 5.5, as the block size increases, the hit ratio increases. Increasing hit ratio indicates that increasing block size is very useful because more requested data are in the cache. However, the time ratio also *increases*. That means *the actual traffic between disk and*

cache is heavier as the block size increases. Therefore, choosing a large block size is not intelligent because it imposes a *heavier traffic* between disk and cache. From this situation, we have observed that hit ratio can only give a *rough* understanding on the performance of a model. Time ratio gives a more concrete understanding on the traffic between disk and cache, which in turn is an accurate indicator of the performance of a model.

Besides, from the above analysis, we have observed that increasing block size has its advantage to capture the spatial locality of references. But owing to the fact that it also takes some extra useless sectors in the cache, it increases the total access time to the disk. This can further verify by the actual number of sectors transferred between disk and cache when the block size varies. Figure 5.6 shows the actual number of sectors transferred for Dbase trace (Cache Size=4M, C1=10, C2=1.5, Set Associativity=2-way). The number of transferred sectors generally increases when the block sizes increases.

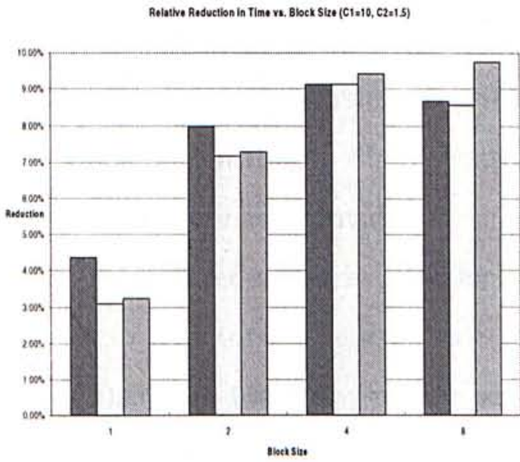
Model	BS=1-sector	BS=2-sectors	BS=4-sectors	BS=8-sectors
2	1,944,730	2,240,546	2,573,892	3,751,840
3	1,955,952	2,266,172	2,587,868	3,764,824
4	1,967,134	2,290,496	2,639,212	3,855,064
5	8,497,649	8,539,820	7,647,628	7,491,760
6	2,039,514	2,254,532	2,398,484	3,443,088
7	1,928,586	2,226,166	2,375,272	3,425,904
8	1,909,416	2,235,700	2,357,828	3,215,992

Table 5.6: Actual Number of Sectors Transferred for Dbase when Varying Block Size

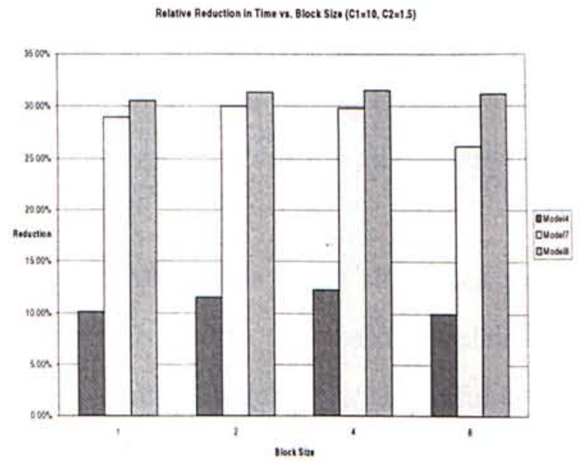
5.2.2 Trends of Relative Reduction in Time

5.2.2.1 Performance Of Model 4, Model 7 And Model 8

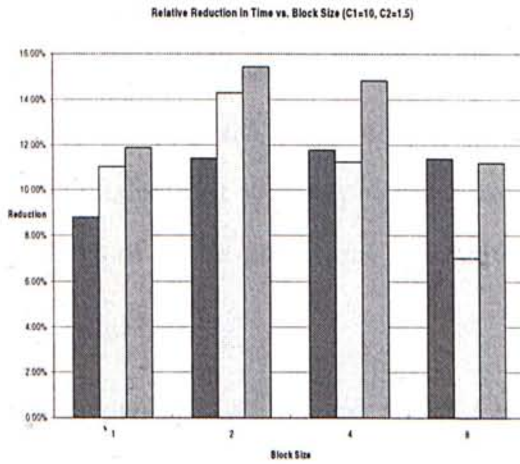
The relative performance is the absolute performance of a model compared with the absolute performance of Model 3. *Although the absolute reduction decreases in general,*



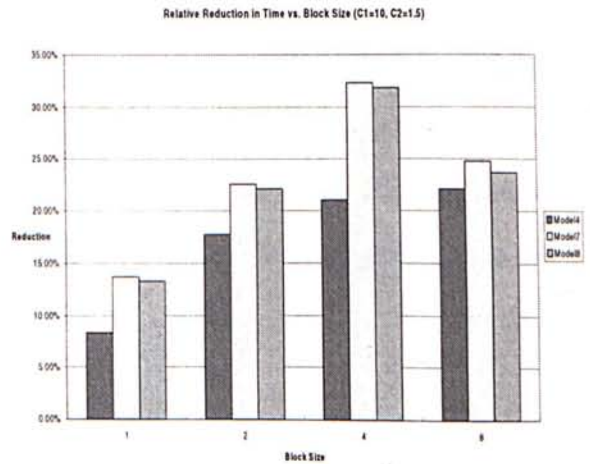
(a) Access



(b) Dbase



(c) Excel



(d) Word

Figure 5.10: Relative Performance of Varying Block Size (without Model 5)

the relative reduction can increase. This means that the drop in performance of a model is less than that of Model 3.

The trends of relative reduction in time are quite similar for different models. When the block size changes from 1 sector to 4 sectors, there is an increase in the relative reduction in time for different models. Although the absolute reduction in time decreases, the effect of slightly larger size of the prefetched block has the advantage that the adjacent data will be referenced soon (refer to the discussion in the Trends of absolute reduction in time). The increase in relative performance verifies that there are actual use of those adjacent data.

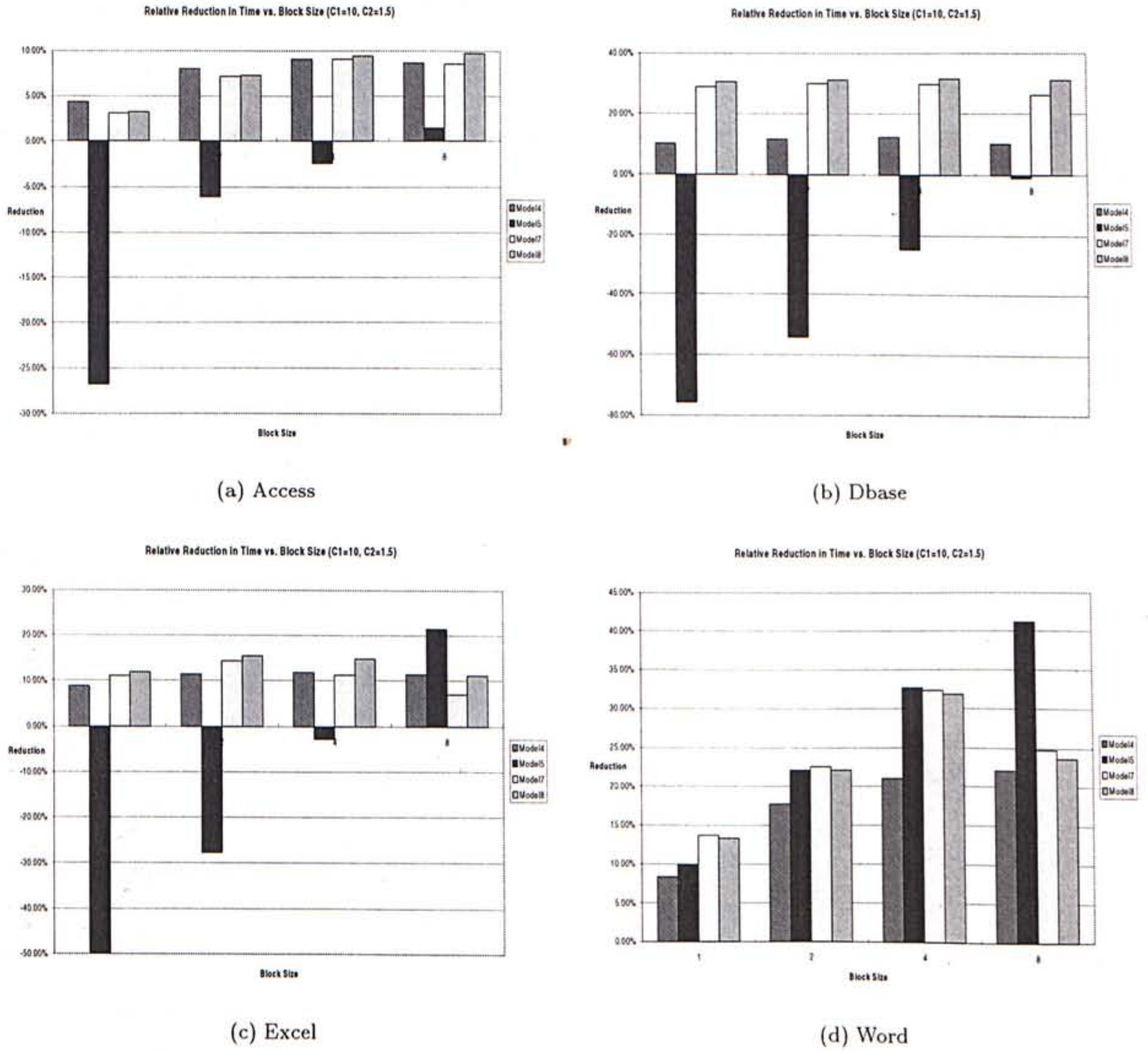
The relative reduction in time for all models generally drops when the block size increases to 8 sectors. The drop in 8-sector block size is due to the reason explained before. There are too many useless sectors being prefetched together with useful sectors. Those useless sectors *pollute* the cache *more* for the models using always prefetch technique.

For Model 7 and Model 8, the increase in relative reduction in time is greater than that of Model 4. The amplitude of increase for Model 7 and Model 8 is larger because they have a *prefetch buffer* to store the prefetched sectors. Those prefetched sectors will be flushed out very rapidly due to the small size of PB. Therefore, PB can reduce the effect of cache pollution, i.e. reduce the number of useless sectors going into the BTC.

5.2.2.2 Performance Of Model 5

We have omitted Model 5 to get a clearer discussion in before. Now, let us focus on the performance of Model 5. Figure 5.11 is plotting for relative performance, including Model 5, for different traces.

For all traces, the relative performance of Model 5 is better and better as the block size increases. This is because Model 5 underuses the cache by storing only the first heading block of each dynamic block. Now, as the block size increases, the first heading block will contain more sectors and in turn, more sectors in each dynamic block are



(a) Access

(b) Dbase

(c) Excel

(d) Word

Figure 5.11: Relative Performance of Varying Block Size

stored in the cache under Model 5. Therefore, the time penalty of getting the next sector is smaller and then the relative performance is better. However, Model 5 is still below the standard in many cases except for the case of 8-sector block size. For 8-sector block size, Model 5 can be better than the standard owing to the fact that the most frequent dynamic block sizes are under 8 sectors. If the system stores 8 sectors as a whole each time, nearly all dynamic blocks are stored in the cache. The system will not need to pay too much time penalty to get the remaining sectors (comparing with the case of block size equal to 1 and storing only the first block). Therefore, more cache hits occur and there is greater reduction in time.

5.3 The Effect Of Set Associativity

As varying the set associativity, we choose a fixed reference point for other cache parameters.

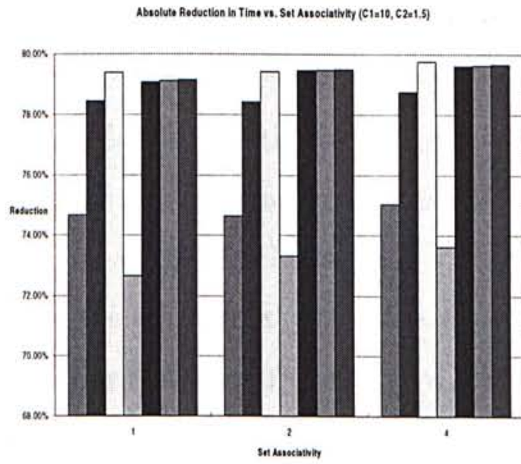
Cache Size = 4M
Block Size = 1 sector
Start-up Time C1 = 10
Transfer Time C2 = 1.5

In the simulation, we have examined 1-way, 2-way and 4-way set associativities.

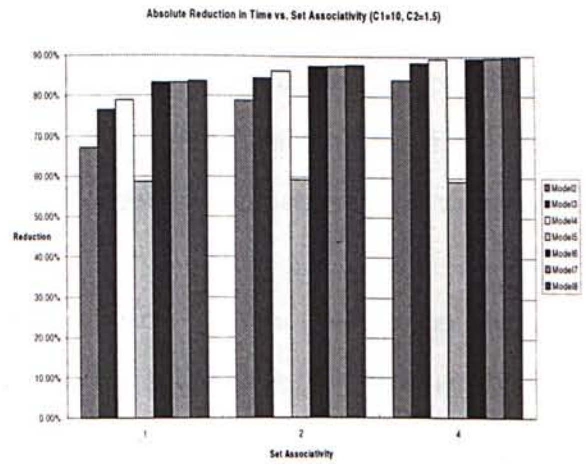
5.3.1 Trends of Absolute Reduction in Time

We have observed that *when the set associativity increases, the absolute reduction in time increases.* This is shown in Figure 5.12.

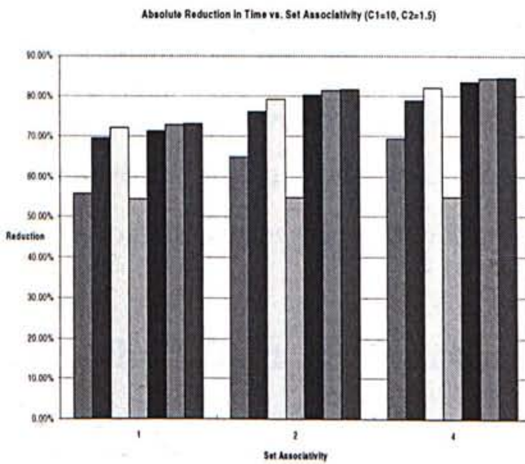
When a block must go in exactly one place in the cache, the placement scheme is called direct mapped or 1-way set associative. When a block can be placed anywhere in the cache, the placement scheme is called fully associative. The intermediate design is called n-way set associative. In a set associative cache, there is a fixed number of locations where each block can be placed. A set associative cache with n locations for a block is called an n-way set associative cache or its set associativity equal to n. Each



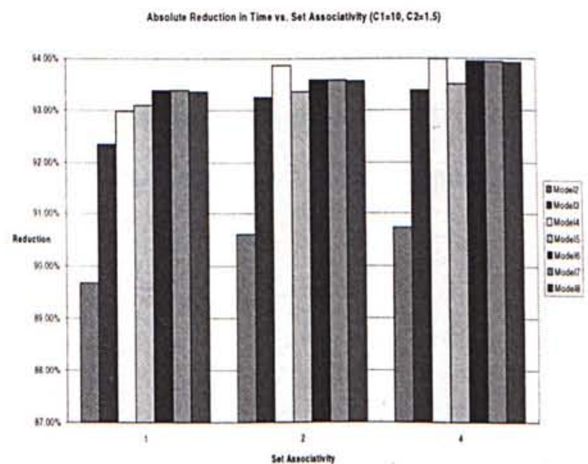
(a) Access



(b) Dbase



(c) Excel



(d) Word

Figure 5.12: Absolute Performance of Varying Set Associativity

block now maps to a unique set in the cache, and a block can be placed in any entries of that set.

We have observed that increasing set associativity can help to increase the performances of all models. This is a fact that increasing the set associativity can reduce the collisions for competing the same entry and can improve the hit ratios of caching models. Fully associative scheme is the best one because each block can store in any place in the cache. Direct mapped scheme is the worst one because each block can go in only one place in the cache. In the case of direct mapped, if there is another block that must go in an entry, the previous stored block must be replaced even if there are other free entries in the cache.

On the other hand, there is a disadvantage if the cache system uses a large set associativity or full set associativity. The time of searching the cache becomes significant and must be taken into account. The accumulated effect of searching time increases greatly when the set associativity increases because this time is counted for each search. Therefore, in general, cache systems use less than 8-way set associativity in order to reduce the time for searching.

Therefore, set associativity is the parameter for real implementation. It is quite independent of which model is chosen. Using large set associativity can improve the hit ratio but increases the search time.

5.4 The Effect Of Start-up Time C1

As varying the start-up time C1, we choose a fixed reference point for other cache parameters.

Cache Size = 4M
Set Associativity = 1 way
Block Size = 1 sector
Transfer Time C2 = 1.5

In the simulation, the values of C1 that we have examined are 5, 10, 15, 20. Note that these values are the ratios of actual values of the start-up time to the use-up time.

5.4.1 Trends of Absolute Reduction in Time

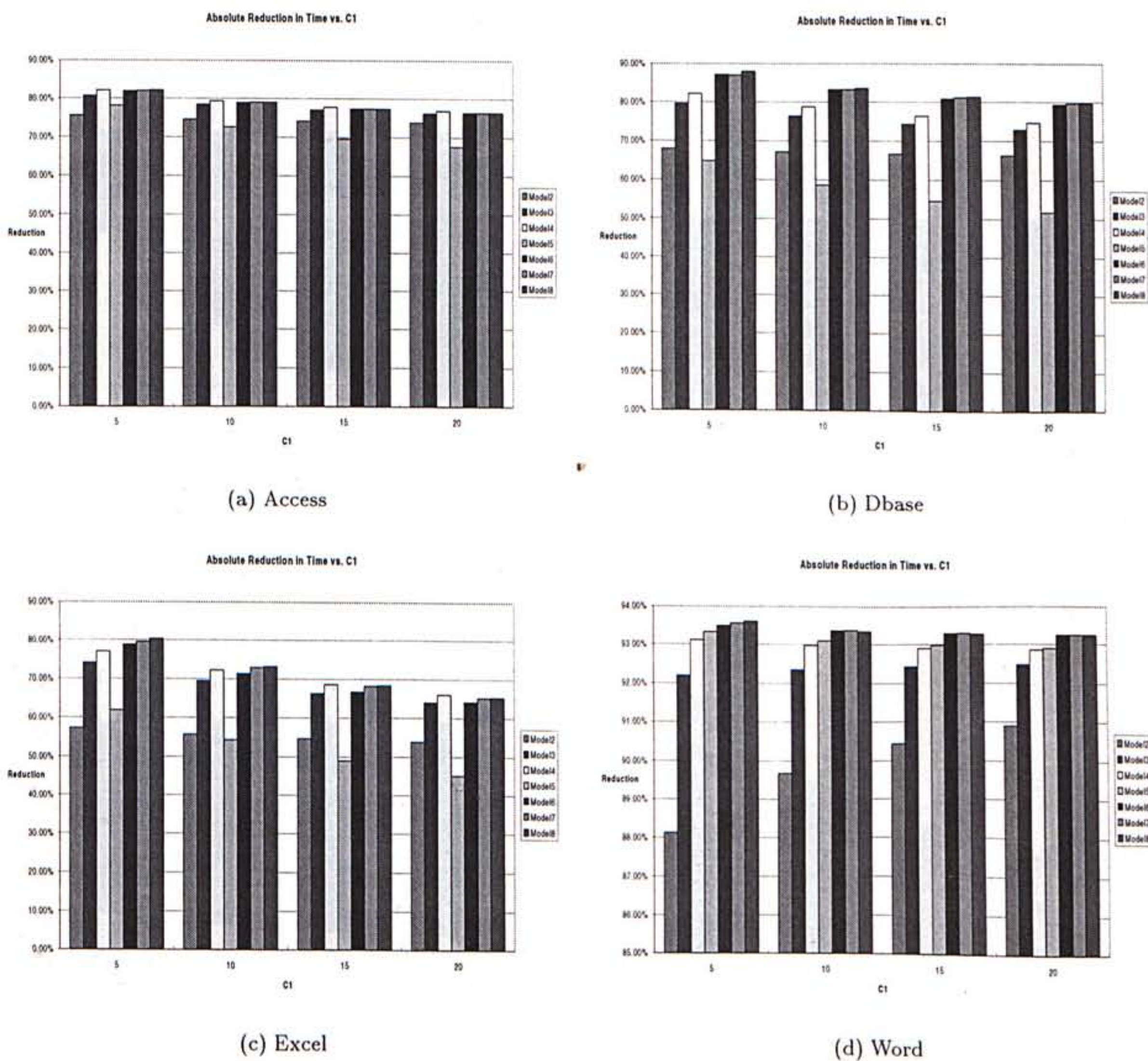
Figure 5.13 shows the absolute reduction in time of all models for different traces. Remind that the *actual time* of disk access for all models increases as C1 increases. This is because the time penalty paid for each cache miss is higher. However, when we calculate the absolute reduction in time, the result of one value of C1 *cannot* compare with the result of another C1 *because their bases are different*, i.e. the total disk access times for different C1 are different. Therefore, the trends of absolute reduction can be in any shape. So, Figure 5.13 is just for reference.

5.4.2 Trends of Relative Reduction in Time

Figure 5.14 shows the trends of relative reduction in time for different models.

When C1 increases, the relative performances of Model 4, Model 7 and Model 8 decreases because the time needed to pay for each cache miss dominates. It covers the effect of other timing factors. Different kinds of prefetching become *less important* when comparing with the overhead of the start-up time C1. Therefore, the performances of Model 4, Model 7 and Model 8 tend to the performance of Model 3. This can be verified by the fact that the relative performances drop as C1 increases.

Start-up time C1 determines the size of starting head that should be stored in BTC for Model 7 and Model 8. As C1 increases, the size of starting head is increased. So, *fewer* extra starting heads can be put into BTC. Therefore, the performances of Model 7 and Model 8 must tend to Model 4. This can be verified by the fact that the amplitude of decreasing performances of Model 7 and Model 8 is larger than that of Model 4. The trend of the relative performance of Model 5 is decreasing because of similar reason in the case of Model 7 and Model 8.



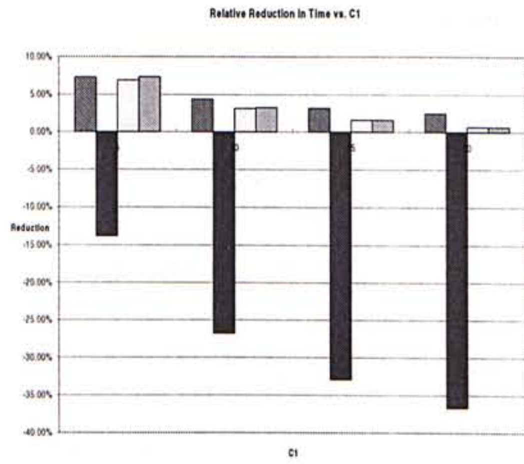
(a) Access

(b) Dbase

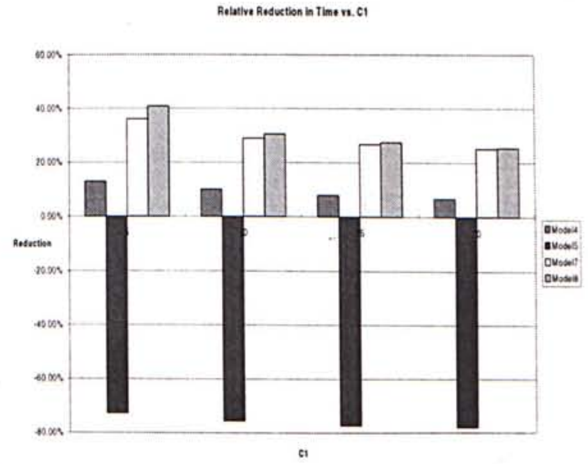
(c) Excel

(d) Word

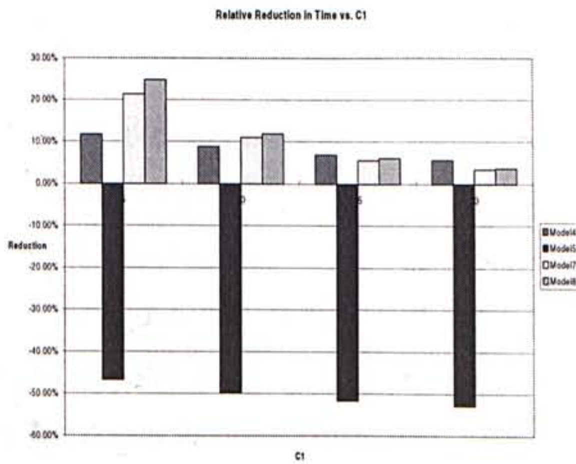
Figure 5.13: Absolute Performance of Varying Start-up Time C1



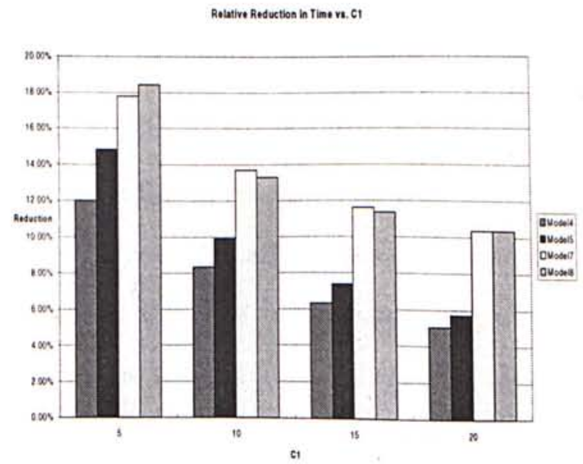
(a) Access



(b) Dbase



(c) Excel



(d) Word

Figure 5.14: Relative Performance of Varying Start-up Time C1

5.5 The Effect Of Transfer Time C2

As varying the transfer time C2, we choose a fixed reference point for other cache parameters.

Cache Size = 4M
Set Associativity = 1 way
Block Size = 1 sector
Start-up Time C1 = 10

In the simulation, the values of C2 that we have examined are 0.5, 1, 1.5. Note that these values are the ratios of actual values of the transfer time to the use-up time.

5.5.1 Trends of Absolute Reduction in Time

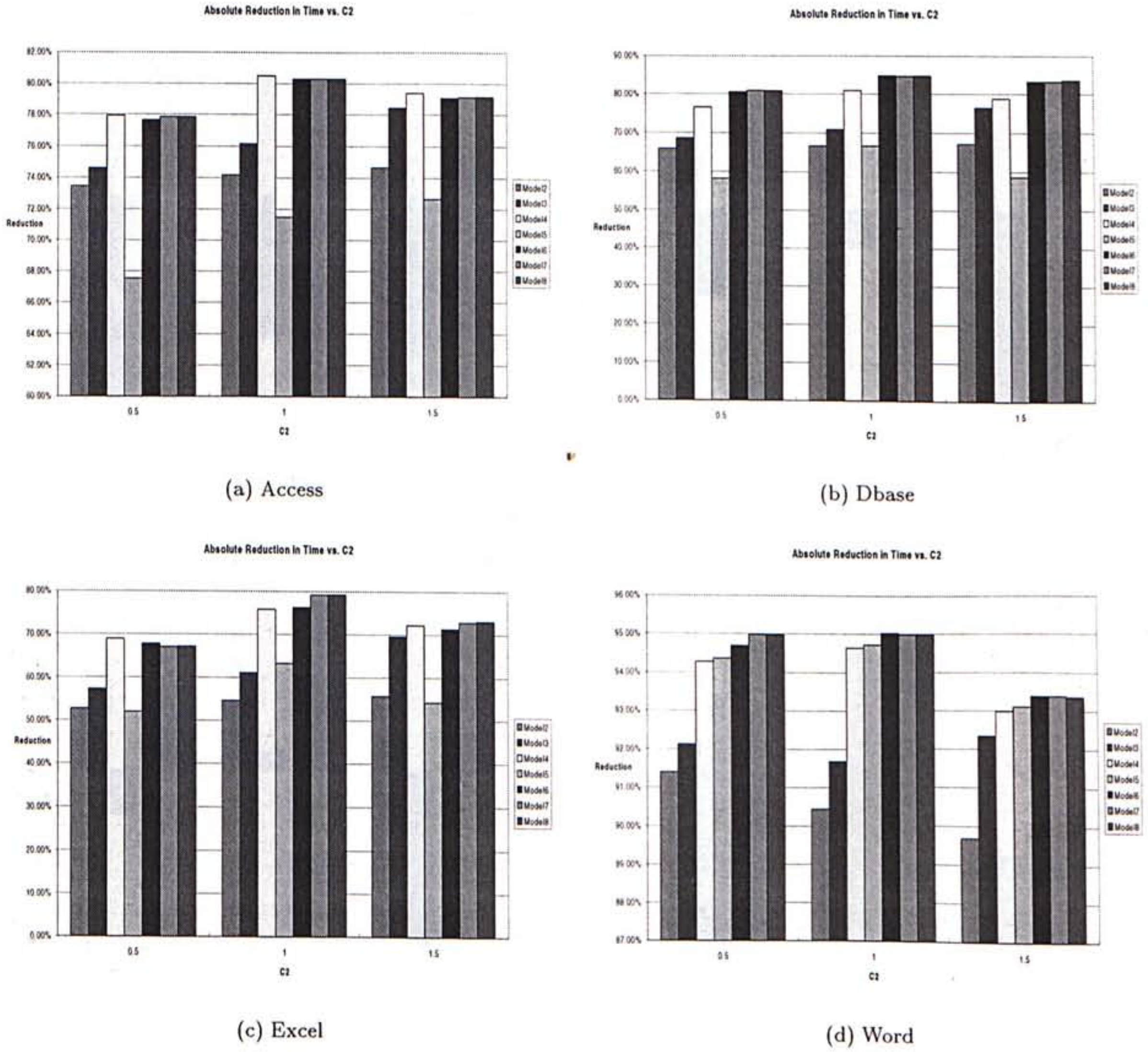
Similar to the case of varying C1, we cannot compare adjacent sets in a graph because the bases are different. Therefore, Figure 5.15 is just for reference.

5.5.2 Trends of Relative Reduction in Time

Figure 5.16 shows the relative reduction in time for different models when varying C2.

Model 4, Model 7 and Model 8 exhibit same patterns for all traces in varying C2. As C2 increases from 0.5 to 1, the relative performances increase for all models in all traces. As C2 increases from 1 to 1.5, the relative performances decrease dramatically. Remind that the value of C2 is the ratio of the actual value to the use-up time. Therefore, there are two cases in these values that are C2 less than or equal to 1, and C2 larger than 1. The value of C2 less than and equal to 1 means that there is enough time to get the next sectors/blocks when the first sector/block has already been placed in the cache. On the other hand, the value of C2 larger than 1 means that there is not enough time to get the next sectors/blocks if only the first sector/block has already placed in the cache.

For C2<=1, only the first block needs to pay the time penalty in cache miss because the following one has enough time to be transferred when the process is using the first



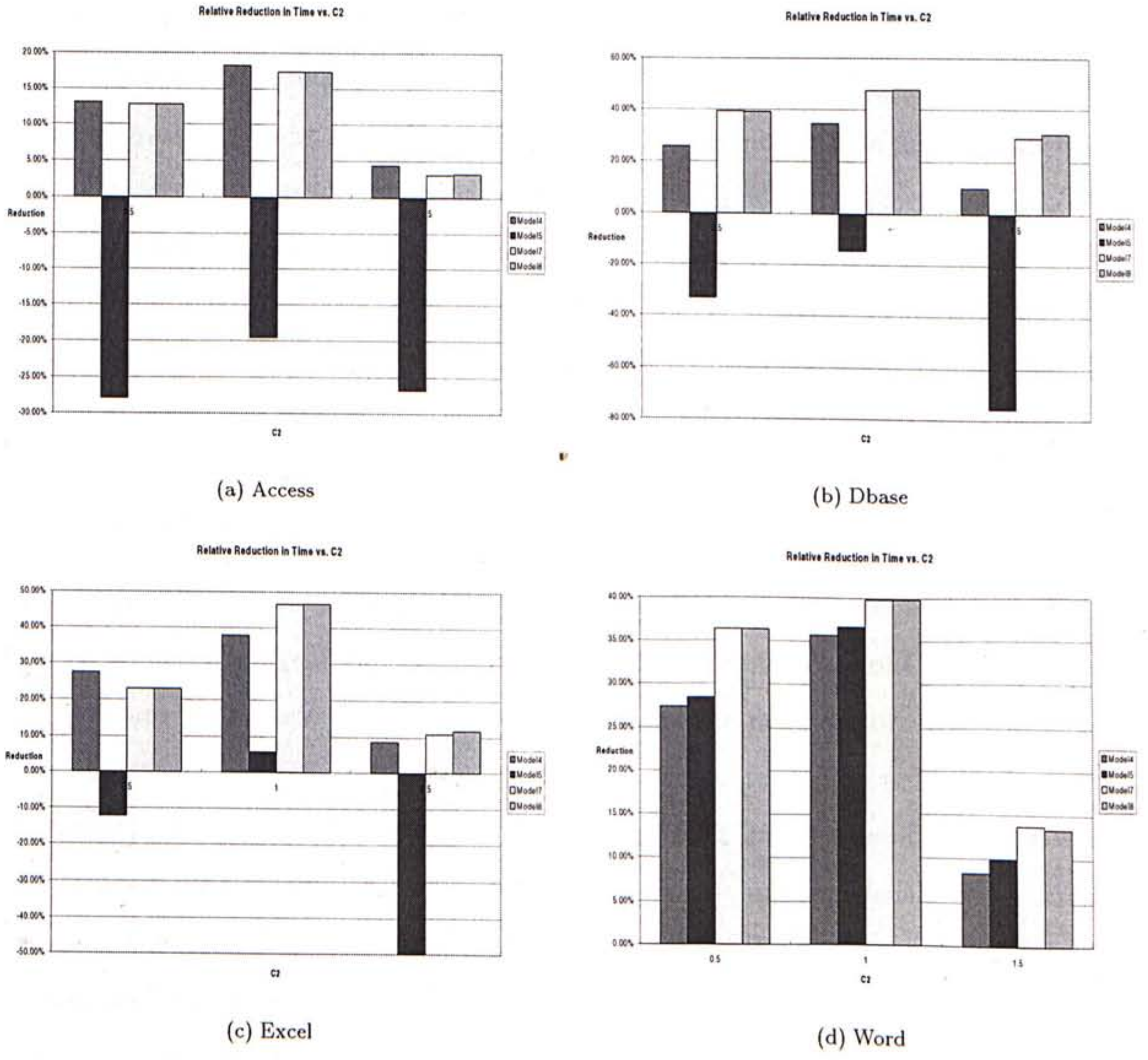
(a) Access

(b) Dbase

(c) Excel

(d) Word

Figure 5.15: Absolute Performance of Varying Transfer Time C2



(a) Access

(b) Dbase

(c) Excel

(d) Word

Figure 5.16: Relative Performance of Varying Transfer Time C2

one. For C2>1, not only the first block needs to pay a time penalty, but all the following ones also need to pay a time penalty in cache miss. This is because there is not enough time to get the following one when the process is using the first one. Therefore, we expect that the performances of the models are much better for C2<=1 than that for C2>1. Our expectation is verified in Figure 5.16.

As C2 increases from 0.5 to 1, the relative performances always increase for all models because the time penalty for getting the sectors/blocks slightly increases for each cache miss. And the standard model, Model 3, has more cache misses than that of Model 4, Model 7 and Model 8 because it only prefetches next sectors on miss. The increase in relative performance also shows that always prefetch is better than prefetch on miss in this case.

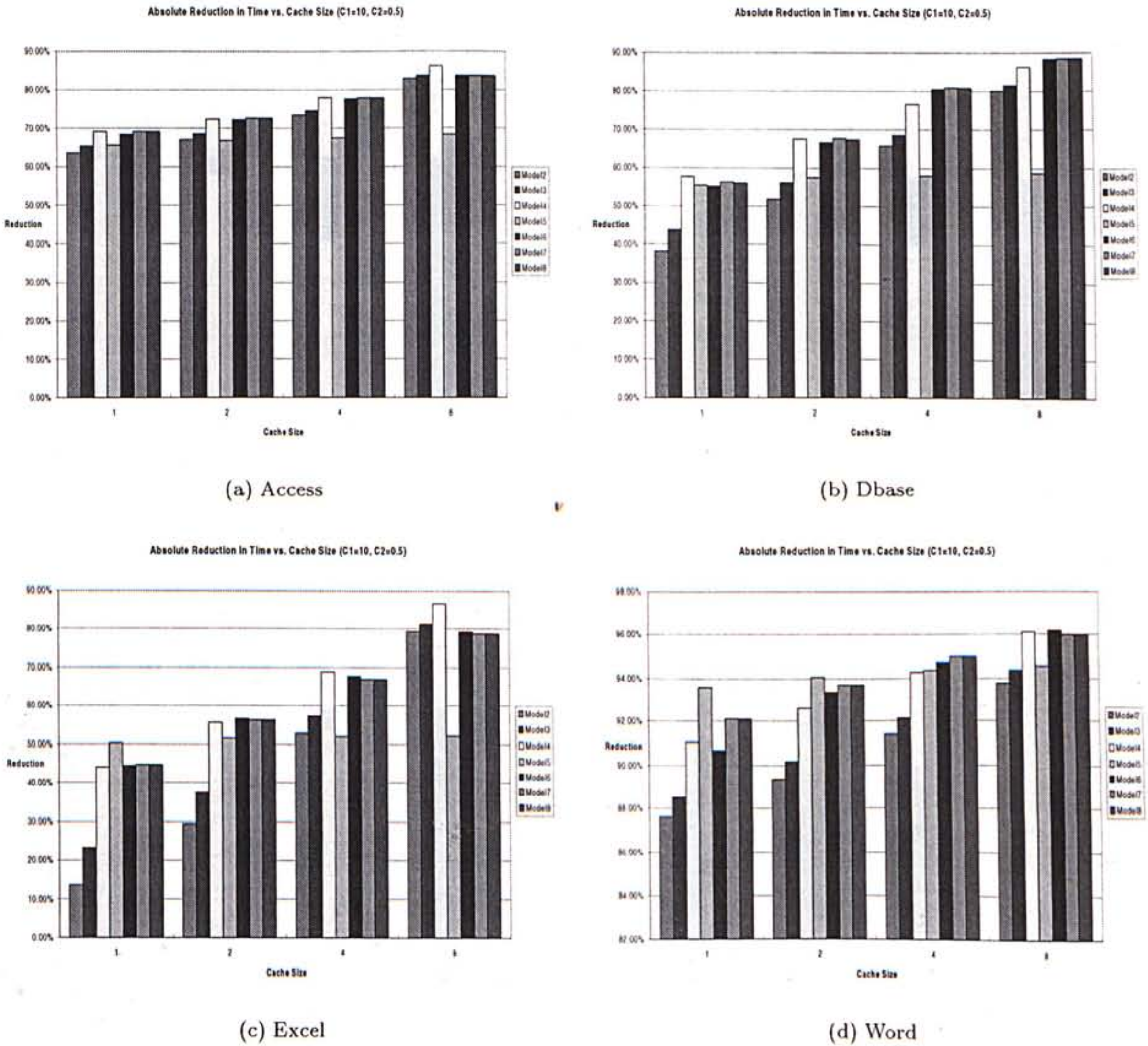
The trend of the relative performance of Model 5 is similar to the cases of Model 4, Model 7 and Model 8 as discussed above.

5.5.3 Impact of C2=0.5 on Cache Size

Transfer time C2=0.5 means that there is enough time to get the next sector when the process is using the current one. This is very different from the case of C2=1.5 because C2=1.5 means that there is *not* enough time to get the next sector. Therefore, we expect that there is some impacts on other cache parameters. The most obvious one is the large increase in the relative reduction in time for C2=0.5 when comparing with that in the case of C2=1.5. The relative reduction usually at least *doubles* the value in the case of C2=1.5. This is due to the fact that always prefetch can cause *more cache hits* than prefetch on miss in this *highly sequential and fast fetching situation.*

However, *not all* parameters will be affected by changing C2 from 1.5 to 0.5. For instance, the general effect of cache size has not been changed by varying C2. Figure 5.17 shows the absolute reduction in time and Figure 5.18 shows the relative reduction in time. The trends are very similar to that in the case of C2=1.5. For relative reduction in time, C2=0.5 has similar effect of zero prefetch time as shown in Figure 5.5. The negative relative performances of Model 7 and Model 8 in Access and Excel are *reduced*

due to enough time for prefetching. Besides, Figure 5.18 verifies that there is a large increase in relative reduction in time when comparing with the case of C2=1.5. The general trend remains the same as the case of C2=1.5.



(a) Access

(b) Dbase

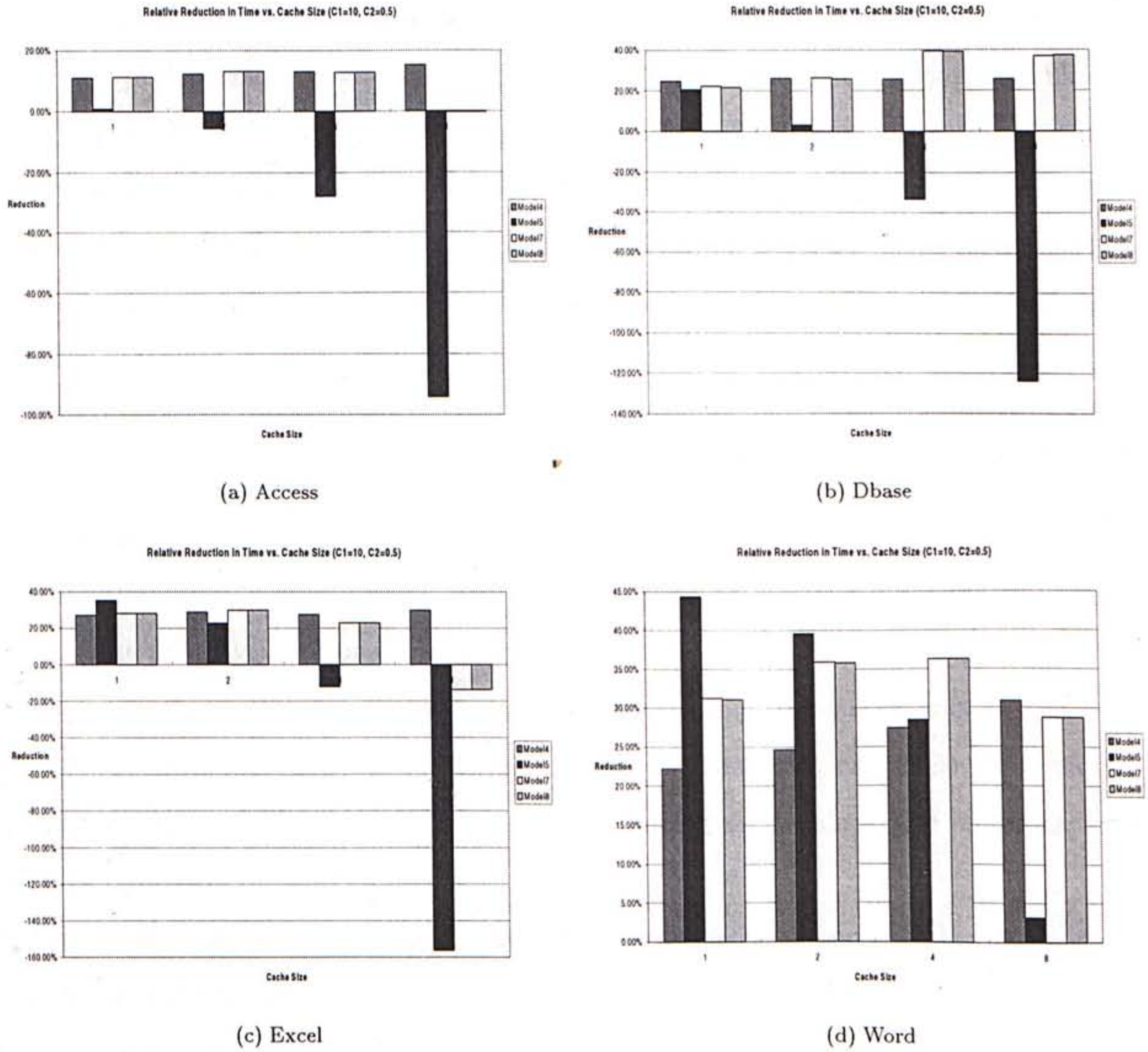
(c) Excel

(d) Word

Figure 5.17: Absolute Performance of Varying Cache Size when C2=0.5

5.5.4 Impact of C2=0.5 on Block Size

Figure 5.19 shows the absolute reduction in time for the four traces. The trends of some models are *different* from the case of C2=1.5. For the case of *C2=1.5*, there is a



(a) Access

(b) Dbase

(c) Excel

(d) Word

Figure 5.18: Relative Performance of Varying Cache Size when C2=0.5

general *decrease* in absolute reduction in time. However, now for the case of *C2=0.5*, there can be an *increase* in absolute reduction in time for Model 5, Model 6, Model 7 and Model 8.

For Model 2, Model 3 and Model 4, their trends are similar to the case of C2=1.5. Their absolute reductions in time decrease when the block size increases. This is due to the reason discussed in the case of C2=1.5 in Section 5.2.1. Block size has a function of implicit prefetching. When the adjacent sectors will be referenced soon, large block size will provide an advantage. However, if the extra stored sectors are not used, they occupy the cache and cause pollution. Now, Model 2, Model 3 and Model 4 store all the requested blocks. Increasing the block size may bring more useless data in the cache although some sectors may have chance to reference later. Therefore, the disadvantage covers the advantage of large block size.

Model 5 has a *more obvious increase* in absolute reduction in time when the block size increases. It is very different from the case of C2=1.5 that it just maintains in a *slightly increasing/decreasing state.* Model 5 stores only the *first* starting block of each dynamic block. As the block size increases, the first starting block size is larger. So the time penalty paid for each cache hit reduced. Since C2=0.5, there is *no* extra *transfer penalty* needed for cache miss, i.e. the cache system *needs only* to pay for the *start-up penalty*, when data fetching overlaps the program execution. Therefore, the *absolute time* of disk access for Model 5 decreases as the block size increases, i.e. absolute reduction in time increases.

For C2=0.5, Model 7 and Model 8 become nearly the same because they both store only the starting head (some heading sectors) for each dynamic block. Therefore, more starting heads can be placed in the cache. For C2=1.5, it is not enough time to get all the remaining sectors by only storing the starting head in BTC, so cache entries need to store the some content sectors. The overhead of including some useless sectors in a large block size (see Figure 5.8) will cover the advantage of getting useful adjacent sectors by a large block size. On the other hand, as discussed above, there is *no* need to pay extra transfer time for C2=0.5 by proper overlapping the program execution

and data fetching. We just *need to pay the start-up penalty.* For Model 7 and Model 8, they have PB to store the prefetched sectors in order to reduce cache pollution due to large block size, which is very useful as discussed in the case of C2=1.5. However, for C2=0.5, it does *not* need to worry the second sectors if the *first one* has already been gotten/stored. Therefore, *guessing the first one* becomes more important for C2=0.5. Model 7 and Model 8 can already *store* more first one than other models. However, they *cannot guess* other *first one.* Larger block size may help to capture other *first ones.* Therefore, Model 7 and Model 8 may have an increase in absolute performance when the block size increases.

Figure 5.20 shows the relative reduction in time for the four traces. The general trend is similar to the case of C2=1.5.

5.6 The Effect Of Prefetch Buffer Size

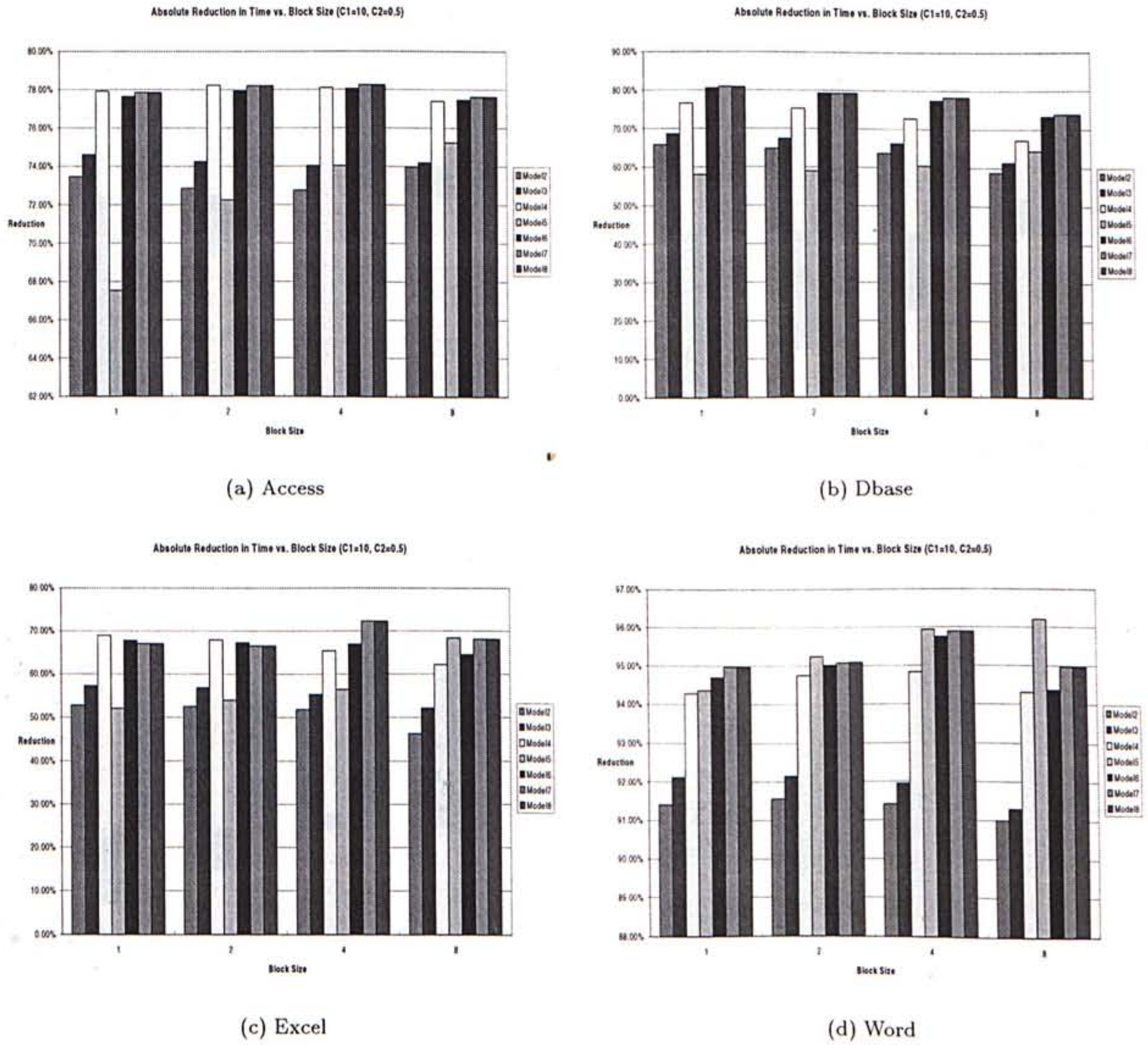
As varying the prefetch buffer size, we choose a fixed reference point for other cache parameters.

Cache Size = 4M
Set Associativity = 1 way
Block Size = 1 sector
Start-up Time C1 = 10
Transfer Time C2 = 1.5

In the simulation, the sizes of prefetch buffer that we have examined are 0.05M, 0.1M, 0.2M, 0.3M and 0.4M.

The prefetch buffer is a small, temporary storage of the prefetched sectors. Its size should be small because all useful data should already be placed in the branch target cache. Now we examine the impact of the prefetch buffer size on Model 7. The impact on Model 8 is similar.

Figure 5.21 shows the absolute reduction in time of Model 7 versus PB size. We note that a small PB size is enough for Model 7. Although increasing the PB size



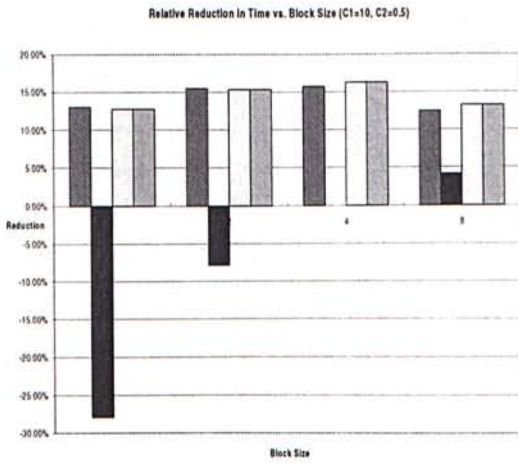
(a) Access

(b) Dbase

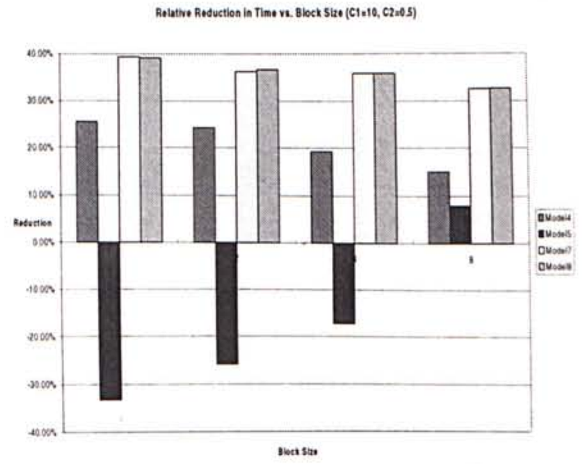
(c) Excel

(d) Word

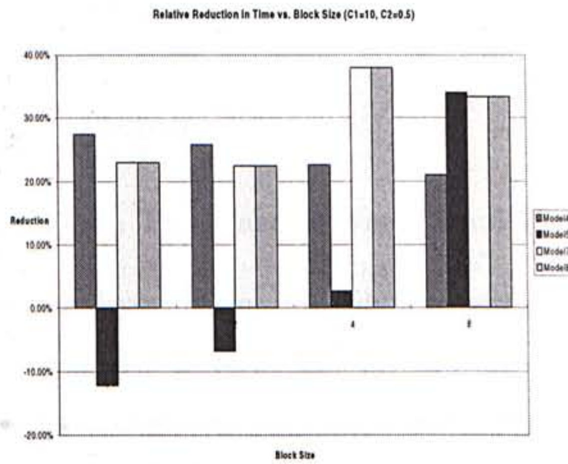
Figure 5.19: Absolute Performance of Varying Block Size when C2=0.5



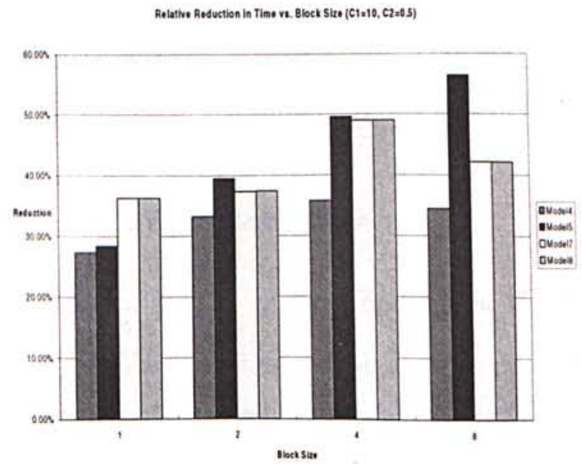
(a) Access



(b) Dbase



(c) Excel



(d) Word

Figure 5.20: Relative Performance of Varying Block Size when C2=0.5

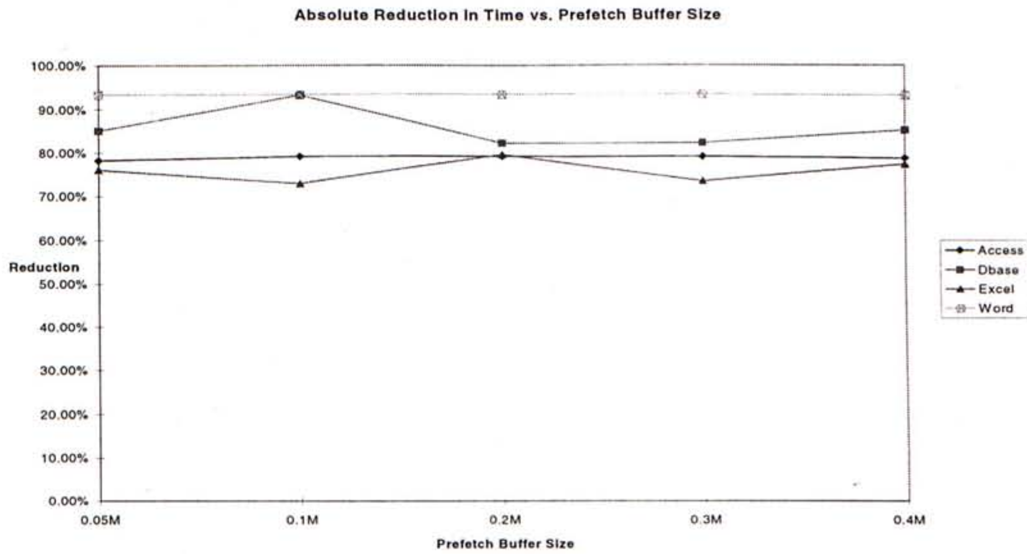


Figure 5.21: Absolute Reduction in Time of Varying Prefetch Buffer Size

may cause a slight increase in performance, it highly depends on the properties of the traces. Therefore, choosing a small PB size is suitable.

5.7 Others

5.7.1 In The Case of Very Small Cache with Large Block Size

For the Excel trace, it exhibits a very strange behavior in 1M cache when block size is equal to *8 sectors*. The hit ratio is only about 10% for unified cache in this case. We have observed a strange fact that the total access time of Model 2, unified cache without prefetch, is larger than that of no cache.

The numbers under *Time ratio* are the ratios of the total access time needed for Model 2 over the total access time of no cache. When the ratio is larger than 1, it indicates that the performance of having a cache is worse than that of no cache. This is due to the fact that the block size is too large. So many nearby sectors are transferred as a whole to the cache. However, the cache size is too small. Those nearby sectors may not have chance to be referenced before they are replaced. Then the cache system

C1	C2	Time ratio
5	0.5	1.0204
5	1	1.0698
5	1.5	1.1023
10	0.5	0.9778
10	1	1.0276
10	1.5	1.0624
15	0.5	0.9479
15	1	0.9961
15	1.5	1.0313
20	0.5	0.9258
20	1	0.9717
20	1.5	1.0064

Table 5.7: Behavior of Model 2 in 1M Cache Size for Excel Trace

needs extra time to get the adjacent sectors that will not be used. This case manifests the disadvantage of large block size. Therefore, choosing a suitable configuration in cache design is very important.

However, this greater than 1 property only observed in the Excel trace, but not in other traces. For other traces, their hit ratios are at least about 50% so more data are being reused even in 1M cache.

5.7.2 Comparing Performance of Model 6 and Model 7

We have not discussed the performance of Model 6 throughout this chapter because its performance is usually worse than that of Model 7 as shown in all graphs of absolute performances. Model 6 and Model 7 use the same policy, ASST, except that ASST in Model 6 applys to each request separately while the ASST in Model 7 applys to dynamic block. This is the difference between request block and dynamic block. Dynamic block provides an environment of larger block size for our algorithms to operate. This shows that the concept of dynamic block is useful in cache design. The difference between applying ASST to each request and to each dynamic block has been discussed in detail in Section 3.3.2 and Section 3.3.3.

	Access		Dbase		Excel		Word	
Cache Size	Model 6	Model 7	Model 6	Model 7	Model 6	Model 7	Model 6	Model 7
1	70.84%	70.92%	59.74%	60.13%	46.70%	48.89%	89.47%	89.69%
2	74.06%	74.11%	71.02%	71.61%	59.68%	62.12%	91.95%	92.03%
4	79.07%	79.11%	83.24%	83.27%	71.30%	72.87%	93.38%	93.39%
8	84.805	84.75%	89.63%	89.76%	82.11%	82.80%	95.05%	94.99%

Table 5.8: Absolute Performance of Varying Cache Size of Model 6 and Model 7

Table 5.8 shows the absolute performance of varying cache size of Model 6 and Model 7. Other parameters are fixed as block size=1, set associativity=1, C1=10 and C2=1.5. The absolute performance of Model 7 is usually better than that of Model 6 except in 8M cache size. This is the situation like Model 7 comparing with Model 4. In very large cache size, the cache is large enough to hold the useful data but Model 7 compulsorily discards more contents sectors than Model 6. There are more accumulated disadvantages of killing correct prefetch and non-heading reuse for Model 7 in 8M cache. Therefore, models that store more sectors for each request can perform better in 8M cache.

5.8 Conclusion

5.8.1 The Number of Actual Sectors Transferred between Disk and Cache

Consider the following case:

Cache Size = 4M
Block Size = 1 sector
Set Associativity = 2 way
Start-up Time C1 = 10
Transfer Time C2 = 2

Table 5.9 shows the number of sectors transferred between disk and cache. We have noted that the numbers of sectors transferred for Model 4, Model 7 and Model 8 are approximately equal to that of Model 2 and Model 3. This means that always prefetch technique does not impose a heavy traffic between disk and cache. On the other hand,

always prefetch can reduce the traffic in some cases. So it is a very suitable technique to incorporate into disk cache design. However, current methods usually do not include the always prefetch technique.

	Access	Excel	Dbase	Word
Model 2	2,350,300	1,944,730	1,852,463	414,742
Model 3	2,363,061	1,955,952	1,855,170	414,807
Model 4	2,373,182	1,967,134	1,859,585	415,076
Model 5	2,749,506	8,497,649	4,929,358	471,906
Model 6	2,372,007	2,039,514	2,117,421	419,053
Model 7	2,366,317	1,928,586	1,940,305	417,119
Model 8	2,362,100	1,909,416	1,890,052	416,170

Table 5.9: Actual Number of Sectors Transferred between Disk and Cache

5.8.2 The Efficiency of Our Models on Common Disk

In conclusion, always prefetch is a very useful technique to capture the highly sequential property of disk access pattern. Simulation verifies that ASST and SEHT can increase the performance of a cache system on the basis of always prefetch. Model 7 and Model 8 usually outperform other models in the intermediate cache size, such as 2M to 4M. The relative performances of Model 7 and Model 8 can double/triple the relative performance of Model 4. This satisfies our aim that the cache can be more effectively utilized by the cache partitioning architecture and the newly proposed control mechanisms when the cache size is limited.

The factors that increase the performance of Model 7 and Model 8 are summarized below:

- intermediate cache size, e.g. 2M to 4M
- small block size, e.g. 1 sector
- large set associativity of the cache, e.g. 4-way set associative
- small prefetch buffer size

- more occurrences of heading reuse
- less occurrences of killing correct prefetch

Chapter 6

Performance Evaluation of High Performance Disk

High Performance disk is characterized by the start-up time C1 that is slightly larger than the transfer time C2. Also, C1 and C2 are both near the value of Tu. In the following discussion, we generally choose C1=2 and C2=1.5. Tu is always set to 1 in order to act as the reference point. In fact, the values 2 and 1.5 are the ratios of the actual values of C1 and C2 to the actual value of Tu. All other timing values are also ratios to Tu.

We focus mainly on the performances of 4 different models: Model 4, Model 5, Model 7 and Model 8. The parameters that we will discuss are the cache size, the block size, the start-up time C1 and the transfer time C2. Others are the same as the case of common disk, so we will not discuss again. In fact, the being discussed parameters are also quite similar to the case of common disk.

6.1 Difference Between Common Disk And High Performance Disk

The difference between common disk and high performance disk is mainly in the value of C1. C2 for both kinds of disk can take small values. High performance disk has smaller C1 than common disk. C1 controls the response time of a disk. As C1 becomes

smaller, the disk responses faster. Therefore, we define the disk having small C1 and C2 as high performance disk. C1 controls the size of the starting head in our algorithms. As C1 approaches C2, the size of the starting head is closer to the size of the stored content blocks. As C1 and C2 become smaller, the size of starting head also decreases. Besides, the time penalty of missing a sector is less than that in case of common disk because the start-up time C1 is smaller. Therefore, these factors may have impacts on the performances of Model 5, Model 6, Model 7 and Model 8 because these models have different treatment to starting heads and content blocks.

6.2 The Effect Of Cache Size

As varying the cache size, we choose a fixed reference point for other cache parameters.

Block Size = 1 sector
Set Associativity = 1 way
Start-up Time C1 = 2
Transfer Time C2 = 1.5

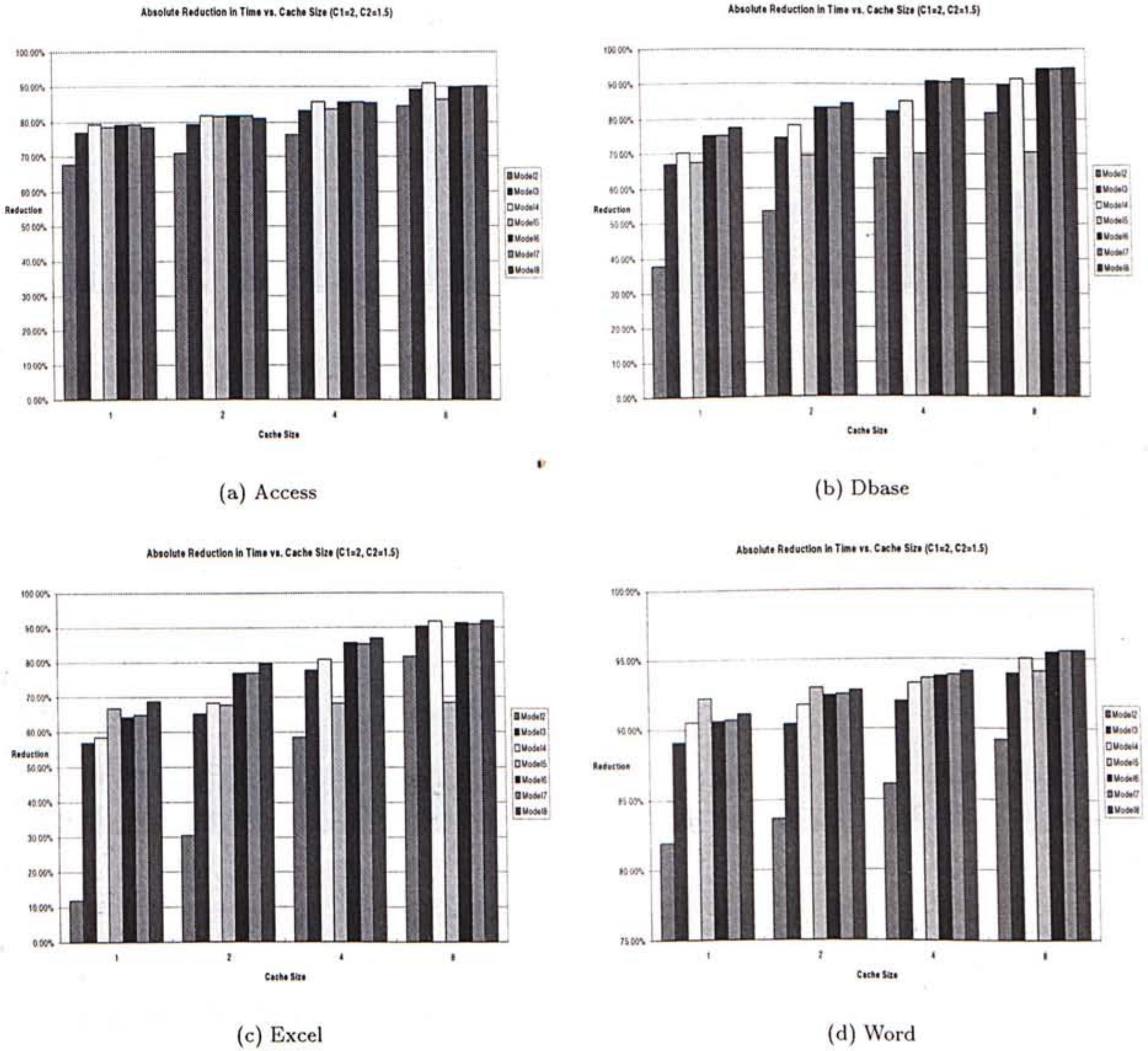
In the simulation, we have examined the cache sizes of 1M, 2M, 4M and 8M.

6.2.1 Trends of Absolute Reduction in Time

We have observed that *the absolute reduction in time increases for all models as the cache size increases* which can be illustrated from Figure 6.1. This is similar to the result in the case of common disk. This is because large cache can store more data, including that has been stored in the smaller cache. Therefore, the cache hit rate must be increased, and disk access time can be further reduced.

6.2.2 Trends of Relative Reduction in Time

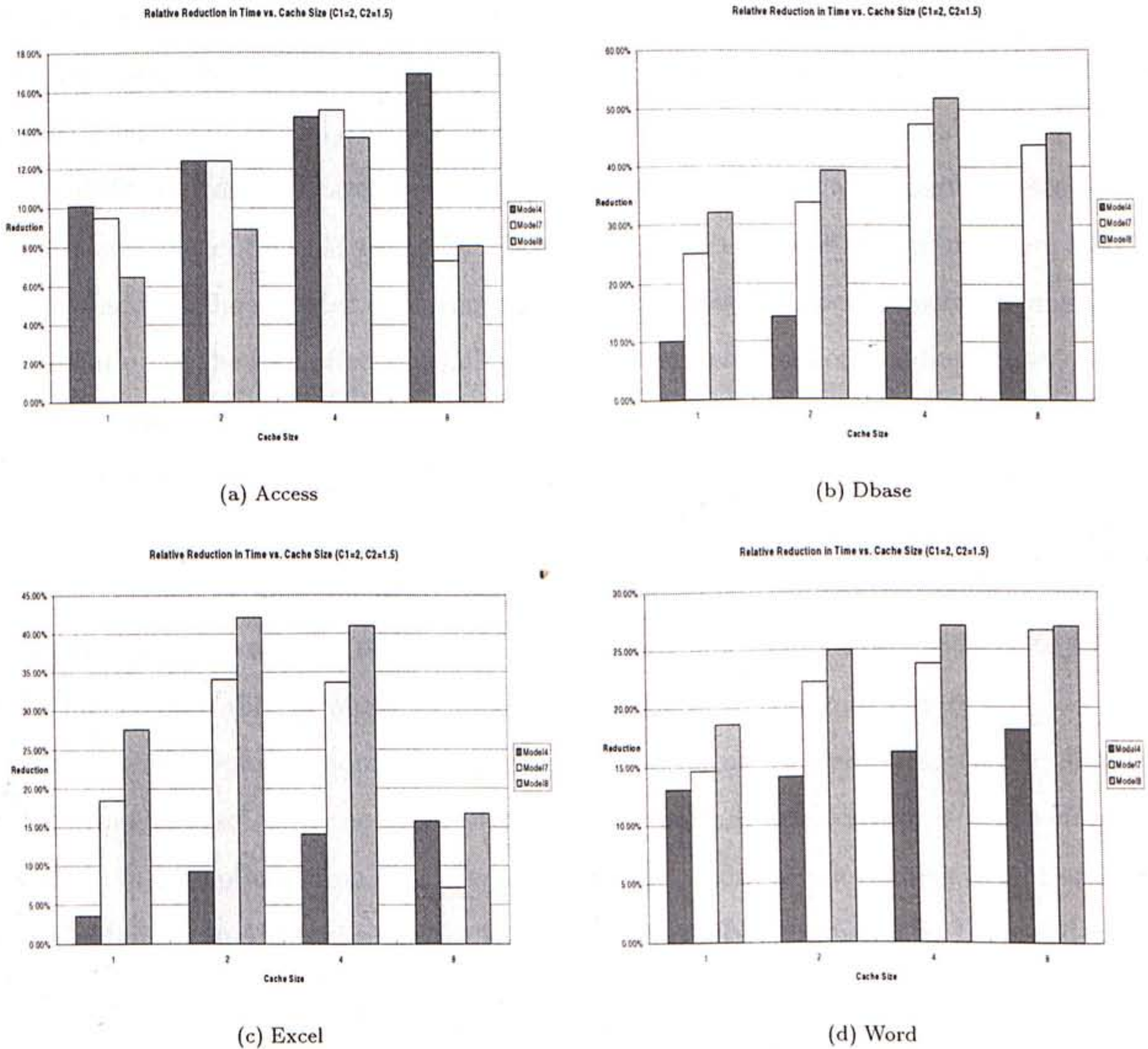
For clarity of graph, we omit the Model 5 in Figure 6.2. From the graph of absolute performance, Figure 6.1, we note that there are *actual reduction* in time when the cache size increases. Therefore, the dropped relative performances mean that the *increase* in



(a) Access (b) Dbase (c) Excel (d) Word

Figure 6.1: Absolute Performance of Varying Cache Size

absolute reductions in time of Model 7 and Model 8 is less than that of Model 3 as the cache size increases.



(a) Access

(b) Dbase

(c) Excel

(d) Word

Figure 6.2: Relative Performance of Varying Cache Size (without Model 5)

6.2.2.1 Performance Of Model 4, Model 7 And Model 8

The trend of relative performance of Model 4 is exactly the same as the case of common disk. Figure 6.2 shows that the relative performance of Model 4 gradually increases when the cache size increases. However, the increase is very little although the cache

size increases 8 times, i.e. from 1M to 8M. The reason is discussed in Section 5.1.2.1 in the case of common disk.

The trends of Model 7 and Model 8 are also similar to the case of common disk. The relative reduction in time first increases when the cache size increases. However, when the cache size changes to 8M, there is generally a drop in relative performance. On the other hand, the relative performance does *not* drop to negative values now. It is very *different* from the case of common disk that there are negative performances of Model 7 and Model 8 in 8M cache for Access and Excel traces as shown in Figure 5.2. As discussed in the chapter 5 of common disk, the negative performance is due to accumulation of the bad effects of killing correct prefetch and non-heading reuse. For high performance disk, since C1 is smaller, the size of starting head decreases. *More* requests can now be stored in the cache, i.e. the effective cache size increases further than that in case of common disk. The further enlarged size *lowers* the effects of the previous two bad factors. Therefore, the relative performance does not drop to negative value. Besides, The relative performances of Model 7 and Model 8 can be *three to four times* higher than that of Model 4 for Dbase and Excel as shown in Figure 6.1.

Our models perform more efficiently in high performance disk. This can be illustrated from the fact that the value of relative performance percentage is *much larger* than that in case of common disk. For instance, the values of the relative reduction in time of Model 7 in 4M cache size for the two kinds of disks are listed in Table 6.1.

Trace	Common disk	High performance disk
Access	3%	15%
Dbase	29%	47%
Excel	11%	33%
Word	14%	23%

Table 6.1: Relative Performance of Model 7 for two kinds of disks

The performance of Model 8 is now generally better than Model 7. This is because Model 8 stores smaller amount of blocks for a fixed number of requests than that of Model 7, and so Model 8 can store more data from more requests, i.e. effectively

enlarges more the cache size than Model 7. Since C1 is small, the time penalty paid for each miss is less than that in the case of common disk. The reuses (cache hit) of the extra stored blocks cover the disadvantage of missing some content sectors. Therefore, if the effectively increased cache size of Model 8 can capture sufficiently more reuses, it can outperformance other models.

6.2.2.2 Performance Of Model 5

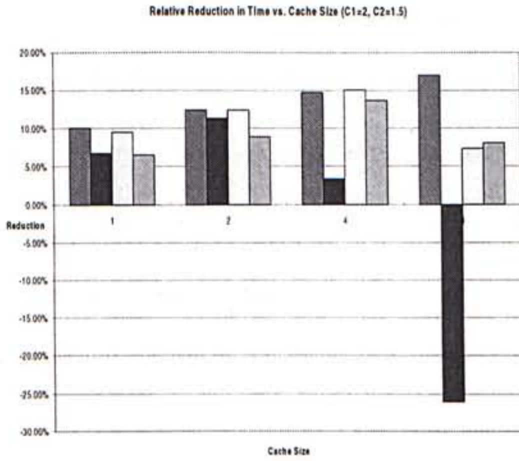
We have omitted Model 5 to get a clearer discussion in before. Now, let us focus on the relative performance of Model 5. Figure 6.3 shows the relative performances of the four models, including Model 5, for different traces.

The trend of Model 5 is similar to the case of common disk. However, the relative performance of Model 5 is much better than that in the case of common disk. Model 5 can outperform Model 3 in small cache size, e.g. 1M cache size for all traces. This can be explained by the fact that for high performance disk, storing more requests can cover the disadvantage of referencing the un-stored sectors because the time penalty paid for each miss is not very large now. Besides, storing more requests can increase cache hits. In small cache size, the cache may not store enough data to capture reuses if all requested sectors are stored in the cache. Now, Model 5 stores only the first block of each non-sequential reference and the time penalty of getting the remaining sectors are smaller than that in the case of common disk. Therefore, Model 5 can perform better than Model 3 due to these advantages in small cache size.

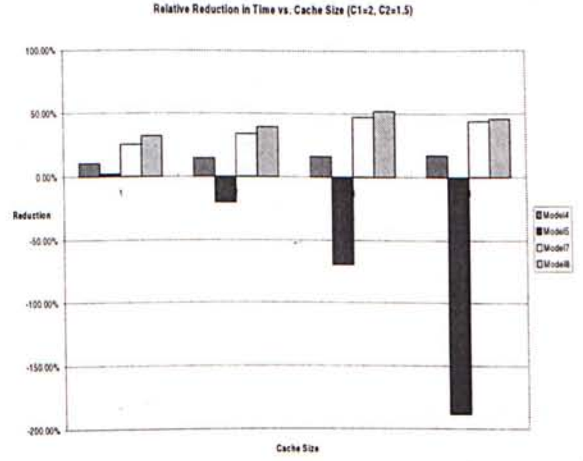
However, for Model 5, it stores only the *first* heading block of a request. No matter hit or miss, it needs to pay more time penalty than other models. So its performance cannot be as good as Model 7 and Model 8. However, the extra stored requests can improve the performance and let Model 5 outperform Model 3 in small cache size.

6.3 The Effect Of Block Size

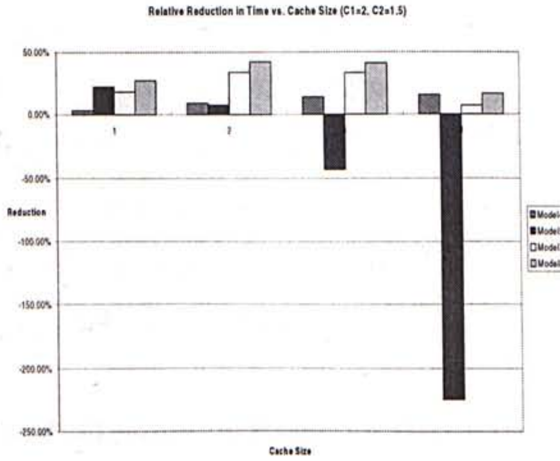
As varying the block size, we choose a fixed reference point for other cache parameters.



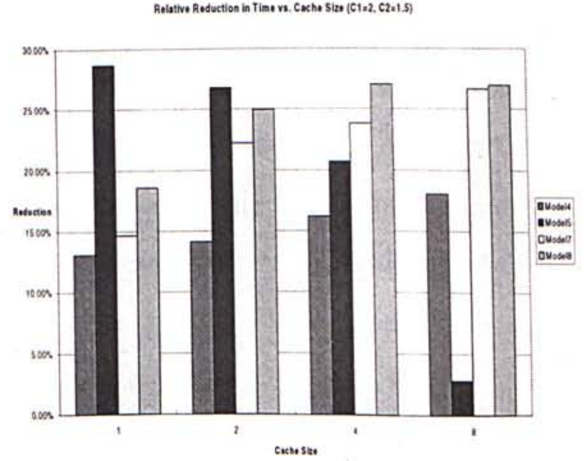
(a) Access



(b) Dbase



(c) Excel



(d) Word

Figure 6.3: Relative Performance of Varying Cache Size

Cache Size = 4M
Set Associativity = 1 way
Start-up Time C1 = 2
Transfer Time C2 = 1.5

In the simulation, we have examined the block sizes of 1 sector, 2 sectors, 4 sectors and 8 sectors.

6.3.1 Trends of Absolute Reduction in Time

We have observed that *in general, the absolute reduction decreases as the block size increases* which is shown in Figure 6.4.

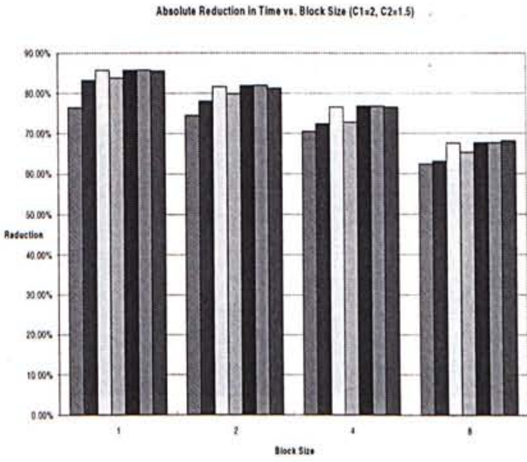
The trend of varying block size is also similar to the case of common disk. The decrease in the absolute reduction means that the absolute performance is poorer in larger block size.

Moreover, we observe that Model 5 is not like other models, the performance of Model 5 may not decrease as the block size increases. This is due to the fact that Model 5 stores only the *first starting block* of a dynamic block. It needs to pay higher time penalty *even to a cache hit.* However, as the block size increases, the first starting block becomes larger. The large block size can provide more time to get the remaining sectors. The penalty of each cache hit is reduced. So, the effect of bringing undesired sectors into the cache can be compensated.

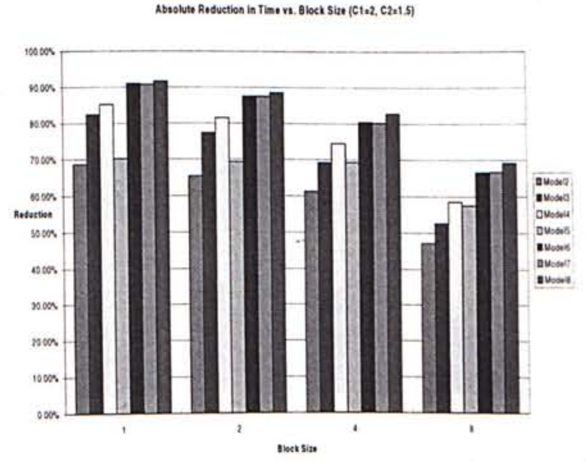
6.3.2 Trends of Relative Reduction in Time

6.3.2.1 Performance Of Model 4, Model 7 And Model 8

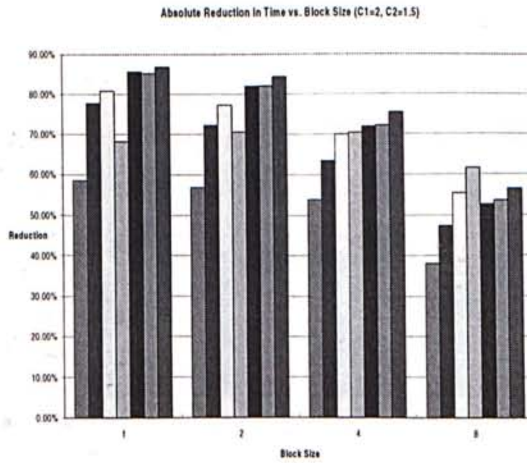
The trend is also very similar to the case of common disk. The relative performance is the absolute performance of a model compared with the absolute performance of Model 3. *Although the absolute reduction decreases in general, the relative reduction*



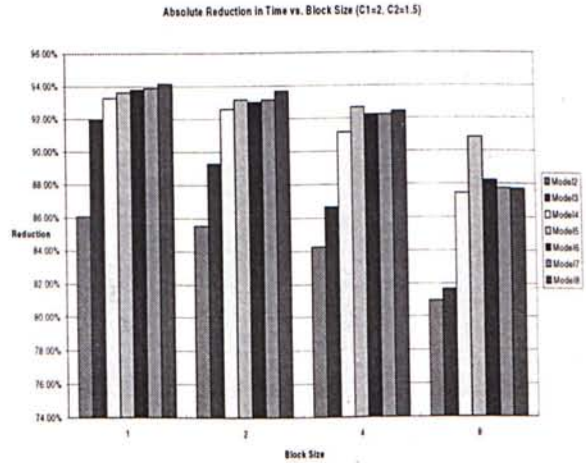
(a) Access



(b) Dbase

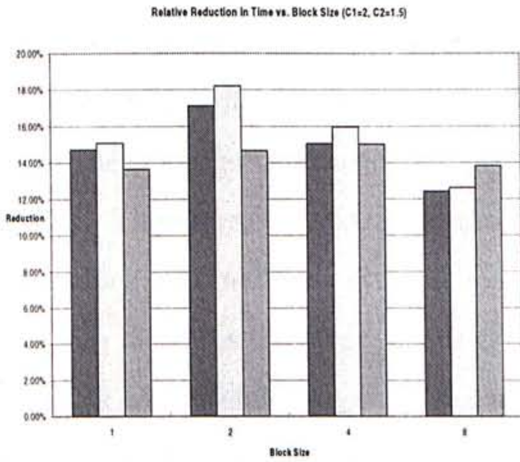


(c) Excel

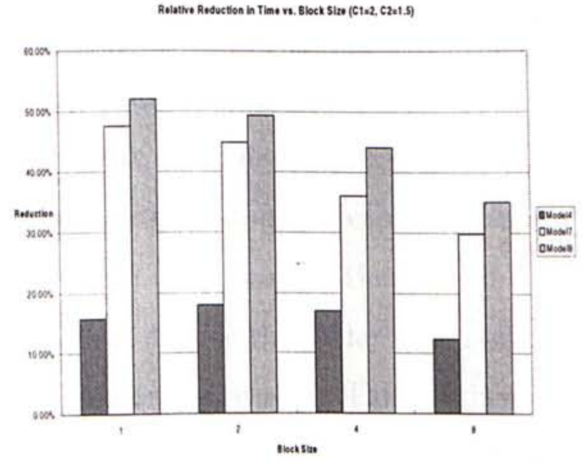


(d) Word

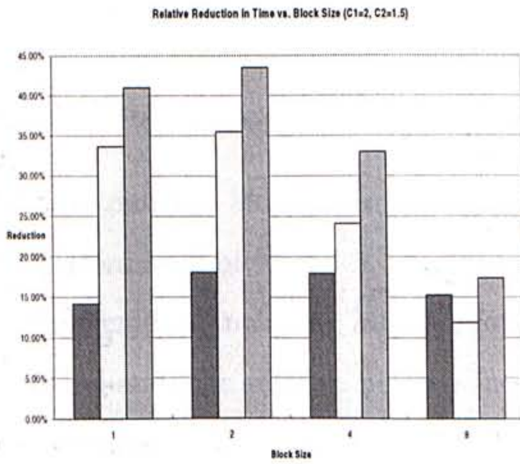
Figure 6.4: Absolute Performance of Varying Block Size



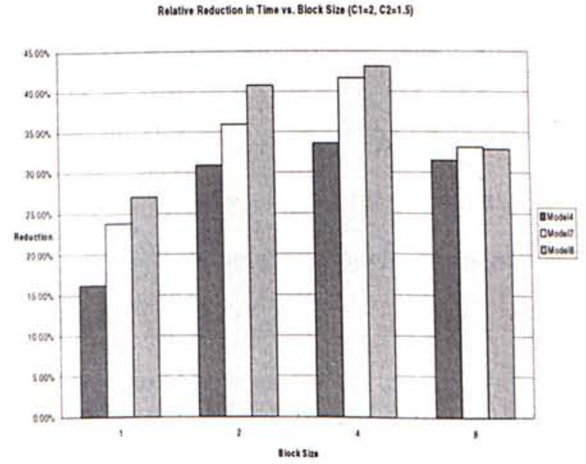
(a) Access



(b) Dbase



(c) Excel



(d) Word

Figure 6.5: Relative Performance of Varying Block Size (without Model 5)

can increase. This means that the performance of a model is much better than that of Model 3 at this situation.

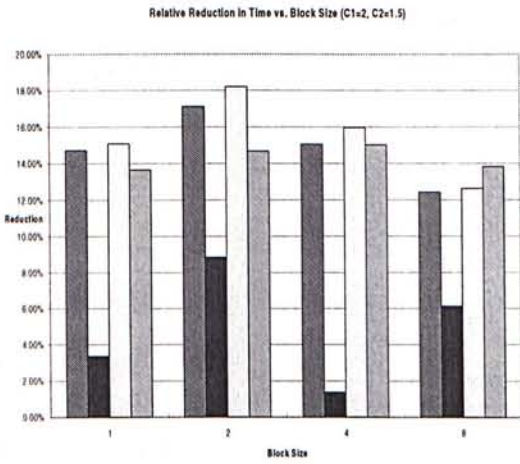
Unlike the case of common disk, the increase in relative reduction of Model 7 and Model 8 is not large. Large block size has an effect of implicit prefetching. It can increase the chance of correct prefetch of adjacent sectors. On the other hand, large block size may bring many useless sectors into the cache and may kick out useful sectors. As discussed previously, the time penalty of a miss is smaller. Model 7 and Model 8 can further enlarge the cache size than that in the case of common disk because C1 is small. Therefore, the extra increased cache size may be enough to capture reuses. If the block size increases little, it may help to capture more reuses. However, if the block size increases too large, it brings too many useless data in the cache. The useless data stick with useful data in a block and make the performance poor. This can justify by the fact that the increase in the relative reduction in time is not too obvious as the block size increases. However, the drop is much dramatic than that in the case of common disk.

6.3.2.2 Performance Of Model 5

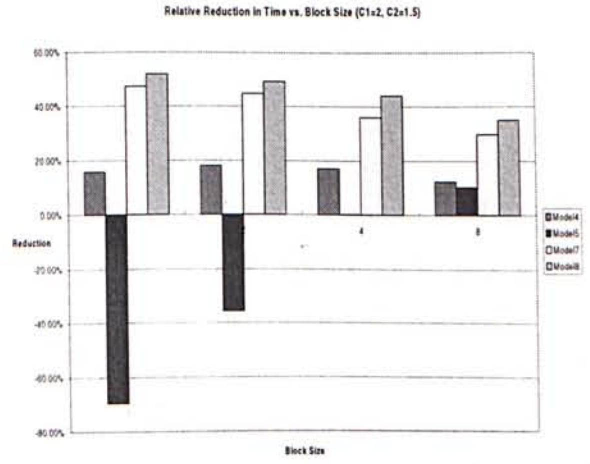
We have omitted Model 5 to get a clearer discussion in before. Now, let us focus on the performance of Model 5. Figure 6.6 are plotting for the relative performances of the four models, including Model 5 for different traces.

In general, the relative performance of Model 5 is better and better as the block size increases. This is because Model 5 underuses the cache by only storing the first heading block of each non-sequential reference in BTC. Therefore, there is not enough time to prefetch the remaining data. Now, as the block size increases, the first heading block contains more sectors and in turn, more sectors of a dynamic block are stored in the cache under Model 5. So the relative performance is better.

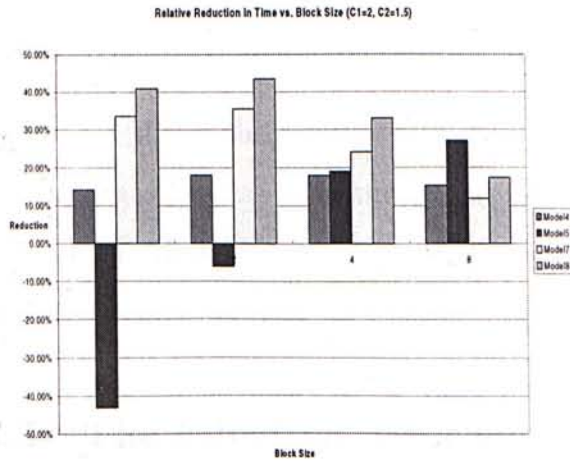
The difference from the case of common disk is that the relative performance of Model 5 can sometimes be better than Model 3 in block size other than 8 sectors. This shows that for high performance disk, small C1, the time penalty for each cache miss



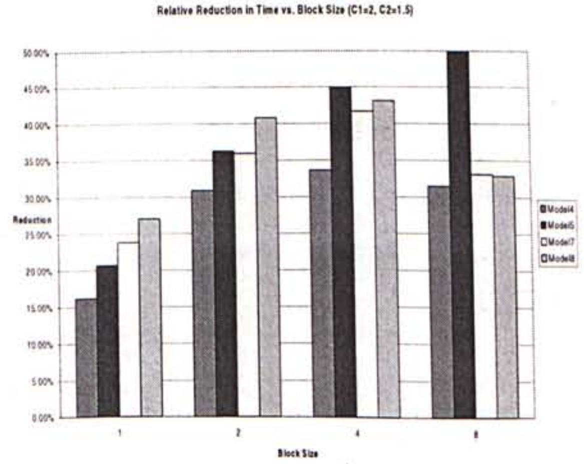
(a) Access



(b) Dbase



(c) Excel



(d) Word

Figure 6.6: Relative Performance of Varying Block Size

is smaller. Model 5 enlarges the effective cache size dramatically. The extra stored sectors contribute more cache hits and they lower the disadvantage of referencing the un-stored sectors.

6.4 The Effect Of Start-up Time C1

As varying the start-up time C1, we choose a fixed reference point for other cache parameters.

Cache Size = 4M
Set Associativity = 1 way
Block Size = 1 sector
Transfer Time C2 = 1.5

In the simulation, the values of C1 that we have examined are 1, 2, 3 and 4. Note that these values are the ratios of actual values of the start-up time to the use-up time.

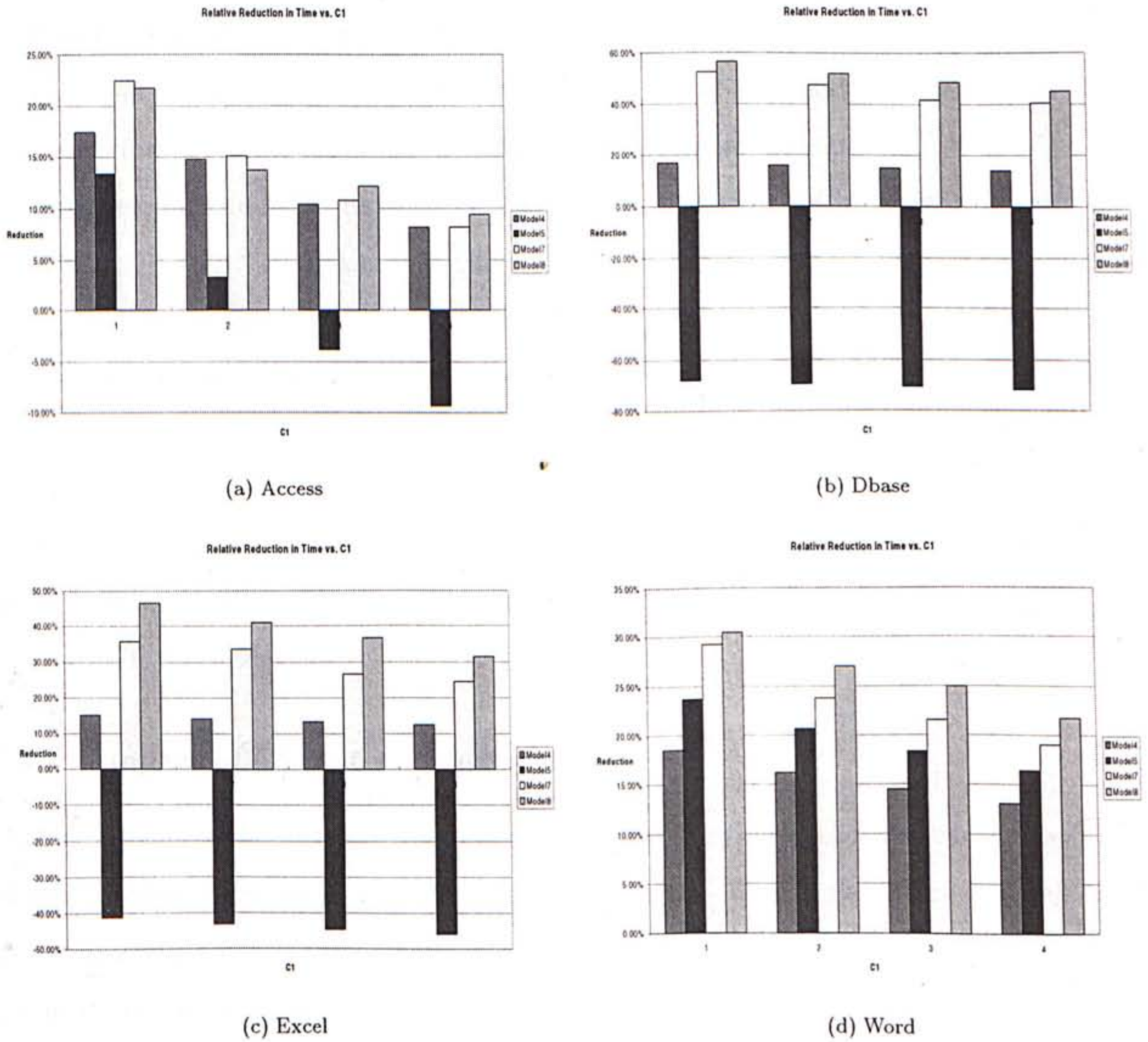
6.4.1 Trends of Relative Reduction in Time

Figure 6.7 shows the relative reduction in time for varying C1. The effect is exactly the same as the case of common disk. When C1 increases, the relative performances of Model 4, Model 5, Model 7 and Model 8 decrease because the time penalty paid for each cache miss dominates.

6.5 The Effect Of Transfer Time C2

As varying the start-up time C2, we choose a fixed reference point for other cache parameters.

Cache Size = 4M
Set Associativity = 1 way
Block Size = 1 sector
Start-up Time C1 = 10



(a) Access

(b) Dbase

(c) Excel

(d) Word

Figure 6.7: Relative Performance of Varying Start-up Time C1

In the simulation, the values of C2 that we have examined are 0.5, 1, 1.5. Note that these values are the ratios of actual values of transfer time to the use-up time.

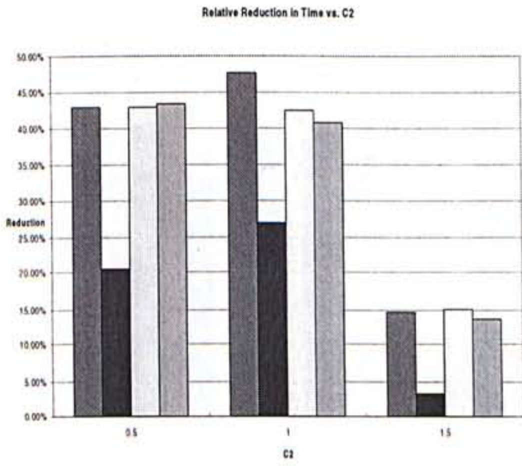
6.5.1 Trends of Relative Reduction in Time

Figure 6.8 shows the relative reduction in time for varying C2. The trends of Model 4, Model 7 and Model 8 are also exactly the same as the case of common disk. As C2 increases from 0.5 to 1, the relative performances of the models increase. As C2 increases from 1 to 1.5, the relative performances decrease dramatically.

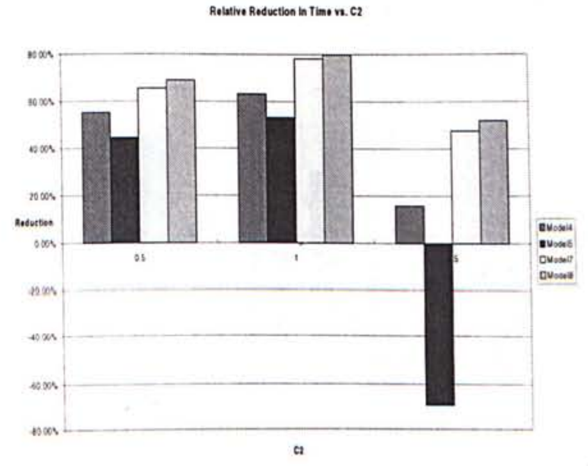
However, for Model 5, its relative performance can now be better than Model 3 when C2<=1. C2<=1 means that there is enough time to get the next sectors after getting the first sector. Therefore, the miss penalty is much smaller than that of C2>1. Besides, C1 is small, so the size of the starting head is also small, approximately 2 sectors. Model 5 always stores the first block of a dynamic block. In this case, the block size is equal to 1. Therefore, the time penalty of reuse is also very small. Furthermore, Model 5 dramatically increases the effective cache size which can store more requests in BTC. The extra stored requests can capture more reuses, so Model 5 can outperform Model 3.

6.5.2 Impact of C2=0.5 on Cache Size

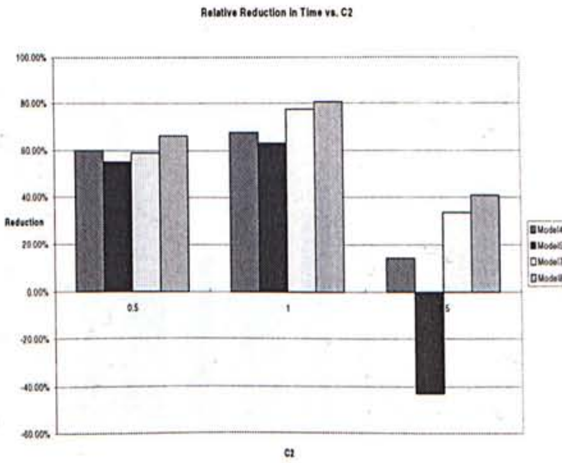
Figure 6.9 shows the absolute reduction in time of varying cache size when C2=0.5. Model 5 outperforms other models in 1M cache size. This is the effect of small C1 so the starting block is very small. Also, C2=0.5 means that there is enough time to get next sectors if the first sector/block has been stored in BTC. In small cache size, there is not enough space to capture the reuses. Therefore, by ignoring some contents sectors, the cache can store more requests and can capture more cache hits. Model 5 is the most vigorous one to discard sectors and let the cache system to get them by overlapping with the program execution. Therefore, in the situation of smaller time penalty of miss, Model 5 can outperform other models. Following the same argument, we predict that the performances of Model 7 and Model 8 are also good.



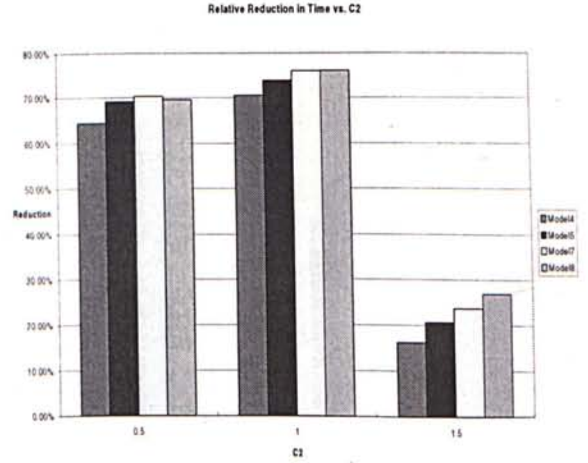
(a) Access



(b) Dbase

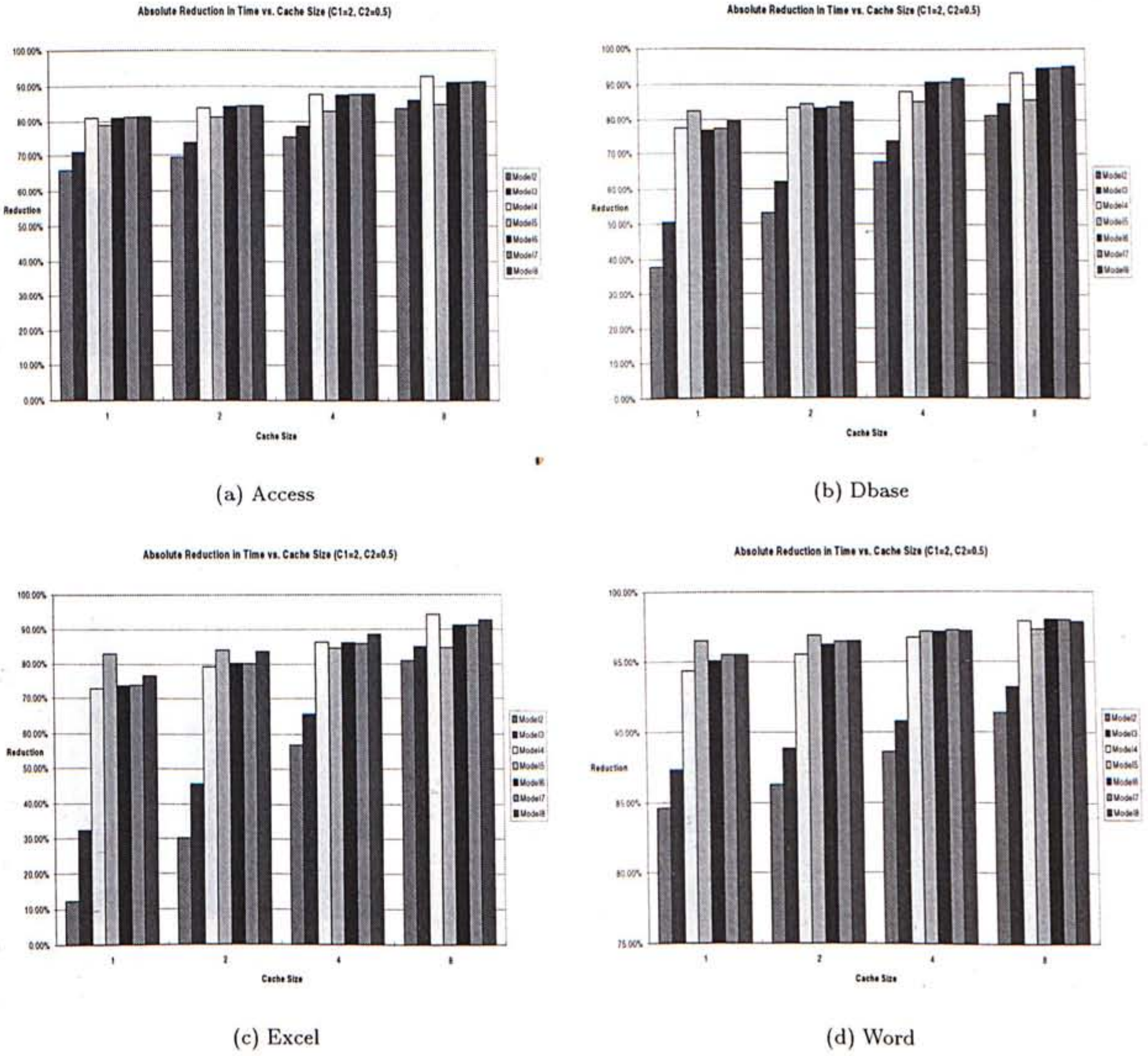


(c) Excel



(d) Word

Figure 6.8: Relative Performance of Varying Transfer Time C2



(a) Access

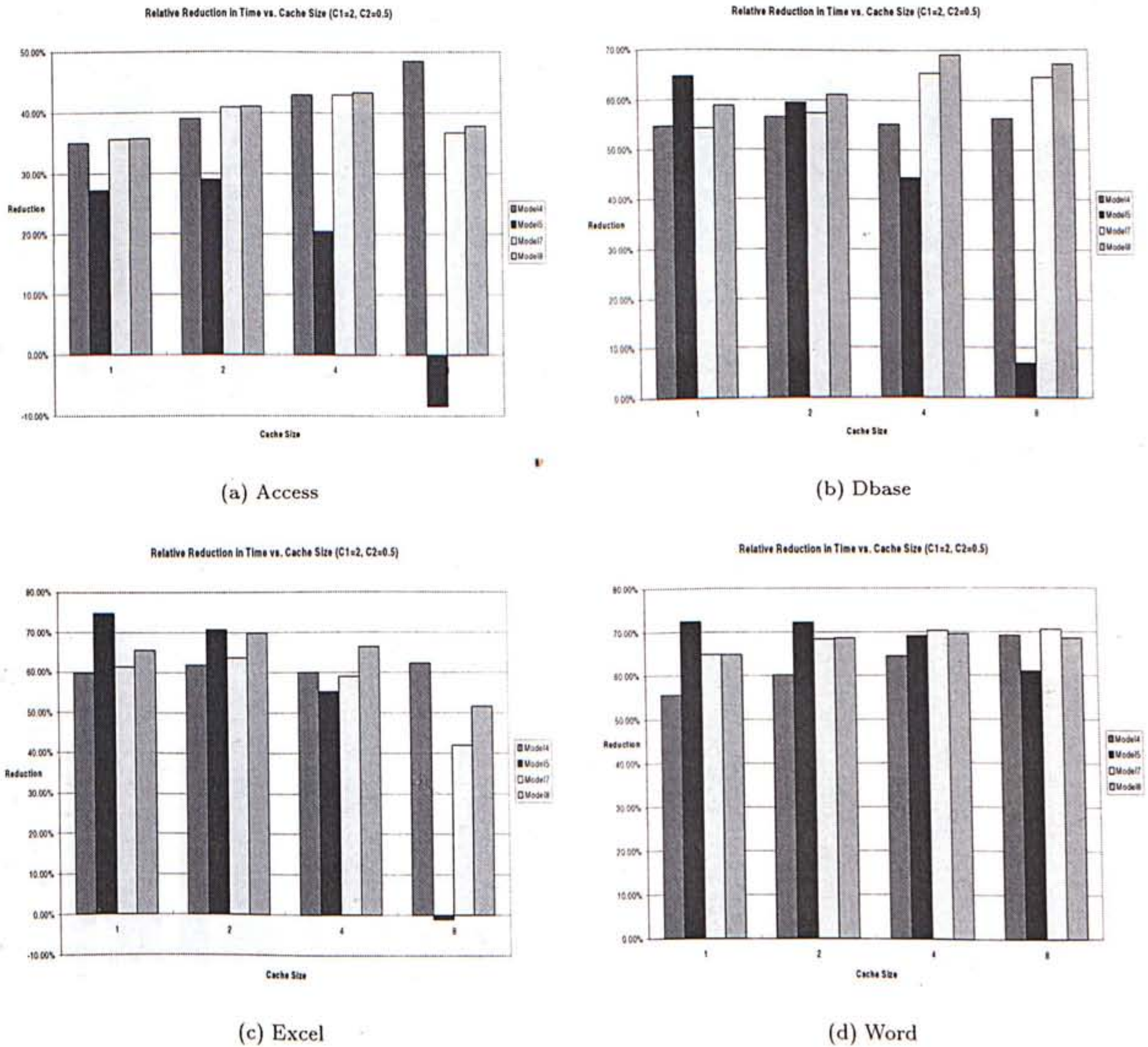
(b) Dbase

(c) Excel

(d) Word

Figure 6.9: Absolute Performance of Varying Cache Size when C2=0.5

Figure 6.10 shows the relative reduction in time for varying cache size when C2=0.5. The trend of relative performance is the same as the case of C2=1.5. However, the relative performance of Model 5 is much better than that in the case of C2=1.5. The reason is discussed as above.



(a) Access

(b) Dbase

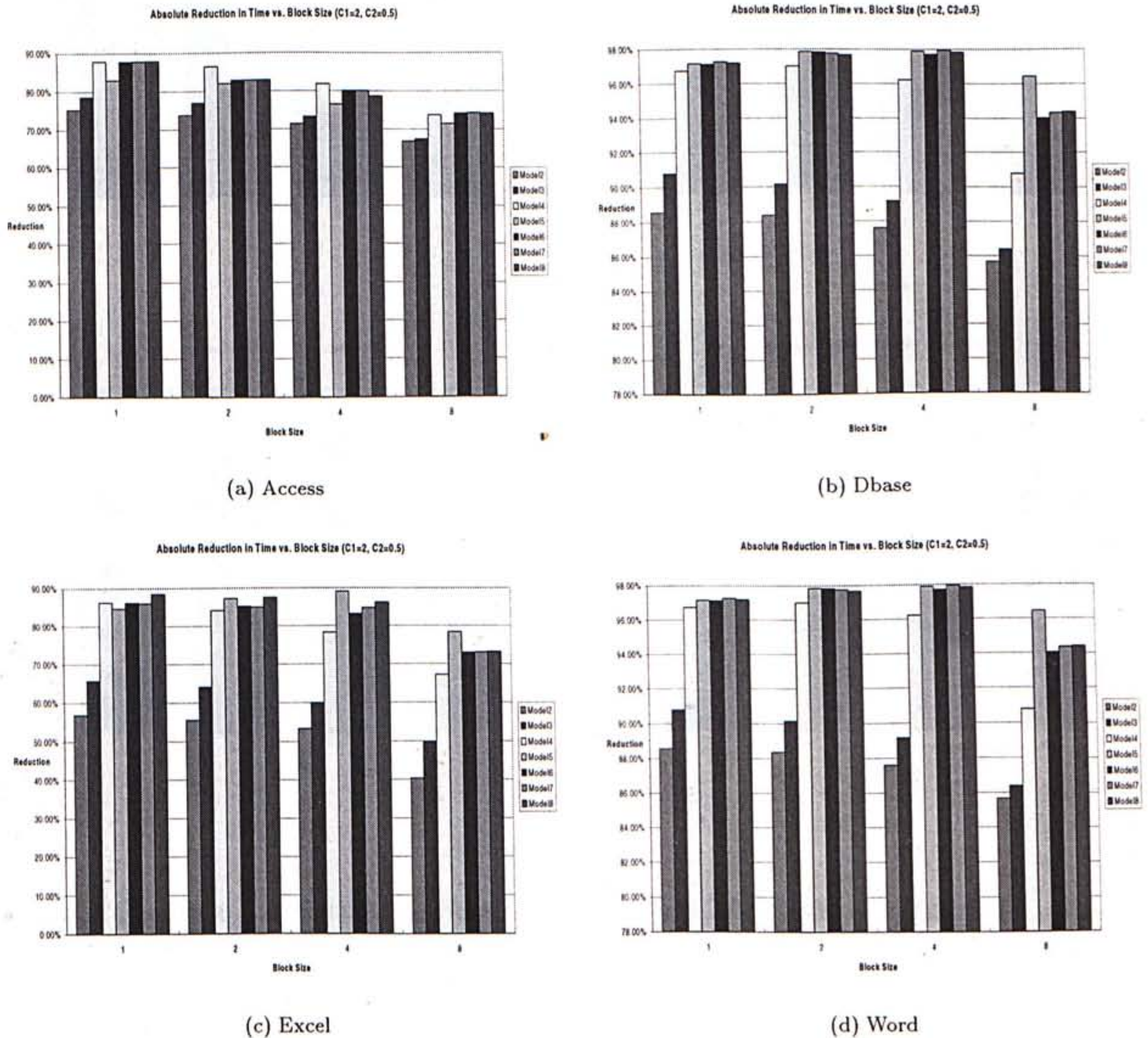
(c) Excel

(d) Word

Figure 6.10: Relative Performance of Varying Cache Size when C2=0.5

6.5.3 Impact of C2=0.5 on Block Size

Figure 6.11 shows the absolute reduction in time for varying block size when C2=0.5. This situation is similar to the case of C2=1.5 except that the performance of Model 5 is much better. The reason is discussed in the previous Section 6.5.2.



(a) Access

(b) Dbase

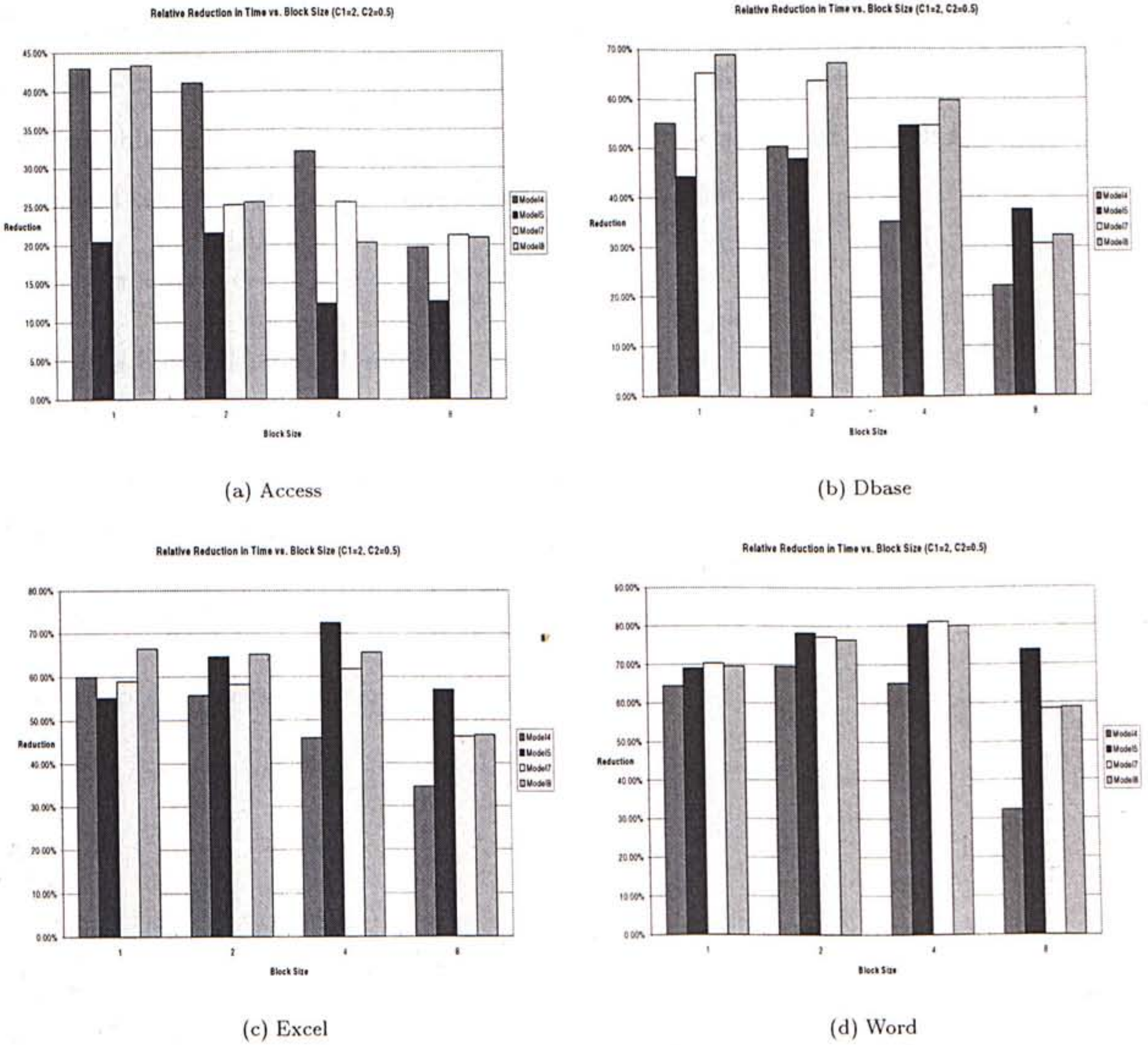
(c) Excel

(d) Word

Figure 6.11: Absolute Performance of Varying Block Size when C2=0.5

Figure 6.12 shows the relative reduction in time of varying block size when C2=0.5. The trend of relative performances of Model 4, Model 7 and Model 8 decreases while that of Model 5 can increase. The reason is discussed in Section 5.5.4 in the case of

common disk.



(a) Access

(b) Dbase

(c) Excel

(d) Word

Figure 6.12: Relative Performance of Varying Block Size when C2=0.5

6.6 Conclusion

High performance disk provides a better environment to ASST and SEHT to work on. Since C1 is small, the starting head is small and so more starting heads can be stored in BTC. Also, the time penalty of each cache miss is smaller than that in the case of common disk. Therefore, the tradeoffs of the ASST and SEHT become less important.

This can be verified by the fact that the relative performances of Model 7 and Model 8 for high performance disk can *triple* their relative performances in the case of common disk as shown in Table 6.1. Besides, the performance of Model 5 is also better than that in the case of common disk. This also can be explained by the above reasons. The most suitable configuration for the best efficiency of the proposed models is similar to the case of common disk and summarized in below again:

- intermediate cache size, e.g. 2M to 4M
- small block size, e.g. 1 sector
- large set associativity of the cache, e.g. 4-way set associative
- small prefetch buffer size
- more occurrences of heading reuse
- less occurrences of killing correct prefetch

Chapter 7

Conclusions and Future Work

7.1 Conclusions

We have discovered that always prefetch performs better than currently used prefetch on miss mechanism because disk access exhibits a highly sequential pattern. The sequential property can be visualized from the formation of large dynamic block size as shown in Table 4.7.

To further improve the cache performance, we have designed few models that are based on the cache partitioning technique and using proposed algorithms to place data in different parts of the cache. The cache is divided into 2 parts: Branch Target Cache (BTC) and Prefetch Buffer (PB). The original algorithm (Model 5) stores the blocks/sectors causing cache misses in BTC and stores all prefetched blocks/sectors in PB. Model 5 is like a cache partitioning model used in CPU cache [Jou90]. The newly proposed algorithms are the Alternative Storing Sectors Technique (ASST) applying to request block (Model 6) and to dynamic block (Model 7) respectively, and the Storing Enough Heads Technique (SEHT) (Model 8). ASST and SEHT are to discard some sectors that need not be stored in the cache by proper overlapping the program execution and the data fetching. The un-stored data are fetched by the cache system during the program execution. The algorithms are designed so that there is little/no time penalty when retrieving the un-stored sectors. Since the algorithms discard some sectors, the cache can store more data from more requests. Therefore, the algorithms can enlarge the effective cache size.

To evaluate these algorithms, we have conducted a simulation study. Our approach is to examine their absolute performances, and relative performances when comparing with the base. We have chosen the base of comparison to be unified cache with prefetch on miss (Model 3). We have also simulated two commonly used algorithms such as unified cache without prefetch (Model 2) and unified cache with always prefetch (Model 4) for comparison.

Model 7 and Model 8 have very good performances under a suitable configuration. *The suitable configuration is a medium cache size (about 2M to 4M), 1-sector block size, higher set associativity (about 4-way to 8-way), and small PB size (about 0.1KB).* Their relative performances can double/triple that of Model 4. This shows that some sectors can actually be discarded if the data fetching can overlap the program execution. This, in turn, has an effect of enlarging the cache size, i.e. the performance of a small cache is as good as an ordinary cache with larger size. For different kinds of disks, the performances of Model 7 and Model 8 still perform well under this configuration. Therefore, we conclude that partitioning the cache into two parts is very useful. ASST and SEHT are good and effective algorithms to control the BTC and PB.

On the other hand, ASST and SEHT have their tradeoff. If the cache size is large enough to hold the working set, their performances will be poor because some sectors are compulsorily discarded. Besides, the accumulated effects of killing correct prefetch and non-heading reuse also lower the performance. If the cache size is too small, the extra stored sectors do not have chance to be reused before they are flushed out. The advantage of enlarged cache size cannot be exploited but the disadvantages of the algorithms still exist. So, the performances of Model 7 and Model 8 are poor when the cache size is too small. There should be a threshold such that beyond the threshold, the extra stored sectors can be reused effectively.

Model 7 and Model 8 perform poorly when the cache size is too small or too large. This is not surprising because the aim of this project is to design effective cache mechanisms so that they can fully utilize the cache when the cache size is limited. The mechanisms perform well in the medium range of cache size. We have verified this view

from the simulation result. Nowadays, multimedia and database applications become more and more popular. The data size is increasing rapidly but the increase in cache size cannot catch up with this speed. Our newly designed models will be very suitable in this situation.

Other parameters also affect the efficiency of different models. An important one is the block size. In current disk cache design, block size is usually set to a large value because this have the effect of implicit prefetching. Current design rarely incorporates the technique of always prefetch. If the block size is large enough, a larger extent of the spatial locality of the references can be captured. However, large block size may bring too many useless data together into the cache. From the simulation, we observe that the performances of Model 7 and Model 8 generally decrease when the block size increases. This is because Model 7 and Model 8 have already incorporated always prefetch mechanism. Always prefetch has the ability to capture spatial locality. Therefore, using a large block size in an always prefetch environment manifests the disadvantages of large block size and degrades the performance.

In the simulation, we have also verified that increasing set associativity improves the performance. However, increasing set associativity increases the time of searching the cache for each request.

From the value of actual sectors transferred, as shown in Table 5.9, we have found that always prefetch does not impose a heavy traffic in the data bus. Always prefetch may reduce the traffic if the trace exhibits a highly sequential property. Therefore, always prefetch is a practical method to improve the performance of a cache.

In conclusion, traditional disk cache design uses very old techniques that were built for CPU cache. It rarely considered the highly sequential interrelationship between successive disk I/O requests. In this project, we have designed a disk cache partitioning architecture controlled by newly proposed algorithms. The main idea is that by proper overlapping the data fetching and the program execution, the cache system can discard some sectors, i.e. some requested sectors need not be stored in BTC. The un-stored sectors can be retrieved by prefetching during program execution. Simulation shows

that the relative performances of the newly proposed models are better than that of unified cache with always prefetch, i.e. Model 4, by as high as 30% in a medium cache size configuration. We conclude that the models are very useful in the design of disk cache.

7.2 Future Work

More traces should be collected from other filesystems to verify the superior performance of the proposed models because the MSDOS filesystem is just one of the many existing filesystems. A more precise simulation should be done to get more accurate performance metrics for disk cache in multi-tasking environment. Besides, there are many write policies such as write back or write through with/without write allocate and periodic update. Their effects on the performances of the proposed algorithms should also be examined.

Bibliography

[Bac86] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1986.

[BF91] B.K. Bray and M.J. Flynn. Strategies for Branch Target Buffer. In *Proceedings of the 24th International Symposium on Microarchitecture. MICRO 24*, pages 42–50, 1991.

[CG94] B. Calder and D. Grunwald. Fast and Accurate Instruction Fetch and Branch Prediction. In *Proceddings the 21st Annual International Symposium on Computer Architecture*, pages 2–11, 1994.

[CGK+88] Peter Chen, Garth Gibson, Randy H. Katz, David A. Patterson, and Martin Schulze. Two papers on RAIDS. Technical Report UCB/CSD 88/479, University of California, Berkeley, December 1988.

[CMCH91] William Y. Chen, Scott A. Mahlke, Pohua P. Chang, and Wen-mei W. Hwu. Data Access Microarchitectures for Superscalar Processors. In *Conference Proceedings, The 24th International Symposium on Microarchitecture*, pages 69–73, 1991.

[CS92] Scott D. Carson and Sanjeev Setia. Analysis of the Periodic Update Write Policy For Disk Cache. *IEEE Transactions on Software Engineering*, 18(1):44–54, January 1992.

[DA95] S. Duvvuru and S. Arya. Evaluation of a Branch Target Address Cache. In *Proceddings of the Twenty-Eighth Hawaii International Conference on System Sciences*, volume 1, pages 173–180, 1995.

[GAN93] Knut Stener Grimsrud, James K. Archibald, and Brent E. Nelson. Multiple Prefetch Adaptive Disk Caching. *IEEE Transactions on Knowledge and Data Engineering*, 5(1):88–103, February 1993.

[GMS88] Hector Garcia-Molina and K. Salem. Disk Striping. Technical report, Computer Research Report, Princeton University, 1988.

[Gon94] A.M. Gonzalez. Design and Evaluation of an Instruction Cache for Reducing the Cost of Branches. *Performance Evaluation*, 20:83–96, May 1994.

[Hos92] Andy Hospodor. Hit Ratio of Caching Disk Buffers. In *COMPCON Spring 1992. Thirty-Seventh IEEE Computer Society International Conference*, pages 427–432, 1992.

[Jou90] Norman P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Conference Proceedings, The 17th International Symposium on Computer Architecture*, pages 364–373, May 1990.

[Kim86] Michelle Y. Kim. Synchronized Disk Interleaving. *IEEE Transactions on Computer*, C-35(11):978–988, November 1986.

[KL91] Alexander C. Klaiber and Henry M. Levy. An Architecture for Software-controlled Data Prefetching. In *Conference Proceedings, The 18th International Conference on Computer Architecture*, pages 43–53, 1991.

[KLW94] Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry. Caching Strategies to Improve Disk System Performance. *COMPUTER*, 27(3):38–46, March 1994.

[KOP+89] Randy H. Katz, John K. Ousterhout, David A. Patterson, Peter Chen, Ann Chervenak, Rich Drews, Garth Gibson, Ed Lee, Ken Lutz, Ethan Miller, and Mendal Rosenblum. A project on high performance I/O subsystems. *Computer Architecture News*, 17(5):24–31, September 1989.

[ME90] Dwight J. Makaroff and Dr. Derek L. Eager. Disk Cache Performance for Distributed Systems. In *Proceedings. The 10th International Conference on Distributed Computing Systems*, pages 212–219, 1990.

[MK89] Y. Manolopoulos and J.G. Kollias. Performance of a two headed disk system when serving database queries under the scan policy. *ACM Transactions on Database Systems*, 14(3):425–442, September 1989.

[Ng91] Spencer W. Ng. Improving Disk Performance Via Latency Reduction. *IEEE Transactions on Computers*, 40(1):22–30, January 1991.

[OS92] Cyril U. Orji and Jon A. Solworth. Write-Only Disk Cache Experiments On Multiple Surface Disks. In *Proceedings. ICCI '92. Fourth International Conference on Computing Information*, pages 385–388, 1992.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Kaltz. A Case for Redundant Arrays of Inexpensive Disks. Technical Report UCB/CSD 88/477, University of California, Berkeley, December 1988.

[Prz90] S.A. Przybylski. *Cache and Memory Hierarchy Design.* Morgan Kaufman Publishers, San Mateo, Calif., 1990.

[PS93] C.H. Perleberg and A.J. Smith. Branch Target Buffer Design and Optimization. *IEEE Transactions on Computers*, 42:396–412, April 1993.

[PSR92] S.T. Pan, K. So, and J.T. Rahmeh. Correlation-based Branch Prediction. In *Conference Record of The Twenty-Sixth Asilomar Conference on Signals, Systems and Computer*, volume 1, pages 51–55, 1992.

[Red92] A.L. Narasimha Reddy. Reads and Writes: When I/Os Aren't Quite the Same. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, pages 84–92, 1992.

[RF93] Kathy J. Richardson and Michael J. Flynn. Strategies to Improve I/O Cache Performance. In *Proceedings of the Twenty-Sixth Hawaii International Conference on System Sciences*, pages 31–39, 1993.

[Ric94] Kanthy J. Richardson. I/O Characterization and Attribute Caches for Improved I/O System Performance. Technical report, Departments of Electrical Engineering and Computer Science, Standford University, 1994.

[SLO90] F. Warren Shih, Tze-Chiang Lee, and Shauchi Ong. A File-Based Adaptive Prefetch Caching Design. In *Proceedings. 1990 IEEE International Conference on Computer Design: VLSI in Computer and Rocessors*, pages 463–466, 1990.

[Smi82] Alan Jay Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[Smi85] Alan Jay Smith. Disk Cache—Miss Ratio Analysis and Design Considerations. *ACM Transactions on Computer Systems*, 3(3), August 1985.

[SO90] J. Solworth and C. Orji. Write-Only Disk Caches. In *Proceedings of the International Conference of the ACM SIGMOD*, pages 123–132, May 1990.

[SV68] D.A. Stevenson and W.H. Vermillion. Core Storage as a Slave Memory for Disk Storage Devices. In *Proceedings of the INFORMATION PROCESSING '68 Conference*, pages F86–F91, 1968.

[TSW92] Dominique Thiebaut, Harold S. Stone, and Joel L. Wolf. Improving Disk Cache Hit-Ratios Through Cache Partitioning. *IEEE Transactions on Computers*, 41(6):665–676, June 1992.

[WEB93] D.L. Willick, D.L. Eager, and R.B. Bunt. Disk Cache Replacement Policies for Network Fileservers. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 2–11, 1993.

CUHK Libraries

000733894