UNIVERSITY of CALIFORNIA
SANTA CRUZ

NON-VOLATILE CACHE MANAGEMENT FOR IMPROVING WRITE RESPONSE TIME WITH ROTATING MAGNETIC MEDIA

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Theodore R. Haining

September 2000

The dissertation of Theodore R. Haining is approved:

Professor Darrell D.E. Long, Chair

Professor Patrick E. Mantey

Professor Jehan-François Pâris

Dean of Graduate Studies

Contents

List of Figures

List of Tables

Abstract

Non-Volatile Cache Management for Improving Write Response Time with Rotating Magnetic Media

by

Theodore R. Haining

A non-volatile write cache is an effective technique to bridge the performance gap between I/O systems and processor speed. Using such caches provides two benefits: some writes will be avoided because dirty blocks will be overwritten in the cache, and physically contiguous dirty blocks can be grouped into a single I/O operation. In this dissertation, we look at how to improve the performance of non-volatile write caches through the use of effective management techniques. We begin by comparing different simple strategies to discover the basic mechanisms governing cache performance. In a series of trace-based simulation experiments, we show that small non-volatile write caches can reduce the amount of data written to disk by as much as 60 percent. Cache performance cannot be measured solely by this metric, however. Policies which best reduce the amount of data written to disk also produce unacceptable numbers of stalled write requests that must wait while cache space is cleaned. Other policies can eliminate stalled writes, but write larger amounts of data to disk. To solve this problem, we present a new block replacement policy that efficiently expels only blocks that are not likely to be accessed again and that coalesces writes to disk. Additional trace based experiments show that a cache employing our new policy is able to match the reductions in data

written to disk and stalled writes of the best simple policies simultaneously. Finally, we observe that cache cleaning produces bursts of writes that increase read response time. We investigate a policy where the cleaning of the cache is delayed until the disk is not in active use, and is unlikely to be actively used again in the near future. We show through analysis and trace-based simulation that while this delay produces a small decrease in mean read response times, it increases write response times because of an increased number of stalled writes.

To my parents, Frank and Sandra Haining, who taught me about the importance of hard work, persistence, excellence, and eduction.

Acknowledgements

I wish to thank Professor Darrell D. E. Long, my dissertation advisor, for his unending patience, support, and technical contributions to this work. I am very grateful for his friendship during my time at the University of California, Santa Cruz. Without his efforts on my behalf and his faith in me during times of trouble, this dissertation would not be complete.

I also wish to recognize the efforts of the other two members of my committee, Professor Jehan-François Pâris and Dean Patrick Mantey for their help in seeing this work to completion. Professor Pâris provided helpful insight when I was unsure how to proceed, and, Dean Mantey showed loyalty which helped to sustain me through this process. I sincerely thank them both.

The support and good wishes of my family and many of friends sustained me during the years in graduate school that brought me to this point. My parents, Frank and Sandra Haining, as well as my sister Sarah, aunt Beverly, and cousins Jim and John all gave me the unconditional love and encouragement I needed through my many years in graduate school. I also complete this dissertation with warm thoughts of my late grand-parents Frank and Anna Rimm in mind. I know the value that they put in education though they had little education themselves; I know how proud they would be if they had lived to see this achievement. Many friends have also helped me during my years in Santa Cruz including: Chane and Helen Fullmer, Joe Pate, Father Roland Bunda, Brother Tony Young, Father Dennis Steik, Jochen and Karin Behrens, Michelle Abram, Ken and Bridget Smith, Veronica Plata, Jorge Garcia, Melissa Cline, Ann Urban, Bill Nitzberg,

Honora Boettcher, Teresa Stirling, Ahmet Amer, and Geoff Bryden. You all helped make Santa Cruz and California into my home and gave the support I needed in moments of crisis.

I would also like to acknowledge the assistance of many members of the technical and administrative staff (past and current) at the Jack Baskin School of Engineering. Terry Figel, Paul Tatarsky, Lylace Blake-Garcia, Lisa Weiss, Helyn Hensley, Marna Evans, Gary Moro, Diane Brookes, and Carol Mullane all provided practical assistance, good will, and support that helped complete this disseration.

This original research in this dissertation was supported by the Office of Naval Research under grant N00014-92-J-1807 and by the National Science Foundation under grant PO-101052754.

Finally, I would like to acknowledge John Wilkes at HP Labs in Palo Alto for providing the trace data and code libraries that made this dissertation possible. The HP disk activity traces were an invaluable tool in the completion of this dissertation. I am grateful for his willingness to make the HP traces available for others to use in their research.

Chapter 1

Introduction

Volatile and non-volatile memory caches offer distinct advantages when used with I/O subsystems such as magnetic disks. Data read from disks can be held in memory to reduce the number of times data is read from disk. Data to be written to disk can also be held in memory where it may be over-written or consolidated into fewer, larger disk I/O operations. Whether for reading or writing, memory caches allow response times much more comparable to CPU speed rather than those of rotating media. With recent advances in technology, random access memory (RAM) and non-volatile RAM (NVRAM) have become more affordable in large quantities for I/O applications.

The focus of this dissertation is on the replacement strategies used to write data from a non-volatile cache to disk in order to make room for current and future write requests. The replacement policy strongly affects cache performance by determining how long data resides in the cache. The amount of cache free space, the number of cache overwrites and the amount of data written to disk are influenced by this duration of

residence in the cache. An effective cache policy will have more overwrites in the cache, write a smaller overall amount of data to disk, and more often have space available to hold data from incoming write requests. This will increase I/O subsystem performance by allowing a maximum number of write requests to complete at memory speed, and produce the smallest waiting time for reads because the disk is not busy performing writes.

In particular, we investigate three key issues related to non-volatile write cache performance that are not adequately addressed elsewhere. We compare the performance produced by different replacement algorithms under similar workload conditions in order to choose an effective strategy, and find the key metrics used to choose between strategies. We employ techniques developed for disk scheduling algorithms to create a new replacement algorithms which provides better overall cache performance. Finally, we investigate the use of idle detection to delay cache cleaning to improve read response times with non-volatile caches.

We examine the problem of choosing an effective replacement strategy in three ways using a combination of analysis and trace-based simulation experiments. We first perform a comparative study of three simple replacement strategies for a non-volatile cache to gain insight into the basic mechanisms (temporal locality, spatial locality, seek distance) at work. Algorithms that exploit each of these mechanisms are used individually in other simulation studies and commercial products but little is known about how they compare under similar working conditions. Our initial work addresses this problem by looking at the performance of different cache management algorithms under similar

conditions from the same trace-based workload.

Results of this simulation study reveal flaws in the use of simple cache cleaning algorithms because both temporal and spatial locality govern cache performance. We use these results to better characterize the disk workloads we use in simulation. We observe that disk blocks fall into two groups based on their frequency of access: hot blocks that are frequently written and will be often updated in the cache, and cold blocks that are written a small number of times over a long period and seldom updated in the cache. Using this observation, we develop a new, original replacement policy to better write dirty blocks from the cache to disk. This new policy divides the cache into hot and cold zones. Recently written blocks are held in the hot zone until they remain unmodified for some period of time, and are then moved to the cold zone. Blocks are cleaned from the cold zone in large groups. We show in a series of simulation experiments that this new policy produces better performance than the three simple replacement policies we tested earlier, with none of their deficiencies.

Finally, our work shows that cleaning the cache causes increases in read response time because writes are made to disk in bursts. These bursts cause read operations to wait for service while the writes are completed. To solve this problem, we defer cleaning the cache until the disk is idle, *i.e.* the disk is not servicing a read operation and is unlikely to do so in the near future. We do this by using a simple idle detector that waits for the disk to be unused for a fixed period of time before writing dirty blocks to disk. An analytical model of the cleaning cycle shows that cleaning with idle detection reduces read response time but also increases write response time because the cache is

kept full during the idle detection period. Trace-based simulations confirm these results.

1.1 Motivation

As processors and main memory become faster and cheaper, a pressing need arises to improve the write efficiency of I/O subsystems. Disks in particular are larger, cheaper, and faster than they were 10 years ago, but their performance still lags that of other subsystems. By looking at the latency and throughput of the different components of a modern computer workstation, this lag is largely due to disk latency.

For example, a modern computer workstation will have a CPU clock rate upwards of 700MHz. The time to access one word of memory from RAM is at most 5 CPU clock cycles, or at most about 7ns. Getting a word of information across a PCI-type bus is a little slower due to a slower bus clock speed of 100MHz, but still is around 60ns. To get a word of data across a SCSI bus requires even more time, approximately 250ns. All of these latencies are within two orders of magnitude of the clock rate of the CPU. The disk latency of disk is about 8ms for the latest generation of hard drives or approximately six orders of magnitude larger than CPU clock speed.

Table 1.1: Approximate timing information for the data path from processor to disk.

	Memory	PCI Bus	SCSI Bus	Disk
Latency	1 clock	3 bus clock	200nS	8ms
Throughput	1 word/3 clock	1 word/2-3 bus clock	20 MB/s	4 MB/s

The effect of this disparity is most felt with I/O intensive tasks, especially write-dominated tasks. Read traffic is affected, but large main-memory caches are an effective technique for reducing the number of reads made to disk [40]. Therefore, the CPU

must most often wait for disk writes to write or update file data or file system data and metadata.

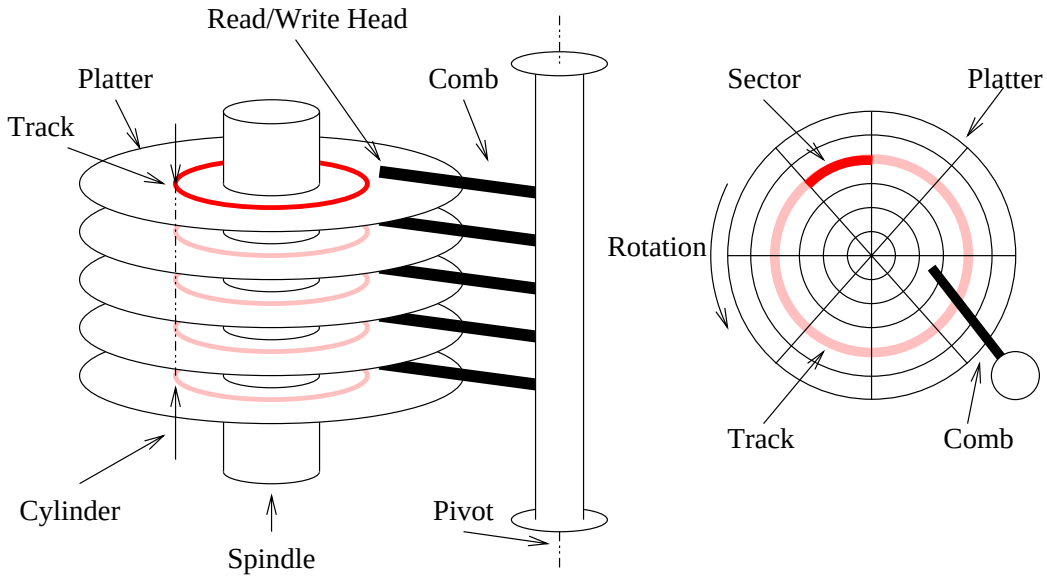
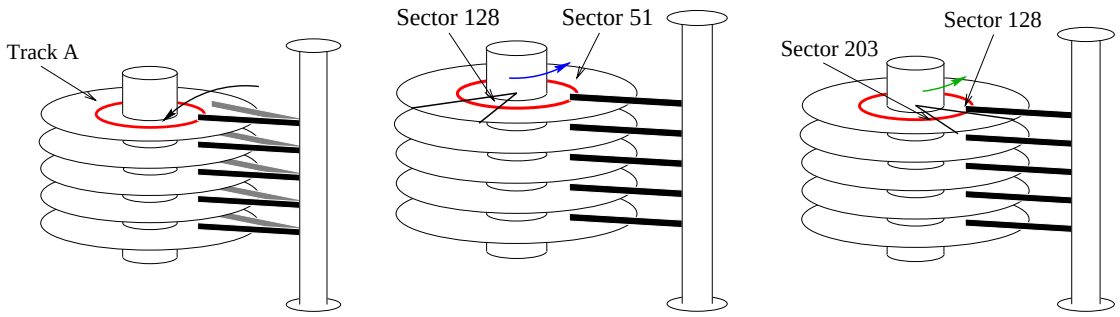


Figure 1.1: The major logical and physical components of a magnetic disk drive. Disk platters are attached to a central spindle which together spin around the spindle's lengthwise axis. One set of magnetic read / write heads per platter are mounted at the end of parallel arms, which together form a comb. An actuator forces the comb to pivot, swinging the read / write heads between the outer and inner edge of the platters. Data is laid out on the platters in concentric rings called tracks. The tracks at the same distance from the spindle on all of the platters form a cylinder. Each track is divided into arcs called sectors.

The write latency of disk arises from the design of the disk itself (shown in Figure 1.1). A magnetic disk satisfies an I/O request by taking at most four distinct steps: seek, settle, rotational delay, and read / write. A *seek* occurs when the disk actuator moves the comb to reposition the disk head over the cylinder containing the track where the next I/O request is located. The disk head is precisely aligned with the track under the disk head during a *settle* after a seek or track switch (when two consecutive I/O operations access data in different tracks in the same cylinder). *Rotational delay* occurs



(a) Seek (b) Settle and rotational delay (c) Read / Write

Figure 1.2: The phases of a disk I/O operation for sectors 128 to 203 of Track A. The read / write heads first move over Track A during a Seek. When the head is positioned over Track A, it must settle to be precisely aligned over the data portion of the track. When the settling is complete, sector 51 is currently under the disk head. The disk must spin until sector 128 moves under the head. Finally, sectors 128 to 203 can be read or written.

while the disk spins until the first sector of the I/O request passes under the disk head. The data is read or written onto the disk during the *read/write* step. The steps in a sample I/O operation are described in Figure 1.2. Much of the available I/O bandwidth that a disk can provide is lost while the disk head is positioned over a requested sector in each I/O operation.

Delaying write operations indefinitely in non-volatile memory before sending data to disk reduces this loss of bandwidth. The longer that data are held in memory, the more likely they will be overwritten or deleted, reducing the necessity for writing them to disk. It is also more probable that data in non-volatile memory can be grouped together to yield larger, more efficient writes to disk. Therefore, a buffer cache can substantially decrease the number of read and write operations actually serviced by the disk.

This substantially reduces the amount of bandwidth lost by eliminating some of the time necessary to reposition the disk head.

1.2 Non-volatile write caching

There are several different ways to incorporate a non-volatile write cache into an I/O subsystem, each with advantages and disadvantages. At the lowest level, a non-volatile cache could be attached to the controller of a hard drive. Such a cache would have exact knowledge about the geometry and layout of the disk and be able to optimize disk access accordingly. A non-volatile cache could also be attached to a channel controller for some I/O channel, such as a SCSI bus controller. This offers the ability to cache data for more than one I/O device, but may require more resources to model all the devices on the channel than controllers currently possess. A still higher level approach would be to place the cache in a block of NVRAM accessible to the kernel as a separate operating system device or mapped into the kernel's memory space. This allows the kernel to potentially combine a non-volatile cache with whatever other I/O buffering methods it uses. It also allows the use of CPU and main memory resources to model the layout of data on all I/O devices connected to the system and actively distinguish between file system data and metadata. The layout information modeled by a kernel would not be as exact as in a disk controller cache, and would possibly expose the cache to corruption caused by crashes in the operating system kernel.

We choose to examine this last approach, by modeling a non-volatile cache as a block of battery backed up memory addressable by the operating system kernel. We

use disk activity traces showing the requests produced by a volatile disk cache in the kernel of the operating system and the write cache we propose operates at the same architectural level. This design enables us to perform trace-based experiments with an I/O subsystem that is minimally modified at the kernel level only. We also believe that this ultimately offers the most flexibility in terms of availability of resources, cache configuration, and cache management. Although this dissertation only discusses caches which interact with one disk, a kernel level cache can be extended to interact with multiple disks. The cache management algorithms we discuss require few resources, but, this design imposes few limits on algorithmic complexity because of the substantial resources available at the kernel level. Changing or updating cache management methods and parameters requires changing kernel level operating system software in this scheme. Making such modifications is a routine task in practice compared to updating software or firmware on channel or disk controllers.

Given that many operating systems already use some kind of file system cache, two approaches can be used for incorporating non-volatile memory into an existing kernel disk cache, *write-aside* and *unified*. In the *write-aside* model, file data are written both into a volatile cache and the NVRAM. The NVRAM is used to protect the permanence of the dirty data in the volatile cache, and is otherwise not accessed after system failure. In the *unified* model, the volatile memory and NVRAM are treated as a single larger, logical cache. An individual block may reside in either memory (but not both), and all dirty blocks are required to be kept in the NVRAM.

We used the write-aside model for our experiments, for the practical reason that

the disk activity traces we used were already filtered through a volatile read cache. It is not possible to reconstruct a unified cache with read and written data together based on this filtered activity. It is possible, however, to approximate total cache performance with a write-aside model by assuming that the volatile cache is partitioned into separately managed spaces for read and written data. Most read hits will occur in the section reserved for read data and these hits are filtered out in the trace. The non-volatile write-aside cache will mirror the contents of the portion of the volatile cache reserved for written data. By explicitly monitoring the write-aside cache, we are then able to take measurements of cache performance in term of block replacement, overwrites of data, and read hits of recently written data.

Finally, it is important to describe the basic operation of the non-volatile write caches we use and define some terms. In our cache design, dirty blocks generated by the file system are written to the cache and eventually updated on disk in physical track-based groups. By writing data from the cache based on physical location, we amortize the cost of the seek phase of a write across a larger number of blocks and increase throughput. We call blocks written into the cache that have not yet been updated on the disk *dirty*. Dirty blocks must be written to disk before they can be overwritten in the cache. Blocks in the cache that have been written to disk are *clean* and are eligible for replacement. The process by which dirty blocks written to disk and become clean is called *cleaning*. Since the act of cleaning blocks and replacing blocks in the cache are very closely connected, we use the terms cleaning policy and replacement policy interchangeably.

The cache cleaning policy is responsible for the management of the number of clean and dirty blocks in the cache. To implement a replacement policy, the cache is organized into a pool containing all the blocks in the cache, and two track-based lists of dirty and clean blocks. Blocks are added to the dirty list when they are written into the cache and become part of the block pool. During cache cleaning, blocks are moved from the dirty list to the clean list in track-based groups in the order chosen by the replacement policy currently in use. If a write request includes blocks from a track on the clean list, the clean blocks already on the track (if any) are deleted from the block pool, and the newly written data in that track moves to the dirty list. When a write request arrives with blocks not already in the pool, clean blocks are removed from both the block pool and clean list. If not enough clean blocks exist to hold the incoming data, the request waits while dirty blocks are explicitly cleaned to disk. This is called a write *stall*.

1.3 Performance measures

We wish to study the impact of cache replacement policy on the utilization of the disk with a specific emphasis on overall response time. Unfortunately, the event arrival times in disk activity traces reflect the interaction between the running processes, the kernel, and the I/O subsystem at the time when the traces were collected. In particular, the times at which certain I/O requests are made depend on when prior I/O requests completed. Altering the behavior of the I/O subsystem with a simulated non-volatile write cache will produce different event completion times, but, the time at which all requests are made is fixed by the trace. This causes differences in the number of I/O

requests waiting for service in certain circumstances and results in incorrect request response times. Therefore, we cannot not simply measure read and write response times directly and use them as accurate measures of system performance.

Instead, we break I/O subsystem response time into two parts. First, we measure *service time* which includes the seek time, settle time, rotational delay, and the time to transfer data to/from the disk. We also measure the *queue time* each request spends waiting to be serviced. The service times for read and write requests are directly comparable between those in the trace because a well-implemented disk simulator will accurately reproduce the service time of the drive it simulates. Use of the queue times is more problematic because the number of requests waiting for service may be inaccurate. We therefore cannot compare simulated queue times and response times to those in the original trace, but we do use them for comparing different simulation runs.

It is also important to measure of how much of the disk activity in the trace is eliminated by using a non-volatile write cache. To do this, we count the number of times that writes were made to the disk by cleaning the cache, and the size of each write used to clean the cache. By comparing the total amount of data written to disk from the cache with the total amount written in the trace, it is possible to get an idea of the amount of work saved through overwrites. The number of cache hits and cache misses add further dimension to how often data is overwritten in the cache. Some read operations are also satisfied by data in the write cache, and a count of the number of read hits indicates further work saved by the write cache. Finally, the number of writes generated by cleaning the cache provides information about the amount of seek time (and potentially wasted

bandwidth) generated by the I/O requests in the trace.

1.4 Organization

This dissertation is organized into several chapters. In Chapter 2, a brief review of related research in the areas of write caching, disk scheduling, and adaptive block placement is presented. Chapter 3 presents a detailed examination of some of the basic strategies we use to test write cache performance. We use the results of these performance tests to develop a new cache management algorithm and present it in Chapter 4. In Chapter 5, we develop a probabilistic model of cache performance and apply it to the problem of modeling simple idle detection for cache cleaning. Chapter 6 describes the simulation environment used as the basis for all of our experiments. The results of our simulation experiments are shown and analyzed in Chapter 7. The conclusions of this work are summarized in Chapter 8.

Chapter 2

Related Work

There is a large continuum of work devoted to hardware and operating system level methods for improving the write latency of I/O subsystems. This has included techniques for managing I/O requests in memory and rearranging data into more efficient patterns on disk. In this chapter we will discuss some of the research related to the original research presented in this dissertation.

2.1 Volatile Write Caches

One way operating systems have traditionally dealt with the problem of disk latency is to use a volatile disk cache in the operating system kernel (such in the BSD UNIX operating system). Read operations are handled in a read-through manner; information is read from disk when requested for the first time, and stored in the cache. The cache satisfies subsequent requests for this data without reading from disk. Space for newly read data is made in the cache by ejecting the least recently used data. Such caches are

very effective, typically satisfying $80 - 90$ percent of read requests with a 512KB to 5MB cache [39]. The I/O requests that remain tend to be write dominated, and the dominance of write I/O requests creates the basic problem we examine in this dissertation.

Smith [48] is among the first to describe how similar techniques might be adapted to reduce the disk latencies associated with writes. He describes two possible types of write caches:

- a write through cache in which all writes are immediately sent to the disk surface with copies kept in memory, and
- a copy back cache which accepts writes and transfers them to the disk when they are pushed out of the cache.

Alhough no data is provided to suggest how effective either type of cache might be, there is some discussion of practical design considerations that others ignore. Removable media pose a particularly interesting problem for write caches; written data must be flushed from the cache to the media before it is removed. Other machines have I/O architectures that are also not amenable to write caching. The IBM count-key-data architecture, for example, initiates a transfer of data first and indicates if data is to be read or written last. This makes it difficult to direct data to a special purpose cache rather than disk.

An implementation of a volatile disk write cache is described by McKusick *et al.* in the description of the Berkeley Fast File System (FFS) for BSD UNIX [33]. When a process writes data to a disk with the Berkeley FFS, data is placed into the write cache with a *write* command. The cache is flushed to disk when the cache becomes full, the writing process issues a *sync* command, or the kernel flushes dirty data in the cache at fixed in-

tervals. The presence of a cache allows the writing process to complete an asynchronous write request in the amount of time necessary to copy data from user memory to kernel memory. If the written data is already present in the cache, it is overwritten and the amount of data written to disk is reduced. Since the cache is volatile and its contents are not immediately written to disk, data can be lost if the computer crashes. Trace based simulation shows that such volatile caches can also be very effective, eliminating 65–90 percent of all disk accesses for file data [40].

While a volatile write cache does decrease the response time of many writes, the approach has serious drawbacks. In order to reduce the amount of written data lost during a sudden system failure, the dirty contents of the cache must be written to disk frequently. These writes may force read operations to wait by increasing the amount of time the disk spends writing data. It also reduces the amount of write activity absorbed by the cache because cache blocks that may be overwritten in the near future are written to disk.

Later work with volatile write caches attempts to reduce the amount of time required to write out the contents of the cache. The Sprite Log-structured File System (LFS) described by Ousterhout and Douglis [39] and Rosenblum and Ousterhout [42] describe a special disk layout to reduce the amount of time needed to clean the cache. Sprite LFS collects newly written data in a file cache in main memory like BSD UNIX, but also abstracts the disk into a sequential, segmented log. When the file cache is cleaned, it is written as a new contiguous segment at the end of the log as a single large I/O operation. Old segments on the disk are periodically coalesced into a new segment at

the end of the log and then marked as free when the new segment is full. Appending writes to the end of a log in this way significantly reduces the amount of time required to clean the cache.

Solworth and Orji propose using in a write cache in a similar way with a different kind of write protocol [38]. Data is written into a main memory write cache, but data is opportunistically written from the cache to a set of reserved blocks on each track. The cache uses the seek and rotational latency associated with a read to write data in the reserved blocks on each track, "piggybacking" writes onto reads. The reserved blocks are eventually cleaned when the data they contain are written back to their regular, fixed locations. Using this method to write data out of the cache is especially effective if the amount of data to be written is small since the number of reserved blocks on each track is small. Since other work has shown that over 90 percent of writes to disk are less than eight kilobytes in size [45], this is an effective approach.

These techniques do reduce the time spent cleaning the cache but the bursty nature of cache cleaning still creates problems. Carson and Setia perform a queuing model analysis to determine the average read response time when a periodic update write policy is used with a write cache [8]. The authors conclude is that bursty arrivals of writes caused by the periodic flushing of the cache creates a "traffic jam" effect that severely degrades service except when the disk workload is highly read-dominated.

This conclusion is confirmed by work with volatile write caches in the read dominated workload of online transaction processing systems (OLTP). Zivkov and Smith [47] use an OLTP system with a cache similar in operation to the one used by the Berkeley

Fast File System. Trace-based simulations show that delaying writes and writing them to disk in batches resulted in no detectable decrease in write traffic, although batching writes may decrease the amount of time spent writing to disk. This work also showed that a mixed write policy that uses a write-back scheme for temporary data and a write-through policy for permanent data increased reliability over a pure write-back cache with very little effect on write traffic.

Carson and Setia extend their previous queuing analysis to determine how batch write size for Sprite LFS affects the response time of read operations [9]. Cleaning the cache and coalescing disk segments require new segments of the disk be created by a set of batched writes. These writes will cause read operations to block until they are completed. Breaking up large batches of writes into smaller write groups can reduce the amount of time that read operations block. This work uses an analytical method to determine the size of those write groups which minimizes read response time while still retaining the advantages of write batching.

The key limitation of the caches described thus far is their volatility. If the system using a volatile cache suddenly crashes, the contents of the cache are lost. Related work with volatile caches attempts to reduce this problem by cleaning the dirty contents of the cache to disk at regular intervals. A frequent sync of the cache reduces cache performance, however, by reducing the number of overwrites in the cache and increasing disk activity. A better way to deal with sudden system failure is needed.

2.2 Non-Volatile Write Caches

A more desirable solution is to make the write cache non-volatile. This is easily accomplished by making the system more robust with an uninterruptible power supply (UPS) or a small amount of battery backed up RAM (NVRAM). It is possible to delay writes indefinitely with a non-volatile cache which provides two major benefits:

- the lifetime of frequently written data in the cache can grow, increasing the number of overwrites in the cache, and,
- the periodic cleaning of the cache is no longer required, potentially reducing the overall amount of disk activity.

If the cache no longer needs to be cleaned at fixed intervals, cleaning is best governed by how many dirty blocks there are in the cache. This creates the potential for some dirty blocks to remain in the cache for longer periods of time, especially when arrival rates are low. A longer cache lifetime for dirty blocks may also mean reduced performance during bursty periods of activity because less clean cache space may be available for use by incoming data.

A common approach to handling cleaning is to use cleaning strategies that examine all dirty blocks in the cache when deciding to clean. Such strategies can include cleaning the cache only when the cache is full, the disk is idle, or some kind of threshold on the number of dirty blocks is reached. This allows a cleaning strategy to choose from the largest possible selection of dirty blocks, making for more (and hopefully better) choices of what to clean. There are several projects that have used this type of non-

volatile cache.

The Rio File Cache, developed by Chen *et al.* [11], uses an uninterruptible power supply and special software to create a non-volatile disk cache capable of surviving operating system crashes. This cache operates much like the volatile cache described by Ousterhout for the BSD 4.2 operating system [40] except that sync calls return immediately, and writes to disk are made when the cache is full. It is important to note that Rio's guarantee of protection is largely provided by software, and as such it does not provide the same level of reliability as a block of NVRAM.

A NVRAM cache is used to improve performance of the Sprite LFS by Baker *et al.* [4] with Sprite servers and disk-less clients. The authors investigate using non-volatile RAM in two places: a non-volatile file cache on client workstations to reduce write traffic, and write buffers for Sprite file servers to reduce disk accesses. Each cache is cleaned when the Sprite segment cleaner coalesces older segments, and when the cache is full.

A similar approach to the Sprite LFS is used with a NVRAM cache by Hu and Yang for the Disk Caching Disk (DCD) [26]. DCD uses a three level hierarchy consisting of a NVRAM buffer, a log structured area of disk consisting of a logical section of one physical disk or a separate physical disk called a cache disk, and one or more disks containing hierarchical file systems. When a write request is made, data are written to the NVRAM buffer and immediately reported as written to disk. The state of the cache disk is checked, and the contents of the cache are written to cache disk as soon as the cache disk is idle. The cache-disk is segmented much like the Sprite LFS [42] and is cleaned using a Last-Write-First-Destage algorithm. As segments are cleaned, data from the cache-

disk are written to their primary physical locations on the data disks. Holes created by overwriting are eliminated as the cache-disk is repacked during cleaning. Extensions of this work include an improved cache structure which requires less NVRAM [27], a patent [55], and the use of the DCD in a RAID array [28].

Work by Hitz, Lau, and Malcolm [24] and Hitz and Cheng [12] describe aspects of the Write Anywhere File Layout (WAFL) and the FAServer NFS appliance by Network Appliance Corporation. Together, the FAServer and WAFL use a NVRAM cache to log incoming NFS write requests and data, as well as hold file system data for generating snapshots. When placed in the cache, each request is split into two separate structures: information representing the sequence number, location, and size of the write, and the data to be written to disk. Writes requests are serviced based on the amount of data they transfer to disk, with the largest requests being serviced first. Overlapping or contiguous write requests in the cache are modified in one of three ways. Contiguous write requests are coalesced into a single larger, higher priority write operation. Exact overlaps of NFS writes (where two writes reference exactly the same disk blocks) cause the write with the lower sequence number to be discarded. Inexact overlaps of NFS writes cause the separate writes to be serviced based on their sequence number, regardless of size.

An alternative approach to cache cleaning involves dividing the cache into segments and using these segment groups to decide when and what data to clean from the cache. This produces more incremental cleaning of the cache rather than large, bursty cleaning operations. The interactions between segment size, read delays, and write batch size remains an open issue, however. Work by Carson and Setia [9] suggests that

the smaller amounts of data produced by incremental cleaning might reduce read wait times, but further work is needed.

The Prestoserve data server from Legato Systems described by Moran *et al.* [35] uses a non-volatile write cache embodied as a bus card with non-volatile RAM. Special device drivers are installed which replace block and raw device drivers for the disks with which the non-volatile write cache is used. These special device drivers process I/O requests, sending the appropriate read and write requests to the cache and others directly to disk. The cache is segmented into a series of small buffers, and the least recently used buffer is written to disk first when cache space is freed. Data is flushed from the cache only when it becomes full or when the number of unflushed buffers falls below a configured threshold. The amount of data flushed from the cache depends on which condition (cache fullness or unflushed buffer threshold) is true. If the cache is full, a pre-configured percentage of the buffers are flushed to disk. Otherwise, buffers are written to disk in small groups (also of a pre-configured size) until the number of unflushed buffers falls below a threshold.

While these systems offer greater guarantees of reliability and higher potentials for overwriting data in the cache, questions about cache performance remain. In particular, little is understood about the relative performance of different management algorithms under similar conditions. The replacement algorithm used in each case is unaccompanied by any analysis or experimental evidence to justify its use. This lack of supporting data leaves unanswered questions about what patterns in disk write access govern the performance of a non-volatile write cache, and what better types of manage-

ment algorithms might exist.

There is also no analysis of the effect of cache cleaning on disk read performance. A segmented approach that may reduce read delays was adopted in some studies and products, but no published work justifies this choice. Work by Carson and Setia [9] with volatile caches suggests that this may be a sound choice, but non-volatile caches use significantly different cleaning techniques. Approaches that better compliment non-volatile cache management policies may exist.

2.3 Caching with Disk Scheduling

One powerful approach to improving overall I/O performance is to schedule disk operations to take advantage of the read and write characteristics of the disk. Such scheduling can reduce the amount of time the disk head spends seeking from track to track, improving mean response time. Log-structured file systems implicitly manage the position of the disk head by appending data to the end of a log, but the LFS write cache needs no special features which distinguish it from caches used by other file systems. Other systems with non-volatile write caches described so far also do nothing to explicitly take advantage of disk scheduling.

We now consider write caches with specific design features for scheduling disk operations. In order for a write cache to exploit disk scheduling, the cache must explicitly incorporate a model describing how data is laid out on disk and including some knowledge of the disk's seek characteristics. Writes to disk are then made from the cache in a manner that profitably exploits the data layout of the disk to reduce the number

and / or duration of the disk operations needed to clean the cache. Caches that use disk scheduling in this way have been already studied in a number of different projects.

Solworth and Orji [49] propose using a NVRAM cache and opportunistically piggyback a cleaning write of a dirty track onto a read request as with their volatile cache work. The physical track with the largest number of dirty blocks also can be written to disk when explicit cleaning of the cache is required. The major results of their work with a model employing one disk surface show that writes become invisible and explicit cleaning of the cache is not required if read rates are sufficiently high. Assumptions are made for extending these results to disks which have multiple tracks per cylinder that do not include head settle time, making them incompatible with the performance of modern disk drives. Additional work by the authors applies these same ideas to multiple disks [37], showing that a single disk model only works well for small caches. A multiple disk model provides much better performance for large caches.

Biswas, Ramakrishnan, and Towsley study write policies for use with non-volatile write caches with disk scheduling [6]. They investigate cache management policies that use one high threshold, or a high and low threshold. The single threshold cache keeps a fixed percentage cache space free. If the number of dirty blocks in the cache rises above this single threshold, the cache is cleaned until the number of blocks falls below the threshold again. The high threshold has a similar function in a dual threshold cache; cleaning begins once the number of dirty blocks rises above the high threshold but stops when the number of dirty blocks falls below the low threshold. The same authors with Krishna extend this single host result to look at the problem of write backs to file

servers [7] in distributed file systems. All caches write data to disk based on which disk track has the most dirty blocks.

An additional study by Chen, Bunt, and Eager [10] investigates further issues of the interaction of non-volatile write caches in distributed file systems. Specifically, the authors examine the effect of relative cache size and cache management policy where write caches exist at both clients and servers. Three policies were examined: least recently used (LRU), write-back with thresholds (WBT), and LRU purge with thresholds (LRUPT). The LRU policy wrote blocks to disk one at a time in LRU order whenever space was needed to make way for incoming blocks in all simulations. The WBT and LRUPT policies wrote blocks to disk in groups. The choice of what group of blocks to purge depends whether the cache is at the client, the server, or in a stand-alone environment. For server or stand-alone caches, blocks are grouped based the number of dirty blocks in a disk track. Client caches group blocks according to the number of dirty blocks in the same file (since disk addresses are unavailable). The WBT policy purges the block cache whenever the number of dirty blocks rises above a given threshold (as in Biswas *et al.* [6]). The LRUPT combines the LRU and WBT approaches, using LRU order but writing groups of blocks in the same track or file as the least recently referenced block.

Fundamental questions about how the different management algorithms compare in the amount of disk activity they generate, the load they can tolerate, and their effect on read response time remain unanswered. Biswas, Ramakrishnan, and Towsley [6] provide an excellent study of the uses of thresholding and piggybacking but only with

a cache using the largest segment per track management algorithm. The study by Chen, Bunt, and Eager [10] examines caches with three different cache management algorithms, but do so in a distributed file system environment with different client/server cache combinations. Exploiting track-based scheduling with thresholds is therefore a useful tool for managing non-volatile caches, but further study is needed to understand which techniques work best with a given disk workload.

2.4 Caching with RAID

Patterson, Gibson, and Katz propose a high performance I/O system called RAID [41] which presents a unique application for non-volatile caching. RAID uses a technique called data striping to store data blocks of files on multiple disks to allow parallel access to data storage. The disks in the RAID are split into reliability groups with disk space containing redundant information that allows for data recovery should one of the disks in the group fail. The same work by Patterson *et al.* provides a taxonomy of five different ways to stripe data to accomplish data striping and redundant data placement called RAID levels 1–5. The techniques used in the different RAID levels range from simple disk mirroring (RAID 1) to distributing data and parity stripes across multiple disks (RAID 5). For the remainder of this discussion, the term "RAID" will refer to a level 5 RAID disk array that stripes data and parity information across multiple disks.

While RAID offers the advantages of improved performance and reliability at reasonable cost, it also has a major drawback. Files in a RAID consist of a set of stripes spread across one or more disks plus a parity stripe for the file on an additional disk. Up-

dating a block in the file requires that the stripe containing the block and the parity stripe for the file both be updated. Therefore, four disk accesses are required to update a single data block: two to read the old data and parity, and two to write new data and parity. This is known as the *small write problem* and causes severe performance degradation for workloads where large numbers of small writes exist.

Menon and Cortney [34] show how NVRAM write caches offer a viable solution to the small write problem by a technique called Fast Writes. A non-volatile write cache is attached to the controller for the RAID array. Writes made to the RAID are written as dirty blocks into the cache and immediately reported as written to disk with only the delay of a write to memory. The cache data are then written to disks in the array by a process called destaging. For the cache that the Menon and Cortney describe, destaging begins once the number of dirty blocks rises above a fixed threshold and the least recently used dirty block is destaged first. Since many applications read data before updating them, it is very likely that the old value of the data block is in the array controller's cache. It is therefore possible to perform a destage write in three operations – one to read parity, and two to write data and parity.

Varma and Jacobson test using Fast Writes with different write policies for destaging with the NVRAM cache [51]. The authors compare the performance of four different algorithms for destaging using synthetic workloads in simulation:

- First Come First Served scheduling which destages based on the order in which write requests are made,
- Least-Cost scheduling which writes out data based on the shortest access time to

complete a destage write,

- High / Low Watermark scheduling uses the dual threshold cache scheme by Biswas, Ramakrishnanan, and Towsley [6] described above, and
- Linear Threshold scheduling that adaptively varies the rate of destaging to disk based on the instantaneous occupancy rate of the cache.

The results of these simulation experiments show that the linear threshold algorithm provides the best read performance of all four algorithms with a high degree of burst tolerance.

A NVRAM cache also improves RAID performance when recovering from a single disk failure. When a single disk in the RAID fails, it is possible to reconstruct data on the fly from the data and parity on the other disks to satisfy incoming requests and rebuild the failed disk when a spare is installed. Since the reconstruction of data requires reading and assembling data from multiple disks as well as possibly writing to a spare, the RAID can support only two-thirds of its normal maximum workload in this degraded state. The RAID experiences total failure if a second disk in the array fails while the first failed disk is being rebuilt with a spare.

Hou and Patt [25] show how an NVRAM write cache can be used to both increase the workload a RAID can handle and decrease the amount of time needed to rebuild a failed disk when a disk in the array fails. In their scheme, the data needed to recreate the failed disk are read from the other disks in the array when they are idle. The NVRAM cache is used as a log for both incoming write requests and the writes to the spare to recreate the failed disk. Writes are made from the NVRAM log to the necessary

disks in the array when those disks are idle.

The use of NVRAM with RAID provides additional ideas and methods for use with other kinds of disk arrays. The study by Varma and Jacobsen adds an additional cache management algorithm, as well as doing some excellent comparative work for cache management algorithms under similar load. The RAID problem domain is significantly different from that for older, more conventional file systems, however, and the RAID results are not applicable. This still leaves problems relating to the choice of caching algorithm and the performance effects of cache cleaning unanswered for other disk configurations.

2.5 Caching with Online Transaction Processing

The use of an NVRAM write cache has also been studied within the context of online transaction processing (OLTP) systems. Many of the techniques used to update or insert data in OLTP environments are similar to those discussed for file systems; others are unique to this problem domain. Due to strict transaction semantics, data is considered to be volatile until commit time. Therefore, data can be explicitly buffered as pages in volatile memory until commit time, similar to the write and sync semantics of a file system. Unlike a file system, however, an OLTP system typically writes out the data twice: once to a log used to reconstruct the consistent state of the database, and once as part of a database page written to disk. The log is typically written out first, making it possible to delay the write of the database page for an arbitrary amount of time. Checkpoints (similar to the fixed interval cleaning of a file system disk cache) periodically flush

database pages to disk to avoid a search of the log arbitrarily far back in time during recovery. Techniques exist which amortize the cost of log and database page writes across more than one transaction [20].

Copeland *et al.* [14] examine using a NVRAM write cache in conjunction with one such technique, "group commit with spooling". In the group commit technique, transactions are held up until a full page of data can be written to the log, or a timeout occurs. Group commit with spooling performs writes out the log data as a background operation when no read I/O is pending. By modeling an OLTP system with a non-volatile cache using some pessimistic assumptions, the authors show that a NVRAM cache is capable of providing excellent system performance. They also show that even if disk unit cost decreases and UPS cost remains the same, NVRAM caching will be more cost effective in the future because of the decreasing cost and power requirements of DRAM.

A more general comparison of I/O and disk costs for Online Transaction Processing (OLTP) systems is performed by Bhide *et al.* [5]. The authors begin by re-evaluating the analysis performed by Gray [19] used to develop the "5 Minute Rule" – if a one kilobyte item is accessed more frequently than once every five minutes, it should be placed in memory rather than on disk. Their results show that because of the constraints of an OLTP system, the five minute rule must be replaced by a range of critical inter-access times ranging from 500 to 10000 seconds. A cost comparison of different hardware solutions under these conditions shows that with a recovery time constraint of less than 10 minutes, it is cheaper to use NVRAM or force data to disk than use volatile memory.

eNVy is a high performance transaction system built by Wu and Zwaenepoel using a 2GB Flash RAM array with a 16MB non-volatile SRAM cache [54]. Flash chips are write once, bulk-erase devices with slow program times, a limited number of program / erase cycles, and memories that cannot be updated in place. The authors overcome these difficulties by using a segmented, log-based copy-on-write scheme (similar to LFS [42]), page remapping, and a NVRAM write cache to provide low latency in-place update semantics. In this case, the Flash RAM EPROMS are analogous to the segments on disk in LFS. As in LFS, data is written to the cache and invalidated in the Flash RAM array. When the NVRAM cache has been filled to some threshold, its contents are written to a empty space on a Flash RAM module. If no such space is available where the cache controller wishes to flush data, a RAM module is cleaned similar to the process used by LFS.

Wu and Zwaenepoel test three different segment cleaning algorithms in simulation: a greedy heuristic that attempts to reclaim segments with the most invalid data first, a locality gathering algorithm that attempts to spread frequently referenced blocks across all segments, and a hybrid approach that combines elements of the first two schemes. The greedy heuristic performs very well for a uniform distribution of block access but degrades rapidly as locality of reference increased. The locality gathering algorithm exhibits the opposite characteristics, performing well in situations of high locality of reference but performing poorly when block distributions were uniform. The hybrid approach partitions the Flash array into two or more parts and uses locality gathering to move frequently accessed blocks between the parts while using the greedy

heuristic to manage segments inside the parts.

The non-volatile buffer cache employed in the Rio file cache [11] is used in two transaction processing systems: a lightweight transaction processor called Vista by Chen and Lowell [32], and a version of the Postgres [50] database system with non-volatile buffering by Ng and Chen [36]. In both cases, the reliable memory used to create the Rio file cache is mapped into the transaction processing system's address space to eliminate double buffering and wasting memory capacity and bandwidth associated with other kinds of usage semantics. For Vista, the focus of the work is on throughput, and the authors claim an improvement in transaction overhead by a factor of 2000 for working sets that fit in main memory. Reliability is the goal of the work with Postgres, with the authors showing that non-volatile memory can be used in database management systems to substantially increase speed without affecting system reliability.

OLTP workloads are generally read dominated, and may not exhibit the same performance characteristics as the conventional file system workloads that are the focus of this dissertation. They do, however, contribute some important economic arguments for the use of non-volatile write caches. The studies by Copeland *et al.* and Bhide *et al.* show how NVRAM caching with disk is a viable economic alternative to the using disk alone. Systems such as eNVy, Vista, and modified Postgres also provide interesting examples of the speed and reliability that non-volatile RAM caching provides. This work still leaves basic questions about the effects of cache management techniques on file system performance unanswered.

2.6 Other Work

Since disk scheduling will be used in the cache management work presented in this dissertation and has been used with similar caches elsewhere, we now discuss some different approaches to the problem. These disk scheduling algorithms use a variety of techniques, many of which provided useful insights for the analysis of non-volatile cache management problems.

Akyürek and Salem use one approach to the problem of reducing seek times by adaptively relocating blocks on the disk [1]. A small number of frequently referenced disk blocks are copied from their original locations to a small, reserved space near the middle of the disk. The choice of what blocks to copy is made by keeping a reference count for each block of the disk requested during a fixed period of time called a monitoring interval. At the end of the interval, the most frequently referenced blocks and their counts are reported and the counts are cleared in preparation for the next interval.

Trace-based simulation experiments use two strategies for placing blocks in the reserved region: serial placement in the order of their block numbers and organ pipe placement [21]. Results show that this technique can cut seek times substantially for only a small fraction of the data on a disk. Not unexpectedly, the rearrangement scheme works best with read-only file systems with a large number of users producing a stable workload. Simulation tests also show that the choice of block placement policy has little effect on overall results. This result contradicts tests with a modified UNIX device driver in a real system [2]. The organ pipe placement policy works best there, with reductions of over 90 percent in seek time in a large read-only file system.

Ruemmler and Wilkes investigate a similar approach for reorganizing disk blocks using the organ pipe heuristic [43]. Noting that Wong proved that the organ pipe heuristic produced an optimal disk ordering for independent accesses [53], they use traces [45] to test its effectiveness with read requests when dependencies in accesses exist.

The results of their experiments show that dynamic disk shuffling can improve performance, but the benefits are not great: 2–15 percent overall. This is in part because 50 percent of the I/O operations in the study required no seeks at all, and cylinder shuffling offers no benefits in this situation. The authors also note that the reductions in seek times from shuffling are becoming smaller and smaller percentages of the long distance seek times in modern disk drives, making the technique less useful with newer disks. Disks shuffled infrequently (once a day to once a week) with small (block to track-sized) shuffling quanta produce the best results in their study.

Others attempt to reduce write response time by using different methods to schedule the movement of the disk head. Seltzer, Chen, and Ousterhout [46] and Jacobsen and Wilkes [29] describe disk scheduling algorithms based on head position. Performed more or less concurrently, both studies focus on using simulation experiments with shortest seek time first algorithms. Methods in both studies are subject to starvation; new requests continue to arrive and may be performed by the disk before others already queued for service with a large seek distance from the current position of the disk head. Starvation can be limited by preventing additional requests from being added to the queue. This causes the population is the queue grow smaller, however, and disk utilization efficiency decreases. Both studies develop a solution to starvation by adding a

time weighting factor to be used in conjunction with seek distance. Weighted shortest seek time first algorithms show both strong average I/O time (within one to two percent of shortest seek time) with a guaranteed bound on maximum response time [46]).

A study by King [30] examines a disk scheduling policy that statistically anticipates the location where the disk head will be needed next, and moves it there during an idle period. A simple analytical model of an anticipatory first come, first served algorithm (called ANTIC) shows that the distance the head travels can be reduced as much 25 percent. If the distribution of requests is not uniform, the algorithm adapts by shifting the ideal anticipatory position toward one of the "hot" spots on the disk. Since the ANTIC algorithm uses idle periods to move the disk head, the approach is less effective when arrival rates are high. Two possible solutions exist: reduce the size of anticipatory seeks, and make seeks interruptible (something not possible with all disk controllers). Results show that both approaches adapt well to high arrival rates. If the pattern of requests from the disk is localized, ANTIC offers little benefit. If it is difficult to detect when the disk it is idle, it is also hard to choose when to perform an anticipatory seek.

Gerchak and Lu [18] extend this work by attempting to find optimal solutions to the problem of anticipatory placement of the disk head. They examine the problem of an anticipatory arm location under two different kinds of head usage scenarios: the left end (LE) scenario where the requested data is read from beginning to end, and the closer end (CE) scenario where the data can be read backwards and re-ordered in a buffer where necessary. They derive optimality conditions showing that the median of the read start block distribution is optimal for LE. No closed form optimality conditions for CE are

presented, but the authors show that a set of optimal anticipatory locations can be found using numerical analysis. They confirm King's result [30] that interruptible anticipatory seeks are required.

This work in disk head scheduling provides useful ideas of the development of cache management algorithms with disk scheduling that have not been tested in with non-volatile write caches. The algorithm of greatest interest is Shortest Access Time First algorithm employed in the studies on disk scheduling based on head position by Seltzer, Chen, and Ousterhout, and Jacobsen and Wilkes. There are also algorithms which use a combination of basic principles to perform effective head scheduling such as the Weighted Shortest Access Time by Jacobsen and Wilkes. Work with block relocation and organ pipe placement provides useful ideas for using frequency of access of information that assisted in some of the modeling and analysis presented here.

2.7 Conclusions

There is a significant body of work related to using non-volatile write caching with individual disks and disk arrays including research studies and commercial products. Some basic questions are left unanswered with regard to cache performance with conventional file systems, however. Systems with non-volatile write caches such as the Legato Prestoserve [35], Write Anywhere File Layout [12], and studies by Solworth and Orji [49], Biswas, Ramakrishnan, and Towsley [6],and Chen, Bunt, and Eager [10] use a variety of cache management schemes. Work by Biswas, Ramakrishnan, and Towsley [6] provides a very good example of the interaction between thresholding and one cache

replacement policy. Different cache replacement policies are tested by Varma and Jacobson [51] with RAID. Yet little work is presented justifying the choice of those policies, or describing the key metrics used to describe cache performance are presented for more conventional file systems. We address this problem with a comparative survey of three basic cache replacement policies in Chapter 3.

There is also a large body of ideas about disk head scheduling algorithms that has not been applied to the problem of developing non-volatile cache replacement policies. In particular, disk head scheduling algorithms like Weighted Shortest Seek Time First [29] show that useful algorithms exist which combine more basic techniques in effective ways. Based on our survey results, we use a similar approach to develop a new cache replacement algorithm in Chapter 4.

Work with volatile write caches has shown that the bursty nature of cache cleaning can cause increases in read response times for workloads which are not read dominated [8]. Non-volatile caches exhibit similar bursts of write activity, but with the possibility of substantially different solutions based on the non-volatile nature of the cache. We describe one such approach employing delays in cache cleaning impossible with volatile caches in Chapter 5.

Finally, we present trace-based simulation results in Chapter 7 using disk activity traces by Ruemmler and Wilkes. These same traces have been used to test a substantially different non-volatile cache system by Hu and Yang [26].

Chapter 3

Cache management policies

A non-volatile write cache can improve the performance of the I/O subsystem in three major ways. Data can be efficiently retained in the cache in such a way to increase the number of overwrites of data. This reduces the number of writes serviced by the disk – writes which would otherwise delay the service of read operations. Clean space in the cache can be managed to reduce or eliminate stalled writes (writes which must wait for clean cache space to become available). This makes I/O subsystem performance resemble that of the cache for the processes that make write requests. When cache space must be cleaned, writes from the cache are scheduled to reduce their overall response time. This reduces the time reads must wait for service from the disk and the amount of time the cache may be unavailable during cleaning and stalls are forced. Each of these cache characteristics is governed by the exact policy used to clean the cache and the cache size.

In this chapter, we focus on the management policies used to determine when

and how to clean the cache. We describe three basic cache policies for scheduling cache cleaning and provide simple models to show how they can improve performance. We also discuss policies that are used to determine when cleaning begins and ends and discuss how exactly to model the writes to disk.

3.1 Cache replacement policies

A fundamental issue in the design of a non-volatile cache is the choice and order of blocks to be cleaned from the cache as new blocks arrive. The blocks a cache replacement policy chooses influences the number of overwrites in the cache because it determines how long certain blocks remain in the cache. The schedule for cleaning blocks is also set by the cache replacement policy. This schedule influences write response time by increasing or reducing disk seek distance and rotational delay.

Several different replacement policies are used in different write caching studies and commercial products. The Least Recently Used (LRU) algorithm is used in the Legato Prestoserve data server [35], and in a distributed file system by Chen, Bunt, and Eager [10]. The Network Appliance FAS Server [12] and the non-volatile write cache research by Biswas, Ramakrishnan, and Towsley [6] both employed the Largest Segment per Track (LST) algorithm. Both Seltzer, Chen, and Ousterhout [46] and Jacobsen and Wilkes [29] examined the Shortest Access Time First (STF) algorithm for use in reducing write response time. Unfortunately, this published work presents each algorithm without any reference to any of the others and it is difficult to compare the relative merits of these algorithms.

To investigate impact of cache replacement policy on cache performance, we will examine all three of these candidate policies under conditions of similar workload. These algorithms are attractive because they employ three basic principles: temporal locality, seek distance, and write size. Comparing the performance of caches using these algorithms under similar conditions will provide insight into the mechanisms at work in the cache, and determine what the important metrics of cache performance are.

The remainder of this section describes the mathematical bases of LRU, STF, and LST algorithms in detail to lay the foundation for their later use and comparison. Descriptions of two of the algorithms (LRU and STF) compare their respective algorithms to a first come, first served approach using queuing theory and probability theory. The description of the LST algorithm shows that the algorithm cleans an optimally large amount of data from the cache under certain conditions. Direct comparison of these three algorithms using trace-based simulation appears in Chapter 7.

3.1.1 Least Recently Used

In the LRU algorithm, the most stale data in the cache is purged first. LRU ignores the number of times data is written into the cache; it keeps track of the oldest dirty data currently in the cache. This beneficially conditions the cache when data already in the cache is modified because its cache lifetime is extended. Longer lifetimes increase the opportunity for further overwrites of that data to be absorbed by the cache. LRU does nothing to ensure that the write is reasonably short to service or is of significant size.

To capture a reasonable model of the behavior of an LRU cache, it is useful to think of the cache as a stack of length m. Write requests in this model are references to

single blocks on the disk or in the cache rather than extents (this does nothing to change the mechanics of the model, but is simply done for ease of explanation). When a write request arrives, the block it references is pushed onto the top of the stack if that block does not already exist in the stack. If there are already m blocks in the stack, the oldest block is ejected. If the new block is in the stack, it is removed from its place and pushed onto the top of the stack.

If the writes are assumed to be statistically independent (though not uniformly distributed), the write request stream becomes a string of independent random variables $r_1, r_2 \ldots, r_t, \ldots, r_n$ with the common stationary distribution $\{\beta_1, \ldots \beta_n\}$ such that the probability $Pr(r_t = i) = \beta_i$ for all $t \geq 1$. The content of the stack s is an ordered set m of blocks taken from from the write request stream $[j_1, \ldots, j_m]$ with a probability β_{j_k} of j_k being requested. The set of all possible orderings of the m blocks in the stack is Q, making the size of Q the permutation of n blocks taken m ways. According to Denning and Coffman [13], the fault rate function $\mathcal{F}$ which describes the number of blocks written out of such an LRU stack is

$$\mathcal{F}(\mathrm{LRU}) = \sum_{s \in Q} D_1^2(s) \prod_{i=1}^{m} \frac{\beta_{j_i}}{D_i(s)}, \tag{3.1}$$

and

$$D_i(s) = 1 - \sum_{k=1}^{m-i+1} \beta_{j_k}.$$

This model is unrealistic because it treats blocks in writes in the request stream independently; in reality, correlation exists between the blocks when processes write to disk. We merely wish to show (pessimistically) how an LRU write cache will write to

disk less by taking advantage of the fact that some blocks will appear in the cache more frequently than others. When considering cache performance with write dependence between blocks, it is easy to see that correlations between blocks mean that frequently written blocks will be more likely to appear in the cache together in temporally local groups. An LRU cache will be more likely to keep these groups together in the cache because of the similar reference patterns and further reduce the number of writes to disk.

It is also possible to create a similar model of an FCFS cache using a set of independent random variables as input. In this case, the FCFS cache is not a stack but a queue of length m with Q again being the set of queue states. As with the LRU cache, Q is assumed to be the set of all orderings of m blocks with the size of Q being the number of n blocks taken m ways. As writes arrive, they added to the head of the queue. If the queue already contains m items, the tail of the queue is ejected. From Denning and Coffman [13], the rate at which blocks are written out of an FCFS cache is

$$\mathcal{F}(\mathrm{FCFS}) = \frac{1}{G} \sum_{s \in Q} D_1(s) \prod_{i=1}^{m} \beta_{j_i}, \tag{3.2}$$

where

$$G = \sum_{s \in Q} \prod_{i=1}^{m} \beta_{j_i}.$$

Modeling the transitions between states for both the LRU and FCFS can be conveniently done with a Markov chain. It is not difficult to show that both Markov chains are irreducible and ergodic. It therefore follows that there are stationary probability distributions for each cache. If we let $\pi(\mathrm{LRU})$ and $\pi(\mathrm{FCFS})$ denote the vector of equilibrium

probabilities for the LRU and FCFS cache respectively, and P_{LRU} and P_{FCFS} denote the corresponding state transition probabilities, the limiting probabilities must satisfy the following two conditions [3]:

$$\sum_{s \in Q} \pi_s = 1$$

and

$$\pi_j = \sum_i \pi_i P_{ij}, j = 0, 1, 2, \ldots .$$

where i represents the possible states and j the next future state both the LRU and FCFS Markov chains. Therefore, the equilibrium equations for the FCFS and LRU caches are $\pi(LRU) = \pi(LRU)P_{LRU}$ and $\pi(FCFS) = \pi(FCFS)P_{FCFS}$.

Looking at the general behavior of both caches, it is now possible to collect some common terms in Equations 3.1 and 3.2. If we let $p_f(s)$ be the probability at equilibrium that a write occurs given that the cache is in state s, the total rate at which an algorithm X writes out blocks is equal to the sum of the product of $p_f(s)$ and π_s over all states, or

$$\mathcal{F}(X) = \sum_{s \in Q} p_f(s)\pi_s .$$

By using this formula to reorganize terms in the equations for $\mathcal{F}(LRU)$ and $\mathcal{F}(FCFS)$, $p_f(s)$ becomes $D_1(s)$ for both the LRU and FCFS caches. The probabilities for each cache being in the state s become

$$\pi_s(LRU) = \frac{\prod_{i=1}^{m} \beta_{j_i}}{\prod_{i=2}^{m} \left(1 - \sum_{k=1}^{m-i+1} \beta_{j_k}\right)} \tag{3.3}$$

and

$$\pi_s(FCFS) = \frac{\prod_{i=1}^{m} \beta_{j_i}}{\sum_{s \in Q} \prod_{i=1}^{m} \beta_{j_i}}. \tag{3.4}$$

Intuitively, the formula for $\pi_s(FCFS)$ represents the stationary probability of the current queue state as a percentage of the total sample space. The formula for $\pi_s(LRU)$ shows that the stationary probability for s is the fraction of the probabilities that the state s occurs (the numerator) and that of an incoming block not already being in the cache (the denominator).

Directly comparing $\mathcal{F}(FCFS)$ and $\mathcal{F}(LRU)$ is a difficult task because the combinatorial size of the set Q makes it hard to analyze these expressions with any rigor. It is possible, however, to make some statements about $\pi(FCFS)$ and $\pi(LRU)$ caches that give an intuitive understanding of how the two caches perform. Rather than try to compare $\mathcal{F}(FCFS)$ and $\mathcal{F}(LRU)$ in general, let us look at the behavior of π_s and $p_f(s)$ for a simple model. Consider a pool of n blocks consisting of two sets of events, A and B containing a and b events respectively that have exponentially distributed inter-arrival times. Each set has a different uniformly distributed reference probability such that

$$\beta_i = \begin{cases} \kappa & 1 \leq i \leq a \\ \lambda & a+1 \leq i \leq n \end{cases}$$

where $\lambda < \frac{1}{n} < \kappa$ and $\alpha = \kappa/\lambda$. Given a cache which holds m blocks, the composition of the cache is based on a binomial random variable X

$$P(X \leq t) = \binom{m}{t} \left(\frac{a\alpha}{a\alpha + b}\right)^t \left(\frac{b}{a\alpha + b}\right)^{m-t}.$$

The expected value of X is $E[X] = (ma\alpha)/(a\alpha + b)$.

Applying the expressions for $\pi_s(FCFS)$ and $\pi_s(LRU)$ to this example, we get

$$\pi_s(FCFS) = \frac{\kappa^l \lambda^{m-l}}{\sum_{s \in Q} \kappa^l \lambda^{m-l}}$$

and

$$\pi_s(\mathrm{LRU}) = \frac{\kappa^l \lambda^{m-l}}{\prod_{i=2}^{m} \left(1 - \sum_{k=1}^{m-i+1} \beta_{j_k}\right)}$$

where l is the number of blocks from set A in each cache for state s. Likewise, the fault rate for the state s is the same for both the FCFS and LRU caches:

$$p_f(s) = 1 - l\kappa - (m-l)\lambda$$

For the statistically expected case, it is possible to obtain expressions for $\overline{\pi}_s(\mathrm{FCFS})$, $\overline{\pi}_s(\mathrm{LRU})$, $\overline{p}_f(s)$ by making $l = (ma\alpha)/(a\alpha + b)$.

Neither of these expressions has a particularly direct reduction that makes comparing them a manageable task. The summation in $\pi_s(\mathrm{FCFS})$ can be reduced to a permutation expression, but no further. The product of sum expression $\pi_s(\mathrm{LRU})$ cannot be reduced at all without making some explicit assumptions about the order in which items appear in the cache.

Some bounds can be put on these two expressions which do allow them to be compared, however:

$$\frac{\kappa^l \lambda^{m-l}}{S_{lower}(n,m)\kappa^l \lambda^{m-l}} < \pi_s(\mathrm{FCFS}) < \frac{\kappa^l \lambda^{m-l}}{S_{upper}(n,m)\kappa^l \lambda^{m-l}} \tag{3.5}$$

and

$$\frac{\kappa^l \lambda^{m-l}}{(1-\lambda)^{m-l}} \leq \pi_s(\mathrm{LRU}) \leq \frac{\kappa^l \lambda^{m-l}}{(1-l(m-l)\kappa\lambda)^{m-l}} \tag{3.6}$$

where the functions S_{lower} and S_{upper} are upper and lower bound approximations of

combinatorial functions using Stirling's approximation:

$$S_{lower}(n,m) = \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{\sqrt{2\pi(n-m)}\left(\frac{n-m}{e}\right)^{\frac{13(n-m)}{12}}}$$

$$S_{upper}(n,m) = \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^{\frac{13m}{12}}}{\sqrt{2\pi(n-m)}\left(\frac{n-m}{e}\right)^{n-m}}.$$

Given these bounds, it it is now possible to make some useful observations. First, the formula for $p_f(s)$ shows that the fault rate for states with more blocks from A is less than those states with less blocks from A. The stationary probability $\pi_s(LRU)$ will also be larger for those states that have larger numbers of blocks from A in the cache. The probability $\pi_s(FCFS)$ is dependent on the size of the cache alone and is therefore the same regardless of the contents of the cache. Therefore, an LRU cache will be more likely to be in a state that has a higher number of blocks from A at a steady state than an FCFS cache. For this reason, the steady state fault rate of an LRU cache is likely to be lower than that of an FCFS cache.

An example illustrating the relative values of π_s and $\overline{\pi}_s$ taken from the Equations 3.5 and 3.6 are shown in Figure 3.1 and 3.2. The increasing stationary probability of states with a higher numbers of blocks from A given different ratios between λ and κ is shown in Figure 3.1. States on the right side of the graph are several orders of magnitude more likely to occur for an LRU cache than the same states in an FCFS cache at steady state. Since these same states have a lower page fault rate, the steady state page fault rate will be lower for the states at the right of the graph as well.

The expected steady state stationary probability for the same cache is shown in Figure 3.2. The higher values of $\overline{\pi}_s(LRU)$ indicate that the LRU cache is more likely to

be (and remain) in a state with at least $(ma\alpha)/(a\alpha + b)$ blocks than an FCFS cache. This again points to an overall lower fault rate for the LRU cache at steady state.

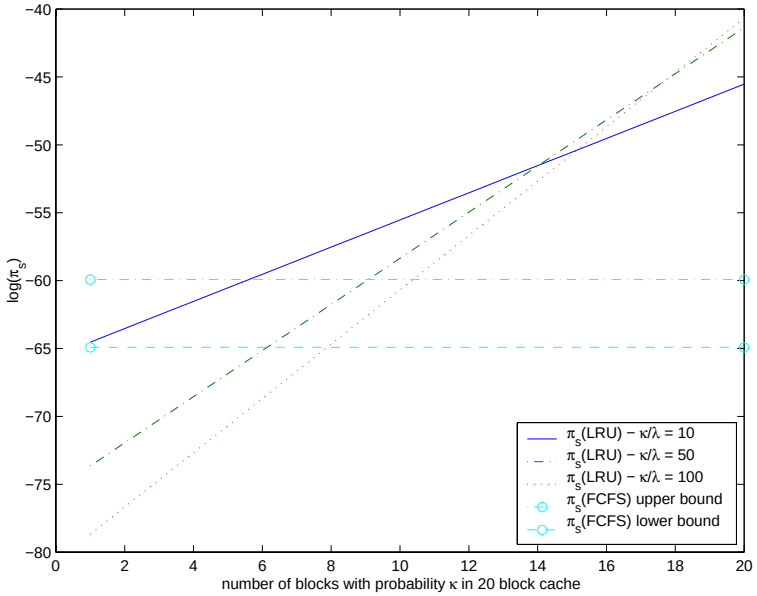


Figure 3.1: The lower bounds of $\pi_s(\mathrm{LRU})$ and the upper and lower bounds of $\pi_s(\mathrm{FCFS})$ for caches where $m = 10$ and $n = 1000$ for different ratios of κ to λ as the number of blocks from A increases across different states.

The LRU algorithm will increase the number of cache hits, but suffers from the weaknesses that it can make inefficient writes to disk and does necessarily maintain large amounts of clean cache space. In certain situations, the blocks that the LRU algorithm picks to clean may result in large amounts of bandwidth being wasted by disk seeks. The writes may also clean space in the cache in small increments, forcing incoming writes to stall.

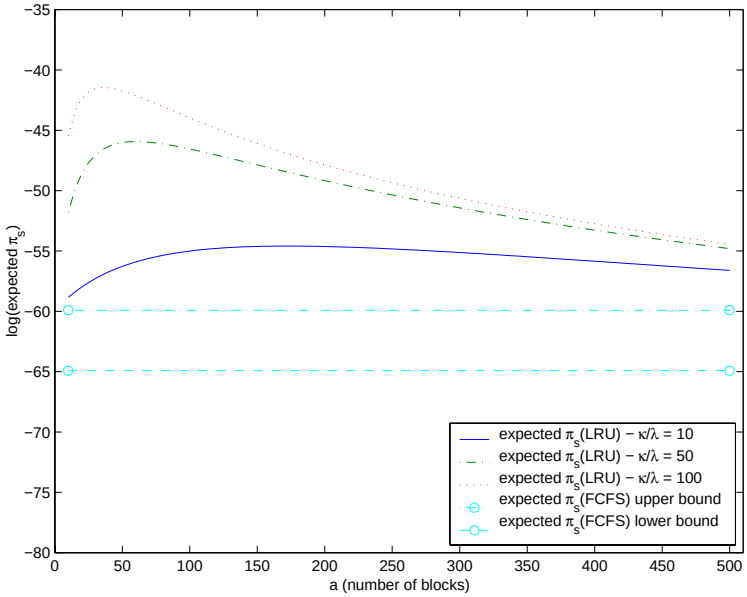


Figure 3.2: The lower bounds of $\overline{\pi}_s(\mathrm{LRU})$ and the upper and lower bounds of $\overline{\pi}_s(\mathrm{FCFS})$ for caches where $m = 10$ and $n = 1000$ for different ratios of κ to λ as the size of A grows.

3.1.2 Shortest Seek Time First

The STF algorithm attempts to minimize the amount of time between when the write is initiated and when the first bit of data is written. The cache controller models the current position of the disk head and writes the data with the lowest sum of seek time and rotational delay. This data may be overwritten in the near future and the amount written may be small. The amount of effort to model the position of the disk head is also non-trivial.

The benefit of this approach can be found by developing an expression for the expected seek time of a series of writes to a hard disk (see Denning [16]). Consider a hard disk with W tracks per platter with the disk arm currently positioned at track k. What is the expected seek time? The answer depends on the number of n operations waiting to be serviced by the disk. We consider these each these requests to be a set of

identically distributed random variables. According to Denning [16], the expected value $E[x]$ of such a set of n variables with a cumulative distribution $\mathcal{F}(u)$ is

$$E[x] = \int_0^\infty (1 - F(u))^n du. \tag{3.7}$$

Given a disk where track 1 is at the outer edge and track W is the inner edge of the disk, it is fairly straightforward to determine the conditional probability of arm movement. Assuming that the head is positioned at track k and $k < W/2$, the disk head can move $k-1$ units toward the outer edge and the inner edge, and k to $W-k$ toward the inner edge. If the requests are assumed to be uniformly distributed across the disk, the probability of a move to each track is probability p where

$$p = \frac{1}{W-1}.$$

This results in the conditional probability distribution shown in Figure 3.3 and the following cumulative distribution function $F(u)$:

$$F(u) = \begin{cases} 0 & u \leq \frac{1}{2} \\ 2pu - p & \frac{1}{2} < u \leq (k - \frac{1}{2}) \\ p + \frac{p}{2}(2k-3) & (k - \frac{1}{2}) < u \leq (W - k + \frac{1}{2}) \\ 1 & u > (W - k + \frac{1}{2}). \end{cases} \tag{3.8}$$

By inserting equation 3.8 into equation 3.7 and integrating, it is possible to develop the following expression for the expected seek time $E[s]$ of n requests:

$$E[s_n] = \left(\frac{W-1}{W}\right)^n \left[\frac{1}{2} + S_{min} + \frac{S_{max} - S_{min}}{2(n+1)} \left(1 + \frac{1}{n+2}\left(\frac{W}{W-1}\right)^{n+1}\right)\right], \tag{3.9}$$

where S_{max} and S_{min} represent the times where the head moves $W-1$ tracks and 1 track respectively. The unscheduled seek time is obtained by setting $n = 1$ in equation 3.9. It

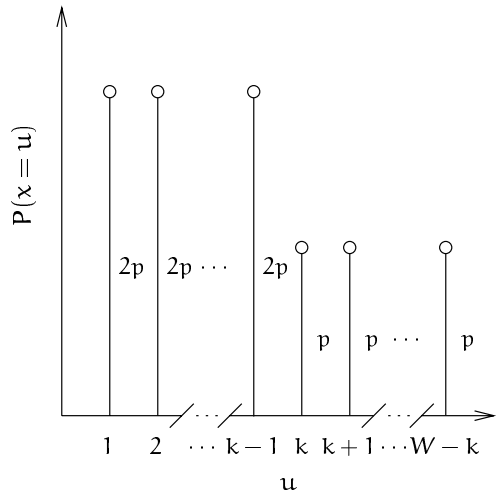


Figure 3.3: Conditional probability that the disk arm moves u units.

is

$$E[s_1] = \left(\frac{W-1}{W}\right)\left[\frac{1}{2} + S_{min} + \frac{S_{max} - S_{min}}{4}\left(1 + \frac{1}{3}\left(\frac{W}{W-1}\right)^2\right)\right]. \tag{3.10}$$

To see how the STF algorithm improves the performance of a non-volatile write cache, consider two non-volatile write caches, one which cleans itself using an STF order and the other using first come, first served (FCFS). If both caches contain the same m blocks which must be written to disk and seeks to clean n of them at once, the STF cache will take $E[s_n]$ time perform this cleaning and the FCFS cache will take $nE[s_1]$ time. This is because the STF cache will use the placement of n blocks together to produce a schedule that reduces the overall seek time required to write all n blocks to disk. The FCFS cache will consider the blocks individually and not take total seek time into account in creating a schedule. The FCFS cache instead writes blocks in the order in which they arrived, in effect creating a schedule equivalent to n STF groups containing only one

block. It is easy to see that $E[s_n]$ will always be less than $nE[s_1]$ for n greater than one, and the STF cache will take less time to complete its cleaning.

Problems with the STF cache management algorithm arise because this approach only considers seek time in its measure of cost. Two other factors can play a role: the availability of free space in the cache and cache hit patterns. By writing out blocks in the track nearest the current position of the disk head, the number of blocks being written out is ignored and may be small. This may create situations where small amounts of clean space are available in the cache and incoming writes will be forced to wait while additional space is cleaned. The extent closest to the disk head may also be overwritten in the near future, and by writing it out disk, an I/O is created that could be avoided.

3.1.3 Largest Segment per Track

With the LST algorithm, the cache controller sorts the dirty list in the cache by the number of blocks in each dirty track. It cleans the track with the largest number of dirty blocks first. LST requires very little state information. It has the advantage that it initiates the purge that will free the largest possible amount of space in the cache.

Cleaning space in a non-volatile write cache is an instance of a fractional knapsack problem. Such problems are posed as follows:

> A thief robbing a store finds n items: the i^{th} item is worth v_i dollars and weighs w_i pounds, where v_i and w_i can be fractions of the i^{th} item. He wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack for some integer W. What items should he take? [15]

In this particular case, each of the items is a dirty track based extent in a snapshot of the

cache at taken at time t. Each of the items is assumed to have unit weight (cost), and so filling the knapsack is merely a matter of choosing the items with the largest size (i.e. the largest value). The knapsack is able to hold whatever pre-determined fixed percentage of the cache necessary to reduce the number of dirty blocks to below a fixed threshold and make the cache “clean”.

The optimal solution to this problem is to put the largest sized dirty track based extent into the knapsack. Once that item is in the knapsack, all or a fraction of the largest remaining dirty extent in the cache is added to the knapsack. This last step is repeated until the knapsack is full.

The proof of the optimality of this solution relies on the fact that a globally optimal solution can be arrived at by making a locally optimal choice. If $S = \{1, 2, \ldots, n\}$ be the set of extents to be cleaned ordered by decreasing size, let 1 be the extent of the greatest size. Suppose $A \subseteq S$ is an optimal solution to cleaning the cache and that the first event in A is k. If $k = 1$, then schedule A begins with the greedy choice. If $k \neq 1$ then let $B = A - \{k\} \cup \{1\}$. Because the size of 1 is less than or equal to that of k with A and B having the same number of extents scheduled, B is also optimal. Thus B is an optimal solution for S that contains the greedy choice of extent 1. Therefore a schedule always exists that begins with the greedy choice.

Once the greedy choice of 1 is made, the problem reduces to finding an optimal solution for those extents that remain in the cache that are compatible with extent 1. If another solution B' that solved $S - \{1\}$ with fewer (at least one larger) extents than $A - \{1\}$, then $B' \cup 1$ would yield a solution with larger extents than A and contradict the optimality

of A. Induction on the number of steps made shows that making the greedy choice at every cleaning step produces the optimal method for freeing cache space.

The weaknesses in this approach arise from relying on a snapshot to decide the cleaning schedule and the assumption that cleaning cost is always proportionate to the extent size. Cleaning cost is made up of three components, seek time, rotational delay, and read/write time, of which only the last is proportional to extent size. Therefore the cost to clean an extent is often not strictly proportionate to the size of the extent and the schedule may incur large amounts of unaccounted cost due to seek time. The use of snapshots to determine the cleaning schedule ignores the fact that certain of the extents in the snapshot may be overwritten in the near future. By writing out these blocks, disk activity is created which could otherwise be avoided.

3.2 Cache eviction policies

The choice of cache replacement policy dictates what blocks to clean from the cache, and influences the frequency and duration of the cleaning proces. It does not, however, describe when cache cleaning begins and ends. The decisions to begin and end cache cleaning are part of a cache eviction policy.

The simplest policy we use to free space in the cache is write behind. Sectors are written to the cache until the cache is full, and then dirty tracks are evicted when a new write request arrives. Because blocks are evicted in cache based groups, the scheduling algorithm can amortize the cost of several write requests in one write. The disadvantage of this approach is that the write request that causes the cache to become full is stalled

until a portion of the dirty list can be written to disk.

A simple way to improve this policy is to add thresholds to the cache to create pro-active eviction policies. A single threshold cache begins to clean after the percentage of dirty blocks in the cache exceeds a *high* threshold. When the threshold is crossed, cache controller sets a “clean request” flag. Once this flag is set, the controller commits groups from the dirty list and moves them to the clean list. The clean request flag is reset when the percentage of dirty blocks falls below the high threshold. A dual threshold cache adds a second *low* threshold. In this case, the clean request flag is not reset until the percentage of dirty blocks is less than the low threshold. The combination of high and low thresholds delays the start and stop of purging from the cache, creating hysteresis.

Adding thresholds has a number of advantages. Thresholds ensure that there always is some free space in the cache and that stalls only occur during bursts in write traffic. The use of thresholds means that cache cleaning does not need to be performed immediately. The controller can attempt to clean the cache when the disk would otherwise be idle, for example. The use of both high and low thresholds means that the clean request flag is set infrequently if the difference between the two is large. The task of cleaning the cache is split into smaller parts, reducing the impact of the additional activity on other I/O requests. If the cache does fill during peak periods of load, it is still possible to immediately clean blocks in the cache until the required amount of free space is available.

To determine the effect of each eviction policy on cache performance, we test each eviction policy in conjunction with each replacement policy in trace-based simula-

tion in Chapter 7.

3.3 Conclusions

In this chapter we examined the basis for three major cache management algorithms that have already been used by either another research project or commercial product. The least recently used (LRU) cache replacement algorithm uses a stack model and temporal locality to write out only those disk blocks that are likely to not be written again in the near future. The STF or shortest access time first algorithm attempts to reduce the inefficiency of writes by ensuring that the writes that have the minimum overall access time for the disk head are written out first. The largest segment per track or LST replacement algorithm creates more efficient writes by making the largest write first reducing the ratio of byte per second of seek time.

Each of these algorithms performs poorly with certain types of write workloads. The LRU algorithm may free blocks in small increments slowly because of long seek times, forcing write stalls. The LST algorithm may clean data from the cache that will be overwritten in the near future. The STF algorithm may suffer from both these problems. This suggests that more complex types of cache replacement policies need to be found which provide better performance in general.

These techniques will be tested in simulation and compared using different metrics in Chapter 7. Their strengths and weaknesses in practice can then be accessed. They will also be used for base line comparison with a new stack model-based policy developed in Chapter 4.

Chapter 4

A Stack Model-Based Replacement Algorithm

Any cache replacement policy must control two things, namely which entities to expel from the cache (the so-called *victims*) and when to expel them. The latter is very important when the processes accessing the storage cannot be delayed. In this case, any write occurring when the cache is full will stall and must wait while victims are being cleaned to the disk. If the selection of these victims is not performed carefully, blocks recently written into the cache will be flushed to disk. Once flushed to disk, the cleaned blocks can be reused and overwritten as additional writes are made. If overwritten, the data from victim blocks will not be present in the cache even though temporal locality dictates that those blocks are the most likely to be accessed again.

In the previous chapter, we described three basic cache replacement algorithms: least recently used (LRU), shortest access time first (STF), and largest segment per track

(LST). Some simple mathematical modeling shows how each of them can potentially improve cache performance by making fewer, shorter, or more efficient writes to disk when cleaning the cache compared to a first come, first served policy. Each of these cache replacement algorithms also have their respective problems however, because they only attempted to improve one metric of cache performance.

To illustrate this problem, consider an LRU managed non-volatile write cache such that the written blocks are spread widely across the disk in large number of contiguous extents which are mostly small in size. The LRU cache replacement algorithm attempts improve cache performance by keeping the most recently referenced blocks in the cache using the assumption that these blocks will be referenced again soon. This keeps the most recently written extents in the cache for a longer period of time to increase the number of overwrites in the cache and reduce the number of blocks to be cleaned from the cache. Because many of the extents to be written to disk are small, cleaning the cache without regard to differences in extent size and access time is inefficient due to wasted bandwidth. Cleaning in this way also may force additional write stalls if cache space is cleaned in small increments.

In this chapter, we address some of the deficiencies of simple algorithms like LRU, STF, and LST. We do this first by characterizing the properties of what we believe is a common I/O workload for disk and presenting some trace-based tests showing that this workload is valid. We then use this information to develop a new hybrid cache replacement algorithm that is well-suited for use with this workload [23].

4.1 A File System-Based Disk Workload

To consider the general I/O workload for a disk, we must first examine some of the common properties of the file systems exhibit when accessing disk. First, file systems read and write disk blocks in groups because of the cost of I/O operations. Each operation incurs a penalty due to seek time and rotational delay. This makes single blocks in the cache very expensive to read or modify; each requires a separate I/O operation. Working with groups of blocks in contiguous segments amortizes the cost of a single I/O operation across several blocks.

Next, we must consider interdependencies between groups of blocks and the relative frequency of access. A file system produces a structured layout of data on the disk. Accesses to this structure will have dependencies. For example, changes in file size in the Berkeley Fast File System [33] will not only produce accesses to the blocks holding the data, but also to i-nodes describing the files. Files are also opened, modified, and closed by programs during specific program runs or in regular bursts for program daemons that provide system services. Accesses of interdependent blocks therefore can generally be expected to occur close together in time during short bursts or regularly across long periods, depending of the behavior of the process modifying the data. A file system may also keep copies of some frequently used structures in memory that must be regularly written to disk to create checkpoints. Therefore, we propose a model of cache behavior that divides I/O operations into *hot* and *cold* groups. Blocks in the hot group are regularly and frequently updated, and, those in the cold group are created, modified a few times, and then not touched again for a relatively long time.

To illustrate the effect of hot and cold groups on cache performance, consider the simple independent reference model used to illustrate the LRU replacement algorithm in Chapter 3. In this model, there are two populations of blocks, A and B, with different uniformly distributed reference probabilities such that blocks from one population are more likely to arrive than the other. If we assume that blocks in the smaller of the two groups, A, have the higher reference probability, κ, then this group will represent the hot blocks frequently used by the file system itself or by long running programs. The remaining blocks in group B represent blocks used in file system operations where a file is created and / or modified, and then not touched for some relatively long period of time with reference probability λ. The expected number of blocks from A that reside in the cache is

$$E[X] = (\alpha m a)/(a\alpha + b) \tag{4.1}$$

where m is the size of the cache, a is the number of more frequently accessed blocks, b is the corresponding number of less frequently accessed blocks, and α is the ratio of κ to λ.

From Equation 4.1, we see that all the blocks from A are expected to be in the cache when cache size is sufficiently large because the number of blocks in the cache from A increases as the cache size increases. Since the cache hit rate depends on the sum of the reference probabilities for all the blocks in the cache and κ is greater than λ, hit rate will grow quickly as more blocks from A are expected in the cache. Once all of the blocks from A are expected to be in the cache, cache hit rate will increase because only because additional blocks from B are expected to be in the cache. Because λ is smaller than κ, these increases in cache hit rate will be relatively small. This produces a knee in a plot of

hit rate to cache size, which is precisely the behavior seen in trace-based simulations as shown in Figure 4.1.

This simple model of cache behavior treats I/O references as single blocks. In reality, I/O requests deal with groups of contiguous blocks, which we call segments. To simplify the eventual presentation of our policy, we define a segment as a set of contiguous blocks located on the same track. By using with a track-based approach to grouping segments, cost is associated with one seek of the disk head and one rotation of a disk platter. This has the advantage of making the cost of writing an individual block inversely proportional to the size of the segment to which it belongs.

More formally, our new replacement policy is based on the following three observations about disk I/O workload:

1. Writes to blocks in the cache exhibit spatial locality: blocks with contiguous disk addresses tend to be accessed together in segments,

2. Writes to blocks in the cache also exhibit temporal locality: the probability that a block in the cache will be accessed again is a decreasing function of the time interval elapsed since it was accessed last, and

3. The curve representing the hit ratio of the cache as a function of its size exhibits a knee: once a given minimum cache size is reached, further increases of its size lead to much smaller increases in the hit ratio (as seen in data from our trace-based simulations shown in Figure 4.1).

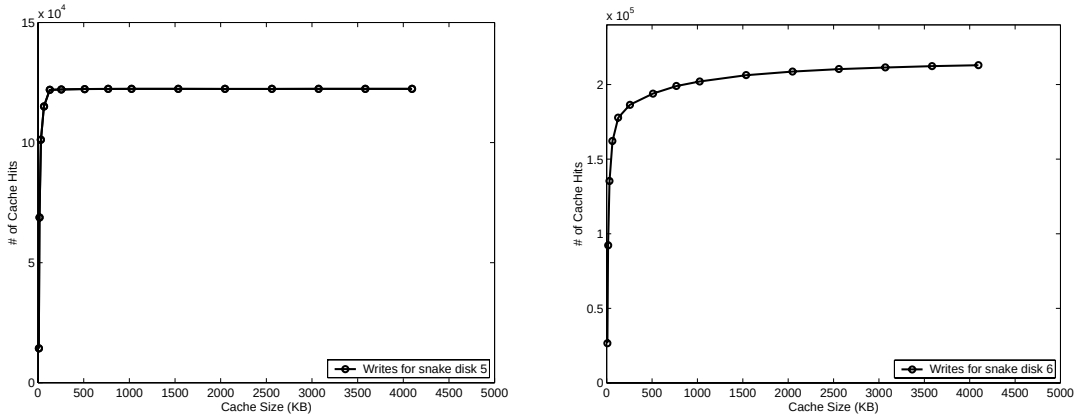


Figure 4.1: Simulated hit ratios as non-volatile write cache sizes increase for the two disks used in our experiments.

4.2 Cache Behavior

Given a cache exhibiting the properties of temporal locality, spatial locality, and a diminishing benefit to hit ratio described above, consider the m segments in S residing at any given time in a write cache. Assume that they are sorted in LRU order so that segment S_1 is the most recently referenced segment. Let n_i represent the size of segment S_i expressed in blocks. If accesses to segments in the cache follow the LRU stack model, each segment has a probability β_i of being referenced next. This probability will decrease with the rank of the segment *i.e.* $i < j$ implies $\beta_i > \beta_j$. Note that β_i represents the probability that keeping segment S_i in the cache will avoid a cache miss at the next disk write.

The contribution of each block in segment S_i to the expected benefit of keeping segment S_i in the cache is given by the ratio β_i/n_i. It makes sense to keep all the segments with the highest β_i/n_i ratios in the cache because this strategy makes the most ef-

ficient use of cache space. Conversely the segment with the *minimum* β_k/n_k ratio should be expelled. The blocks of that segment have the lowest probability of avoiding a cache miss during the next write.

Using these probabilities, the segments residing in the cache are partitioned into two groups. The first group contains segments recently written into the cache; these are the most likely to be accessed again in the near future. These *hot* segments should remain in the cache. The second group contains the segments not recently accessed and therefore much less likely to be referenced again. These *cold* segments are all potential victims for the replacement policy.

These two groups of segments are identified based on the knee in the curve representing the hit ratio of the cache as a function of its size. Let s_{knee} be the size in blocks of the cache at the knee and let C_j be the sum of the sizes of the first j segments in the LRU stack ordering of all segments in the stack:

$$C_j = \sum_{i=1}^{j} n_i. \tag{4.2}$$

The hot segments are the k most recently referenced segments that could fit in a cache of size s_{knee}, that is, all segments S_i such that $i \leq k$ where k is given by:

$$\max\{j \mid j \geq 1 \text{ and } C_j \leq s_{knee}\}. \tag{4.3}$$

All cold segments are assumed to be good candidates for replacement. We infer from the hit ratio curve that the β_i/n_i ratios for cold segments differ very little from each other. We would expect to see a greater increase in hit rate where the hit rate is nearly constant past the knee otherwise. Therefore the most efficient choice is to clean

the largest segment in the cold region. This cleans the most cache blocks for the cost of one disk seek and at most one rotation of the disk platter.

In Chapter 3, we discussed how a write behind replacement policy that never expels segments until the cache is full can often cause writes to stall for lack of available space in the cache. We also discussed how this can be avoided by setting an upper threshold on the number of dirty blocks in the cache to force block replacement to begin. By using a threshold to maintain some clean space, say 10 percent of the cache space, the cache is able to absorb short-term bursts of write activity and prevent stalls. Our cache replacement policy then has two thresholds: one to determine when replacement should begin in order to keep a minimum amount of clean space, and one to determine when it ends based on the location of the knee.

The algorithm to select the segment to be expelled can thus be summarized as follows:

1. Find s_{knee} the size of the cache for x-value of the knee of the hit ratio curve.

2. Order all segments in the cache by the last time they were accessed: segment S_1 is the most recently accessed segment.

3. Compute successive $C_j = \sum_{i=1}^{j} n_i$ for all $C_j \leq s_{knee}$.

4. Let $k = \max\{j \mid j \geq 1 \text{ and } C_j \leq s_{knee}\}$.

5. When 90 percent of the cache is full of dirty pages, expel the segment S_v such that S_v has $\max\{n_i \mid i > k\}$ until $S_v = S_k$.

4.3 Conclusions

In this chapter, we discussed the shortcomings of simple cache management algorithms like LRU, STF, LST. These algorithms only attempt to improve one metric of cache performance, such as seek time, write size, or number of writes. This can lead to situations where the cache performs well according to one metric but not others, creating poor overall performance. For example, a management algorithm may effectively reduce the amount of data written to disk but forces an unacceptable number of stalled writes because the amount of clean space in the cache is small. We believe that this is true for a common I/O workload where there are frequently referenced hot blocks on the disk and cold blocks that will be created and then updated infrequently.

To solve this problem, we proposed a new segment-based cache replacement algorithm that exploits both temporal locality and write size to make more efficient writes of cache stale data during cleaning. To do this, the cache is divided into hot and cold regions based on the ratio of time since last access to segment size. The hot region contains the most recently written dirty segments. Segments move to the cold region when the ratio of time since last access to segment size increases past a threshold value. Once in the cold region, segments become candidates for cache cleaning. Segments are written to disk in track based groups according to the amount of dirty data per track. This cache replacement policy will be tested in trace-simulation in Chapter 7.

Chapter 5

Idle detection

A practical problem with non-volatile write caches is that cleaning the cache causes read events to wait while data is written from the cache to disk. The thresholds that make such caches more effective by reducing disk write activity and stalled writes also cause read delays because the thresholds causes cache cleaning to occur in bursts. These bursts occur without any knowledge of recent disk utilization and cleaning may begin during a period of high read activity for the disk. The writes used to clean the cache create congestion and increase read response time by forcing reads to wait for service by the disk while the writes are performed [9]. They also increase total disk seek time by eliminating the benefit of any spatial locality in the read request stream [30].

One possible way to ameliorate these read response increases is to delay the cleaning of the cache until the disk is not servicing a read request and unlikely to service another for some time. In this state, the disk is considered to be *idle*. If cleaning is performed while the disk is idle, no read requests must wait for additional time for

service while the cache is cleaned. This also has the potential benefit that additional overwrites may occur in the cache by forcing a delay in cleaning dirty blocks to disk. These benefits are not without cost: the cache remains full of dirty blocks and stalls occur as write requests arrive that do not overwrite blocks in the cache. Thus the overall effectiveness of this idle cleaning approach depends on finding a good balance between increase stalled writes the reduction in read response time.

In this chapter, we evaluate the relative merits and deficiencies of idle cleaning. We will focus on a very simple constant-time idle detection technique that waits a fixed amount of time after each request is complete. If no read requests requiring disk access arrive during this time, the disk is considered to be idle and cleaning can begin. If a read arrives, cleaning is interrupted and the read is serviced. The disk must then remain unused for the fixed idle detection period before cleaning can resume. To facilitate our analysis, we will develop two mathematical models: one describing how often the cache must clean data to disk, and another for calculating read and write response time during idle detection and cache cleaning. The results of this analytical analysis will then be compared with simulation results in Chapter 7 of this dissertation.

5.1 Modeling cache behavior

Consider the following model of an I/O subsystem with a non-volatile LRU write cache: the system consists of a disk with an associated event queue and the non-volatile write cache itself (see Figure 5.1). I/O requests are made by the computer's process population and arrive at the cache in two separate streams of read and writes.

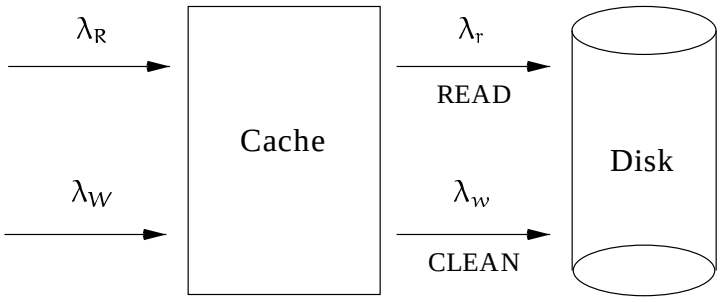


Figure 5.1: Modeled request streams into and out of the disk cache.

Each stream has a fixed associated arrival rate; reads arrive at a rate λ_R and writes at λ_W. Each request stream has its own separate hit probability for the cache, h_r and h_w. The I/O events are filtered by the cache based on these hit probabilities and then passed to the event queue of the disk.

For modeling purposes, h_r and h_w are assumed to be independent of each other and calculated differently. The read hit probability, h_r, is assumed to be a fixed constant representing the probability that a recently updated block is found in the write cache such that $0 \leq h_r \leq 1$. The write hit probability, h_w, varies with the contents of the write cache. The cache will contain at most n blocks where each block has an associated hit probability p_i and the blocks are ordered by increasing hit probability ($p_i \leq p_{i+1} : 1 \leq i \leq n-1$). The probability that the next written block will be in the cache can be expressed as:

$$h_w(n) = \sum_{i=1}^{n} p_i.$$

The events that arrive at the disk are also separated into two event streams (see Figure 5.1). Reads arrive at the rate $\lambda_r = (1 - h_r)\lambda_R$.

When the cache becomes full and must be cleaned, writes are made from the

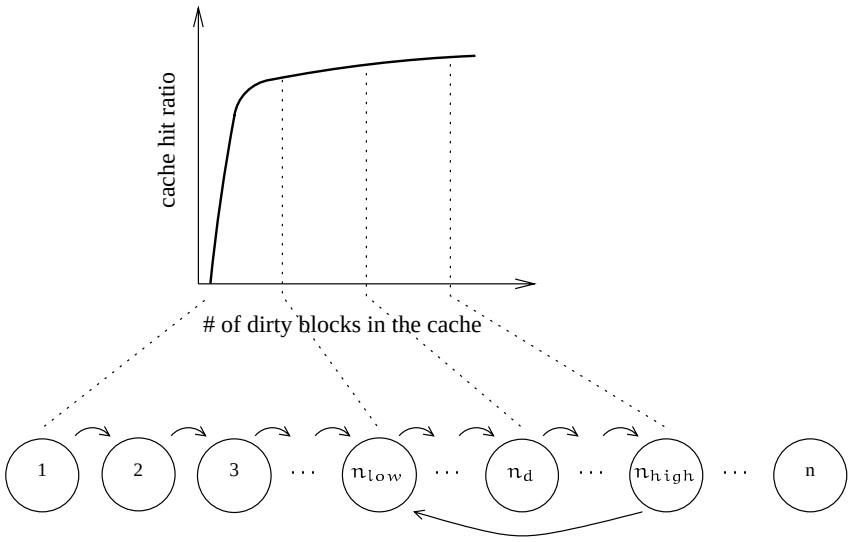


Figure 5.2: The relationships of the thresholds for the minimum and maximum number of dirty blocks in a cache of size n. At any given time the cache has n_d dirty blocks in it. After some period of time, n_d will be greater than n_{low} and then $n_{low} \leq n_d \leq n_{high}$ from then on.

cache to disk at the rate λ_w. The choice of when to clean the cache and how much must be cleaned is based on two cache parameters, n_{high} and n_{low}, such that $0 \leq n_{low} < n_{high} \leq n$ as shown in Figure 5.2. Cleaning begins when the number of dirty blocks in the cache is greater than or equal to n_{high} and ends when the number falls below n_{low}. The probability p_{clean} that a given write event will force the cache to be cleaned is:

$$p_{clean} = (1 - h_w(n_{high}))p(N = n_{high})$$

where N is the number of dirty blocks in the cache. From this, we calculate the disk write arrival rate to be $\lambda_w = p_{clean}\lambda_W$.

To evaluate the probability that there will be n_{high} blocks in the cache, consider that the cache is a Markov chain that moves between at most n different discrete states as dirty blocks are added and removed. To understand the transitions between each of

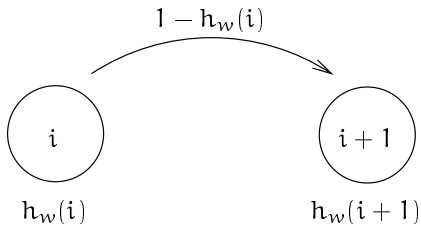


Figure 5.3: The state transition when an additional block is added to a write cache containing i blocks.

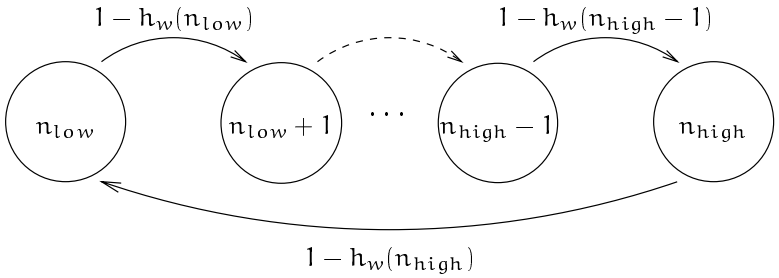


Figure 5.4: The state transitions made around the n_{low} and n_{high} boundaries of the write cache.

these states, consider the change in state of the cache when one block is added to the cache shown in Figure 5.3. Before a new block is added to the cache, there are i dirty blocks in the cache with an associated hit probability of $h_w(i)$. A state transition only occurs if the number of dirty blocks in the cache increases because the arriving block isn't in the cache or is in the cache and clean. The probability of a state transition is therefore $1 - h_w(i)$ and the hit probability of the $i + 1$ state is $h_w(i + 1) = h_w(i) + p_i$. Looking at the boundary conditions at n_{low} and n_{high}, cleaning the cache introduces a cycle into the overall state transition graph, shown in Figure 5.4.

Ignoring the initial state transitions where the number of dirty pages is below

n_{low}, the state transition graph can be translated into the following probability matrix:

$$P = \begin{bmatrix} h_w(n_{low}) & 1-h_w(n_{low}) & 0 & \cdots & 0 & 0 \\ 0 & h_w(n_{low}+1) & 1-h_w(n_{low}+1) & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & h_w(n_{high}-1) & 1-h_w(n_{high}-1) \\ 1-h_w(n_{high}) & 0 & 0 & \cdots & 0 & h_w(n_{high}) \end{bmatrix}.$$

It is not difficult to show that this matrix is irreducible because the there is always a positive probability of the cache moving between two states in an arbitrary number of steps. Likewise, each of the states also communicate because the state transition probabilities are non-zero. If the state transitions governed by $h_w(n)$ remain constant over time, the matrix is also aperiodic. If the Markov chain is irreducible and aperiodic, it must be ergodic and a set of steady state limiting probabilities must exist. This can be verified using numerical analysis as the converging values of p_{clean} show as P is multiplied by itself in Figure 5.5.

In order for the state transition probabilities between states to remain constant over time, the function $h_w(n)$ must be of the form

$$h(n+1) = h(n) + \epsilon$$

where the ϵ is approximately constant for $n_{low} \leq n < n+1 \leq n_{high}$. This means that while the blocks are written into the cache at an exponentially distributed rate, the probabilities that dirty blocks in the cache will be referenced again differ very little from each other. Were this not the case, the probability of creating a new dirty block in the cache would depend on the order in which blocks were written into the cache, the Markov chain would no longer be homogeneous, and the model would break down.

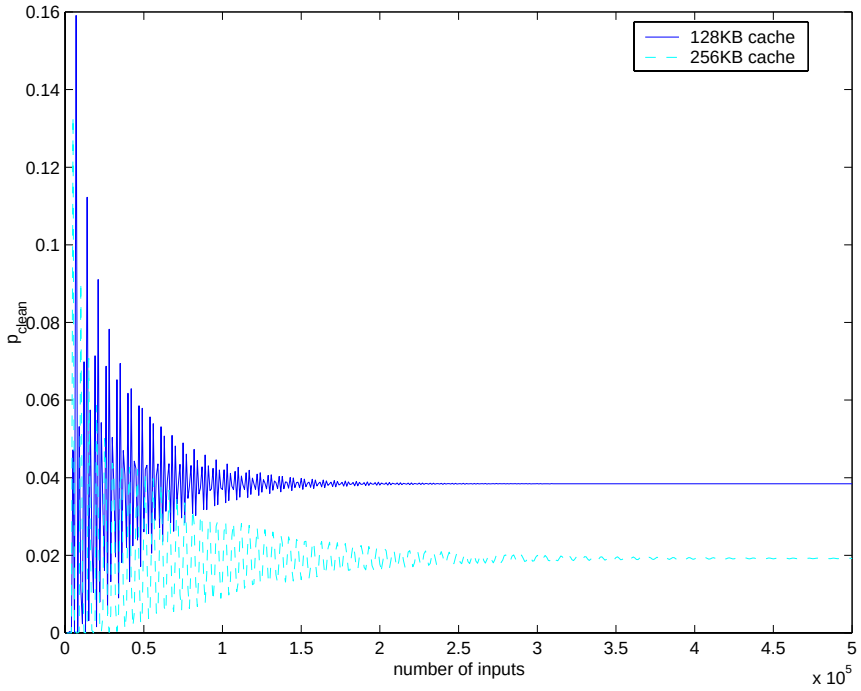


Figure 5.5: The value of p_{clean} obtained from P^1 to P^{500000} in increments of 1000 for a cache of 52 blocks with an incremental hit probability of 0.000067 every time a dirty block is added to the cache.

Fortunately, we believe that block reference probability distributions supporting a time homogeneous matrix are common. Two examples of reference distributions that create homogeneous transition matrices are the reference model described for the LRU cache in Chapter 3 and the access patterns we assume for the stack model based-algorithm in Chapter 4. In both cases, the stream of write operations consists of events referencing either a small set of blocks very frequently or a larger set infrequently. Given a cache of sufficient size, the cache will be expected to contain all of the frequently referenced data. Changes in contents of the cache will come from infrequent writes to the larger set of data where writes are assumed to be equally likely to all the blocks. Therefore, by making n_{low} sufficiently large to include all of the frequently referenced blocks in the cache, both of these examples produce homogeneous transition matrices. We will test this assertion by comparing model results with trace-based simulation in Chapter 7.

5.2 Modeling cache cleaning

The analytical model for cache cleaning we present treats each cleaning interval independently assuming that cleaning operations occur approximately t_{cycle} seconds apart where $t_{cycle} = 1/\lambda_w$ and $\lambda_w = p_{clean}\lambda_W$. The model describes a cleaning cycle that begins with a cache full of dirty blocks needing to be cleaned, followed by fixed sequence of steps. First, there is a waiting period to detect if there is an interarrival time in the read request stream larger than a pre-set constant. When such an interarrival time occurs, the disk is declared idle and the disk's event queue is allowed to drain. The contents of the cache are then cleaned to disk, followed by the service of any reads which

arrived while the writes were performed, the service of the reads that arrived while those reads were serviced and so on, until no further requests remain. After this cleaning cycle is complete, the disk system becomes an M/G/1 server that services read requests until the next time cleaning is required. The relationship between the various times can be seen in Figure 5.6. This model resembles that developed by Carson and Setia [8] for their periodic update analysis.

The first interval in the cleaning cycle, t_{detect}, is the period when the disk acts as an M/G/1 server and the cache determines if the disk is idle. The duration of the test interval, t_{idle}, is a parameter set as a part of the tuning of the cache. The actual amount of time spent waiting is a function of the arrival rate λ_r of requests. Intuitively, let X be a geometric random variable representing a series of Bernoulli trials every t_{idle} seconds where the trial is true if no reads arrive since the last trial and false otherwise. The probability p that a Bernoulli trial succeeds is the probability that no event arrives during t_{idle} seconds or

$$p = e^{-(\lambda_r t_{idle})} \frac{(\lambda_r t_{idle})^0}{0!} = e^{-\lambda_r t_{idle}}.$$

Therefore, the time to detect an idle period, t_{detect} is the product of one plus the expected value of X and t_{idle} or

$$t_{detect} = t_{idle} \left(1 + \frac{(1 - e^{-\lambda_r t_{idle}})}{e^{-\lambda_r t_{idle}}} \right)$$

with a variance of

$$\frac{(1 - e^{-\lambda_r t_{idle}}) t_{idle}}{e^{-2\lambda_r t_{idle}}}.$$

Next, any waiting reads events are allowed to drain out of the disk's queue. Assuming that the arrival pattern of reads is relatively independent of the cleaning activity of the disk, the system is operating as an M/G/1 server when cleaning commences. The interval t_r is the time required to service reads that are in the queue. Since reads and cleaning activity are assumed to be independent, the cleaning cycle commences at a random time as far as read arrivals and service completions are concerned. Therefore, the value of t_r is at most the "virtual waiting time" of a random arrival from Pollaczek's formula:

$$W_q = \frac{\rho_r W_s (1 + C_s^2)}{2(1 - \rho_r)} = \frac{\lambda_r E[s^2]}{2(1 - \rho_r)}$$

where $\rho_r \equiv \lambda_r W_s$ is a measure of the probability that the disk is servicing a read. The squared coefficient of variation term C_s^2 includes a measure of the random variability in each disk service operation.

The fact that idle detection occurs before the read queue is allowed to drain affects the length of t_r, however. Because no read request has arrived for t_{idle} seconds in order for the disk to be considered idle, the draining of the disk's queue actually begins at the start of the idle period. Therefore, the complete expression t_r is:

$$t_r = \begin{cases} W_q - t_{idle} & W_q > t_{idle} \\ 0 & W_q \leq t_{idle}. \end{cases}$$

The third interval of length t_w is the time required to complete writes that are needed to clean the cache. Given that every cleaning cycle begins t_{cycle} seconds apart on average, the number of writes to be cleaned is the product of the number of writes in the

cache and the average service time for each, or

$$t_w = (n_{high} - n_{low})W_s.$$

A series of read intervals now follows before the system returns to an M/G/1 queueing state. These intervals are of average length $t_i, 1 \leq i \leq \infty$. The first read interval of length t_1 is the time to complete read requests that arrive during t_w, so $t_1 = t_w \rho_r$. In general, t_i is the time required to service read requests that arrive during $i-1$. Thus, $t_i = \rho_r t_{i-1} = t_0(\rho_r)^i$. The total length of the active part of the cleaning cycle is

$$t_{active} = \sum_{i=0}^{\infty} t_i = \frac{t_0}{(1-\rho_r)}.$$

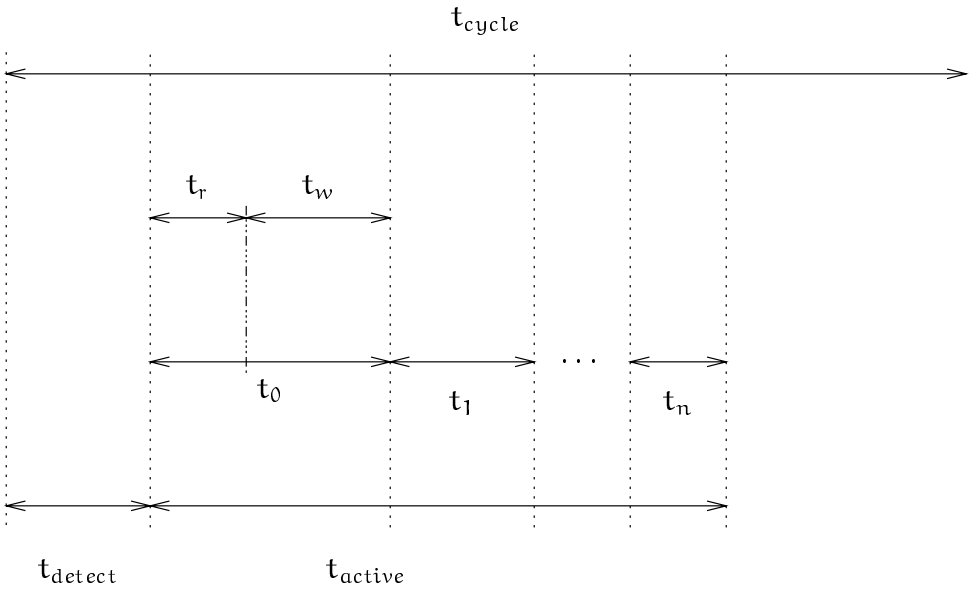


Figure 5.6: Relative times in a cache cleaning cycle of a non-volatile cache.

The average response time for reads can now be calculated for each of the intervals in which reads are serviced. This is identical to an expression developed by Carson and Setia [8]. Let R_1 be the the average read response time during period t_1. This is equal

to the average amount of time spent waiting during the preceding interval plus the average time to be serviced during the current interval. The average waiting time during the interval when the request arrived is the mean residual amount of time left for a random arrival, $t_0/2 + \sigma_{t_0}^2/2t_0$, where $\sigma_{t_0}^2$ is the variance of t_0. The average time for service in the system includes both queuing and service time. Let $A[k,t]$ be the probability that k read operations arrive during a period of t seconds. Since the read arrival process is Poisson, this is

$$A[k,t] = e^{-\lambda_r t}\frac{(\lambda_r t)^k}{k!}.$$

Given that there are k arrivals in the system during time t, the average waiting time $\overline{W}[k,t_0]$ is

$$\overline{W}[k,t_0] = \sum_{i=0}^{k}\frac{iW_s}{k}.$$

Therefore, the average time in the system during the interval t_1 is:

$$\begin{aligned}
\sum_{k=1}^{\infty}\overline{W}[k,t_0]A[k,t_0] &= \sum_{k=1}^{\infty}\sum_{i=0}^{k}\frac{iW_s}{k}A[k,t_0] \\
&= \sum_{k=1}^{\infty}\frac{W_s}{k}\sum_{i=0}^{k}iA[k,t_0] \\
&= \sum_{k=1}^{\infty}\frac{W_s}{k}\frac{k(k+1)}{2}A[k,t_0] \\
&= \frac{W_s}{2}\sum_{k=1}^{\infty}kA[k,t_0] + \frac{W_s}{2}\sum_{k=1}^{\infty}A[k,t_0] \\
&= \frac{W_s}{2}(\lambda_r t_0 + 1 - A[0,t_0]) \\
&= \frac{1}{2}(t_1 + W_s(1 - e^{-\lambda_r t_0})).
\end{aligned}$$

The value of this last expression lies between $t_1/2$ and $(t_1 + W_s)/2$. Assuming the lower value results in closed form expression with a slightly more optimistic prediction of response time. Carson and Setia report that assuming either bound makes little difference in the final response time calculation. Using this bound provides the following average response time requests serviced during t_1:

$$R_1 = \frac{1}{2}\left(t_0 + \frac{\sigma_{t_0}^2}{t_0} + t_1\right). \tag{5.1}$$

The variance of the time t_0 in Equation 5.1 is the combination of the variances of t_r, and t_w. The variance of t_r is the variance of the queue waiting time for an $M/G/1$ system when $t_r > 0$ or

$$\frac{\lambda_r^2 E^2[s^2]}{4(1-\rho_r)^2} + \frac{\lambda_r E[s^3]}{3(1-\rho_r)}$$

The variance of the cache cleaning period is expressed as the variance of of a random number of random variables, or $E[s^2](n_{high} - n_{low})$. Therefore the overall variance for the t_0 is

$$\sigma_{t_0}^2 = \frac{\lambda_r^2 E^2[s^2]}{4(1-\rho_r)^2} + \frac{\lambda_r E[s^3]}{3(1-\rho_r)} + (n_{high} - n_{low})E[s^2]. \tag{5.2}$$

During the remaining read intervals where $i \geq 2$, the average response time is calculated in a similar way. During the interval $i-1$, a random number of read operations arrive. The variance of the aggregate time to service those reads is the variance of the sum of a random number of variables, or:

$$\sigma_{t_{i-1}}^2 = \lambda_r t_{i-2}\sigma_s^2 + W_s^2\lambda_r t_{i-2} = E[s^2]\lambda_r t_0 \rho_r^{i-2}.$$

The response time for requests serviced during interval $i \geq 2$ is the sum of the waiting time during the $i-1$ interval and the waiting and service times during interval i, or

$$R_i = \frac{t_{i-1}}{2} + \frac{\sigma^2_{t_{i-1}}}{2t_{i-1}} + \frac{t_i}{2} = \frac{1}{2}\left(t_{i-1} + \frac{E[s^2]\lambda_r}{\rho_r} + t_i\right).$$

The probability that a read request arrived during interval $i-1$, and is serviced during interval i is, on average, $t_{i-1}/p_{miss}\lambda_W$. The contribution to the average read response time of the read requests that arrive during the cleaning interval is

$$\sum_{i=1}^{\infty} R_i Pr(i) = \frac{t_{active}}{t_{cycle}}\frac{1}{2}\left(t_0 + \lambda_r E[s^2] + \frac{\sigma^2_{t_0}}{t_{active}}\right).$$

After this cleaning activity is complete, read requests are serviced by a simple M/G/1 server which had an average response time of $R = W_s + \lambda_r E[s^2]/(2(1-\rho_r))$. The probability that a read request arrives during the "non-active" portion of the interval is $(t_{cycle} - t_{active})/t_{cycle}$. Thus, the overall response time for reads during a cleaning interval is

$$\begin{aligned} R_{read} = {} & \frac{t_{active}}{t_{cycle}}\frac{1}{2}\left(t_0 + \lambda_r E[s^2] + \frac{\sigma^2_{t_0}}{t_{active}}\right) + \\ & \frac{(t_{cycle} - t_{active})}{t_{cycle}}\left(W_s + \frac{\lambda_r E[s^2]}{2(1-\rho_r)}\right). \end{aligned} \tag{5.3}$$

Calculating the response time for writes is a much simpler problem. The write cache absorbs writes with negligible response time (assume zero response time) whenever a write does not stall. The only time when stalled writes can occur during the cleaning cycle is during t_{detect} and t_r; writes can immediately be moved into the cache during t_w and after. The average waiting time for a write during is the residual amount of time for a random arrival during t_{detect} and t_r or

$$\frac{t_{detect} + t_r}{2} + \frac{\sigma^2_{t_{detect}}}{2t_{detect}} + \frac{\sigma^2_{t_r}}{2t_r}$$

where $\sigma^2_{t_{detect}}$ and $\sigma^2_{t_r}$ are the variances of t_{detect} and t_r respectively. Since all of these expressions are already known, the response time for a write is

$$R_{write} = \frac{t_{detect} + t_r}{t_{cycle}} \left(\frac{t_{detect} + t_r}{2} + \frac{\sigma^2_{t_{detect}}}{2t_{detect}} + \frac{\sigma^2_{t_r}}{2t_r} \right). \tag{5.4}$$

To show how the read and write response times are affected by increasing idle detection times, we performed tests with a hypothetical disk and write cache. Rather than obtain disk moment information for the disks in our traces, we used the moment information suggested by Vongsathorn and Carson [52] and used by Carson and Setia [8]. The values we used for cache size and p_{clean} are those suggested by the performance of a 128K cache with snake disk 5 with the model described in the previous section. We used mean arrival rates of reads and writes similar in relative proportion to the rates found in snake disk 5 trace. The parameters are summarized in Table 5.1. We perform these tests to show the relative effect of increasing idle detection times, and to gain insight into the strengths and weaknesses of our model.

Our results (shown in Figure 5.7) show that idle detection may do little to improve read response time, and increase write response time. This indicates that the bursts of writes produced by the cache do not often encounter the disk in a busy state. The number of reads queuing for service when a cleaning commences is small, and therefore the benefit of waiting for that queue to empty does little to improve overall read response time. This observation will be compared with trends in the response time results from trace-based simulations in Chapter 7.

Our tests with the model show a major flaw with respect the relative sizes of t_{detect} and t_{cycle} that limit the range of t_{idle} values we could test. An increasing value

Disk mean service time (ms)	15
Standard deviation (ms)	8
Second moment (s^2)	0.0012
Third moment (s^3)	0.000006
Coefficient of variation	1 / 3
p_{clean}	0.03
Cache size (blocks)	52
λ_R	12
λ_W	8

Table 5.1: Disk and cache model parameters for read and write response time tests.

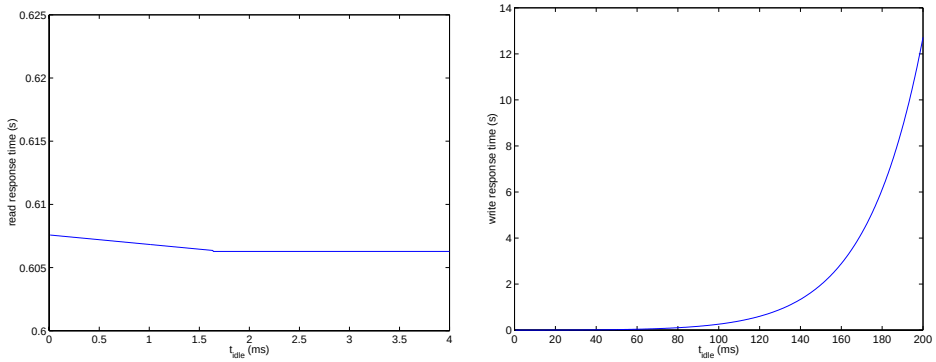


Figure 5.7: Read and write response times for a hypothetical disk with a non-volatile write cache as the idle detection t_{idle} is increased.

of t_{idle} quickly produced values of t_{detect} that were larger than t_{cycle}. This breaks the assumption that the cleaning cycle is at most t_{cycle} in length. This does not directly affect the write response time, but calls the read response time results into question because of the method used to calculate R_{read}.

The model also does not accurately take the effects of stalled writes into account. The non-volatile write cache used in simulation produces stalled writes during the detection period if it is excessively long. The model makes the simplifying assumption that the number of stalls are few as long as idle detection times are relatively short and assumes that all waiting writes can be written into the cache during t_w. This simplifies the analysis of the model significantly, but it also reduces the amount of time required to find the disk idle. We believe that this still gives a decent approximation of disk behavior for small idle detection times, however.

5.3 Conclusions

In this chapter, we examined idle detection as a solution to problems created by non-volatile cache cleaning. When a non-volatile write cache becomes full it begins writing dirty blocks to disk in bulk. The batches of cleaning writes arrive at the disk in bursts, causing read response time to increase while read operations to wait for service. To reduce read response time, we investigate delaying the cleaning of the cache until such time when the disk is no longer servicing a read and unlikely to service another in the near future. This keeps the cache full for a period of time and increases the possibility of stalled writes, but reduces the waiting times that cleaning writes cause arriving reads.

To develop a model for estimating read and write response time when using a simple constant time idle detector, we first use a Markov model to estimate how often the cache must be cleaned. We then break down cache cleaning into time periods: the time to detect if the disk idle, the time to let any pending reads finish, the time to write from the cache, and the time to perform any reads that arrived while the cache was being cleaned. We develop mathematical expressions for each length of time based on the assumption that the disk performs as an M/G/1 queue.

Based on tests with our model, we note that the time needed for any pending reads to be serviced decreases only slightly as the idle detection time increases. Small decreases in queue drain time change produce few changes in the number of reads serviced in the later in the cycle very and do little to decrease the overall read response time. At the same time, longer idle detection times increase the probability of a write arrives and a stall is forced. This results in an increase in write response time. These conclusions will be compared to cache behavior in trace-based simulation in Chapter 7.

Chapter 6

Simulation environment

In order to create a test bed for evaluating different cache management schemes, we implemented our own models of the HP2200C and the HP97560 disks. Each disk model is implemented in C++, and designed to support multiple disks connected to one or more data buses within the same simulation. Two different types of buses can currently be used: the HP97560 can be attached to a SCSI 2 bus, and the HP2200C to a HP-IB (IEEE-488) bus. The simulation currently uses the Sim++ event simulation package [17], but can be easily ported to another environment. The design is also meant to be extensible and support additional disk models and bus types over time.

6.1 Implementation

The models that we used to simulate both the HP97560 and the HP2200C are based on three major sources. A paper by Ruemmler and Wilkes [44] described the basic mechanisms and simulation parameters needed to build an effective disk simulator. A

document by Kotz, Toh, and Radhakrishnan [31] provided routine specifications, simulation times, and core data structures that we adapted for our simulator. The SRTheavy C++ code library by Ruemmler and Wilkes provided additional hints about the behavior of each disk and access to file system traces.

While each of these sources helped us create accurate models, we needed to make a few assumptions to make each simulator complete:

- The size of read and write requests are limited to the size of the buffers on each the disks of each type. Analysis of trace data showed no I/O events larger than the buffer cache size of the HP97560. For the HP2200C, there were a small number of calls for swap space which were larger than the buffer on the disk. Since the number of these calls was small, they were filtered out of the simulation.

- Read and write fence sizes were equal and corresponded to the values listed by Ruemmler and Wilkes as “Read fence size”.

- Cylinder skew and cylinder seek time subsumes track skew and head switch costs when crossing cylinder boundaries for both disks.

- Data provided in the SRTheavy library for the head switch time of the HP2200C ($4.5ms$) conflicted with the time in the article by the same authors ($2.5ms$). While calculations show that $4.5ms$ is a better offset for the skew factor of the HP2200C, we decided to use the value published in the article.

- Two guesses were used for the delays needed to get the use of each type of data bus. A value of $50\mu s$ was used for the time needed the SCSI bus; this was the value

used by Kotz *et al.*. A value of $400\mu s$ was used for the for the HP-IB bus. The SRTheavy library provided a bus delay time of $4ms$ for the HP-IB bus, but testing showed that it didn't work well with our simulation. John Wilkes said that this reflected driver overhead as well as bus acquisition time, and, the driver wasn't modeled by this simulator.

- The sizes for the "Disk Request" (10 bytes) and "Done Messages" (1 byte) for both disks were the same as Kotz *et al.* used for their HP97560 simulator.

- Controller overhead values were assumed to be the same as those published in the article by Reummler and Wilkes.

- Neither disk has a segmented buffer cache, command queuing, or multiple zones (though the code will support multiple zones). Evidence in the traces showed that the HP2200C did shortest seek time first reordering of write operations. Our HP2200C simulator did not reorder operations in this way.

- A head settling time of $1.0ms$ for the HP97560 and $1.5ms$ for the HP2200C was included for write requests.

- Transfers of read data from the buffer cache to the bus for the HP2200C commenced when the head crossed a track or cylinder boundary. Reummler and Wilkes did not describe this behavior in their article, but it did appear in STheavy code.

6.2 Structure of the simulation

The simulation model consists of two Sim++ Facility objects, two major loops of simulation events, and several supporting member functions. The disk's Facility objects provide serialized access to the disk and to the bus to which the disk is connected. One event loop functions as the disk's DMA engine, getting or filling buffer memory, performing bus transfers, and managing the bus if the buffer is full or data is not ready. The second loop simulates the disk's mechanism by performing seek, settle, and rotation operations, getting or filling the disk's buffer memory, and requesting DMA transfers if the disk's buffer is full or not ready. The supporting member functions send and receive control messages and calculate delays associated with the disk's mechanism.

A disk I/O request begins as follows: When the user calls Request_Disk, a request is made for the Facility representing the disk. After this request is successfully made, a request is made for the Facility representing the bus. When that second request is complete, the simulation waits the amount of time needed to send a command message to the disk's Controller. After that time has elapsed, the bus Facility is released and the Controller starts. The Controller determines if the new request is a read or a write, and handles each operation accordingly. If the new operation is a READ, the Controller schedules the start of the disk's mechanism event loop to move the simulated disk head to the appropriate location to begin the read. If the operation is a WRITE, the Controller starts both event loops to write the data to the disk's buffer and move the head to start the write.

Since the HP97560 supports buffer caching, read ahead, and immediate report-

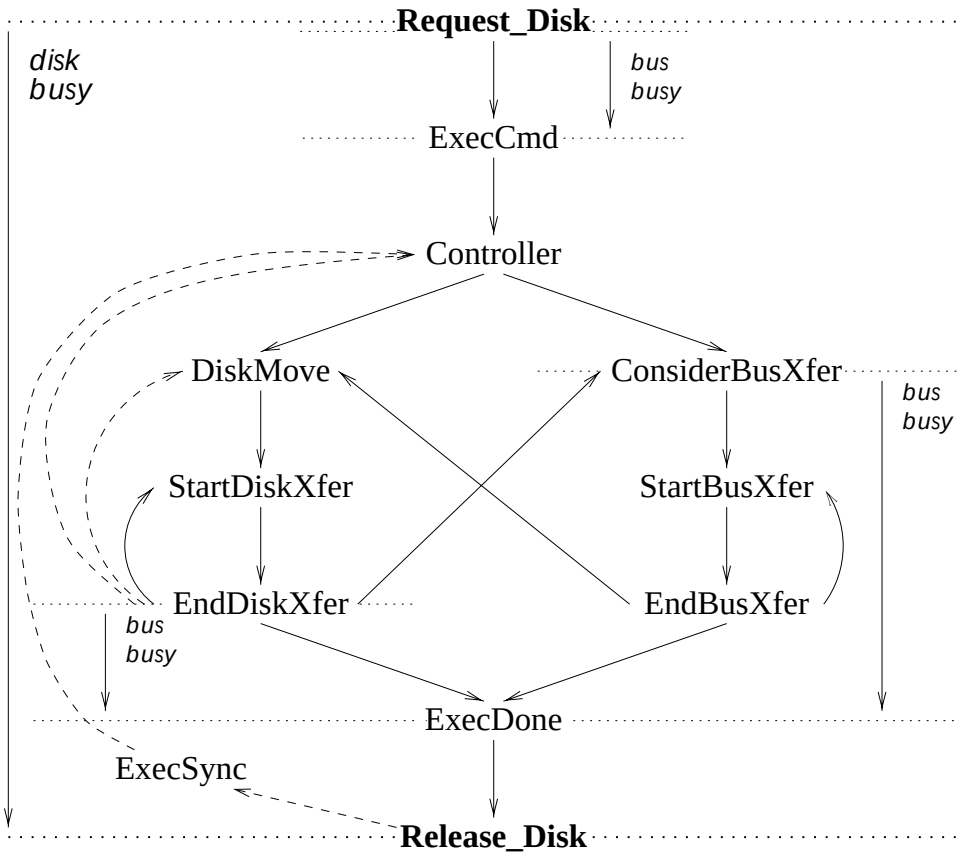


Figure 6.1: Event graph of our simulation. Bold names denote class member functions. Other names represent simulation event routines, which are implemented as C++ friend functions. Solid lines represent event transitions common to all disks. Dashed lines represent transitions due to read ahead, immediate reporting, and sync-ing operations. Arrows and dotted lines represent Facility objects that are busy for the duration of event.

ing, the Controller for the HP97560 must perform a few additional functions. As each new READ request arrives, it checks for a buffer cache hit. If there is a read hit, it activates the disk's DMA event loop to transfer the data across the bus and flushes the buffer cache to the first requested sector. If there is a read miss or a write, the whole cache is flushed.

In the disk mechanism transfer loop, requests are processed on a sector per loop iteration basis using logical sector numbers. The loop begins when DiskMove calculates the time needed to move the disk head to the beginning of first sector of the new I/O request. DiskMove then schedules a StartDiskXfer to occur when the move is complete. StartDiskXfer updates the state of the disk's buffer to show the read or write of the next sector has started, and schedules an EndDiskXfer for when the sector I/O is finished. EndDiskXfer sets the current state of the disk's buffer to the values previously set by StartDiskXfer, and schedules a StartDiskXfer if further head movement is required. EndDiskXfer also attempts to start the DMA transfer loop in case any new data needs to be transferred to or from the buffer. If the data transfer is complete and the current operation is write, EndDiskXfer schedules an ExecDone to notify the host that the write is complete.

The DMA transfer loop works by transferring bytes from the bus to the buffer one sector at a time. ConsiderBusXfer decides if a DMA transfer loop should be started. Whether or not the loop is started depends on a set of circumstances that varies with the type of operation and the type of disk. Some factors that need to be considered are:

- if the bus is already in use,

- if the Controller indicates that data needs to be transferred on the bus,
- if the last sector already has been written to the buffer (for writes), or read from the buffer (for reads),
- if the buffer is sufficiently full (for reads) or sufficiently empty (for writes) to merit a DMA transfer, or
- if the disk begins read DMA immediately or on a track crossing.

If the appropriate conditions are met, then ConsiderBusXfer requests the disk's bus Facility and schedules a StartBusXfer for when that request is successful.

StartBusXfer schedules an EndBusXfer event after a sector of disk is transferred over the bus. EndBusXfer continues the loop by scheduling a StartBusXfer if the current DMA transfer is incomplete and the cache contains non-transferred data (if reading) or the buffer is not full (if writing). If the loop is not continued, then ExecDone is scheduled to finish the current request. EndBusXfer may also start the disk transfer loop if the cache filled during a read, or emptied the cache during a write. Since the disk transfer calls ConsiderBusXfer as it iterates, the bus transfer will be restarted when conditions have changed.

Since the HP97560 prefetches reads and immediately reports writes, certain additional state transitions occur. Because the HP97560 performs read prefetching, the disk mechanism loop transfers data from the disk until the buffer is full. When the Controller is called during read prefetching and there is a cache miss or a write after a read, the mechanism transfer loop may be busy and events may be cancelled. Because the

HP95760 immediately reports writes, ExecDone will be scheduled as soon as the DMA transfer of the data to the buffer is complete. If another operation is requested before the buffer is written to disk, the data is either appended to the cache (if it is a contiguous write) or the operation must wait until the write is complete. If the operation must wait, the disk Controller is restarted after the write is finished. Setting the event token's sync flag to TRUE causes the HP97560 to wait until the write is complete before scheduling ExecDone. Execution of the I/O request finishes when Release_Disk calls ExecDone. The Facility representing the disk is then released, and execution completes.

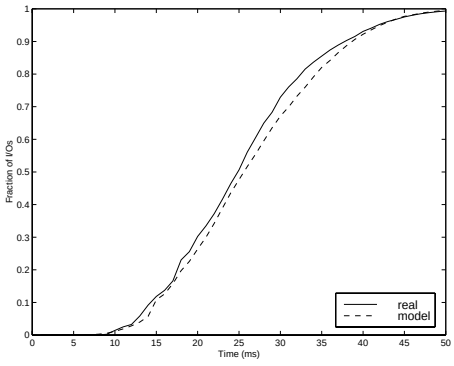
6.3 Validation

To validate our simulation models, we used trace data used by Ruemmler and Wilkes for their study. To test the HP2200C, we looked at events for disk numbers 0 and 1 from the `hplajw` trace set from 4/18/92 to 5/11/92. For the HP97560, we chose events from the `snake` traces from 4/25/92 through 4/30/92, disk numbers 5 and 6. We filtered the traces for events from the appropriate disk drives and ran them through our disk model. All writes were treated as immediate reported.

The method we use to evaluate our simulations is the same used by Ruemmler and Wilkes to test their disk simulators [44]. Simply comparing mean I/O execution times for simulated and real disks provides little information about the accuracy of the simulation model. Instead, we plot cumulative time distributions of execution times for the real and model disks, and measure the difference. The root-mean-square of the horizontal distance between the two curves is used as a figure of *demerit*, and represented

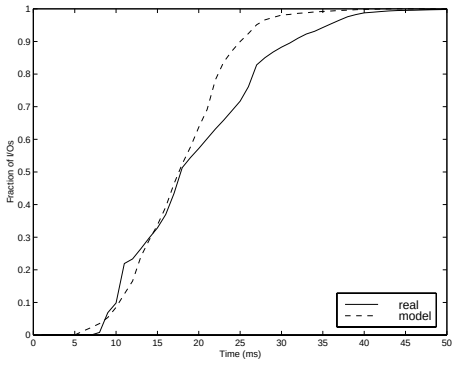
in absolute (as a difference in milliseconds) and relative terms (as a percentage of mean execution times). The real trace has a demerit of zero; it matches itself exactly.

Each set of traces produced some trace events that had access times that were extremely long. For the `snake` traces, Kotz, Toh, and Radhakrishnan consulted Chris Ruemmler about these long requests, and discovered that these long requests were most likely due to thermal recalibration. Since neither we, Kotz *et al.* , nor Ruemmler and Wilkes modeled these events, we simply discarded them. In the case of the `hplajw` traces, the trace data revealed that these events are extremely large (> 32Kb) reads and writes of swap data. Since the sizes of these events were larger than the buffer in the disk controller and sent to the disk by a unknown protocol, they were discarded as well. Both types of events were very rare, and removing them gave us a closer match.



(a) hplajw disk 0

	Mean	Demerit	
Simulation	26.39ms	1.06ms	4.0%

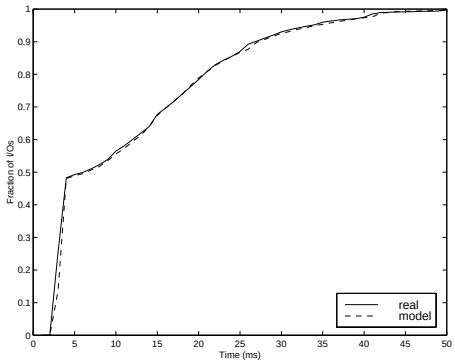


(b) hplajw disk 1

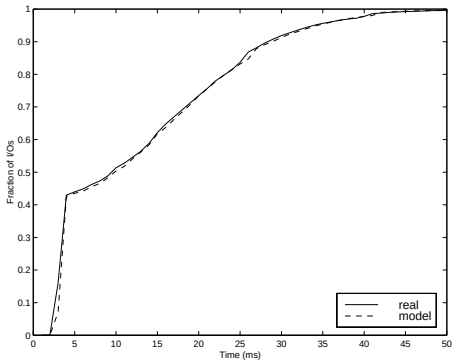
	Mean	Demerit	
Simulation	17.75ms	5.48ms	30.1%

Figure 6.2: I/O time distributions for our model of the HP2200C. Input data is taken from the `hplajw` trace set from 4/18/92 to 5/11/92.

We simulated each disk independently, and offset trace events for consecutive



(a) snake disk 5



(b) snake disk 6

	Mean	Demerit	
Simulation	11.73ms	0.88ms	7.5%

	Mean	Demerit	
Simulation	13.01ms	0.55ms	4.3%

Figure 6.3: I/O time distributions for our model of the HP97560. Input data is taken from the snake trace set from 4/25/92 to 4/30/92.

days with the number of microseconds in a day to create large continuous sets of accesses. Mean simulated access times, demerit times, and demerit percentages, as well as distributions for real and simulated disks is shown in Figures 6.2 and 6.3.

Testing showed that our simulators were accurate to 4.0–8.0% for three out of the four disks we looked at. The reason for the large figure of demerit for Disk 1 is unclear. The trace headers for the hplajw traces indicate that one unmonitored section of Disk 1 was used to write out trace information. This may be the source of the demerit.

To test the different properties of NVRAM caches, we modified the simulator to support a write cache. The cache was implemented in a way similar to the disk simulators; each type of cache was implemented as a C++ class with similar interfaces. Caches with four different management policies have been implemented: LRUcache using a Least Recently Used policy to age data from the cache, SSTcache using Shortest Seek

Time, LSTcache using the Largest Segment per Track in the cache, and StModcache using the Stack Model-based algorithm we develop in this dissertation.

To clean each cache object, we used two basic mechanisms: either a routine which explicitly cleans the cache while an incoming write waits, or a background cleaning process similar that found in many virtual memory systems. The use of each mechanism depends on the cache eviction policy (see §3.2). A write behind cache is only cleaned when space is needed for an incoming write, therefore only explicit cleaning is used. The high and low cleaning thresholds pro-actively evict sectors from the cache using both explicit and background cleaning when used. Writes generated by the background cleaner are marked with a flag, but have the same queuing priority as other reads and writes.

6.4 Conclusions

In this chapter, we described the specific implementation details of the HP97560 and HP2200C disk simulators we developed. In particular we discussed details that differentiate our simulators from similar simulators developed by other research groups. We described their use in trace-based simulation and show figures of demerit compared the performance of actual disks. We showed that our disk simulators perform very accurately in three out of four cases with 4.0% demerit for `hplajw` disk 0, 7.5% demerit for `snake` disk 5, and 4.3% demerit for `snake` disk 6.

Chapter 7

Simulation results

To compliment the mathematical analysis of the cache management techniques presented in the previous chapters, we now use trace-based simulation to study the impact of each technique. Our goal is to see how write cache management affects overall response time for both read and write requests. The major results of these experiments confirm the hypothesis that non-volatile write caching is a huge win: a well-managed non-volatile write cache of sufficient size can result in order of magnitude decreases in mean write service time. In fact, some cache management policies work so well that there are no stalled writes and the write performance of the I/O subsystem mimics the write performance of the cache. The effect of write caching on read response times is generally moderate, with the read delays caused by bursty cleaning writes being infrequent. A discussion of the strengths and flaws of each cache management technique based on our simulations follows in the rest of this chapter.

For our experiments, we collect several related time measurements to deter-

mine the effectiveness of different cache management algorithms in trace-based simulation. We model the I/O subsystem as the classic queue/server combination used in queuing theory. Requests are read from the trace, placed in the queue, and wait for the server perform them in the order that they are issued. The *queue time* is the amount of time the request spends waiting in the queue. The server simulates the actions necessary to complete the request at the head of the queue, removes it from the queue, and then repeats this cycle with the new head of the queue until the queue is empty. The *service time* is the time for each request to be completed by the server and depends on the type of I/O request being serviced and the cache state. The time necessary for cache hits and writes smaller than the amount of clean cache space is zero. The times to complete stalled writes, reads, and cleaning writes that require disk I/O include the disk seek time, settle time, and rotational delay as well as time to transfer data to or from the disk. The request *response time* is the sum of the queue time and service time.

We also collect information to measure how much disk activity from the trace the non-volatile write cache eliminates. The number of cache hits and cache misses provide information about how much data is overwritten in the cache. We record the amount of disk activity generated by the write cache, including the number of cleaning writes and their size. The total amount of data written to disk from the cache is a good indicator of the amount of disk activity saved through overwrites when compared to the amount written in the trace. The request times, service times, and number of cleaning writes made by the cache provide information about the contribution of cache cleaning to queue congestion and the amount of potentially wasted disk bandwidth. Finally, the

number of stalled writes is a good measure of how effectively a management technique keeps clean space available in the cache.

We wish to use I/O request response times for measuring the effectiveness of cache replacement techniques. We find that inaccuracies in the queue times of our results make this information unreliable for comparison with the times in the trace, however. These inaccuracies are caused by two factors:

1. Disk activity traces record I/O request times that represent the interaction between the processes, the operating system kernel, and the I/O subsystem of a computer. Adding a non-volatile write cache changes this interaction, but the request times remain fixed and do not correctly reflect interaction with the modified system.

2. Small inaccuracies of less than a millisecond in the service times of our simulators skew the queue times of long strings simulated read events compared to those in the traces. There are several such long groups of reads of consecutive sectors in the traces (presumably to back up the disk) where each read explicitly begins within a millisecond of the completion of the one before it. Our disk service times were slightly too large (< 1ms), creating long sequences of events waiting for service by the disk where none existed in the trace.

Both of these factors result in skewed queue lengths which sometimes (especially in the case of the consecutive reads) skew mean queue times by several milliseconds. This makes comparing the response times of the trace-based simulations and the traces impossible. We do however compare response times between traces, especially for our idle detection experiments. To avoid queue time skew for some experiments where response

time was especially important, the first 90 minutes of each trace was truncated to remove the groups of consecutive reads causing the skew.

Rather compare response times between the traces and our simulations, we rely on metrics which are accurate and indicate trends in response time when a non-volatile write cache in use. Of these, service time was the most important because our simulators are shown to be very accurate in Chapter 6. This allows us to directly compare mean service times between different simulation runs with the service times in the original traces. The number of cache misses and the number of stalled writes are also important because they affect how many write requests must wait for disk I/O to occur before being written into the cache. Finally, the number of writes and the amount of data needed to clean the cache are strong metrics for cache efficiency because increased cleaning activity increases the queue and response times of read operations.

For our experiments, we concentrate on disks 5 and 6 of the `snake` trace set and disk 0 from the `hplajw` set. Some of the important static and dynamic characteristics for these data sets are summarized in Table 7.1. Write requests in these traces are all sent to the cache and then eventually updated on disk with the exception of large writes ($>$ 16KB), which are sent straight to disk. Writes larger than 16KB are passed directly to the disk because they constitute a large portion of a disk track for the disks we examine. This makes them relatively efficient to write immediately and avoids consuming large portions of cache space in the smaller caches we test.

The remainder of this chapter is divided into four major sections. The next section describes simulation results comparing the three different management algorithms

Table 7.1: Characteristics for disks used in our analysis.

Disk	# of Read I/Os	Mean Read Size (KB)	Mean Service Time (all) (ms)	Block Size (bytes)
5	134420	5.354	11.735	512
6	146782	6.152	13.039	512
0	41080	4.409	25.346	256

Disk	# of Write I/Os	Mean Write Size (KB)	Mean Service Time (writes) (ms)
5	129379	6.876	12.388
6	240632	6.726	12.714
0	82054	6.024	27.475

presented in Chapter 3. Section 7.2 presents simulation results for the stack model-based algorithm described in Chapter 4. The simulations showing the effects of the idle detection technique in Chapter 5 on response times is found in Section 7.3. Finally, the last section summarizes the important results of these simulation studies.

7.1 Cache management results

For our first set of experiments, we concentrate on the relative merits of the cache management techniques presented in Chapter 3 [22]. The goal is to understand the basic mechanisms that control cache performance and examine how thresholding techniques and cache size improve cache performance. The following three subsections describe the performance of the least recently used (LRU), largest segment per track (LST), and shortest access time first (STF) algorithms with different thresholding schemes and cache sizes. Section 7.1.1 describes the performance of a very simple write behind non-

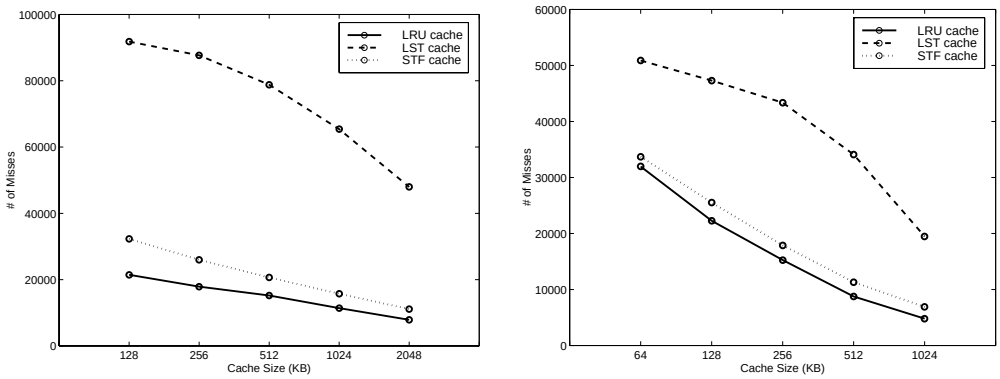
volatile write cache. Because this simple technique yields a good reduction in disk write activity but a high number of stalled writes, simulation results for single and dual threshold caches of different sizes are shown in Section 7.1.2. The good performance of dual threshold caches motivates a series of experiments to examine the relationship between replacement algorithm and cache size. The results of these simulations are shown in Section 7.1.3. This first series of experiments concentrates on `snake` disk 5 and `hplajw` disk 0.

7.1.1 Write behind cache management

A write cache contains the modified disk blocks which must be eventually committed to disk to make room for new blocks as write events occur. A simple policy to evict blocks from the cache is to wait until the cache is completely full and then purge the cache with a write behind strategy, as explained in Chapter 3. Because the cache operates at a nearly full steady state, writes frequently stall while room is made for the new data. The idea of simple non-volatile write behind caches was first suggested but never tested by Biwas, Radhakrishnan, and Towsley [6] because such policies "would result in unacceptably poor performance" because of frequent stalling.

We performed our own tests with write behind caches. We varied the cache size starting at 128KB and doubled the size on each run until we reached 2MB. For the three disk scheduling algorithms we tried, LRU performed the best in these tests, closely followed by STF, and then LST. Our service time measurements show that LRU and STF decreased the mean service time by at least 25%, scaled well as cache size increased, and reduced the number of writes to disk by at least 75% for both disks. The LST managed

cache produced little reduction in the number of writes to disk especially for small caches with a correspondingly small reduction in service times.



(a) Disk 5 (b) Disk 0

Figure 7.1: Cache misses for disks with a write behind cache.

Our results for stalled writes in some measure dispute the assumption that write behind caches are ineffective because of the high number of stalled writes (see Figure 7.1). While the cache miss rate is high for LST caches (nearly 100% of all writes are cache misses for small caches), cache misses for the LRU caches were much smaller (less than 20% for disk 5 and 40% for disk 0) with the performance of the STF cache in between. These results show that write behind caches can be effective and offer significant performance improvement without having any cache parameters to tune.

7.1.2 Cache management with thresholds

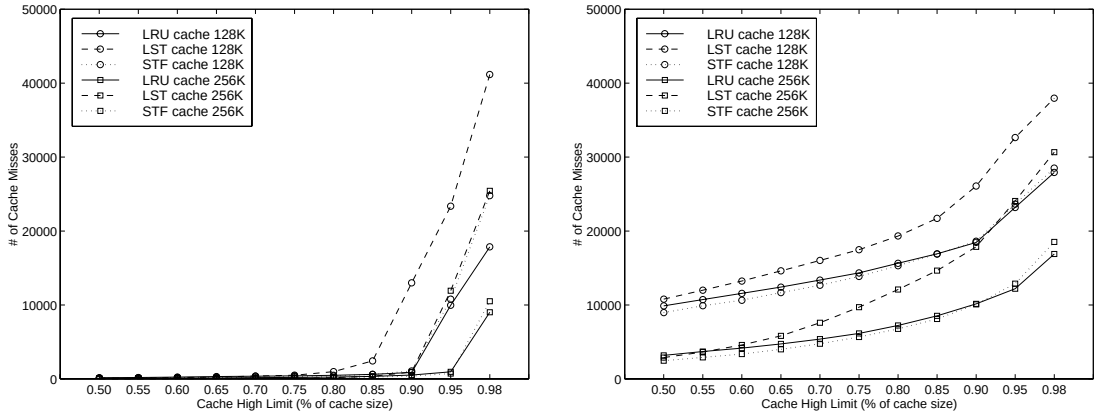
While a simple write behind cache can improve cache performance, the best cache management algorithm we examined only showed 30–50% improvement in mean

service time. For this reason, we also study a track based purge policy triggered by thresholds to prevent the cache from becoming completely full or (in some cases) completely empty. By preventing the cache from becoming completely full, stalled writes are reduced because some clean space is always available to hold new data. Since the cache does not completely empty, some data that will be overwritten in the cache in the near future will (hopefully) not be written to disk.

First we consider a single threshold variant of this cache eviction policy. With this type of cache, the high and low thresholds are set to the same value. The resulting cache is similar to a write behind cache, with some improvements. Because there is always some clean space in the cache, writes to the cache stall less frequently and cache writes can be made by the background cleaner. The amount of cache space cleaned in the background is small and the cache must be cleaned frequently.

We performed experiments to investigate the sensitivity of the cache to the threshold setting. We looked at single threshold caches of two different sizes for each disk; the caches were 128KB and 256KB for disk 5 and disk 0. Cache sizes were small to prevent the cache from holding the working set of written blocks. At the same time, we wanted to get some feeling for how these parameters change with cache size.

The number of writes to disk for a single threshold cache was within 10% of that of a write-behind cache of the same size for the LRU and STF algorithms until the threshold rose above 90%. The write traffic of the LST cache for disk 5 was also within 10% for both sizes, but the single threshold LST cache improved about 30% for most threshold values. The least amount of write traffic was produced for all single threshold



(a) Disk 5 (b) Disk 0

Figure 7.2: Number of cache misses for disks with a single threshold cache.

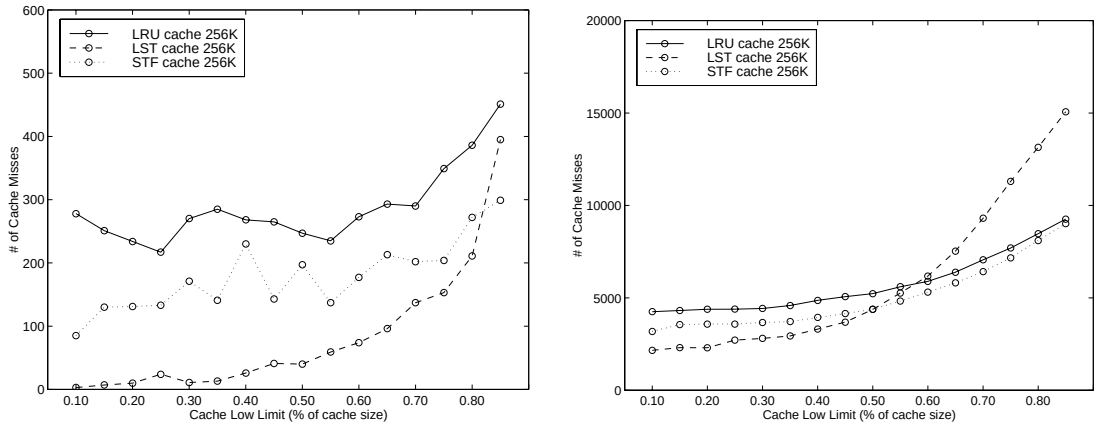
caches when the threshold was set at 98%. This single threshold cache produced 25–50% fewer writes than the write behind cache.

Our results also show that adding a single threshold only improved the number of stalled writes produced by the write behind cache (see Figures 7.1 and 7.2). The reduction in stalls is attributed to the clean space kept available by the threshold; data is written instead of forcing stalls. Since only the number of stalls changes, temporal locality still dominates and the LRU cache performs best.

Our results show that choosing a good cache threshold is a trade-off between the numbers of stalled writes and writes to disk. To obtain good overall performance, we try for a balance that decreased the number of writes to disk, but avoided sharp increases in the number of stalled writes. Based on these criteria, we note that the best choice for a high limit threshold is in the range of 90–95% for both disks for all algorithms.

Using a single threshold improves cache performance, but it is difficult to find a good balance between writes to disk and stalled writes. To avoid this problem, we also look at caches using high and low thresholds to create hysteresis in the cache purging process. Because the amount of space cleaned using a dual threshold scheme is larger, the number of stalled writes will decrease.

We began by testing how the interaction of the high and low threshold values affected performance. We fixed the high threshold at the single threshold values and varied the low threshold value from 10–85%. For these experiments, cache size was set at 256KB. The number of cleaning writes and cache misses for each cache are found in Figures 7.3 and 7.4.

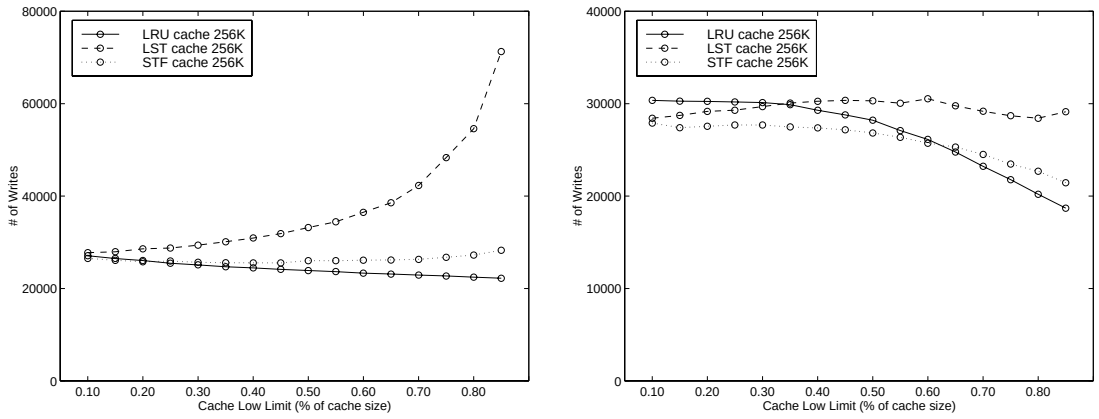


(a) Disk 5

(b) Disk 0

Figure 7.3: Number of cache misses for disks with a high and low threshold cache.

Finding a good value for the low threshold depends on what metric is used. Based on the number of cache misses for each cache, all three algorithms perform best



(a) Disk 5 (b) Disk 0

Figure 7.4: Cache generated write traffic for disks with a high and low threshold cache.

when the low limit was set in the 15–20% range. It is more difficult to measure the performance of the write cache based on the number of writes generated to clean the cache. The LRU cache also shows the unusual property that number of writes decrease by almost 5% as the lower limit is raised. The LST and STF algorithms perform well for both disks when the low limit is set in the 15–20% range. The number for both algorithms is not minimal for disk 0, because of the unusual rise and fall in the number of writes.

In terms of performance, adding a second threshold causes mixed results. It reduces the number of stalled writes for the LST cache to almost zero, and reduces the number of writes for the LST cache to that of the other algorithms (see Table 7.2). At the same time, it degrades the performance of the LRU algorithm. Cleaning large portions of the cache benefits the LST cache because it will always use the fewest number of large writes to clean cache space. This same action reduces the number of dirty blocks in the

Table 7.2: Write traffic for disk 5 and disk 0 generated by a 256k cache.

	Write Behind	Single Threshold	Dual Threshold
LRU	22194	21155	26039
LST	98117	83010	28589
STF	31773	30227	25791

Disk 5

	Write Behind	Single Threshold	Dual Threshold
LRU	30187	17083	30243
LST	53569	29928	29165
STF	34633	20573	27553

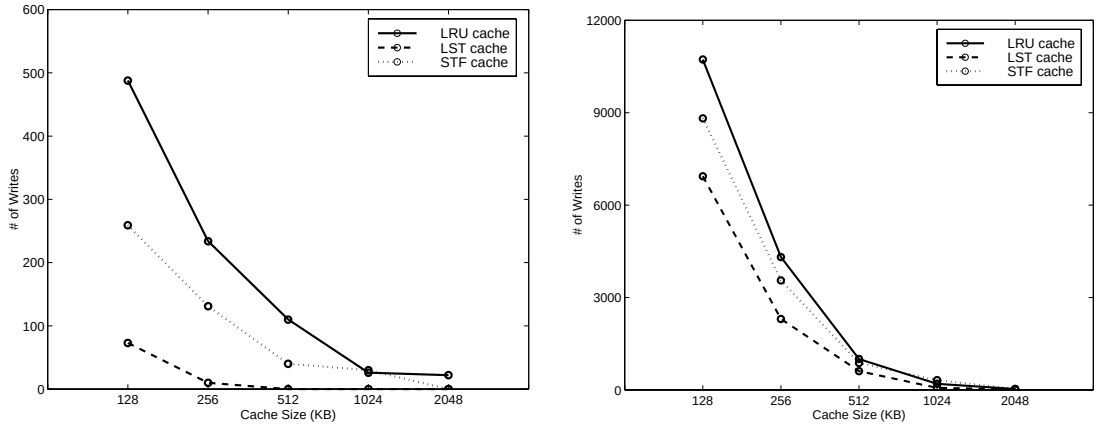
Disk 0

LRU cache whenever cleaning is performed. The LRU algorithm may be able to make better choices when there are more dirty blocks in the cache. In spite of this limitation, the LRU algorithm performs well, confirming that temporal locality is still important.

7.1.3 Cache size variation

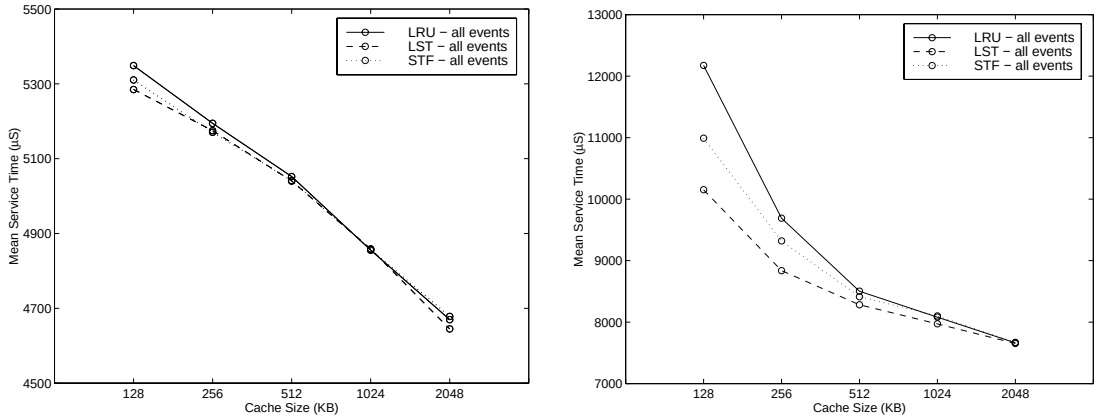
The performance of the dual threshold caches leaves an open question: is it better to stall less or write to the disk less? We looked at the answer to this question while examining the impact of cache size for the LRU, STF, and LST algorithms. We used the number of cache misses as our best metric for setting the high and low thresholds for each cache. High threshold values were set to the values used in our lower bound tests: 90% for the upper thresholds for all caches. Low threshold values were set to 20% for disk 5 and 15% for disk 0. For these experiments, we varied cache size from 128KB to 2MB, doubling memory size for each successive run.

The results from our cache size experiments show that all three management



(a) Disk 5 (b) Disk 0

Figure 7.5: Write cache misses for disks with different dual threshold write cache sizes.



(a) Disk 5 (b) Disk 0

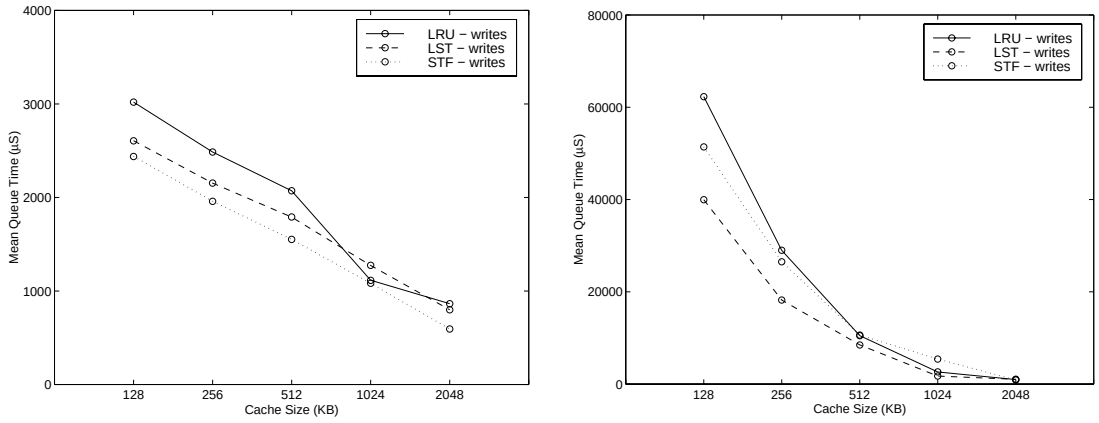
Figure 7.6: Mean service times for disks with different dual threshold write cache sizes.

algorithms scale equally, though the head position aware algorithms (LST and STF) may scale slightly better than LRU. Looking at the number of writes generated by each cache, the LST, STF, and LRU caches produced almost identical numbers of writes for disk 0, and LRU produced approximately 10% more writes for disk 5. The rates of decrease for STF and LST were were slightly higher than that for LRU for disk 5. Perhaps temporal locality becomes less significant as cache size grows larger for that disk. At that point, algorithms that take advantage of head position may become more useful.

Looking at the mean service and cache miss data, fewer cache misses have a direct and beneficial effect on the service time (see Figures 7.5 and 7.6). The LST cache produces the lowest service times for both disks, and produces zero stalled writes with the smallest amount of cache space for both disks. Service times for all trace events quickly converged to an average dominated by the service time of read events for both disks.

Lower service times are only beneficial if they contribute to lower overall response times. While the LST algorithm does produce consistently fewer stalls and better service times, it also tends to write more often to disk. If these additional writes are increasing the amount of time that other events spend queueing for service, then an algorithm other than LST is a better choice.

To check to see if this was happening, we collected queue time information for two sets of events in the disk traces. The results in Figure 7.7 show mixed results. For disk 5, the STF cache produces mean queue times for write events that are better than either LRU or LST. The set of write events from disk 0 shows that the LST cache has a



(a) Disk 5 (b) Disk 0

Figure 7.7: Mean write queue times for disks with different dual threshold write cache sizes.

lower mean queue times than the other caches though the values are close enough not to be significant. The read queue times for disk 5 were abnormally long (for reasons mentioned at the beginning of this section), but, the read queue times for both disks did show trends similar to their write counterparts.

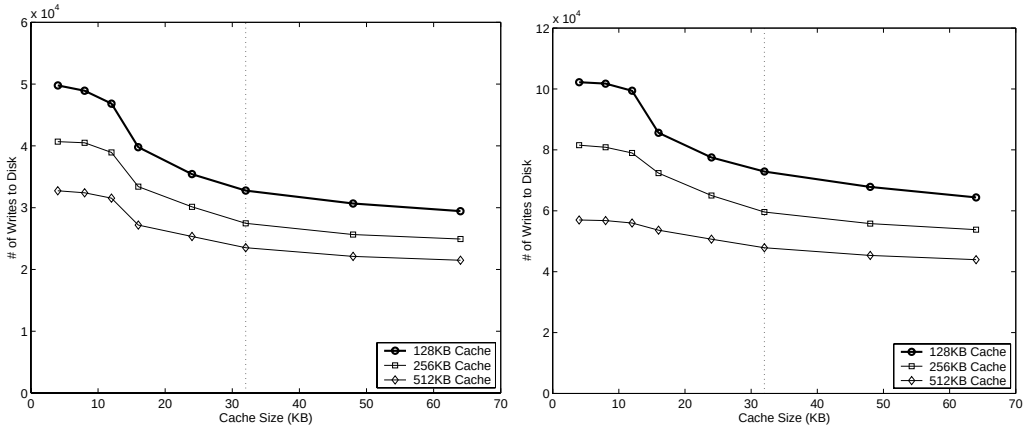
7.2 Simulations using the stack model algorithm

As results in the previous section show, the LRU, LST, and STF algorithms can very effectively reduce the amount of write traffic sent to the disk and produce a comparatively small percentage of stalled writes. No one algorithm is clearly the best for all cache performance metrics, however. The LST algorithm produces a small number of stalled writes, but writes to disk the most often. The STF replacement policy writes the least amount to disk, but produces and higher number of stalled writes. The LRU

cache performs somewhere in between LST and STF in terms of stalled writes and disk activity. These results motivated the development of the stack-model based algorithm presented in Chapter 4. Our experiments with this algorithm focus on disks 5 and 6 from the snake traces.

To validate our approach of grouping segments into hot and cold regions, we examine the effect of manually varying the size of the hot region of our cache [23]. If this approach is correct, the number of writes to disk should be high when the hot region is small because the large hot segments are frequently being replaced. As cache size increases and approaches the knee, the number of writes to disk should decrease rapidly because more hot segments fit into the hot region. The number of writes should then decrease gradually past the knee as all the hot segments are now in the hot region. The results (see Figure 7.8) showed precisely this type of behavior, with decreases of as much as 25 percent in the number of writes before the knee and as little as 4 percent after it.

We now compare the performance of our replacement policy to those that use temporal locality or spatial locality but not both. We employ two other policies for points of comparison: the LRU replacement policy and the LST replacement policy. We expected the LRU and LST policies to perform worse than our policy overall. The LRU policy handles hot segments well, but makes costly small writes to disk. The LST policy makes efficient writes of large segments, but this is only useful when the segments are cold. Since our policy attempts to deal with both hot and cold segments, we expect that it will perform comparably (at least) to the best metrics for LRU and LST. A comparison of the results shows this is true for the number of writes made to disk and the



(a) Snake disk 5 (b) Snake disk 6

Figure 7.8: The effects of varying the size of the hot region of the cache for three different cache sizes with snake disks 5 and 6. The dotted line indicates the location of the knee in the hit ratio curve.

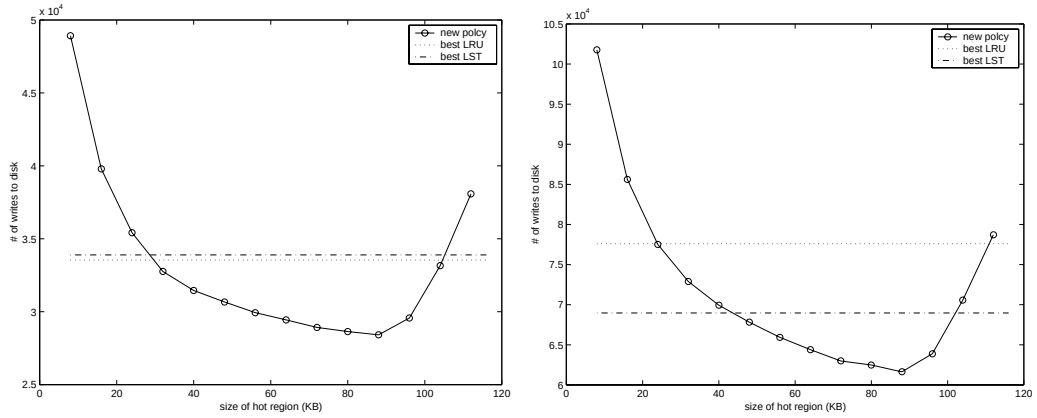
Table 7.3: A comparison of three metrics for snake disks 5 and 6 for a 128KB cache.

	Snake disk 5			Snake disk 6		
	LRU	LST	New	LRU	LST	New
writes to disk	33538	33895	32758	57059	54057	59592
stalled writes	418	85	97	398	0	0
cache overwrites	79678	76061	79980	143191	141814	145871

number of stalled writes (see Table 7.3). This comparison also shows that our new policy consistently overwrote data in the cache more often than the LRU or LST policies.

To see how our new policy performs as the size of the hot region changes, we considered the hot region to be a fraction f of the total cache size and varied f from zero to one with a 128KB cache. A small cache was used because larger caches produce fewer cache writes and stalled writes, obscuring the relative performance of the different replacement policies. We adjusted the size of the hot region of the cache between 8KB

and 112KB in 8KB increments to obtain different values of f. We didn't use a hot region size larger than 112KB because the high threshold of the cache was set to require that at least 10 percent of the cache be kept free of dirty blocks. The effect of changing hot region size on write activity and cache stalls for the Snake disk 5 is shown in Figure 7.9.



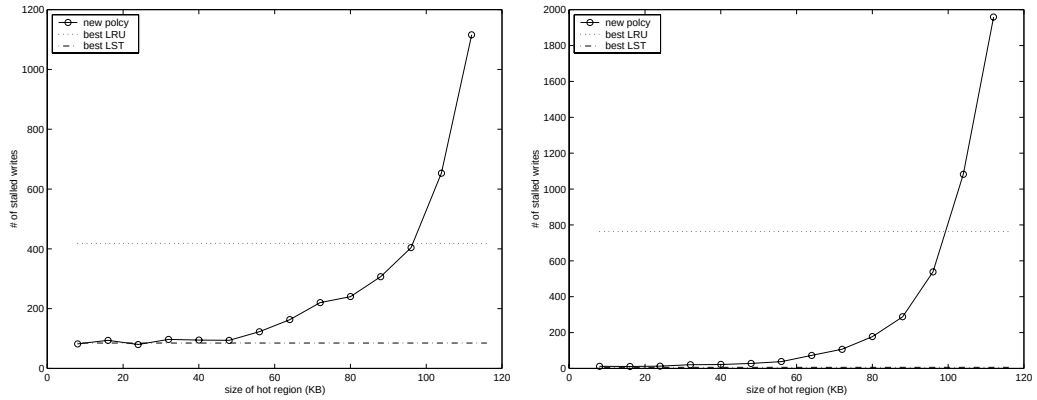
(a) Snake Disk 5 (b) Snake Disk 6

Figure 7.9: Writes to disk made with the new policy as the size of the hot region varies

The changes in the number of writes made to disk with the increasing size of the hot region confirm our assumptions about cache behavior. The number of writes to disk is high when the size of the hot region is very small because hot segments are being replaced in the LST portion of the cache. The number of writes decreases sharply as the hot region grows but then flattens out once the size of the knee value is reached. The increase in write activity as the hot region grows very large is because the effective amount of space that can be cleaned is small. The small amount of clean-able space forces explicit writes as the cache often stalls and frequent purges of the small amount of

clean-able space.

Our sensitivity experiments reveal that the number of stalled writes determines the upper bound in hot region size (see Figure 7.10). As the hot region grows, the percentage of dirty blocks in the cache more often approaches the 90 percent upper limit for dirty blocks in the cache. This causes the number of stalled writes to begin to increase rapidly at a smaller cache size than the number of writes shown in Figure 7.9. Obtaining the best overall performance requires a hot region size that provides the right balance between the number of writes to disk and stalled writes.



(a) Snake disk 5 (b) Snake disk 6

Figure 7.10: Stalled writes made by the new policy as the size of the hot region varies

Finally, we looked at how well our new replacement policy scales as cache size increases. Changes in the number of writes to disk (shown in Figure 7.11) reveals that the cache scales well. The number of disk writes decreases at a nearly linear rate as cache size doubles.

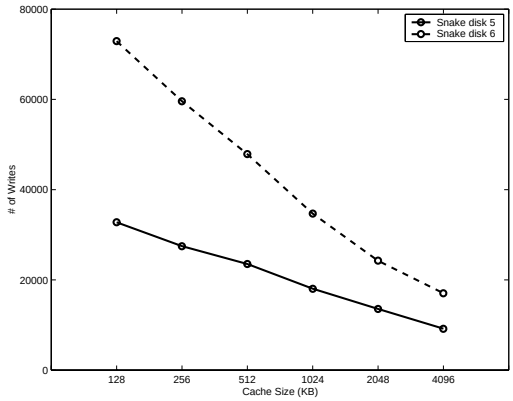


Figure 7.11: Writes to disk using our new policy as cache size increases.

7.3 Idle detection

One consequence of using a non-volatile write cache is that writes are made to disk in bursts as the cache is cleaned. These bursts can potentially delay read requests made to the disk by forcing them to wait while groups of cleaning writes are serviced. A possible remedy to this situation is to wait until the disk is idle *i.e.* not servicing a read request and is unlikely to do so in the near future. This allows cleaning write requests to be serviced while the disk is otherwise idle and creates minimal increases in read response time. The experiments we performed to examine the effects of idle detection focus on snake disk 5.

Our work with idle detection produced two basic models to describe cache performance in Chapter 5. The first uses a Markov chain to model the growth in the number of dirty blocks in the cache. This Markov chain has a set of steady state limiting probabilities that can be used to predict the probability that an arriving write will cause the cache to begin cleaning dirty blocks to disk. The second model uses queuing theory to

investigate the effects of waiting a fixed period of time to determine if the disk is idle before writing dirty blocks to disk during a cleaning cycle.

To test our model for predicting how often the cache will be cleaned, we instrumented our disk simulation programs to produce special event tags indicating when cleaning cycles start. Scripts counted the number of cleaning cycles that occurred in a given experimental run. The probability that a write forces a cleaning cycle to begin is this cycle count divided by the number of write events in the source traces.

We quickly discovered one major flaw with our simple Markov model: the model does not correctly handle write dependencies between blocks on the disk. The Markov model we construct treats each block independently, where there is one state in the Markov chain for each dirty block in the cold region of the cache. A write event in the trace rarely writes a single block, however. Blocks are written to the cache in groups of one to sixteen kilobytes, causing additional state transitions that our simple Markov chain does not model.

In an effort to account for these write dependencies between blocks, we try modeling the cache using a *grouping factor* to reduce the number of states in the cache. The rationale behind the grouping factor is this: a disk will generally write out data in certain contiguous file system block and fragment sizes. If we have the states in the Markov chain represent the number of these dirty blocks or fragments in the cache, a better probability estimate might be possible. Therefore, we calculated the length of the Markov chain in our model according to the following formula:

$$(\text{\# of Markov states}) = \frac{(80\% \text{ of total cache size in KB}) \times (\text{\# of blocks per KB})}{(\text{grouping factor})}$$

The eighty percent cache size figure in the formula assumes that twenty percent of total cache space was assumed to be reserved by settings of n_{high} and n_{low}. This is based on experiments in Section 7.1.2 showing the need to set the high and low cleaning thresholds to improve cache performance by reducing stalls and keeping hot blocks in the cache.

The one remaining required input of for the model is the state transition probability for the Markov chain using the formula

$$h(n+1) = h(n) + \epsilon.$$

The value ϵ was estimated by plotting a histogram of the write frequency of every block in the experimental trace sorted by increasing frequency. This created a tail at one end of the histogram that estimated the number of "cold" blocks in the trace that are written a small number of times. The grouping factor was divided by this number to produce a value for ϵ.

Some simple tests with different grouping factors with data taken from snake disk 5 show that a grouping factor of 8 with an estimated number of cold blocks in the trace of 119400 models cache performance well. The results are shown in Figure 7.12. This indicates that the Markov chain with grouping factor approach has some merit, but further research is required to more correctly deal with dependencies between written blocks. This may include additional analysis to correctly determine the grouping factor based on the properties of the trace, or a second order model that uses dependency information to better model state transitions.

In Chapter 5, our model of cache cleaning with idle detection suggests that cleaning the cache during idle periods improves read response time very little and de-

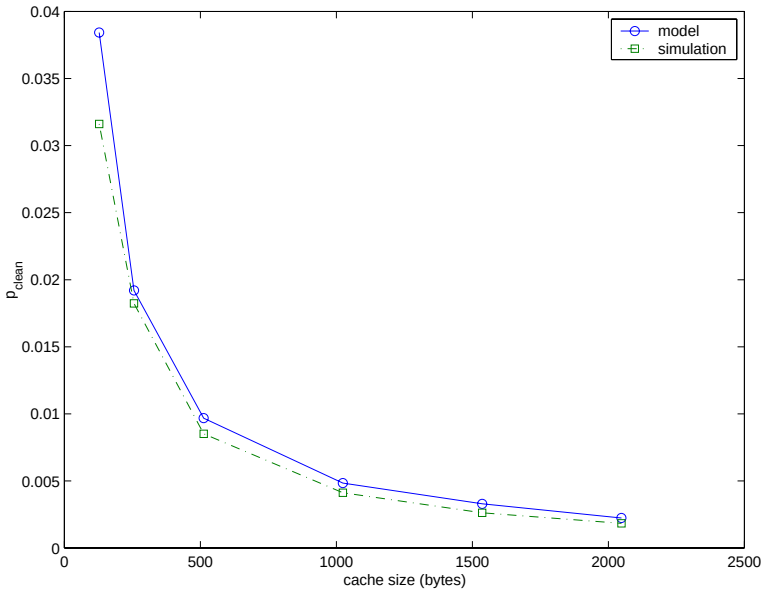


Figure 7.12: Simulated and model predicted cache cleaning probabilities for an LRU managed cache with snake disk 5. The model values used a grouping factor of 8. The number of cold blocks in the trace was estimated to be 119400.

grades write response time. To prove or disprove this assertion, we altered our simulators to wait a fixed period of time after the last serviced event before cleaning the cache. We then conducted experimental runs with a wide range of idle times and collected simulated read and write mean response times for each run. The results of these simulations are shown in Figure 7.13.

The simulated read and write response times repeat the general behavior seen in the model in Chapter 5 with a sample disk. The read response time stays approximately constant for small idle waiting times, until a cross over point as waiting time increases. After this crossover is reached, the read response time decreases slightly and remains nearly constant at this lower value. At the same time, write response time remains constant for low values of the idle detection time and then increases when the idle

detection time grows large. These are all characteristics observed in both the model and the simulated data.

This behavior suggests that the number of reads waiting for service when a cleaning cycle commences is rather small. It also suggests that no additional decreases in read response time are possible once an idle detection period allows these reads to be serviced. It does suggest that one way to improve read response time is to issue write requests to clean the cache in small groups rather than all at once. This is an approach actively pursued analytically by Carson and Setia [9] with the Sprite LFS.

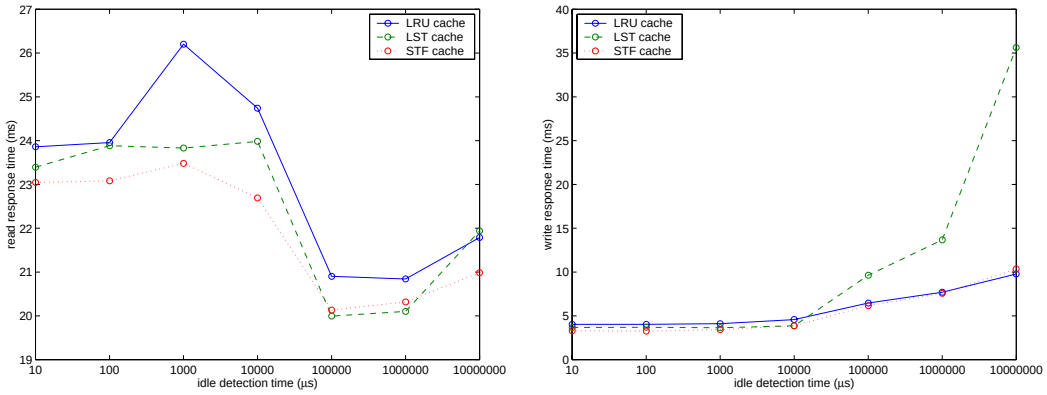


Figure 7.13: Read and write response times for snake disk 5 with a non-volatile write cache with idle detection.

7.4 Conclusions

This chapter describes a series of simulation experiments designed to increase our insight about how to manage of non-volatile caches and either confirm or deny our ideas about how such caches function. In general, these results show that the caching and delaying of writes to disk is an excellent idea. Even the simplest write behind cache

significantly reduced the amount of data written to disk by 80 percent by overwriting data in the cache when correctly managed. Slightly more complicated two threshold caches are able to increase the number of cache overwrites and dramatically decrease stalled writes (writes which must wait while dirty cache space is cleaned). Some cache management algorithms were even able to reduce the number of stalled writes to zero, making the I/O subsystem appear to have a write service time identical to that of the cache.

Our first series of experiments with the basic cache management algorithms presented in Chapter 3 reveal that there is more than one important mechanism governing cache performance. Each of the simple algorithms has some kind of performance flaw; the STF algorithm produces too many stalls, the LST cache writes to disk too often, and the LRU algorithm shows mediocre performance in between the two other algorithms. This suggests that a simple technique that takes advantage of only one property of the write reference stream cannot always produce the best cache performance.

We perform similar experiments with the stack model based algorithm described in Chapter 4. These experiments show that the new algorithm offers the best aspects of the performance of the LRU and LST algorithms; it produces a number of stalled writes comparable with LST, and writes as much data as LRU to disk. Further tests reveal that this algorithm also scales well.

Finally, we use instrumented simulators to investigate using idle detection with cache cleaning. The results confim trends found in the model presented in Chapter 5. Idle detection improves read response times only slightly while increasing write re-

sponse times. Work with the Markov model to predict how often the cache cleaning is required reveal that dependencies between written blocks make the number of states in the Markov chain difficult to calculate. By grouping blocks together and altering the model to work with these groups instead of single blocks, it is possible to predict the probability that the cache will be cleaned, although more work is required to validate this technique.

Chapter 8

Conclusions

As processors and main memory become faster and cheaper, a pressing need arises to improve the write efficiency of disk drives. Today's disk drives are larger, cheaper, and faster than they were 10 years ago, but their access times have not kept pace. The microprocessors of today have a clock rate 50 times faster than their predecessors of 10 years ago did. At the same time, the average seek time of a fast hard disk is at best between one half and one third of its predecessors from the same period. Some technique must be found to bridge the performance gap if I/O systems are to keep pace with processor speed.

In the preceding chapters of this dissertation, we examine several aspects of non-volatile cache management used in conjunction with delayed writes to address this problem. Non-volatile disk caches can reduce disk workload more effectively than volatile kernel buffers or disk caches because they allow disk writes to be safely delayed. This approach allows some writes to be avoided because the blocks will be overwritten or

deleted while they are still in the cache. The remaining disk writes also can be organized more efficiently by writing contiguous blocks in a single I/O operation. As disk subsystems continue to lag the performance of ever faster processors, non-volatile caches are an increasingly important way to remove a critical performance bottleneck.

Our goal with this dissertation is to compare different cache management techniques by looking at different ways to implement and clean disk write caches of NVRAM. We examined three issues not adequately addressed in prior work with non-volatile write caches:

1. How do different basic cache replacement algorithms affect cache performance under similar file system workloads and what are the key metrics that describe cache performance?

2. Can techniques used in developing disk scheduling algorithms be used to develop new replacement algorithms which provide better overall cache performance?

3. Can idle detection be used to delay cache cleaning to improve read response times with non-volatile caches?

The answers to these questions form the basis of the work of this dissertation. In Chapters 3 and 4, we present mathematical models and algorithms describing four replacement algorithms for cleaning the cache. Three of these algorithms (in Chapter 3) were used previously in a research study or product involving non-volatile caching or disk scheduling. The algorithm presented in Chapter 4 is new and uses design principles borrowed from disk scheduling research. Chapter 5 describes a method for predicting how often the cache requires cleaning and a model for investigating of delaying cache

cleaning until the disk is not servicing any I/O requests. This work is complimented in Chapter 7 by a series of trace-based simulation experiments which measure cache performance when these different techniques are used. Key measures of performance include the number of write cache hits and misses, the mean service times, and the number of writes to disk for two types of disks. The simulators for these disks are described in detail and validated in Chapter 6.

The remainder of this chapter is split into four sections summing up our conclusions for the different aspects of this work and describing future work. The next section describes the key results for a comparative study of three basic cache replacement algorithms. Next, Section 8.2 presents our conclusions regarding the use of a new stack-based replacement algorithm that combines aspects of two simpler algorithms. In Section 8.3 describes the effects of using cache cleaning with idle detection. Finally, we end with some directions the research in this dissertation could take in the last section.

8.1 Cache management

Our comparative study with NVRAM caches shows that temporal locality is a key to cache efficiency for many caches, especially small ones. We implemented and tested models of caches using a simple write-behind purging model and write-behind with thresholds. We used the least recently used (LRU), largest segment per track (LST) and shortest seek time first (STF) algorithms to decide what data to clean from the cache. Comparison of the data for the number of (cleaned) writes to disk and stalled writes shows that the LRU algorithm works best in many situations. Algorithms which attempt

to use head position to determine what to clean next (LST and STF) can produce fewer stalled writes, but generally write to disk more often. For many kinds of caches, the cost of the additional writes outweighs the benefits of fewer stalls.

Our initial work with write-behind caches shows that the LRU managed cache was the most effective, by far. Others rejected simple write behind caches under the assumption that writes to these caches would frequently stall. Caches of this type using algorithms depending intrinsically on disk geometry, particularly LST, perform poorly, sometimes no better than the cache-less disk itself. The LRU managed cache significantly reduces the number of writes to disk and improved response time. The number of stalled writes is large compared to more complex cache management policies, but the policy is very simple and does not need tuning.

We find that one of the few cases where temporal locality is not dominant is when high and low thresholds are used. The large hysteresis of such a cache substantially reduces the number of writes that the LST algorithm must make to clean the cache, making its write performance equivalent to other algorithms. This performance improvement combined with the smaller number of stalled writes creates reduced service times. A study of how these properties scale shows that head position algorithms like LST may perform better than LRU when cache sizes are large.

8.2 A stack based replacement technique

Our prior results show that although adding thresholds to caches improved their performance, each basic method we examined performed poorly in some way.

In particular, caches managed using a greedy LST policy produced the fewest number stalled writes but produced larger amounts of write activity. The STF policy produced the least amount of write activity, but also produced an unacceptable number of stalled writes. The LRU policy performed between the other two in terms of write activity and stalled writes.

We presented a new block replacement policy that organizes efficiently disk writes while keeping the blocks that are the most likely to be accessed again in the cache. Our policy partitions the blocks in the cache into two groups: the hot blocks that have been recently referenced and the remaining blocks that are said to be cold. Hot pages are guaranteed to stay in memory. Whenever space must be made in the cache, the policy expels the cold blocks that belong to the largest set of contiguous segments within a single track until enough free space has been made.

Experimental results show that the use of our new replacement policy with a correctly tuned, modestly sized cache reduces writes to disk by 75 percent on average and the policy frequently did better. The results show the stack model policy to be more effective than cache replacement policies which exploited either spatial or temporal locality, but not both. In particular, data is overwritten in the cache more often using our policy than the others, reducing the number of writes to disk. The new replacement policy also gives the relative benefits of such policies without their unattractive features. It often provides the least number of writes to disk of any of the policies we used for comparison. At the same time, it often produces no stalled writes when other policies produce hundreds.

8.3 Idle detection

A potential problem with non-volatile write caching is that it produces bursts of writes to clean the cache. These groups of writes can increase read response time by forcing read requests to wait while writes are serviced. We examined a method which forces clean writes to be made when the disk is idle *i.e.* the disk is not servicing a read request and unlikely to do so in the near future.

The first part of this investigation attempted to predict how often a non-volatile write cache requires cleaning. We first developed a model of cache behavior using a Markov chain. Using assumptions about the cache workload, the Markov chain produced a expression for stable mean probability for cleaning the cache over time.

Tests with trace-based simulations show that interdependencies between disk blocks in the disk traces make the calculation of the number of states in the Markov chain difficult. In particular, the Markov chain models individual disk blocks where real disk workloads rarely access blocks one at a time. We used the observation that disks frequently write data in groups of contiguous disk blocks based on a file system block size or fragment size to compensate for these dependencies. By employing a grouping factor to make the states in the Markov model correspond to groups of sectors, model predictions closely approximate observed values. The properties of this grouping factor technique are not well understood and require further investigation and validation.

This cache model was then extended with queuing theory techniques to show that the delays in cleaning induced by idle detection have mixed effects for many practical workloads. By modeling a statistically average cleaning cycle, we show that mean

read response times decrease only slightly while write response times increase dramatically due to stalls for larger idle detection times. These results are confirmed in trace-based simulations.

These results suggest that few, if any, read requests are waiting for service at the disk when cleaning begins. The delay of the start of the cleaning cycle until the disk is idle allows only a few reads to be serviced and has a small impact on mean read response time. The real impact of cache cleaning is caused by reads waiting for service once cache writing has begun. Cleaning writes should be made from small cache segments to more effectively reduce read response time, not from the entire cache at once.

8.4 Future work

This work can be extended in several ways. One of the most immediate aspects of this work requiring more research is the method to determine the size of the hot zone for the stack model-based replacement algorithm. We determined the best size for the hot zone empirically in our experiments. A more ideal solution would be some form of online method that uses information about the disk workload from some preceding period and determines the hot zone size based on those conditions.

Additional work is also required to identify the workloads where the stack-model based algorithm works best. We only examined the traces of two disks for our simulation results, and both were taken from a single file server. Other workloads exist with different access characteristics where the algorithm may not perform as well. A test for separating good and bad workloads for this algorithm must be devised, and,

a measure of how poorly this algorithm performs with bad workloads must be found. This can include experiments with systems that may produce substantially different disk workloads like databases and transaction processing systems

The idle detection work suggests that the cache should be cleaned in segments to reduce impact on read response time. Determining the size of these segments and the amount of time between the cleaning of the segments is an open research issue. This will require additional mathematical analysis of the cleaning cycle and further experimentation.

The work in this dissertation only deals with the interaction between the cache and one disk. Additional studies are required to extend this work to systems with multiple disks. Issues such as threshold placement, the correlation between cache size and the number of disks, and the merits of each cache replacement policy must be investigated.

Finally, this work needs to be included in a device driver implementation. Trace-based simulation has flaws because it fixes request arrival times according to the interaction between the processes, operating system, and I/O devices used by a certain machine at a certain time. Using a non-volatile write cache changes the performance characteristics of the I/O subsystem, and the arrival times of certain I/O requests should be different as a result. Simulation provides excellent indications of how performance will improve, but the kind and size of the real performance improvements produced by this approach will not be known until an implementation is done.

This dissertation examines several important aspects of cache management for non-volatile write caches to reduce write latency. It includes a comparative survey of

simple cache management techniques. It also presents a new hybrid algorithm that provides better performance. It details two disk simulators that can be used in other I/O related research. Finally, it discusses the advantages and drawbacks of delaying cache cleaning until the disk is idle. All of this work will be useful in guiding future work in non-volatile write cache management.

Appendix A

Simulator Usage

This section describes the steps required to create and initialize the data structures needed for a simple disk simulation program. This discussion will be limited to the public members of all simulation classes. More details about the private members of classes and the interaction between simulation objects can be found in the simulation source code.

Three different C++ classes were developed to implement and validate our design: DiskNULL, Disk335, and Disk975. The classes Disk335 and Disk975 simulate the HP2200[1] and the HP97560 disks respectively, and, both classes have identical interfaces. The class DiskNULL is used to replay a disk trace using starting and ending times found in an existing disk trace, and has slighty different interface semantics.

In general, a disk object is created by calling a constructor which supplies it with information about the bus to which is to be attached (see Figure A.1). This information is

[1] The HP2200 and the HP335h are identical disks. Source code in the SRTheavy library from HP frequently refers the HP2200 as the HP335. For this reason, we used 335 instead of 2200.

```
Token_List tlist;

diskioBus *theHPBusType = new diskioBus("hp_ib");
Facility *theHPBus = new Facil-
ity("HP bus", 1);

diskioBus *theSCSIBusType = new diskioBus("scsi");
Facility *theSCSIBus = new Facil-
ity("SCSI bus", 1);

DiskNULL *theDiskNULL = new DiskNULL();
Disk335 *theDisk335 = new Disk335(theHPBus, theHPBusType);
Disk975 *theDisk975 = new Disk975(theSCSIBus, theSCSIBusType);
```

Figure A.1: Constructors and necessary data structures for simulation objects.

in two parts: a Sim++ Facility object to control access to the bus, and a diskioBus object to provide information about bus speed and access time. The exception to this rule is the class DiskNULL. Because a DiskNULL object is used to play back existing traces, it doesn't model bus contention and no bus information is necessary. The construction of disk objects as pointers is recommended, because Sim++ does not support a destructor for the Facility class. Using pointers avoids the creation of unused Facility objects in the simulation. All simulation events are referenced from a common event list (of type Token_List) which must be declared by the simulation program writer.

```
int theDisk_idNULL = theDiskNULL->AddToArray();
int theDisk_id335 = theDisk335->AddToArray();
int theDisk_id975 = theDisk975->AddToArray();
InitTokenList(&tlist);
```

Figure A.2: Initialization steps for simulation objects.

Once the disk objects have been declared, two more data structures must be

initialized (shown in Figure A.2). In order for the simulation events to locate the state information of the disk object with which they are associated, each disk object must be added to a common array of disks known to all simulation events with an AddToArray call. The id of the disk object assigned (and returned) by AddToArray is used internally by the simulation as an id to relate specific events and specific disks. The declaration of the simulation event list must also be made known to the disk objects through a call to InitTokenList.

```
token a_token

a_token.id = theDisk975->Id();
a_token.type = <READ, WRITE, EOF>
a_token.sector = <logical sector number>;
a_token.size = <size in sectors>;
a_token.start = <time since simulation start in microseconds>;
a_token.sync = <TRUE or FALSE>;

tlist.add(a_token);
```

Figure A.3: Declaration and input fields of a request token.

All three simulation classes accept I/O requests as tokens which contain information about the request. An event token requires a minimum of five pieces of information (see Figure A.3): The token's id is the identifier returned by the AddToArray call for the disk object processing the event. This id can be obtained from the disk object with an Id() call. The event_type is the type of the requested operation. Currently supported request types are limited to reads and writes (the EOF type indicates that the simulation is finished). The sector number is the logical sector number representing the starting location of the request. The size of the event is the length of the request in number of sectors.

The start is the simulation time when the request will be issued. The sync flag indicates whether or not the request is to be followed by a sync operation (if it is supported).

Each simulation token must be added to the event list prior to the start of its execution. The last event added to the event list should be an EOF event.

```
a_token.wait = <time since simulation start in microseconds>;
a_token.end = <time since simulation start in microseconds>;
```

Figure A.4: Additional input fields for the DiskNULL class.

Since the DiskNULL class replays existing traces, two additional fields shown in Figure A.4 are required. The wait time, normally an output, is the time from the trace when the simulation began servicing the request. The end time, also usually an output, is the time from the trace when the request was completed.

```
theDisk->Request_Disk();
```

Figure A.5: Starting a service request.

Each I/O event is serviced by calling the disk object's member function named Request_Disk (shown in Figure A.5). Because this call should be made during a Sim++ event, the time of the request should already be known. The simulation is complete once all I/O requests have been processed.

To obtain simulation results after the simulation has completed, individual tokens can be retrieved from the token list and their outputs read as seen in Figure A.6. The wait time is the simulation time when the servicing of the I/O request actually started (the time in the queue is the difference between the wait time and the start time). The end

```
tlist.get(token_index);

a_token.wait = <simulation time when request service began>;
a_token.end = <simulation time when request service completed>;
```

Figure A.6: Obtaining simulation results.

time is the time when the request completed (the service time is the difference between the end and wait times).

Bibliography

[1] Sedat Akyüreck and Kenneth Salem. Adaptive block rearrangement. *ACM Transactions on Computer Systems*, 13(2):89–121, May 1995.

[2] Sedat Akyüreck and Kenneth Salem. Adaptive block rearrangement under UNIX. *Software–Practice and Experience*, 27(1):1–23, Jan 1997.

[3] Arnold O. Allen. *Probability, statistics, and queuing theory: with computer science applications.* Academic Press, Inc., second edition, 1990.

[4] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-volatile memory for fast, reliable file systems. *Operating Systems Review*, 26(Special issue):10–22, Oct 1992.

[5] Anupam Bhide, Daniel Dias, Nagui Halim, Basil Smith, and Francis Parr. A case for fault-tolerant memory for transaction processing. In *Digest of Papers FCTS-23 23rd International Symposium on Fault-Tolerant Computing*, pages 451–60. IEEE Computer Society Press, Aug 1993.

[6] Prabuddha Biswas, K. K. Ramakrishnan, and Don Towsley. Trace driven analysis

of write caching policies for disks. *Performance Evaluation Review*, 21(1):12–23, Jun 1993.

[7] Prebuddha Biswas, K. K. Ramakrishnan, Don Townsley, and C. M. Krishna. Performance analysis of distributed file systems with non-volatile caches. In *Proceedings of 2nd International Symposium on High Performance Distributed Computing*, pages 252–62. IEEE Computer Society Press, 1993.

[8] Scott Carson and Sanjeev Setia. Analysis of the periodic update write policy for disk cache. *IEEE Transactions on Software Engineering*, 18(1):44–54, Jan 1992.

[9] Scott Carson and Sanjeev Setia. Optimal write batch size in log-structured file systems. In *Proceedings of USENIX Workshop on File Systems*, pages 29–91. USENIX Association, May 1992.

[10] Kerhong Chen, Richard B. Bunt, and Derek L. Eager. Write caching in distributed file systems. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 457–66. IEEE, Jun 1995.

[11] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The Rio file cache: surviving operating system crashes. *SIGPLAN Notices*, 31(9):74–83, Sep 1996.

[12] Yu-Ping Cheng and David Hitz. High-performance non-volatile RAM protected write cache accelerator employing dma and data transferring scheme. United States patent 5701516, Patent and Trademark Office, Dec 1997.

[13] Edward G. Coffman, Jr. and Peter J. Denning. *Operating systems theory*. Prentice-Hall, 1973.

[14] G. Copeland, T. Keller, R. Krishnamurthy, and M. Smith. The case for safe RAM. In *Proceedings of the 15th International Conference on Very Large Databases*, pages 327–35. Morgan Kaufmann, Aug 1989.

[15] Thomas H. Corman, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to algorithms*. McGraw-Hill Publishing Company, 1989.

[16] Peter J. Denning. Effects of scheduling on file memory operations. In *Proceedings of AFIPS Spring Joint Computer Conference*, pages 9–21. AFIPS Press, Apr 1967.

[17] Paul A. Fishwick. Simpack: getting started with simulation programming in C and C++. Technical Report 92–022, University of Florida, Department of Computer Science and Information Science and Engineering, Gainesville, FL, Jul 1992.

[18] Yigal Gerchak and Xinjian Lu. Optimal anticipatory position of a disk arm for queries of random length and location. *INFOR*, 34(4):251–61, Nov 1996.

[19] J. Gray and F. Putzolu. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time. *SIGMOD Record*, 16(3):395–8, Dec 1987.

[20] Jim Gray and Andreas Rueter. *Transaction processing: concepts and techniques*. Morgan Kaufmann Publishers, 1993.

[21] D. D. Grossman and H. F. Silverman. Placement of records on a secondary starage device to minimize access time. *Journal of the ACM*, 20(3):429–38, Jul 1973.

[22] Theodore R. Haining and Darrell D. E. Long. Management policies for non-volatile write caches. In *1999 IEEE International Performance, Computing and Communications Conference*, pages 321–8. IEEE, Feb 1999.

[23] Theodore R. Haining, Jehan-François Pâris, and Darrell D. E. Long. A stack-based method for non-volatile cache management. In *Proceedings of the Eighth NASA Goddard Mass Storage Conference*, pages 217–23. IEEE, March 2000.

[24] D. Hitz, J. Lau, and M. Malcolm. File system design for a NFS server appliance. In *Proceedings of the Winter 1994 USENIX Conference*, pages 235–46. USENIX Association, Jan 1994.

[25] Robert Y. Hou and Yale M. Patt. Using non-volatile storage to improve the reliability of RAID5 disk arrays. In *Proceedings of the 27th Annual International Symposium on Fault-Tolerant Computing*, pages 206–15. IEEE Computer Society, Jun 1997.

[26] Yuming Hu and Qing Yang. DCD – disk caching disk: a new approach for boosting I/O performance. *Computer Architecture News*, 24(2):169–78, May 1996.

[27] Yuming Hu and Qing Yang. A new hierarchical disk architecture. *IEEE Micro*, 18(6):64–76, Nov-Dec 1998.

[28] Yuming Hu, Qing Yang, and Tycho Nightingale. Rapid-cache – A reliable and inexpensive write cache for disk I/O systems. In *Proceedings Fifth International Symposium on High-Performance Computer Architecture*, pages 204–13. IEEE, Jan 1999.

[29] David M. Jacobson and John Wilkes. Disk scheduling algorithms based on rotational position. Technical Report HPL–CSP–91–7, Hewlett-Packard Corporation, Concurrent Systems Project, Palo Alto, CA, Feb 1991.

[30] Richard P. King. Disk arm movement in anticipation of future requests. *ACM Transactions on Computer Systems*, 8(3):214–29, Aug 1990.

[31] David Kotz, Song Bac Toh, and Sriram Radhakrishnan. A detailed simulation model of the HP 97560 disk drive. Technical Report PCS–TR94–220, Department of Computer Science, Dartmouth College, 1994.

[32] David E. Lowell and Peter M. Chen. Free transactions with Rio Vista. In *16th Association for Computing Machinery Symposium On Operating Systems Principles*. Association for Computing Machinery, Oct 1997.

[33] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–97, Aug 1984.

[34] Jai Menon and Jim Cortney. The architecture of a fault-tolerant cached RAID controller. *Computer Architecture News*, 21(2):76–86, May 1993.

[35] Joseph P. Moran, Russel P. Sandberg, and Donald C. Coleman. Method and apparatus for enhancing synchronous I/O in a computer system with a non-volatile memory and using an acceleration device driver in a computer operating system. United States patent 5359713, Patent and Trademark Office, Washington, D.C., Oct 1994.

[36] Wee Teck Ng and Peter M. Chen. Integrating reliable memory in databases. In *Proceeddings of the 23rd International Conference on Very Large Data Bases*, pages 76–85. Morgan Kaufmann, Aug 1997.

[37] Cyril U. Orji and Jon A. Solworth. Write-only disk cache experiments on multiple surface disks. In W. W. Koczkodaj, P. E. Lauer, and A. A. Toptsis, editors, *Proceedings Fourth International Conference on Computing and Information*, pages 385–8. IEEE Computer Society Press, 1992.

[38] Cyril U. Orji and Jon A. Solworth. Write twice disk buffering. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 27–34. IEEE Computer Society Press, Sep 1994.

[39] John Ousterhout and Fred Douglis. Beating the I/O bottleneck: a case for log-structured file systems. *Operating Systems Review*, 23(1):11–28, Jan 1989.

[40] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. *Operating Systems Review*, 19(5):15–24, 1985.

[41] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference*, pages 109–16. Association for Computing Machinery, Jun 1988.

[42] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of 13th ACM Symposium on Operating*

Systems Principles, pages 1–15. Association for Computing Machinery SIGOPS, Oct 1991.

[43] Chris Ruemmler and John Wilkes. Disk shuffling. Technical Report HPL–91–156, Hewlett-Packard Laboratories, Oct 1991.

[44] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, Mar 1994.

[45] Chris Ruemmler and John Wilkes. Unix disk access patterns. In *USENIX Technical Conference Proceedings*, pages 405–20. USENIX, Winter 1993.

[46] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. In *Proceedings of the 1990 Winter Usenix*, pages 313–24. USENIX, Jan 1990.

[47] Barbara Tockey Sivkov and Alan Jay Smith. Disk caching in large databases and timeshared systems. Technical Report CSD-96-913, University of California, Berkeley, Berkeley, CA 94720-1776, Sep 1996.

[48] Alan Jay Smith. Disk cache – miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, 3(3):161–203, Aug 1985.

[49] Jon A. Solworth and Cyril U. Orji. Write-only disk caches. *SIGMOD Record*, 19(2):123–32, Jun 1990.

[50] Michael Stonebraker. The design of the Postgres storage system. In *Proceedings of 13th International Conference on Very Large Data Bases*, pages 289–300. Morgan Kaufmann, Sep 1987.

[51] Anujan Varma and Quinn Jacobson. Destage algorithms for disk array with non-volatile caches. *IEEE Transactions on Computers*, 47(2):228–35, Feb 1998.

[52] P. Vongsathorn and S. D. Carson. A system for adaptive disk rearrangement. *Software Practice and Experience*, 20(3):225–42, Mar 1990.

[53] C. K. Wong. Minimizing expected head movement in one-dimensional and two-dimensional mass storage systems. *Computing Surveys*, 12(2):167–78, Jun 1980.

[54] Michael Wu and Willy Zwaenepoel. eNVy: A non-volatile, main memory storage system. *SIGPLAN Notices*, 29(11):86–97, Nov 1994.

[55] Qing Yang and Yuming Hu. System for destaging data during idle time by transferring to destage buffer, marking segment blank, reordering data in buffer, and transferring to beginning of segment. United States patent 5754888, Patent and Trademark Office, Washington, D.C., 1998.