Storage Management for Large Scale Systems

A Thesis Submitted to the College of

Graduate Studies and Research

in Partial Fulfillment of the Requirements

For the Degree of Doctor of Philosophy

in the Department of Computer Science

University of Saskatchewan

Saskatoon, Saskatchewan

by

Wenguang Wang

Permission To Use

In presenting this thesis in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
University of Saskatchewan
Saskatoon, Saskatchewan, Canada
S7N 5A9

Abstract

Because of the slow access time of disk storage, storage management is crucial to the performance of many large scale computer systems. This thesis studies performance issues in buffer cache management and disk layout management, two important components of storage management.

The buffer cache stores popular disk pages in memory to speed up the access to them. Buffer cache management algorithms used in real systems often have many parameters that require careful hand-tuning to get good performance. A self-tuning algorithm is proposed to automatically tune the page cleaning activity in the buffer cache management algorithm by monitoring the I/O activities of the buffer cache. This algorithm achieves performance comparable to the best manually tuned system.

The global data structure used by the buffer cache management algorithm is protected by a lock. Access to this lock can cause contention which can significantly reduce system throughput in multi-processor systems. Current solutions to eliminate lock contention decrease the hit ratio of the buffer cache, which causes poor performance when the system is I/O-bound. A new approach, called the multi-region cache, is proposed. This approach eliminates lock contention, maintains the hit ratio of the buffer cache, and incurs little overhead. Moreover, this approach can be applied to most buffer cache management algorithms.

Disk layout management arranges the layout of pages on disks to improve the disk I/O efficiency. The typical disk layout approach, called Overwrite, is optimized for sequential I/Os from a single file. Interleaved writes from multiple users can significantly decrease system throughput in large scale systems using Overwrite. Although the Log-structured File System (LFS) is optimized for such workloads, its garbage collection overhead can be expensive. In modern and future disks, because of the much faster improvement of disk transfer bandwidth over disk positioning time, LFS performs much better than Overwrite in most workloads, unless the disk is close to full. A new disk layout approach, called HyLog, is proposed. HyLog achieves performance comparable to the best of existing disk layout approaches in most cases.

Acknowledgements

The writing of a Ph.D. thesis can be a daunting and tasking experience. Fortunately, I was lucky to get guidance, friendship and support from many people.

I would like to express my sincere appreciation to my supervisor, Dr. Rick Bunt, for his tremendous guidance, encouragement, and support for my Ph.D. research. During my six years of study, he challenged my ideas, encouraged me to pursue whatever caught my fancy, and gave me the freedom to make my own discoveries and mistakes. He not only guided my research project, but also taught me the methodology and ethics of doing research. His way of promoting ideas to a higher level and handling things in a very organized way also affected me greatly. I believe what I have learned from him will continue to benefit my career. He sponsored me to attend many of the best conferences in systems research (such as SIGMETRICS, SOSP, USENIX, FAST) when I don't have publications there. These are opportunities that most graduate students in the world don't have. These invaluable experiences greatly opened my mind and inspired new ideas.

This thesis research started from a project supported by IBM's Centre for Advanced Studies (CAS). During my eight month stay at the Toronto Lab, I got tremendous help from many people in the DB2 performance group and the DB2 development group. It was a wonderful place to work and an exciting experience for me. I am grateful to many people in IBM CAS for their help and support. In particular I would like to thank Berni Schiefer, Kelly Lyons and Joe Wigglesworth for providing a nice environment and giving me the support for my research. I would like to thank Steve Rees, Keri Romanufa, Aamer Sachedina, Yongli An, John Li, Larry Menard and many other people who answered numerous technical questions. I am especially thankful to Miso Cilimdzic, Wing Lau, Garfield Lewis and Peter Shum, for their help on porting a trace package. And I would like to thank Chantal Buttery for her caring approach to students.

I would like to thank Dr. Derek Eager in my DISCUS research group. As one of my committee members, he gave me many insightful comments when I encountered

difficulties in the research. His suggestions on my thesis were also very helpful. I would like to thank my other committee members, Dr. Hanan Lutfiyya (external), Dr. Raj Srinivasan (cognate) and Dr. Ralph Deters for there valuable comments.

I would like to thank Dr. Dwight Makaroff in my DISCUS research group for his support and his feedback on my research.

I would like to thank Kevin Froese in my group for developing some initial tools and simulators, and Lixin Wang for the helpful discussions.

The experimental parts of my research required an enormous amount of computing resources. I was fortunate to get all the help I needed from the excellent system administrators of our department and the DISCUS group. I am especially thankful to Greg Oster, Brian Gallaway, Cary Bernarth and Dave Bocking. Besides system administration, Greg Oster also answered many technical questions and gave me helpful feedback on my research.

The office staff members of my department have been wonderful. Their prompt support to my queries made my life much easier. I especially want to thank Jan Thompson, our graduate correspondent, for her love and caring during my study.

I am grateful to many people outside of the University for various types of help in my research. Specially I want to thank Edward Lee from UC Berkeley for his comments on my paper, Greg Ganger from Carnegie Mellon for answering many of my questions, Daniel Ellard from Harvard for providing the NFSEmail traces, Said Elnaffar from Queen's University for providing the TPC-W benchmark suite, Ken Bates from the Storage Performance Council for answering questions about the Financial and WebSearch traces, and Song Jiang from the College of William and Mary for providing the LIRS simulator and test traces.

The TPC-H and openmail traces were provided by John Wilkes and Hernan Laffitte from HP Labs.

Financial support for this thesis research came from IBM's Centre for Advanced Studies (CAS) and the Natural Sciences and Engineering Research Council of Canada (NSERC).

Finally, I would like to give my wife, Dr. Yanping Zhao, my deepest

appreciation for her love, support and collaboration with my research. She has always been my best friend and research partner. I also want to thank my parents and younger sister for their support of my studies.

Table of Contents

List of Tables

List of Figures

List of Acronyms

2Q – Two queue replacement algorithm

DBMS – Database management system

DGCLOCK – Dynamic generalized clock replacement algorithm

DMA – Direct memory access

DSS – Decision support system, also called OLAP system

E-commerce – Electronic commerce

EELRU – Early eviction least recently used replacement algorithm

FBR – Frequency-based replacement algorithm

FFS – Fast file system

GB – Gigabytes (2^{30} bytes)

GCLOCK – Generalized clock replacement algorithm

HyLog – Hybrid log-structured disk layout

ILRU – Inverse least recently used replacement algorithm (for caching B^+-tree pages in DBMS indexes)

IRR – Inter-reference recency, used by the LIRS replacement algorithm

KB – Kilobytes (2^{10} bytes)

LFS – Log-structured file system

LFU – Least frequently used replacement algorithm

LRFU – Least recently/frequently used replacement algorithm

LIRS – Low inter-reference recency set replacement algorithm

LRU – Least recently used replacement algorithm

MB – Megabytes (2^{20} bytes)

MRU – Most recently used replacement algorithm

OLAP – On-line analytical processing, also called decision support

OLTP – On-line transaction processing

OLRU – Optimal least recently used replacement algorithm (for caching B^+-tree pages in DBMS indexes)

QLSM – Query locality set model

SCSI – Small computer systems interface

SMP – Symmetric multi-processor

TPC – Transaction Processing Performance Council

TPC-B – TPC benchmark B, a database stress test benchmark with OLTP-like workload (obsolete as of June 6, 1995)

TPC-C – TPC benchmark C, an OLTP benchmark

TPC-H – TPC benchmark H, an ad-hoc decision support benchmark

TPC-W – TPC benchmark W, a transactional web e-commerce benchmark

Chapter 1

Introduction

1.1 Background and Motivation

Storage is an important part of all large computer systems. Storage is where data are stored persistently over a long period of time. It represents 40%-60% of total hardware cost, and its management cost comprises 60%-80% of the total cost of ownership [2]. Storage is often the performance bottleneck of the system, especially for large scale systems accessing large amounts of data, such as storage servers, file servers, web servers, email servers, and database servers.

Storage servers, such as the EMC Symmetrix series [118], provide disk storage to other systems. A storage server typically divides the storage into volumes, and provides an abstraction of pages with linear addresses in each volume. Storage servers often provide raw storage services to file servers and database servers. File servers, such as the FAS series of NetApp Inc. [81], provide file services to other systems. A file server maintains the metadata of a file system (bitmaps for space allocation, directories for data organization, etc.) and provides security and data access consistency. File servers often provide file services to other systems, such as web servers, email servers, and database servers.

In all these systems, data are stored on magnetic disks which are cheap but slow to access. Storage management is used in these systems to speed up accesses to data on disks. Storage management employs an in-memory buffer to cache popular disk pages, and also manages how data are placed on disks. The part of the system controlled by storage management is called the storage subsystem, whose performance is often crucial to the performance of the whole system.

Recent years have seen the capacity and the bandwidth of memory and disks increase dramatically, with their prices dropping greatly [43]. As a result, the performance of the storage subsystem in small systems is much less an issue, since adequate performance can usually be achieved by provisioning more advanced hardware. However, large enterprise systems are supporting more users, larger data sets, and new applications, as the storage technology improves. The demand for high performance of storage subsystems in enterprise systems continues to increase. Many of the performance requirements cannot be met simply by deploying more hardware because the cost would become unbearably high or no existing hardware could satisfy the needs. The goal of this thesis research is to investigate techniques to improve the performance of storage subsystems in large scale systems.

A good understanding of the characteristics of workloads running on the storage subsystem is a prerequisite to its performance study. The workloads running on the storage subsystem vary substantially. Typical types of workload include research workloads [32], email workloads [32], web server workloads [113], and database workloads [52, 113].

Studies on file server and email server workloads [32, 94, 129] have found that:

- A large proportion of the requests to the storage subsystem are writes, because many reads can be cached at the client side.
- Many reads and writes are random.
- Sequential reads and writes do not typically span a large number of pages, since most files are small.

These are also the workload characteristics of storage servers which reside below file servers and email servers [113, 137, 138, 141].

Database workload is an important class of workload in the storage subsystem. Typical database systems can be roughly divided into two categories: on-line transaction processing (OLTP) and decision support. As a new way of doing business through the Internet, e-commerce applications play an increasingly

important role in database applications. The workload characteristics to the database server of an e-commerce system lie between that of OLTP and decision support [33].

In a typical decision support system (DSS), the user queries the data in the database to search for certain patterns, answer some business questions, or predict future business patterns. The results of these queries can help managers to make better decisions. Typically these queries are complex. They read a large amount of data in the database. Normally, queries are read-only, and only one user runs the query. Since a large amount of data is often read sequentially from the disks, the storage subsystem can easily achieve its maximum transfer throughput. The performance bottleneck of a DSS is often the CPU instead of the storage subsystem.

OLTP is a class of application that manages data entry retrieval transactions in many industries, including banking, supermarkets, airlines, rental services, and manufacturing. A typical OLTP workload contains simple transactions from many terminals with strict response time requirements (several seconds). Many of these transactions update the database. In OLTP workloads, traffic to disks is characterized by small random reads and writes but few sequential I/Os. Storage is often the bottleneck in such a system.

The workload of the storage subsystem in many large scale systems, such as storage servers, file servers, email servers, and OLTP systems, is characterized by small reads and updates to a large amount of data on disks. Effective management in the storage subsystem can significantly improve the overall performance of such systems. These workloads are the main focus of this thesis research.

Since the storage subsystem is often complex, studying it directly in the context of a real system is difficult. A combination of analysis, trace-driven simulation, and measurement approaches is used in this research. First, direct measurement experiments are performed on real systems to understand the important behavioural elements of a real system. Then, analytical models and trace-driven simulators are developed (and verified) to study key parts of the

storage subsystem. Next, new algorithms are designed and evaluated using analytical models and simulation. Finally, the new algorithms are implemented and evaluated in real systems to confirm the modelling and simulation results. Use of analytical models and simulations dramatically reduces the effort that would be needed to handle complex real systems.

1.2 Storage Management Overview

Data are placed on secondary storage (i.e., the disks) and managed by the storage subsystem. An important design objective of storage subsystems is to provide fast access to the data that are stored. Disks have different performance characteristics than memory. Although disks are cheap and large (e.g., many giga-bytes or tera-bytes), their access time is slow (e.g., orders of milliseconds) compared with that of memory (e.g., orders of nanoseconds). Disk has high maximum transfer bandwidth, but it is achievable only when accessing large chunks of data sequentially. These disk characteristics determine the common approaches used in storage management to improve performance.

Storage management can be divided into four functional layers, as illustrated in Figure 1.1:

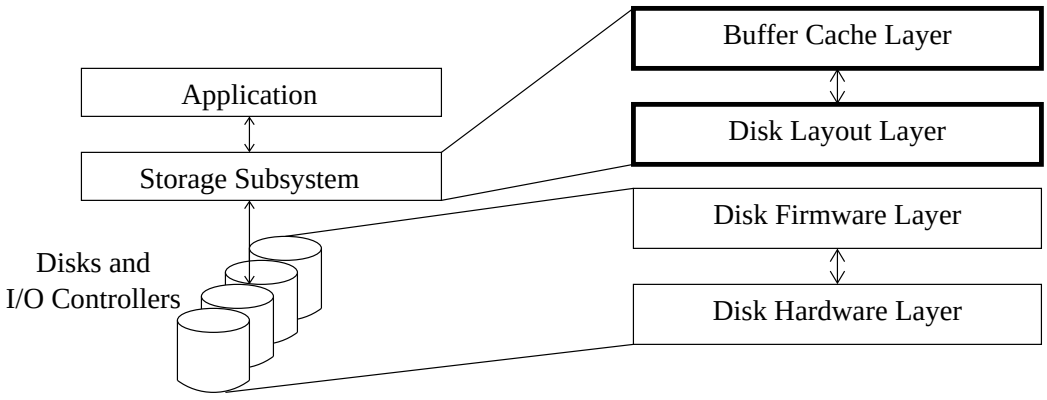


Figure 1.1: Storage Management in DBMS
[The two layers within bold boxes are the focus of this thesis study.]

1. The **buffer cache layer** dedicates an in-memory buffer called the *buffer cache* to store popular disk pages. A buffer cache management algorithm is used to decide what pages are kept in the buffer cache. The performance of the buffer cache layer is crucial to that of a large scale system.

2. The **disk layout layer** manages the on-disk data structures and data placement strategies. The goal of this layer is to improve the I/O efficiency by reducing the mechanical movement of the disk arms. This layer has been studied extensively in the design of file systems [40, 74, 75, 96, 140].

3. The **disk firmware layer** utilizes the physical properties of a disk to reduce the average data access time. A number of approaches, such as disk cache, disk scheduling, and Redundant Array of Independent Disks (RAID) [89], have been proposed for this layer. This layer often resides in disk firmware or RAID controllers, where the designers of large scale systems typically do not have control. Moreover, it is difficult to improve the disk I/O efficiency within the physical disk layer if the upper layers are not properly designed. This layer is not studied in this thesis research.

4. The **disk hardware layer** is the actual hardware of the disk, including platters, disk arms, magnetic surfaces, etc. The hardware technology, especially the storage density of the disk has been constantly improving in the past. However, because of engineering and manufacturing constraints, this improvement is taking place at a slower rate than that predicted by Moore's Law (the speed or capacity doubles every 18 months), which predicts the improvement of silicon chips (memory and CPU).

1.2.1 The Buffer Cache Layer

The buffer cache layer plays a critical role in a storage management subsystem. A buffer cache management algorithm is used to manage the pages in the buffer cache. The access to data often exhibits temporal locality, i.e., a small amount of

data are frequently re-referenced during a short interval [29]. By exploiting this locality, a buffer cache with a relatively small size compared to that of the disks can absorb most I/O requests to the disks. As a result, the storage subsystem can achieve an average access time close to that of the memory while keeping the average storage cost close to that of the disks.

The main functions of the buffer cache management algorithm include the standard functions – fetch, placement, and replacement. The fetch algorithm brings new pages into the buffer cache. Normally, fetch on demand is used, but when there are sequential accesses to pages, prefetch can be a helpful supplement to fetch on demand. The placement algorithm determines where a page is placed in the buffer cache. Its decision is often determined by the replacement algorithm that decides how the free pool of the buffer is maintained. Replacement on demand makes a free space only when needed. Pre-replacement tries to always keep some spaces available by writing back to disk the changed pages that are deemed not to be needed. Since the replacement function is crucial to buffer cache management, a buffer cache management algorithm is often called a buffer cache replacement algorithm.

Although many replacement algorithms have been described in the literature[1], real systems tend to use very simple algorithms such as LRU [80, 91, 105, 119] or CLOCK [46] for the following reasons:

- *Tuning difficulties.* Most advanced algorithms require the proper setting of one or more parameters. The tuning of these parameters is often done by trial and error. Since real systems already have many parameters to tune, simple replacement algorithms like LRU or CLOCK are often selected to simplify the tuning of the buffer cache layer.
- *Unclear advantage under large buffer cache sizes.* Previous studies of replacement algorithms show that the hit ratios of all replacement algorithms

[1]For example, CAR and CART [8], ARC [76], LIRS [59], LRFU [66], EELRU [112], SEQ [41], 2Q [60], LRU-K [84], FBR [93], Application/File-level Characterization [20], Unified Buffer Management [64], ILRU and OLRU [98], HotSet [99], QLSM and its variants [15, 21, 34, 67], Page Fault Frequency [22], Working Set [29], and Cache Partitioning [116]

converge on large buffer caches, which are used in large scale systems. One may be tempted to conclude that the performance benefit of advanced algorithms over simple algorithms becomes small under large buffer caches, which reduces the incentive of adopting advanced algorithms in real systems.

- *Access contention for concurrent accesses.* Large database servers running OLTP or e-commerce applications often support a large number of concurrent users. When multiple users access the buffer cache at the same time, the global data structure that the buffer cache replacement algorithm employs must be locked to avoid corruption. This lock may become a contention point which limits the system throughput dramatically. Although the buffer cache management in DBMSs are managed by software and any replacement algorithms can be used, systems such as SQL Server 7.0 use CLOCK-based replacement algorithms, which do not change the global data structure on buffer cache hits, in spite of hit ratios that are lower than other replacement algorithms [25].
- *Limitation of hardware support.* In virtual memory management, buffer cache hits must be managed by hardware. For every buffer cache hit, current hardware sets the reference bit and the changed bit if this page is modified. Modern operating systems, such as FreeBSD, NetBSD, Linux, Mac OS X, and Windows 2000/XP, employ unified buffer caching [109], which unifies virtual memory management and the file system buffer cache. Only CLOCK-based algorithms can be used in these systems to cache the file system data in the unified buffer cache.

1.2.2 The Disk Layout Layer

Disks have very different access characteristics from memory. Disks have high maximum transfer bandwidth, which can be achieved only when transferring large blocks of data sequentially. When accessing data scattered over the disk, a great deal of time is spent on the mechanical movements of disk arms and rotations of

disk platters, resulting in less than 10% disk transfer bandwidth being utilized. Disk layout management decides where data are placed on the disks to improve the utilized disk transfer bandwidth. With efficient disk layout management, many small data transfers on different locations of the disk can be merged into a few large transfers, and the mechanical delay between accesses can be significantly reduced. As a result, the access performance can be dramatically improved.

Many disk layout approaches are optimized for sequential read within a file or files of one directory [40, 74, 75, 140]. Data belonging to one file or one directory are placed close to each other on disks. Since changed data are overwritten on top of old copies (called *Overwrite*), the write performance is also improved when the I/O pattern of writes resembles that of reads. Such disk layout approaches work well in systems with a small number of active applications. Large scale systems need to support many users who access multiple files at the same time. The interleaved requests from different users generate disk I/Os scattered over the disk, which dramatically reduces the disk performance. These disk I/O requests appear to be random at the disk level. Since read requests can be effectively cached by the buffer cache layer, these random I/Os are mainly write requests [88].

The Log-structured File System (LFS) [96, 106] was designed to improve the write performance for workloads with random updates while maintaining comparable read performance. Instead of using Overwrite to handle updates, LFS accumulates small random writes and writes them to the disk in a large contiguous write. LFS has the potential to achieve good write performance [96], but, since the data are written to a new location every time they are updated, their old copies must be reclaimed by a garbage collection process called *segment cleaning*. In OLTP-like workloads, segment cleaning results in high overhead and severely decreases the overall system performance [108].

1.3 Contributions of the Thesis

This thesis studies the storage subsystem on large scale systems, addressing several performance issues of the buffer cache layer and the disk layout layer. This includes analyzing the characteristics of typical workloads running on large scale systems, identifying the performance bottleneck of the storage subsystem, modeling and simulating key components of the storage subsystem, and designing and evaluating new algorithms which can ease the tuning task or achieve better performance. Direct experimentation, trace-driven simulation, and mathematical modeling are used in this study. The main contributions of the thesis are:

- A new algorithm is proposed to automatically tune parameters of buffer cache management in order to achieve good performance. Its effectiveness is comparable to the best manually tuned algorithm.
- The problem of lock contention in buffer cache management is investigated. A new approach called the *multi-region* cache is proposed to eliminate the lock contention of the buffer cache. This approach can work together with most buffer cache replacement algorithms. It does not compromise the overall hit ratio of the buffer cache, and incurs little overhead.
- Different disk layout management approaches are modeled and their performance characteristics are analyzed. A new approach called *HyLog* is proposed. HyLog achieves performance close to the best of existing approaches in most configurations.

1.3.1 Self-tuning of Buffer Cache Management

A buffer cache management algorithm in a real system often has many parameters that need to be tuned for the particular workload and system configuration at hand. However, such tuning is often not easy. The buffer cache management algorithm in IBM's DB2 database management system employs page cleaners to write the changed pages (dirty pages) in the buffer cache asynchronously so that

expensive synchronous writes can be avoided when the space occupied by these pages are needed. Through measurement and trace-driven simulation, it was found that the number of page cleaners is important to the throughput of the overall system, but it is difficult to tune this parameter manually. A self-tuning algorithm is proposed to automatically tune the page cleaning speed. Simulation results show that this algorithm can achieve throughput comparable to the best manually tuned system.

1.3.2 Lock Contention of Buffer Cache Management

Most buffer cache replacement algorithms use a global data structure to manage all pages in the buffer cache. In large scale systems with many processors and threads, different threads may access the buffer cache simultaneously. The global data structure of the replacement algorithm must be protected by a lock to avoid corruption. This lock may become the contention point (called lock contention) and limit the system throughput. Real systems solve the contention problem in different ways, depending on their particular requirements. SQL Server 7.0 uses a CLOCK-based algorithm to reduce contention [25], since this kind of algorithm does not modify the global data structure on buffer cache hits. However, a CLOCK-based algorithm typically has a lower hit ratio than other replacement algorithms and may cause poor system performance. Berkeley DB and ADABAS use different variations of LRU without global data structures [10, 105], but these approaches either have high overhead or cannot be applied to other replacement algorithms. These current practices in buffer cache management motivate the search for a better approach to reduce lock contention without compromising the overall hit ratio.

A new buffer cache management approach, called the *multi-region* cache, is proposed for this purpose. The multi-region approach divides the buffer cache into many fixed-size regions, each of which is managed by an instance of a replacement algorithm. Each page is mapped into a unique region. Since lock contention

happens only when pages within the same region are accessed at the same time, and there are a large number of regions, lock contention almost never happens in the multi-region cache. Both analysis and simulation results show that a large buffer cache with hundreds of thousands of regions has almost the same overall hit ratio as the traditional approach, and the overhead is negligible. Multi-region cache can be applied to most replacement algorithms.

1.3.3 Disk Layout Management

With a large buffer cache, most disk reads can be resolved in memory [88]. As a result, in many systems, write requests make up a large portion of the total traffic to the disks [32, 113]. These write requests are from many users and often scatter over the disks, which results in low utilization of the disk transfer bandwidth when the disk layout is managed by the commonly used Overwrite approach. LFS was designed to provide good write performance while maintaining comparable read performance in such systems, but, the high segment cleaning overhead of LFS decreases its performance dramatically [108].

The write performance of Overwrite and LFS is modeled and the impact of changing disk technology on their performance is investigated. Because of the much faster improvement in disk transfer bandwidth than disk positioning time [43], it is found that LFS significantly outperforms Overwrite under modern and future disks over a wide range of system configurations and workloads.

LFS performs worse than Overwrite, however, when the disk space utilization is very high because of the high segment cleaning cost. A new approach, the Hybrid Log-structured (HyLog) disk layout, is proposed to overcome this problem. HyLog uses a log-structured approach for hot pages to achieve good write performance, and Overwrite for cold pages to reduce the segment cleaning cost. An adaptive separating algorithm is designed to separate hot pages from cold pages under various workloads and system configurations. This algorithm works in real time and incurs little overhead. Simulation results under a range of system

configurations and workloads show that, in most cases, HyLog performs comparably to the best of the existing disk layout approaches.

1.4 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 discusses related research pertaining to the storage subsystem. Chapter 3 describes the methodology, the typical workloads, and the characteristics of the workloads. Chapter 4 studies the self-tuning of buffer cache management. The design and evaluation of a self-tuning algorithm for buffer cache management are presented. Chapter 5 discusses lock contention in buffer cache management. The problem of lock contention is identified and analyzed. A new approach called multi-region cache is proposed and evaluated. Chapter 6 investigates performance issues of disk layout management. The performance of existing approaches is modeled and analyzed. A new disk layout approach called HyLog is proposed and its performance is studied by analysis and simulation. Finally, Chapter 7 summarizes the thesis and gives directions for future work.

Chapter 2

Related Research

Magnetic disk remains the dominant medium for secondary storage. In large scale systems, large amounts of data (from many giga-bytes to many tera-bytes) reside in disk storage. Since the access time of disks is much slower than memory, disk access is a performance bottleneck in many large systems. Numerous approaches have been proposed to improve the performance of the storage subsystem. They can be categorized roughly into four layers:

- The *Hardware Layer* includes the hardware of the disks. Improvements in hardware architecture and engineering make the disks faster, which improves the overall performance of the system.
- The *Firmware Layer* resides in either disks or disk controllers. This layer organizes the disk I/O requests to achieve good disk I/O performance. Typically the firmware layer does not have knowledge of the data semantics of the application layer.
- The *Disk Layout Layer* organizes the data placement strategies on disks to improve disk I/O efficiency. This layer normally has some knowledge of the semantics of the application data.
- The *Buffer Cache Layer* uses a fast in-memory buffer cache to store popular disk requests so that the number of requests to disks is reduced.

Previous research on improving the performance of the storage subsystem is organized into these layers and briefly discussed in Section 2.1, 2.2, 2.3 and 2.4. Some approaches that exploit the knowledge of different layers are discussed in

Section 2.5. Since the hardware and firmware layers reside in the storage hardware, typically they cannot be controlled by designers of large scale systems. Therefore, the other two layers, the disk layout layer and the buffer cache layer, are the focus of this thesis study.

2.1 The Hardware Layer

A magnetic disk consists mainly of a set of parallel platters constantly rotating around a fixed axle and a movable read/write head on each surface of the platter. Data are stored in concentric *tracks* on the platters. In each track, data are stored in fixed size sectors, each of which typically contain 512 bytes data. All read/write heads are attached to a disk arm, which can move to different tracks. Once the disk arm moves to the desired track, it waits until the desired sector rotates under the read/write head before data on this sector can be accessed. The same tracks on all platters compromise a *cylinder*.

The latency of a disk access can be broken down into three main parts: *seek time*, *rotational latency*, and *transfer time*. Seek time is the time that the disk arm takes to move to the desired track. This may involve a mechanical movement of the disk arm. Rotational latency is the time that the read/write head waits for the desired sector to rotate under it. The sum of seek time and rotational latency is called the *positioning time*. Transfer time is the time it takes for the desired sectors to pass under the read/write head, which is affected by the rotational speed, the circumference and storage density.

Since three components of the disk access time all involve mechanical movements, reducing them is constrained by engineering and manufacturing limitations. Memory capacity and CPU speed improve about 60% every year, following Moore's Law (i.e., doubling every 18 months). Thanks to the fast improvement of storage density, disk capacity increases at the same rate (60% every year) [43], and sequential transfer bandwidth improves 28% every year [43]. The average seek time decreases 12% every year [101], and the rotational speed

increases 10% every year [3]. Under this technology trend, the performance difference between disks and silicon chips (memory and CPU) becomes larger and larger. The performance advantage of conducting sequential disk I/Os over random disk I/Os also becomes larger and larger. Because of this characteristic, most strategies to improve the performance of disk storage attempt to reduce the seek time and rotational latency between requests.

Since accessing data sequentially on a cylinder is the fastest, the sectors are numbered in a way so that when all sectors in the disk are accessed sequentially, sectors in the outermost cylinder are accessed first, sectors in the second outermost cylinder are accessed next, and so on. Under this sector addressing scheme, the performance specification of the disks can be summarized as the *unwritten contract* between hosts and disks [103]:

- Sequential accesses have the best performance, much better than non-sequential.
- Accessing a sector with an address close to the previously accessed sector is usually faster than accessing a sector with an address that's far away.
- Ranges of sector addresses are interchangeable, and the absolute sector addresses do not affect performance.

This unwritten contract is used by system designers to improve the performance of disk storage.

Several new secondary storage technologies are being actively researched. A new storage technology based on microelectromechanical systems (MEMS), for example, employs many small mechanical probe tips to access magnetic surfaces. MEMS-based storage could provide access times 6.5X faster than disks [44], and storage systems partially using MEMS-based storage could provide dramatically better performance and cost/performance [128].

2.2 The Firmware Layer

The firmware layer resides in disks or disk controllers. It organizes disk requests to achieve better performance. The common approaches used in the firmware layer include disk scheduling, disk cache, and disk array.

2.2.1 Disk Scheduling

In large scale systems, typically many requests are pending at the disk. Without disk scheduling, these requests are served in a first-in-first-out order. When the requests are scattered over the disk, most of the access time is spent in the seek time and the rotational latency. A disk scheduling algorithm reorders the requests to achieve good I/O throughput [115]. Since algorithms generating high I/O throughput often increase the maximum response time by delaying some requests, tradeoffs between these two factors should be considered.

The simplest scheduling algorithm is first-in-first-out (FIFO), which does not conduct any scheduling. Many scheduling algorithms attempt to reduce the seek time. The Shortest Seek Time First (SSTF) algorithm always serves the first request which incurs the shortest seek time [28]. The problem of SSTF is that it may always select requests close to the current disk head position and starve other requests. The SCAN algorithm can prevent this kind of starvation [28]. In the SCAN policy, the disk head moves in one direction to satisfy all waiting requests. It changes the direction of head movement only at the innermost and outermost cylinder. The disk head then moves in the reverse direction and again picking up all requests in order. LOOK, a variant of SCAN, changes direction if no more requests exist in the current direction [79]. SCAN and LOOK favour requests in the middle of the disk and requests on the outer and inner cylinders have low priority. Cyclical SCAN (C-SCAN) and cyclical LOOK (C-LOOK) treat all cylinders equally by fulfilling requests only when the head is moving in one direction. VSCAN(R) gives a continuum of algorithms between SSTF and LOOK [39]. R denotes how likely the scheduler wants to maintain the current

direction of travel. VSCAN(0) is SSTF, and VSCAN(1) is LOOK. VSCAN(0.2) was suggested to be a good balance between the average response time and starvation resistance [39].

The Shortest Positioning Time First (SPTF) algorithm [56, 107, 139] takes into account both seek time and rotational latency when selecting the next request. It gives better performance than algorithms considering only seek time [107, 139], but the performance advantage of SPTF over other algorithms is limited under modern and future disks [54]. Moreover, the computation overhead of SPTF is high [54, 139] and must recognize the existence of disk cache and optimize for it [139].

All the above algorithms may favour the latest arriving jobs and starvation may happen if requests for the same cylinder keep coming. The N-step strategy can be applied to these algorithms to address this problem [115]. This strategy segments the disk request queue into sub-queues of length N. Only one sub-queue can be processed at a time. All new incoming requests are added to other queues. Similar to the N-step strategy, the aging strategy [107, 139] can also be applied to the scheduling algorithms to avoid starvation. In the aging strategy, requests that have longer waiting time receive higher priority so that they will not be delayed indefinitely. Both the N-step strategy and the aging strategy increase fairness at the expense of reduced throughput [107, 139].

In MEMS-based storage devices, because the access times depend on the relative locations of the data, scheduling algorithms can be used to reduce access times. Simulation results show that most of the algorithms and insights from previous disk scheduling research have similar benefits to systems with MEMS-based storage devices [44, 45]. There are also scheduling algorithms designed specifically for MEMS-based storage [49].

2.2.2 Disk Cache

Most disks employ an on-board fast cache to match the speed between the bus and the disk media and cache disk blocks. If the write back of the disk cache is enabled, the write request is considered finished by the host once all data are transfered to the disk cache. Disk cache can be used to store prefetched data of the same track to decrease the access time of sequential requests.

The zero-latency access employed in many modern disks can eliminate rotational latency when reading or writing all sectors on the same track [101]. In zero-latency access, the disk firmware can start reading or writing these sectors in any order. As a result, when all sectors of a track are accessed in one sequential request (called a *track-aligned extent*), the rotational latency and the time that the disk arm switches tracks can be avoided [101]. If the data are allocated on disk regardless of the track boundary (called a *track-unaware extent*), the access may span two tracks and incurs much longer access time. Track-aligned extents can be used to optimize disk accesses for certain workloads [101, 102].

Because the requests to the disk are often filtered by upper layer buffer caches, these requests exhibit poor locality [137]. When this happens, the benefit of a large disk cache is limited [142].

2.2.3 Redundant Array of Independent Disks (RAID)

Since one disk cannot provide the capacity, performance, and reliability required by large scale systems, it is now common to organize many disks together to form a Redundant Array of Independent Disks (RAID) [89]. RAID is a set of disks (also called a disk array) managed by a controller and viewed by the host as a single logical drive.

RAID has several levels that provide different I/O characteristics and redundancy methods. The most commonly used levels are RAID-0 (non-redundant), RAID-1 (mirrored), and RAID-5 (block-interleaved distributed parity).

In RAID-0, the data are distributed across all the disks in the array. It provides good performance but does not contain redundancy information. In RAID-1, all data are duplicated on different disks. It provides good performance and redundancy is achieved. The problem of RAID-1 is that only half of the disk space can be used to store user data.

RAID-5 uses an extra disk worth of space to store parity information, so that a single disk failure does not cause data loss. Row-Diagonal Parity (RDP) uses two extra disks worth of space to protect RAID against double disk failure [24]. Both RAID-5 and RDP provide good read performance. Their write performance is also good when writing large blocks since new parity information can be calculated without extra disk I/Os. When writing small blocks of data, however, the performance of RAID-5 and RDP is poor because of the extra disk I/Os required to calculate parity. This is called the *write penalty* for small writes.

It is difficult to select and manage the levels of RAID in a disk array. RAID levels can be programmatically selected given characteristics of workloads [4]. AutoRAID [136] dynamically configures the disk array using a two-level storage hierarchy. The higher level uses RAID-1 to store actively updated data, while the lower level uses RAID-5 to reduce the space overhead of redundant information. Data migrates between these two levels automatically depending on the available space and the activity of the data.

2.3 The Disk Layout Layer

The disk layout layer uses disk layout management approaches to decide the placement of data on disks. The design objective of disk layout management is to improve the disk I/O performance by reducing the mechanical movement of disk arms. Disk layout management approaches have been studied extensively in the context of file systems [40, 74, 96, 140].

2.3.1 Disk Layout Optimized for Large Sequential Access

The traditional UNIX file system [122] is an ancestor of many modern file systems, such as FFS, UFS, and ext2. The basic data structures used in the traditional UNIX file system are still popular in those modern file systems. In the traditional UNIX file system, a tree structure called *inode* is used to control the allocation of disk pages belonging to a file or a directory. All inodes are stored in the beginning of the disk. A free block list is used to manage the free space of the disk. A block size of 1KB is used as the unit of space allocation.

Since the traditional UNIX file system does not consider the placement of data on disks, it has several performance problems. When accessing a file, both its inode and the data blocks need to be accessed. However, since all inodes are stored at the beginning of the disk and the data blocks may be stored far away from their inodes, long seek time is involved when the disk arm moves back and forth between the file data and its inode. The free block list manages all free blocks as a stack. After many files are created and deleted, the blocks stored in the free block list become randomly organized. When creating a new file, these blocks are taken off from the free block list and assigned to the file. Therefore, the blocks belonging to new files scatter over the disk and are not allocated sequentially. When these files are accessed sequentially, which happens very often [87], the disk must spend most time to position the disk head on the desired block of the file. Finally, the 1KB block size is small (i.e., a file contains many blocks), which exacerbates the above performance problems. As a result, less than 2% of the disk bandwidth can be achieved when accessing a file sequentially after the file system has been used for some time [73, p. 269].

The Fast File System (FFS) [74] was designed to overcome these performance problems. FFS uses a larger page size (4KB to 16KB) to reduce the number of pages and thus the disk seeks. FFS divides the whole disk into one or more areas, each of which is called a *cylinder group*. Each cylinder group consists of one or more consecutive cylinders on the disk. For each file, FFS attempts to allocate

space close to its inodes, and it tries to allocate all data pages belonging to the same file contiguously in one cylinder group. FFS was further improved using I/O clustering so that small requests to the same track on the disk are merged into large accesses [75].

Modern file systems, such as BeFS [40], NTFS [97], XFS [117], ext3 [127], and yFS [140], employ similar disk layout approaches to support the sequential access of a single file effectively.

2.3.2 Disk Layout Optimized for Metadata Updates

FFS uses carefully ordered synchronous writes to maintain the integrity of the on-disk metadata structure in the event of system failure. These synchronous writes are the bottleneck of the file system in metadata-intensive workloads [86]. A disk check program is used to restore the integrity of metadata after a failure. Since the whole disk needs to be scanned, the check may require hours for large file systems.

NVRAM (non-volatile RAM) is used in many storage systems to speed up synchronous writes and cache bursts of writes [118]. The use of NVRAM also eliminates the need of a full file system scan after system failure. Since NVRAM is constrained in size due to its high price, Disk Caching Disk (DCD) [53] employs a log disk to substitute for NVRAM and achieves similar write performance. The problem with these two approaches is that they require special hardware, and they achieve high write performance only in systems with many idle periods.

The Soft Updates approach [37] uses in-memory ordering to eliminate synchronous writes to metadata while preserving their on-disk consistency. However, Soft Updates cannot support sophisticated data structures such as B-trees which are often used in block allocation [117, 140] and directories [40, 97, 117, 140].

Journaling file systems, such as Be File System [40], ext3 [127], GPFS [104], JFS [58], NTFS [97], ReiserFS [92], XFS [117], and yFS [140] use a write-ahead log

to record metadata updates so that the synchronous writes can be eliminated. Sophisticated data structures such as B-tree can be supported by journaling. Journaling file systems improve the performance of metadata writes and support fast recovery after system failure.

Journaling file systems do not speed up the I/O performance for file data. The writing to the log generates extra disk traffic than Soft Updates since each metadata update is written twice: one in the write ahead log, one in the original location of the metadata.

2.3.3 Disk Layout Optimized for Small Writes

FFS and journaling file systems are designed to provide efficient access to sequential I/O of a single large file. However, most files are small. Moreover, when multiple users access many files simultaneously in large scale systems, requests from different users interleave with each other. As a result, requests to disks are often small I/Os scattered over the disks. This kind of disk I/Os is called *random* I/Os. The disk spends most time in moving the disk head to the desired locations when processing random I/Os. Since most of the read operations can be absorbed by the file buffer, writes constitute a large proportion of these random I/Os [32, 113]. Efficiently handling these random writes is crucial to the overall performance of the system.

The Log-structured File System (LFS) [88, 95, 106] was designed to achieve good write performance in systems with such random writes. In LFS, the whole disk is viewed as an append-only log, containing a linked list of segments. Each segment is a large fixed-size contiguous disk space, typically larger than 0.5MB for a single disk system[1]. LFS accumulates individual writes, regardless of which files they belong to, into a segment-sized contiguous block, which is then written to a free segment found in the list of available segments. Since data are always written to new locations, LFS is also called a non-overwrite approach, while the traditional

[1]In a disk array, a segment should span all disks, and the stripe size on each disk is typically larger than 0.5MB

approaches are called *Overwrite*. Because the disk seek time and rotational latency are dominated by the long data transfer time when writing large contiguous segments, the write performance of LFS is better than Overwrite. Moreover, accumulating small writes into large writes can avoid the write penalty of small writes in RAID-5 and RDP disk arrays [24, 89].

Since the data are written to the end of the log instead of their original locations, their old copies are invalidated (called *dead pages*). As new data are written to the disk, the disk space can be used up eventually, if these dead pages are not reclaimed periodically for reuse. The process of reclaiming dead pages is called *segment cleaning*, which is performed by a *cleaner*. The cleaner first selects some candidate segments for cleaning, then reads these segments into memory, and finally writes the alive pages of these segments out to empty segments. The space occupied by the candidate segments can be marked as free after the cleaning. The selection of segments for cleaning is crucial for the performance of LFS [78, 96]. Segments with an old age and many free spaces are good candidates for cleaning. Another method to reclaim space from dead pages is called *hole-plugging* [136]. In hole-plugging, the cleaner reads the candidate segments into memory, and writes the alive pages into holes found in other segments. The cost of normal segment cleaning increases quickly as the disk space utilization becomes high [96], where hole-plugging becomes the preferred method to reclaim free space [72]. Segment cleaning can be done as a background activity when the file system is lightly utilized or has bursty behaviour (*background cleaning*). This cleaning can also be performed on-demand when the free segments are almost used up (*on-demand cleaning*).

Previous research [106, 108] found that on-demand cleaning adversely impacts system performance, especially in OLTP environments. Because updated data are randomly distributed on the disks in such workloads, most segments are fairly full before cleaning. This makes the segment cleaning overhead very high. Cleaning has been observed to cause performance degradation of 35%-50% under these workloads, which makes LFS perform comparably to or sometimes worse than FFS

with I/O clustering [108].

A number of cleaning strategies have been proposed to improve the performance of LFS. The adaptive cleaning approach [72] selects between cleaning and hole-plugging based on current free space presented and workload characteristics. The heuristic cleaning algorithm [12] determines disk idle periods and attempts to perform all cleaning during these periods. PROFS [130] attempts to improve the performance of LFS by placing hot data in the faster zones of the disks and cold data in the slower zones during the cleaning. Write Anywhere File Layout (WAFL) [48] and Log-Structured Array (LSA) [77] use LFS and NVRAM to manage disk layouts. WAFL also maintains multiple snapshots of the file system. WAFL avoids doing cleaning by plugging data into holes found in segments. However, when free spaces are not contiguous, the write performance is compromised. Although NVRAM eliminates writes for keeping the metadata integrity and improves write performance by absorbing bursts of writes, the high cleaning cost of LFS is not addressed.

WOLF [131] is a recent proposal to reducing LFS cleaning overhead. WOLF separates hot and cold pages when they are written to the disks. It usually writes two segments of data to the disks at one time. Pages are sorted based on their update frequencies before being inserted into the segment buffers. The rationale is that the segments containing pages with higher update frequencies will soon become low-utilized since the pages in them are likely to be updated again in a short time, thus reducing the cleaning overhead. This approach works well only when about half of the pages are hot and half are cold, so that they can be written into separate segments. In other cases, WOLF has little advantage over LFS, as shown in Section 6.4 (page 137).

2.4 The Buffer Cache Layer

In the buffer cache layer, disk pages are cached in the buffer cache and managed by a buffer cache management algorithm. The objective of buffer cache

management is to reduce the number of disk accesses by keeping popular pages in the buffer cache, thereby reducing the average access time of disk pages.

The buffer cache replacement algorithm is the most crucial part of buffer cache management. The design of buffer cache replacement algorithms is based on anticipated characteristics of page reference patterns in the buffer cache, such as temporal locality, spatial locality, etc. The general reference characteristics and some specific reference characteristics of typical applications are discussed in Section 2.4.1. Some existing replacement algorithms are reviewed in Section 2.4.2. Other issues for buffer cache management are discussed in Section 2.4.3.

2.4.1 Characteristics of References to the Buffer Cache

Understanding the characteristics of references to pages in the buffer cache is a prerequisite to the design of effective buffer cache replacement algorithms. One or more of the following typical reference characteristics may be observed in real workloads:

- *Temporal locality*: To the extent to which reference patterns exhibit temporal locality, pages that have been referenced recently tend to be referenced again in the near future. Temporal locality is observed in many access patterns, such as program page accesses [29], database accesses [61], CPU cache accesses [115], and file buffers. Temporal locality is the main design assumption of many replacement algorithms, which are called *locality-based* algorithms.

- *Spatial locality*: To the extent to which reference patterns exhibit spatial locality, pages whose addresses are close to recently referenced pages are likely to be referenced in the near future. This is a common reference pattern in buffer cache management. Spatial locality can be utilized by using a large page size. When an item in a page is referenced, the whole page is loaded into the buffer cache so that references to other items in the same page can be satisfied without additional disk I/Os. When references exhibit spatial

locality, it is likely that there will be repeated references to a page once it is brought into the buffer cache. These references are called *correlated references* [8, 60, 84].

- *Sequential accesses*: In sequential references, pages are referenced once from the beginning to the end. If the addresses of these pages are consecutive, prefetching can be used to improve performance. Normally prefetching is not part of a buffer cache replacement algorithm, but it can be applied to any replacement algorithm.

- *Looping accesses*. In looping references, the same set of pages is referenced in the same order repeatedly. Contrary to the temporal locality pattern, the most recently accessed page is the one that will be accessed again farthest in the future. This pattern could cause locality-based algorithms such as LRU to perform poorly if the loop cannot fit in the buffer cache. For this pattern, the Most Recently Used (MRU) algorithm, which selects the most recently accessed page as replacement, achieves the optimal hit ratio.

None of the above assumptions alone can describe well all the reference patterns that occur in real applications. Algorithms that are designed based on a single assumption work well only when this assumption holds. To overcome this problem, some algorithms have been designed based on a combination of several assumptions.

2.4.2 Replacement Algorithms

Locality-based replacement algorithms make extensive use of the notion of *recency of reference*. The recency of reference to a page refers to the "time" that has passed since the previous reference to the page, where each page reference is considered as one unit of time. The recency is often used to predict the time when this page will be referenced next.

The LRU (Least Recently Used) algorithm is the simplest locality-based algorithm. When the buffer cache is full and room is needed for a new page, the page with the largest recency is selected for replacement. LRU can be efficiently implemented using a linked list with constant time overhead.

LRU performs well in common workloads and is the most popular replacement algorithm in real systems (e.g., LRU is used in Database 2 [119], ADABAS [105], MySQL [80], and PostgreSQL [91]). LRU performs poorly, however, when the temporal locality assumption does not hold. For example:

- When the number of pages in a loop is larger than the size of the buffer, LRU always replaces the page that will be used the soonest. A better replacement algorithm would select the page that has just been referenced, since it will be used the farthest in the future.
- After a burst of accesses to some infrequently used pages, e.g., a sequential scan, these pages will replace many commonly used pages in the buffer cache.

Many algorithms have been designed to overcome these problems of LRU. These algorithms use additional information to make better replacement decisions. Some algorithms use the recency of several past references to a page. Some algorithms detect the sequential scan and loop patterns and handle them differently. Some algorithms use frequency of references. Some algorithms detect or use application-level knowledge. These algorithms are listed in Table 2.1.

Table 2.1: Categories of Replacement Algorithms

Additional Information Used	Replacement Algorithms
Recency of more than one past references	LRU-K, 2Q, LIRS, ARC, CAR, CART
Sequential scan and loop	SEQ, EELRU
Reference frequency	LFU, FBR, LRFU
Application knowledge	Application Controlled Caching, DEAR, AFC, UBM, ILRU, OLRU, Hot Set, QLSM

The LRU-K algorithm [84] replaces the page whose *kth* last reference has the largest recency. The suggested value of K is 2 [84] and LRU-K is called LRU-2.

LRU-2 has $O(\log n)$ overhead, where n is the size of the buffer cache. 2Q [60] is an approximation of LRU-2 with constant overhead. LIRS [59] replaces the page whose difference of recency between the last two references is the largest. ARC [76] has a similar data structure and idea as 2Q but can dynamically adjust itself according to workload changes. Since the information of the last several references of a page is retained in these algorithms, the correlated references to a page caused by spatial locality can let these algorithms erroneously decide that such a page is popular in the long term [84]. Tunable parameters are used in LRU-K and 2Q to filter out correlated references. ARC does not consider correlated references, and thus its performance can be adversely impacted when correlated references exist. CART [8] is an improvement of ARC which filters correlated references. LIRS utilizes the recency of the last reference instead of the reference difference between the last two references when the former is larger. This mechanism filters correlated references automatically after the correlated references to a page finish. All these algorithms can handle sequential access patterns well. LIRS can handle large loops well, but LRU-2, 2Q, ARC, and CART have similar problems to LRU when handling large loops.

The SEQ algorithm [41] detects long sequences of access patterns with consecutive addresses and applies a different replacement algorithm to these sequences. The early-evict LRU (EELRU) [112] algorithm use the recency distribution of referenced pages to decide whether to evict a page from some pre-defined early eviction points, so that pages with smaller recency could be evicted. Both algorithms can handle the sequential scan and large loop patterns that LRU cannot handle well.

The Least Frequently Used (LFU) algorithm replaces the page that has been referenced the least number of times. One problem with LFU is that some pages may have built very high reference frequency, and therefore cannot be replaced even after they have not been referenced for a long time. Some variations of LFU (e.g., LFU*, LFU-Aging, and LFU*-Aging) have been proposed to overcome this problem [6, 93, 137]. Frequency Based Replacement (FBR) [93] is a variant of LFU

in which recently referenced pages do not accumulate reference counts so that the correlated references are filtered out. FBR has several parameters that can affect performance but cannot be tuned easily. The LRFU (Least Recently/Frequently Used) algorithm [66] combines both the recency and the frequency information of past references when making replacement decisions. It is like a LFU algorithm which continuously ages its frequency values. A parameter can be used to control how fast this aging is, which must be tuned to suit different workloads. All these frequency based algorithms cannot handle large loops well, because the frequency information does not help to select the best replacements for looping references. Since a priority queue is often needed to maintain the frequencies of all pages in these algorithms, the overhead of LFU and its variants is $O(\log n)$, where n is the size of the buffer cache.

Application-controlled caching [16] uses application-specific knowledge to improve buffer cache performance. In this algorithm, each application explicitly specifies the replacement policy and priority of its data. The problem with this approach is that all applications must be modified to give the hints explicitly. This increases the burden to application developers and may not be feasible for some applications. DEAR (DEtection-based Adaptive Replacement) [19] and its variants Application/File-level Characterization (AFC) [20] and Unified Buffer Management (UBM) [64] detect typical reference patterns and apply different replacement policies to different patterns. Four typical reference patterns, i.e., sequential pattern, looping pattern, temporal localized pattern, and probabilistic pattern, can be detected in real time. The DEAR/AFC/UBM approach works well in applications where each file has a clear reference pattern. A simple reference pattern does not exist if a file is shared by many processes/threads (which is typical in database systems).

Because of the regularity of data references in database applications [61, 62], many replacement algorithms have been proposed for the DBMS buffer cache. Two replacement algorithms, Inverse LRU (ILRU) and Optimal LRU (OLRU), were designed to manage index pages which are stored as B^+-trees [98]. The level

of the page in the B^+-tree is used to determine how pages are placed in the buffer cache. ILRU works well in small buffers or when the access to the leaf pages of the B^+-tree are skewed. OLRU works well when the access to the leaf pages of the B^+-tree are evenly distributed. The Hot Set model [99] analyzes the execution plan of a query to find the amount of buffers required to fit the looping pattern, which is called the *Hot Set*. The Query Locality Set Model (QLSM) model [21] advances the idea of Hot Set and identifies several access patterns found in the query execution plan. Different replacement policies are used for different access patterns. The Hot Set model and the QLSM model can be used to make load control and prefetch decisions for the buffer cache [15, 34, 67]. In many database workloads, multiple concurrent queries access shared data and their reference behaviours overlap in complex ways. These algorithms do not work well in such a situation.

In virtual memory, the buffer cache hit must be handled by hardware. The overhead of the replacement algorithms discussed above is too high to be used directly in virtual memory management. The CLOCK algorithm is an approximation of LRU that can be used in virtual memory. CLOCK uses a reference bit to remember whether a page is referenced recently. Some variants of the CLOCK algorithms, such as the Generalized CLOCK (GCLOCK) and the Dynamic GCLOCK (DGCLOCK), use more than one bit for each page to help make better replacement decisions [31, 82], but workload-specific tuning is often required for such algorithms. CLOCK performs similarly to LRU and suffers the same problems as LRU. Clock with Adaptive Replacement (CAR) [8] uses CLOCK to approximate ARC. CAR with Temporal filtering (CART) is a variant of CAR that filters correlated references [8]. CAR and CART perform similarly to ARC and, like ARC, cannot handle large loops well. CLOCK can be modified to approximate LFU with much lower overhead. The basic idea is to associate each page with a counter to record the number of times the page is referenced. Linux uses this algorithm to manage its virtual memory [30].

2.4.3 Other Issues of Buffer Cache Management

Cache Partitioning

Traditional algorithms such as LRU consider all pages in the buffer when selecting a candidate for replacement. These algorithms are called *global algorithms*. Another category of algorithm divides the buffer cache into several partitions. When a replacement is needed, only pages in the same partition are considered, and so the behaviour of one partition does not affect the other partitions. Optimal sizes for all partitions can be selected based on the *marginal gain* of each buffer cache [116]. Marginal gain is the increase of number of hits of a partition when a buffer is added to this partition. When the marginal gains of all partitions are the same, the partition allocation is optimal. Cache partitioning needs to work together with a global algorithm (or several global algorithms) to manage each partition.

Self-tuning in Buffer Cache Management

Because of the complexity of real systems, the effective configuration and tuning of any management algorithm is a significant challenge for the administrators. Some goal-oriented self-tuning algorithms have been proposed to ease the task. In a DBMS, response time goals are first specified by the administrator. The buffer cache is then partitioned. Goal-oriented self-tuning algorithms can be used to dynamically adjust the partition sizes to meet the response time goal [13, 23]. The algorithm collects system states periodically and adjusts the buffer cache allocated for each transaction class. Response time goals of different transaction classes must be specified by the administrator, which could be difficult in complex systems.

2.5 Mixing the Layers

The storage subsystem is divided into different layers. The lower layer hides its implementation details and provides a simple interface to the upper layer. Each

layer cannot make assumptions about how other layers work beyond the given interface. This design philosophy is commonly seen in the design of network protocol stacks and complex software systems. The cost of a layered design is somewhat compromised performance. If the higher layer knows more details about the lower layer, or vice versa, informed decisions can often be made to achieve better performance. Many approaches were proposed in storage management to utilize information of other layers. They are discussed in Section 2.5.1 and 2.5.2. The difficulties of mixing layers are discussed in Section 2.5.3.

2.5.1 Using Lower Layer Knowledge

Disks communicate with the upper layer using a simple protocol. The starting logical block number and the number of blocks to be read or written are given to the disks, and the notification of operation completion and the data (if this is a read request) are returned to the upper layer. Disks are highly sophisticated and intelligent under this simple interface [3, 54, 100].

Static information such as track boundaries, and dynamic information such as disk head position, seek time and rotational latency of the next request, can help the upper layer utilize the disk more effectively. If an extent is placed within a disk track (called a track-aligned extent), the access to it is much faster than track-unaware extents [101]. Carefully placement of data with the knowledge of track boundaries also allows efficient access to both contiguous and certain non-contiguous blocks [102]. As a result, dramatic performance improvements can be achieved in certain applications [101, 102].

The freeblock scheduling algorithm uses detailed dynamic information of the disk to effectively utilize unused disk bandwidth for background requests without affecting foreground requests [70, 71, 120]. This algorithm utilizes the rotational latency between foreground requests opportunistically to serve background requests. These background requests may be served out-of-order. The detailed characteristics of some disks can be automatically extracted [100] and utilized for

freeblock scheduling outside of disk firmware [70].

Virtual log utilizes the head position to improve performance of small synchronous writes [132]. Similar to LFS, virtual log writes data to a new location. Unlike LFS, it selects the new location based on the current position of the disk head to minimize seek time and rotational latency. Since virtual log can utilize any free space on the disk, no garbage collection activity is required.

When the information inside a disk array is exposed, the file system can be built to utilize this information to achieve better flexibility, reliability, manageability, and performance [27].

2.5.2 Using Upper Layer Knowledge

Data that appears to be the same in the lower layer may represent different entities in the upper layer. For example, a disk block in a storage server (the lower layer) may represent file data, a directory, an inode, a bitmap or a free block in a file server (the upper layer). This extra information is called the semantics of the upper layer. When these semantics are known, the lower layer can often act appropriately to achieve better performance and functionality. The semantically-smart disk [111] infers the on-disk structures of the file system running in the upper layer, and detects the semantics of the file system operations automatically. These semantics can then be used to make better decisions, such as allocating files within track-aligned extents, caching the metadata of files, or securely delete files. When the disk array understands the on-disk structures of the file system, the key metadata and the frequently used files of the file system can be duplicated to a high degree to achieve high availability [110].

Multiple levels of buffer caches are often used in storage management. The upper level buffer cache greatly changes the characteristics of the references to the lower level buffer cache [137]. The DEMOTE approach was proposed to let the two buffer caches work together, so that the lower level cache knows what pages have been evicted from the upper level cache [138]. DEMOTE requires changes to

the communication protocol between the upper level and lower level.

The approaches discussed in Section 2.5.1 can be implemented inside the disks. Disks containing these extra logic are called active disks [1] or intelligent disks [63]. This kind of smart storage can be used to perform general processing, such as database queries [50] and image filtering [55].

2.5.3 Difficulties of Mixing Layers

Although mixing the layers can bring dramatic performance improvement, these approaches have their difficulties. The first difficulty is the increased complexity. Mixing the layers somewhat compromises the main advantage of using layered design: reducing complexity. Increased complexity translates to higher development and test cost. Since the layers depend on each other's internal details, one change within a layer may break components in other layers. Some approaches for mixing the layers use other than standard protocol between layers, which increases the difficulty of deployment.

Knowledge of static elements of the other layer, such as the track boundary in a disk, is relatively easy and safe to obtain. However, knowledge of dynamic elements, such as the disk head position, is much harder to get. As modern disks become more and more sophisticated, predicting the activity of disks outside of disk firmware is becoming a challenge [54]. One difficulty for embedding extra logic in disk firmware to utilize the knowledge of dynamic elements of the disk is that a disk may be used stand-alone, or may be part of a disk array. When a disk is part of a disk array, the array controller is a better place to embed this extra logic, but the knowledge of the dynamic elements of the disk, such as the disk head position, is again not available at the array controller.

Chapter 3

Research Methodology

Storage management in large scale systems is complex. It is difficult to analyze an existing algorithm, let alone implement a new one and evaluate it, in a real system. Modeling and simulation are two cost-effective alternatives. By modeling the key components of a system mathematically, important insights can be gained, and this can also shed light on possible solutions. A wide range of workloads and system configurations can be tested easily in a simulator, and various algorithms can be implemented and evaluated with much less effort than in a real system.

For this thesis research, a blend of direct experimentation, trace-driven simulation, and modeling was used. First, direct experimentation was used to understand how the real system behaves. Analytical models and simulators were then built to simulate key parts of the storage subsystem. The models and simulators were validated against the real system, and new algorithms were analyzed in the models and evaluated in the simulators. Different simulators, and analytical models were used to study different aspects of the storage subsystem. The validation and verification of them are discussed separately in Chapters 4, 5, and 6.

In the rest of this chapter, Section 3.1 describes the basic characteristics of typical workloads studied in this thesis. Section 3.2 discusses the characteristics of the TPC-C benchmark, which is an important workload in this research.

3.1 Typical Workloads

Understanding the basic workload characteristics of the storage subsystem is a prerequisite to studying its performance. The workloads presented to the storage subsystem can be classified roughly into database workloads and file server workloads. Since storage servers often sit below other applications which already employ large buffer caches, such as database servers or file servers, the requests to a storage server may already have been filtered by upper layer buffer caches, and so may exhibit different characteristics [137, 138, 141]. These three categories of workloads are discussed separately in the following subsections.

3.1.1 Database Workloads

Database servers support a wide range of applications. They can be classified roughly into two classes: on-line transaction processing (OLTP) systems and decision support systems (DSS). DSSs are also called on-line analytical processing (OLAP) systems. Real database applications have the properties of both [51]. As a new way of doing business through the Internet, e-commerce applications play an increasingly important role in database applications. The database workloads in e-commerce applications can be viewed as a mix of OLTP and decision support workloads [33].

Since real database workloads are hard to obtain, benchmarks were used as the database workloads in this research. The TPC benchmarks developed by the Transaction Processing Performance Council (TPC) [124] are widely accepted for testing the performance of database systems under various benchmarks.

3.1.1.1 OLTP Workloads

OLTP workload is important in large database systems. OLTP applications are used in many industries for data entry and data retrieval transactions. OLTP is the cornerstone by which a great deal of modern business is done.

Most OLTP transactions are quite simple. The execution time of each

transaction is short (typically within a second), and there are upper bound requirements for response time. An OLTP application has many terminals connected to one or more central database servers through a network as shown in Figure 3.1. The client/server model is typically used for OLTP applications. Different terminals initiate various transactions to the server independently. The database on the server is updated frequently by these transactions. These updates are typically small and to random places of the disks. Because of the I/O-intensive nature of OLTP applications, the storage subsystem is often the performance bottleneck. The two major design challenges for the storage subsystem to achieve good performance for OLTP applications lie in the buffer cache layer and the disk layout layer. The buffer cache of the DBMS must be managed effectively to reduce the number of disk accesses. Since most reads are absorbed by the buffer cache, random updates make up a large proportion of requests to disks. The disk layout must be organized to handle the random updates efficiently. The TPC-C benchmark [124], which is a standard benchmark representing OLTP workloads, is used in this thesis to study the performance of storage management under OLTP workloads.

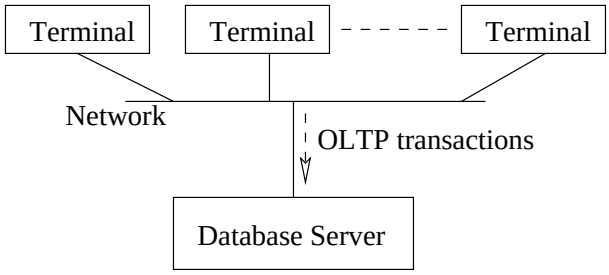


Figure 3.1: Architecture of an OLTP Application

3.1.1.2 Decision Support Workloads

In a DSS, users query business data to get answers to critical business questions. In decision support workloads, complex queries are used to search a large amount of data in the database. The execution time of each query is long (from tens of seconds to several hours), and the queries mainly read the database sequentially.

The storage subsystem can easily achieve its maximum transfer bandwidth under such I/O patterns. The performance bottleneck of DSS is often the CPU computation power instead of the storage. The TPC-H benchmark [124] represents an ad hoc decision support workload, which means no advance knowledge of the queries is available. The DBT3 benchmark developed by the Open Source Development Lab [26] is a simplified implementation of the TPC-H benchmark, and this is used as the workload when studying the performance of the buffer cache layer.

3.1.1.3 E-commerce Workloads

E-commerce is a new way of selling products or services through the Internet. Customers access an e-commerce web site from their web browsers. A typical e-commerce system contains a back-end database server and several front-end servers, including web servers, web caches, and image servers. Figure 3.2 shows a typical structure of an e-commerce environment. The web servers provide web pages to browsers. The image servers provide images to browsers. The web cache servers cache the search results of client requests to reduce the load on the database server. The back-end database server stores the information of customers and products and processes user transactions.

Current network load-balancing technology makes it very easy to add front-end servers when they become a bottleneck of the system [17]. It is much more difficult to use more than one database server, however, since distributed database systems are hard to build and manage. In a typical e-commerce system, front-end servers are built on many cheap machines, while the database server is built on a very expensive machine. In many large e-commerce configurations [126], the cost of the database server hardware represents 30% to 65% of the total hardware cost, although there is one database server but several dozens of front-end servers. Therefore, improving the performance of the database server can greatly reduce the total cost of the system.

The primary function of the web cache is to reduce the load on the database

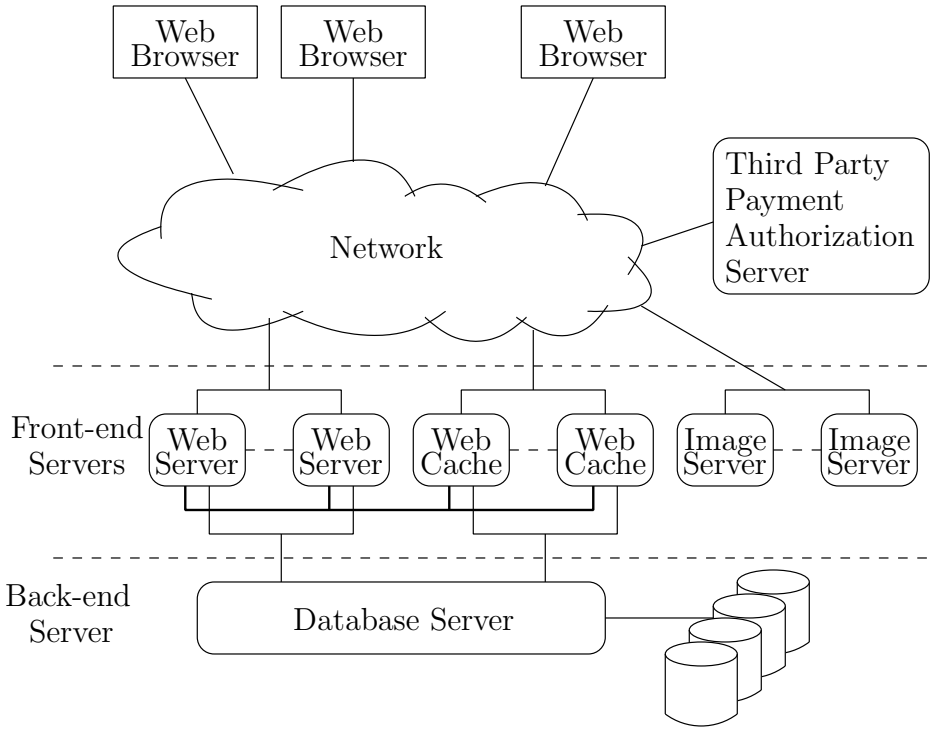


Figure 3.2: E-commerce Environment

server. In e-commerce web sites, customers can search for products or services by sending complex queries to the database server. The search results of these queries are cached in the web cache after being generated by the database server. These cached results can satisfy some subsequent search requests (even from other users) without contacting the database server again. Without dynamic caches, the database server can easily become a performance bottleneck. When customers buy products or services, the transactions sent to the database server are not cacheable, since they contain updates to the database, which cannot be used by other customers and must be processed by the database server within a strict response time bound. These simple transactions are similar to the transactions in OLTP workloads. As dynamic cache technology improves, more query results can be cached at web caches, and the workload seen by the database server is more like OLTP. The TPC-W benchmark [124] is designed to represent such workloads and is used as the e-commerce workload when studying the buffer cache layer.

3.1.2 File Server Workloads

File servers support a wide range of applications. The characteristics of file system activities depend on the type of applications running on it. File servers are supporting more users, higher data rates, more files, and more space as time goes on [7, 87]. Studies of file system workloads exhibit some common characteristics despite these changes [7, 32, 87, 94, 129]:

- Most accesses are to small files, while most bytes are from big files.
- A large proportion of file blocks die quickly (within seconds or several minutes).
- Most accesses to files are sequential accesses, but are to small files.
- A large proportion of bytes are accessed randomly.
- As the file client employs larger caches, significantly more writes are observed at the file server [32].

The presence of client cache increases the proportion of writes in the server and changes the temporal locality characteristics of requests to the file server buffer cache [36], which affects the design of the buffer cache replacement algorithm used in file servers.

3.1.3 Storage Server Workloads

Storage servers typically run under file servers and/or database servers. Because of the use of buffer cache in the file servers and database servers, the requests to storage servers exhibit different characteristics [36, 113, 137, 138, 141]:

- Since many reads are cached by upper level buffer caches, write requests make up a large proportion of the total requests. Some traces contain more writes than reads [113].

- Much less temporal locality is observed in the requests to the buffer cache of storage servers than is observed in upper level buffer caches, since these requests are misses of upper level caches. Recency-based algorithms such as LRU that are designed to utilize temporal locality, will therefore perform poorly, while frequency based algorithms show advantages [137].

3.2 TPC-C Workload Characterization

3.2.1 Overview of TPC-C

The TPC-C benchmark models the order processing operations of a wholesale supplier with some geographically distributed sales districts and associated warehouses. The business environment is illustrated in Figure 3.3. In the TPC-C benchmark, the number of warehouses is a variable which determines the scale of the benchmark. Each warehouse has 10 sales districts and each district serves 3000 customers. The supplier has 100,000 items for sale. The initial size of one warehouse is about 100M bytes data.

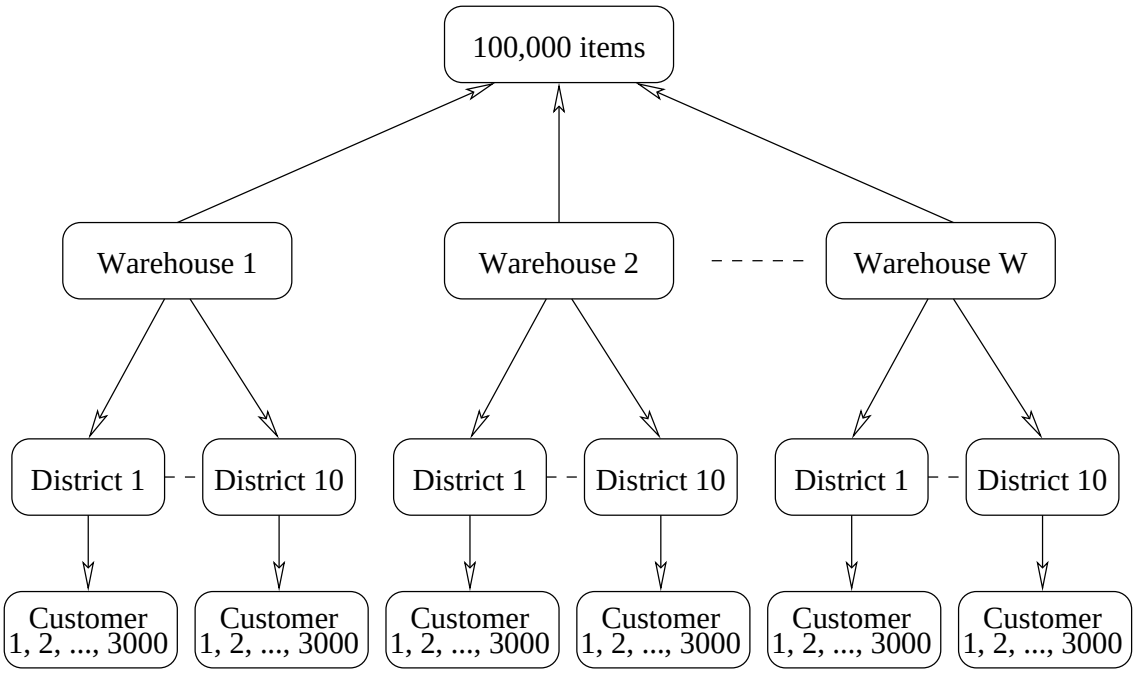


Figure 3.3: TPC-C Business Environment

Five basic transactions that represent essential features of the application are defined by the benchmark. These are listed in Table 3.1. When the benchmark is run, performance is expressed in terms of *transactions per minute*, defined as the number of New Order transactions completed per minute.

Table 3.1: TPC-C Transactions

Transaction	Characteristic	Percentage
New Order	read-write, mid-weight	45%
Payment	read-write, light-weight	43%
Order Status	read only, mid-weight	4%
Delivery	read-write	4%
Stock Level	read only	4%

The database of the TPC-C benchmark consists of nine tables. The relationships among these tables are defined in the entity-relationship diagram in Figure 3.4 [124].

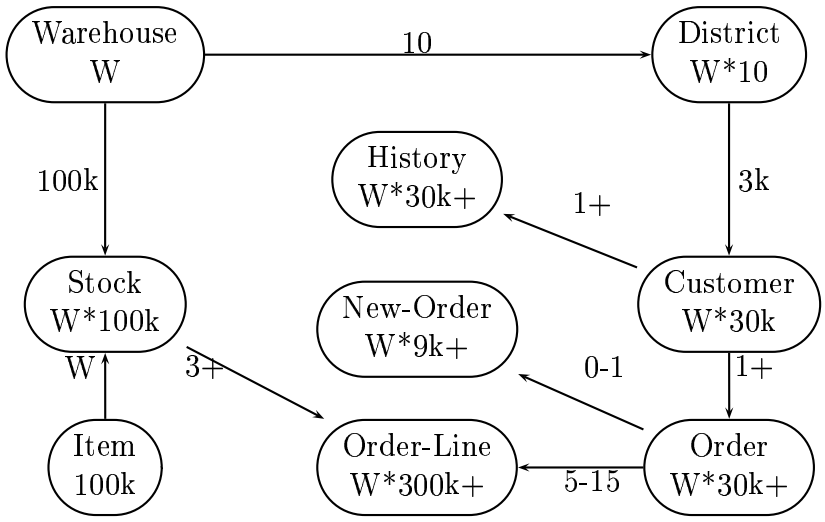


Figure 3.4: TPC-C Tables and their Relationships [124]

The numbers in the entity blocks represent the number of rows of that table. The size of most tables scales with the number of warehouses W. The number next to each relationship arrow represents the cardinality of that relationship (average number of children per parent). The plus (+) symbol indicates more rows will be added to the table as the benchmark runs.

In order to simulate the skewness of the distribution of accesses in real applications, a non-uniformly distributed random number generator is used to populate the database and generate transactions. This non-uniform random function $NURand_A(x, y)$ is defined as:

$$NURand_A(x, y) = ((rnd(0, 2^A - 1)|rnd(x, y)) + C) \bmod (y - x + 1) + x,$$

where:

1. $a|b$ stands for the bitwise logical OR operation between a and b.
2. $a \bmod b$ stands for a modulo b.
3. $rnd(a, b)$ stands for a randomly selected uniformly distributed number within $[a, b]$.
4. C is a random constant in the interval $[0, A]$ that does not affect the distribution of the numbers but affects the "hot" values of the generated numbers. The TPC-C documentation states that C must be selected so that it does not alter performance.
5. A is a constant that can affect the skewness of the distribution of the random numbers generated. The logical OR of $rnd(0, 2^A - 1)$ affects the lowest A bits of the value returned by $rnd(x, y)$. Each of these affected bits is the logical OR of two bits from two uniformly distributed random numbers ($rnd(0, 2^A - 1)$ and $rnd(x, y)$). Table 3.2 shows that the probability of value "1" is 75%, and the probability of value "0" is 25%, which makes the numbers generated by $NURand$ non-uniformly distributed. The larger the value of A, the more skewed the distribution that the $NURand$ function generates.

Table 3.2: Logical OR of Two Bits

a	b	$a\|b$
0	0	0
0	1	1
1	0	1
1	1	1

When each district is populated with 3,000 customers, 1000 unique random last names are used for the first 1000 customers and the $NURand_8(0, 999)$ function is used to select a name from these 1000 names for the remaining 2000 customers. Figure 3.5 shows the cumulative distribution function (CDF) of the access skewness of the last names after the initial database population. The most popular last name is used by 60 customers. The cross in the figure shows that 21% of the last names are used by 60% of the customers.

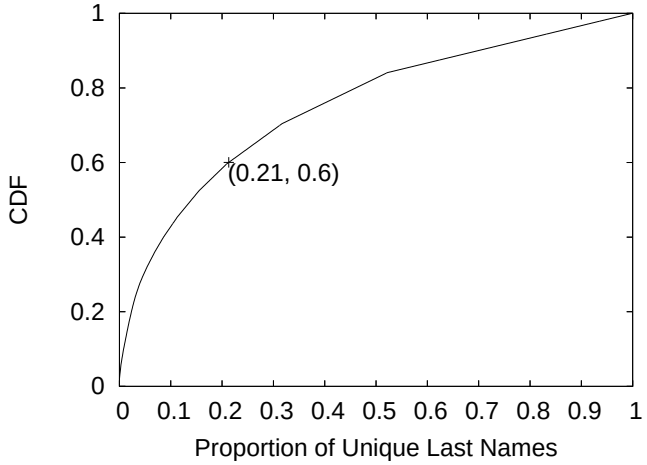


Figure 3.5: Access Skewness of Last Names

During the execution of the benchmark, when the Item table is accessed by the New Order, Payment, and Order Status transactions, the item ID is selected using the $NURand_{13}(1, 100000)$ function. When a customer is selected from the Customer table by a New Order transaction, the $NURand_{10}(1, 3000)$ function is used to select a random customer ID. When a customer is selected by the Payment or Order Status transaction, it is selected by the last name generated by the $NURand_8(0, 999)$ function 60% of the time, and selected by the customer id generated by the $NURand_{10}(1, 3000)$ function 40% of the time. The distributions of the access skewness of customers and items referenced by the TPC-C transactions are shown in Figure 3.6. The item ID distribution is more skewed than the customer ID distribution. As shown in the figure, 16% of the item IDs contribute 80% of the references, and 33% of the customer IDs contribute 80% of

the references.

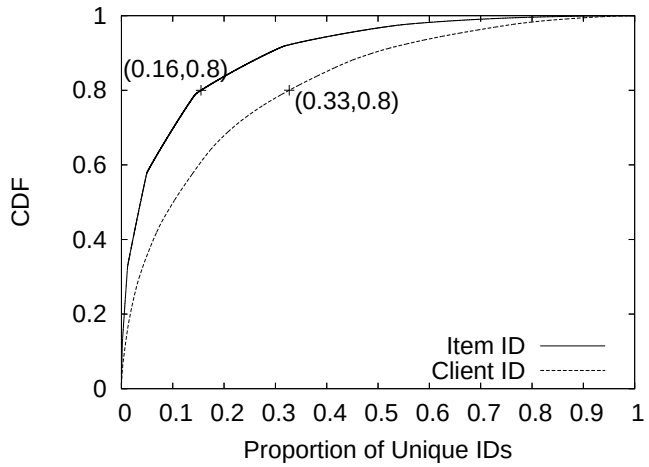


Figure 3.6: Access Skewness of Item IDs and Client IDs

3.2.2 System Configuration and Trace Collection

In order to measure performance results and collect traces, TPC-C was configured to run on an IBM PC Server 704 in the DISCUS laboratory at the University of Saskatchewan. At the time this research was done, this machine was configured with 4 PentiumPro 200MHz processors, 512 MB of memory and 12 4.3GB hard disks attached to two PCI Wide Ultra SCSI-2 buses. The data transfer rate of each SCSI bus was 40MB per second. 10 disks were IBM ST34571WC, and 2 disks were IBM DCHS04Y. All these disks have a rotational speed of 7200RPM, and an average seek time of 7.5ms (DCHS04Y) or 10ms (ST34571WC). The DBMS was IBM DB2 for Windows, version 7.1.0. The number of warehouses in the TPC-C database was 50.

The buffer cache in DB2 employs a fix/unfix mechanism for requesting pages [31]. When the upper layer needs to access a page, it sends a *fix* request to the buffer cache. The page is read from the disk if it is not yet in the buffer cache. The address of the page in the buffer cache is returned to the upper layer as the result of the *fix* request. After a page is fixed, it cannot be evicted from the buffer cache until an *unfix* request of this page is received. As shown in Figure 3.7, a

trace point was placed between the upper layer of the DBMS and the buffer cache to catch all buffer cache fix/unfix requests when running the TPC-C benchmark. The tracing package was ported from one developed by Hsu [51, 52].

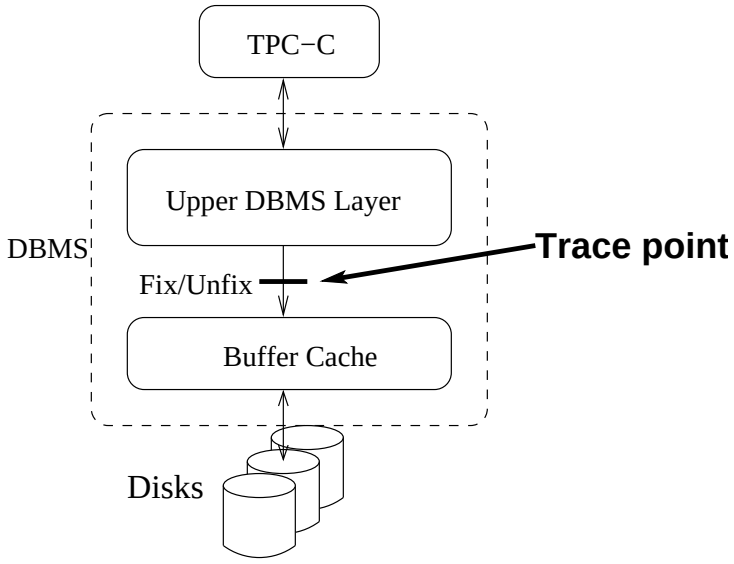


Figure 3.7: Trace Collection Point

All necessary information related to buffer cache requests was recorded in the trace file. Table 3.3 presents the important fields of a trace record. The *fix mode* defines two kinds of fix: *exclusive* and *shared*. If a page is fixed in the exclusive mode, it can be read and updated during the fix period. A page can have at most one exclusive fix at any time. If a page is fixed in the shared mode, it can be read but not updated during the fix period. A page can be fixed in shared mode by many threads at the same time. The collected trace has about 200 million requests, 84% of which are reads. About 1 million distinct pages (page size is 4KB) are referenced in the trace. 60 users were used for the TPC-C benchmark when collecting the trace.

Since temporal locality is one of the most important properties for the design of buffer cache replacement algorithms, the temporal locality in the TPC-C workload is analyzed in the rest of this chapter.

Table 3.3: Important Fields of the Trace

Field	Value
type	the type of the request, must be either *fix* or *unfix*
user id	the user who sends the request
object type	the type of the requested page, must be either *index* or *data*
table id	the id of the table to which the requested page belongs
page number	the logical page number of the requested page
fix mode (only for fix)	*exclusive* or *share*
modified (only for unfix)	whether or not this page has been modified

3.2.3 Reference Characteristics in the TPC-C Benchmark

The reference characteristics of different tables and objects (data and indexes) were studied. The *LRU stack depth* was used to study the reference characteristics of a particular trace. To measure the LRU stack depth, all referenced pages are ordered by their recency of reference. For each reference, the LRU stack is searched for the requested page. If it is found, the LRU stack depth of this reference is the number of pages that have smaller recency than this page. This page is then moved to the top of the stack. If the referenced page is not found in the stack, the LRU stack depth of this reference is infinite, which happens only when the page is first referenced.

If a page is referenced twice in succession, an *immediate re-reference* is noted. For most buffer cache replacement algorithm, immediate re-references are always cache hits and so they do not change the behaviour of the buffer cache. Immediate re-references make the temporal locality of the trace appear better than it actually is. Therefore, immediate re-references were removed from the trace before the following analysis was done.

The cumulative distribution function of the LRU stack-depth probability $F(x)$ is used to describe the reference behaviour of the trace, where x is the LRU stack depth. For a buffer cache of size x, $F(x)$ is also the buffer cache hit ratio if the buffer cache is managed by the LRU algorithm. A more skewed $F(x)$ distribution

implies higher buffer cache hit ratio of the trace. The skewness of $F(x)$ can be caused either by the temporal locality of references, or by the skewed access probability of references.

3.2.4 Single-user Workload Characteristics

In the TPC-C benchmark, reference characteristics do not change over time, and all users have identical characteristics. There are about 3.3 million trace records for every user. Preliminary analyses of the traces found that the first 250,000 records of a user have similar characteristics as the whole trace of this user. Thus only the first 250,000 trace records of the first user are used for the analyses presented in this subsection to reduce the computation resources required.

3.2.4.1 Overall Characteristics

Figure 3.8 shows the LRU stack depth distribution of a single user referencing about 10,000 pages. The cumulative LRU stack depth is about 80% when the buffer cache size is only 1000. This implies that the overall trace can achieve high hit ratio on a small buffer cache. The rightmost point of the line in the figure indicates that about 85% of the page references have finite LRU stack depth.

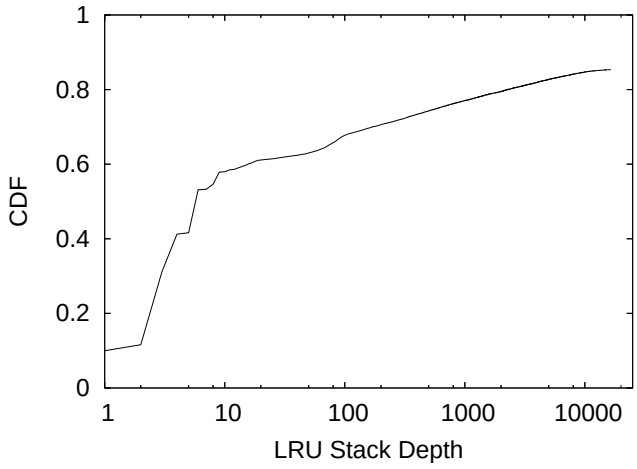


Figure 3.8: The LRU Stack Depth for a Single User

3.2.4.2 Characteristics of Different Tables

There are nine tables in the TPC-C benchmark (see Figure 3.4 on page 42). The references to each table are extracted from the trace so that their LRU stack depth distribution can be studied separately. The results show that different tables exhibit different degrees of skewness in their cumulative LRU stack depth distributions. The nine tables are organized into three categories in Table 3.4 to focus the following discussions.

Table 3.4: Tables with Different LRU Stack Depth Distributions

Table	Skewness of Cumulative LRU Stack Depth Distribution
Warehouse, OrderLine	High
District, Item, NewOrder, Order, Stock	Medium
History, Customer	Low

Figure 3.9 shows the LRU stack depth of the Warehouse and OrderLine tables, which have high skewness in their cumulative LRU stack depth distribution.

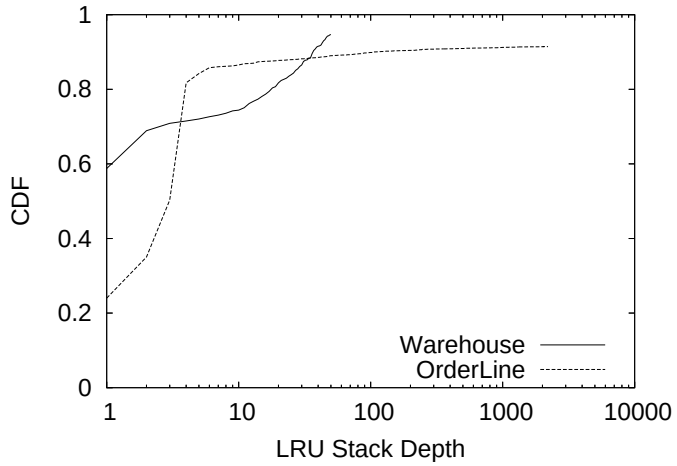


Figure 3.9: The LRU Stack Depth of the Warehouse and OrderLine Tables

For Warehouse, no stack depth greater than 50 was found, because this table has only 50 pages. Many transactions access the same record of the warehouse table for many times during its execution, which makes the cumulative LRU stack depth distribution of this table highly skewed.

In OrderLine, more than 80% of the references have stack depth less than 5. In one New-Order transaction, several new records are inserted into OrderLine. These records might be searched in several other transactions. As a result, OrderLine has highly skewed cumulative LRU stack depth distribution.

Five tables (District, Item, NewOrder, Order, and Stock) have medium skewness in their cumulative LRU stack depth distribution. Their LRU stack depth plots have similar shape. For clarity, the data for only three tables (Item, Order and Stock) are plotted in Figure 3.10. The references to the Item table are non-uniformly distributed random accesses. Its skewed cumulative LRU stack depth distribution is due to the skewed distribution of the accesses as shown in Figure 3.6 (page 45).

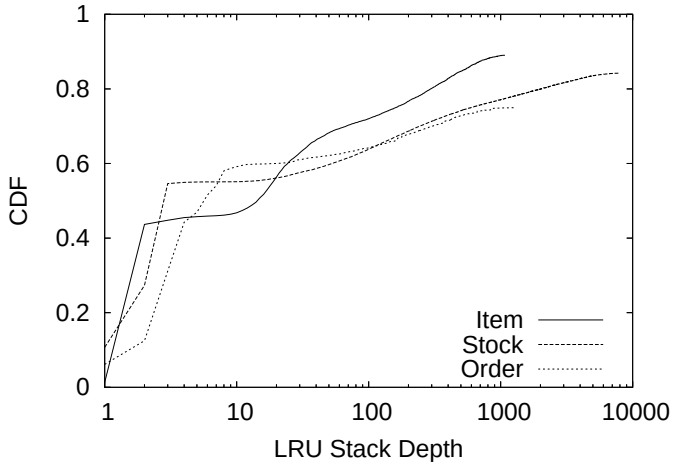


Figure 3.10: The LRU Stack Depth of the Item, Stock and Order Tables

Figure 3.11 shows the cumulative LRU stack depth distribution of History and Customer, both of which have low skewness in their cumulative LRU stack depth distribution. This figure shows that in the History table, about 80% of its references have infinite stack depth. This is because data are appended at the end of the History table and are not accessed again in the TPC-C benchmark. 45% of the references to Customer have infinite stack depth. Customer is characterized by non-uniformly distributed references. It exhibits low skewness in the cumulative LRU stack depth distribution because the non-uniform random function used for

the Customer table has low skewness (see Figure 3.6) and the Customer table is very big.

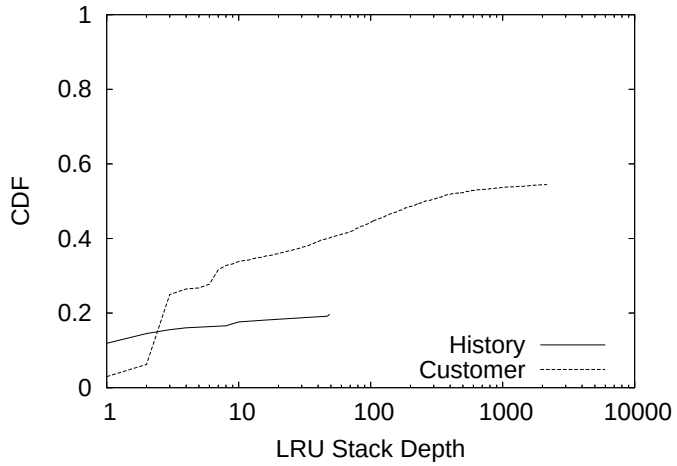


Figure 3.11: The LRU Stack Depth of the History and Customer Tables

Figures 3.9, 3.10, and 3.11 show a sharp increase in the cumulative LRU stack depth for almost all tables when the buffer size is less than 5 pages, which is reflected as the "knee" in the figures. Since one page contains many table rows or index values, the upper layer of a DBMS typically accesses a database page several times during a short interval. These references are correlated references, and the period that correlated references occur is called the *period of correlated references*. The sharp increase in the cumulative LRU stack depth is caused by correlated references. This sharp increase does not reflect the real access probability of the page references in the long term. Therefore, it is important to treat the correlated references differently than other references when designing buffer cache replacement algorithms. In the LRU-K algorithm [84], a parameter is used to define the period of correlated references so that re-references occurring in this period are not counted as the k^{th} last reference to the page. In the 2Q algorithm [60], the length of the short term queue reflects the estimation of the period of correlated references. In the CART algorithm [8], the length of the L1 queue reflects the estimation of the period of correlated references.

As shown in figures of this subsection, different tables often have different

reference characteristics. This is caused by the different ways the applications operate on the tables. If the data in a table are accessed in a highly skewed manner (as in Item), the high popularity of references to the hot data can cause highly skewed cumulative LRU stack depth distribution. The reference behaviour of each table can guide the partitioning of the buffer cache. Tables with similar skewness of their cumulative LRU stack depth distribution can get a high hit ratio with a small buffer cache. The hit ratio will not increase much if more buffer cache space is given to these tables. On the other hand, the hit ratio of tables with less skewed cumulative LRU stack depth distribution will keep increasing even when the buffer cache is large. This suggests that tables with skewed cumulative LRU stack depth distribution should be put into a small partition, and tables with less skewed cumulative LRU stack depth distribution should be put into a large partition.

3.2.4.3 Characteristics of Data And Indexes

The structures of data pages and index pages are different. Data pages have a linear structure, and are often stored sequentially on disks. Index pages employ the B^+ tree structure to facilitate fast search by key values. Accesses to index pages always start from the root and go through all levels of the tree until either the search fails or the appropriate leaf is reached. Therefore, different referencing behaviour is expected on data pages and index pages.

Figure 3.12 shows the stack depth distribution for references to data pages and index pages. 70% of references to data pages have infinite stack depth, while the cumulative LRU stack depth distribution of the index pages is much more skewed.

Analysis of the reference behaviour for data pages and index pages can guide the partitioning of the buffer cache. Because data pages and index pages have different reference behaviour, they could be put into different partitions of a buffer cache and managed separately.

Recall that Figure 3.11 shows that references to the Customer table have low skewness in their cumulative LRU stack depth distribution. The data pages and the index pages are studied separately and their LRU stack depth distributions are

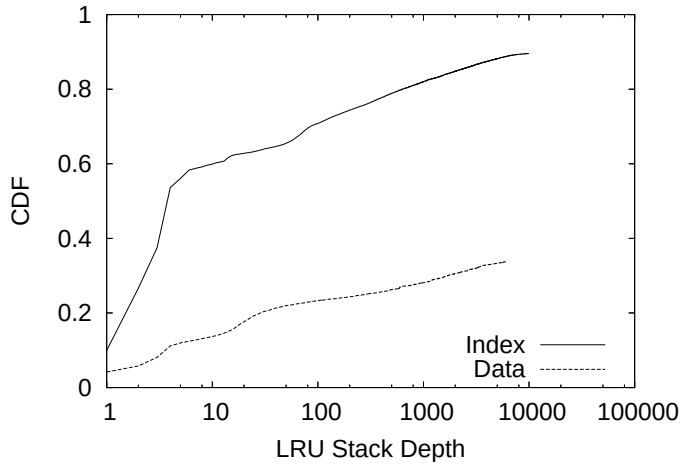


Figure 3.12: The LRU Stack Depth of Data and Indexes

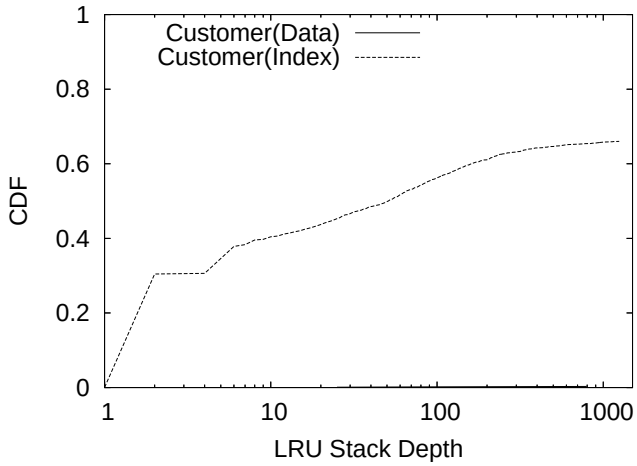
shown in Figure 3.13. Nearly all the data pages have infinite stack depth while the index pages have shape similar to that of all index pages. In fact, all index pages have similar access patterns regardless the table they belong to. Note that if a larger size of the sample is studied, the percentage of data pages that have infinite stack depth may decrease, although the shape of the line is expected to be similar.

3.2.5 Multi-user Workload Characteristics

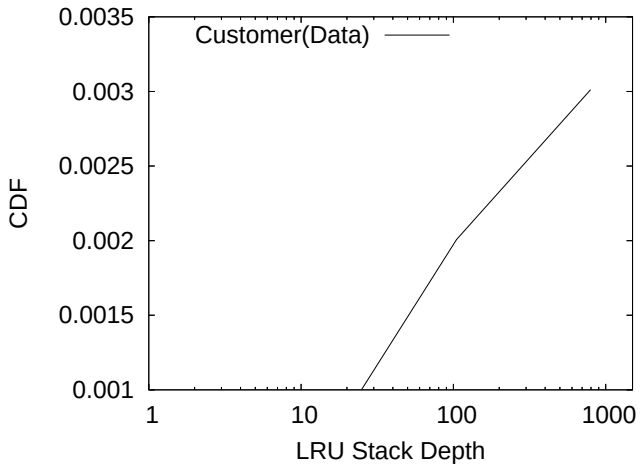
All the above analyses are based on the requests generated by a single user. When many users use the DBMS at the same time, as in the case of the TPC-C benchmark, different users share pages of the same database. The aggregate reference behaviour is affected by this sharing. Since the workload does not change over time, the first 100,000 requests sent by each user formed the basis of the following analyses.

Figure 3.14 shows the LRU stack depth of the overall reference trace for 1 user and 60 users. When the buffer cache is larger than 125 pages, the hit ratio of the trace with 60 users is similar to that of the trace with 1 user. This is because all users in the TPC-C benchmark access the database in the same pattern as defined by the non-uniform random generator.

As in the single user case, the sharp increase in the buffer cache hit ratio at



(a) Data and Index



(b) Zooming in on the Data line

Figure 3.13: The LRU Stack Depth of the Customer Table
[Figure 3.13(b) is a zoom in of Figure 3.13(a) so that the line of *Customer(Data)* is visible.]

small buffer cache sizes (less than 125 pages) in Figure 3.14 indicates that there are correlated references. Since the correlated references from different users interleave with each other, the period of correlated references increases when the number of users increases. This can be seen in the figure where the knee occurs at a larger buffer cache size. If the buffer cache replacement algorithm needs to estimate the period of correlated references (e.g., LRU-K and 2Q), this parameter should be tuned according to the workload to get good performance.

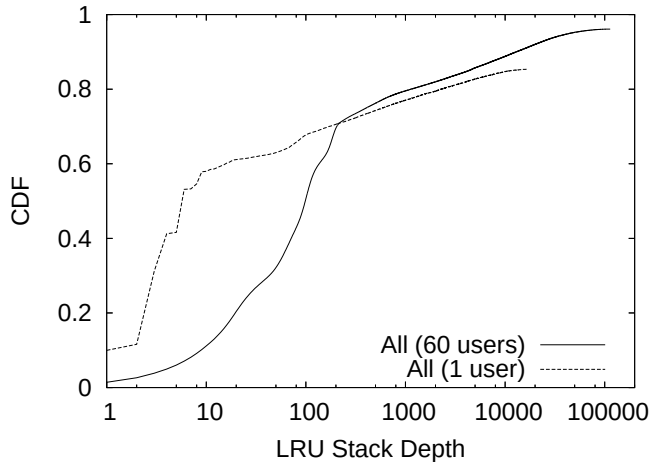


Figure 3.14: The LRU Stack Depth for Sixty Users

3.2.6 Summary

This section presents the results from an analysis of the reference behaviour of the TPC-C benchmark. Correlated references exist in most tables. The period of correlated references is affected by the type of references to tables, type of objects (data or index), and number of users. If the period of correlated references is used as a parameter in the buffer cache replacement algorithm (as it is in LRU-K and 2Q), this parameter should be tuned according to these factors.

Different tables exhibit different reference behaviour, depending on how the data are accessed by the applications. The data pages and index pages also exhibit different reference behaviour. The index pages of all tables have similar reference behaviour. The reference behaviour of different tables and the data/index pages suggest that the buffer cache can be partitioned to group pages with similar properties into one partition.

Chapter 4

Self-tuning of Buffer Cache Management

Tuning a buffer cache management algorithm to achieve good performance in a real system is often complex. Many parameters must be turned according to the workload and system configuration. Such tuning is often difficult, however, and in many cases can be done only by trial and error.

This chapter investigates the tuning of buffer cache management in the context of a specific DBMS, IBM DB2 7.1.0 for Windows. Because OLTP workloads have high I/O demand, buffer cache management is crucial to the performance in OLTP systems. The TPC-C benchmark, which represents an OLTP workload, is used as the workload. Although a specific system and benchmark workload were used in this study, the methodology used is generalizable and can be applied to other systems and workloads.

The remainder of this chapter is organized as follows. Section 4.1 discusses the buffer cache management algorithm being studied. Section 4.2 presents the methodology used. Sections 4.3 studies the I/O activities in the buffer cache and the impact of important parameters in buffer cache management. Section 4.4 describe a new self-tuning algorithm to automatically tune the page cleaning activity of buffer cache management. Section 4.5 presents simulation results of the performance of the new algorithm. Section 4.6 summarizes the chapter.

4.1 Buffer Cache Management Overview

Figure 4.1 shows the structure of the DB2 buffer cache, which is managed by a replacement algorithm and a page cleaning algorithm. Since many users can use

DB2 simultaneously, there is one database clerk (a thread or a process) corresponding to each active user. Each clerk processes that user's queries and accesses database pages in the buffer cache.

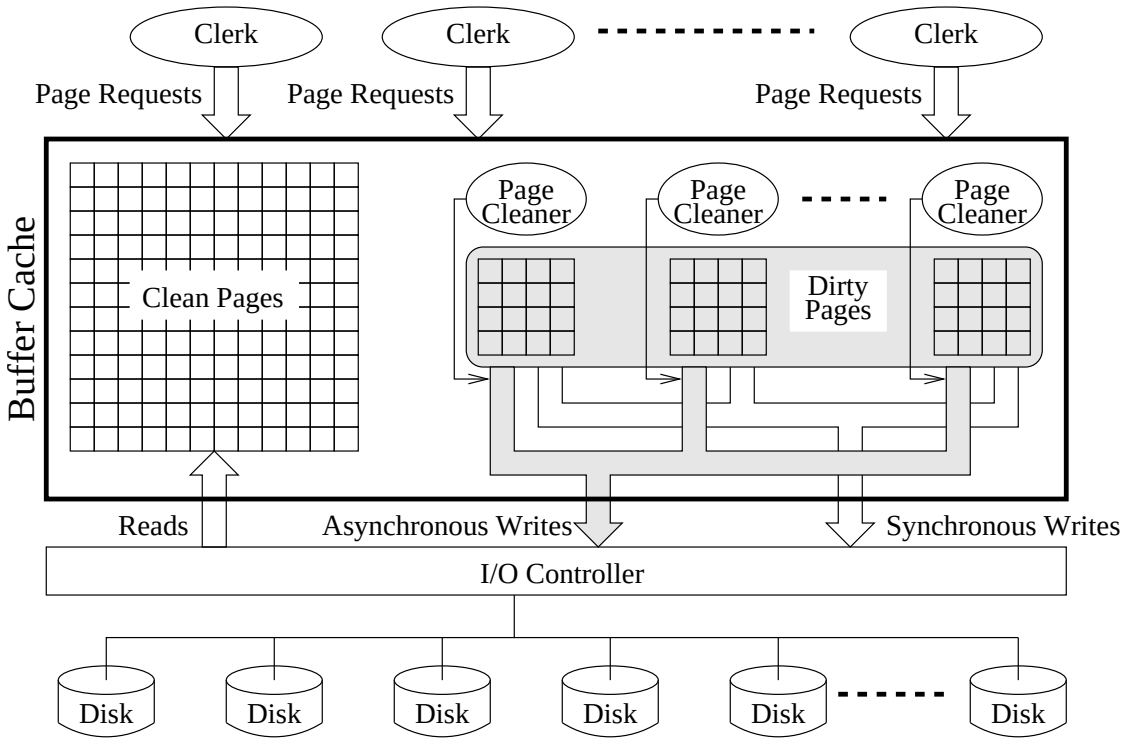


Figure 4.1: The Structure of the Buffer Cache

When a new page needs to be read into a full buffer cache, the replacement algorithm must select a page for replacement. When a page is selected, its status is checked; if it is clean (i.e., unchanged since it was fetched into cache), the space it occupies can be used immediately, but if it is dirty (i.e., changed), a *synchronous write* must take place before the user clerk can fetch the new page. This is a blocking activity that seriously impacts performance.

In addition to synchronous writes, DB2 also uses page cleaners to perform *asynchronous writes* to maintain a pool of free pages for replacement. It does this to avoid the blocking inherent in synchronous writes. Each page cleaner manages a subset of the dirty pages as shown in Figure 4.1. All page cleaners are asleep initially. When a page cleaner wakes up, it collects some dirty pages and writes them to disk. Since the writes generated by page cleaners are performed

asynchronously, the user clerks are not blocked for the writes. Thus the user clerks can continue to request buffer cache pages from the buffer cache while asynchronous writing is taking place.

Because read latency is reduced when free pages are available for incoming pages, the page cleaning speed can affect system throughput significantly. The page cleaning speed can be controlled by the number of page cleaners, which can be set by the database administrator before the database application starts. Simulation results presented later in this section show that tuning the number of page cleaners to a proper value can improve system throughput by as much as 19%. If this parameter is manually tuned, when the system configuration or workload changes (e.g., more disks or memory are used, the database becomes larger, or more users are using the system), the tuning must be performed again. Moreover, because of the complexity of different workloads, manual tuning usually is done by experiments, which is difficult for the following reasons:

- A workload of sufficient length must be available to determine how the system performs under a particular setting.
- Each performance "experiment" must run long enough to skip the buffer cache warmup period and to eliminate statistical fluctuations resulting from short-term transient effects.
- The database must contain enough data to provide a realistic operating environment.

4.2 Methodology

Both simulation and measurement were used to study the tuning of the buffer cache management. As was stated in Section 3.2.1 (page 41), the performance metric of interest is *throughput*, measured as the number of New Order transactions completed per minute.

4.2.1 System Configuration and Experimental Setup

A TPC-C test environment was configured on the PC Server 704 described in Section 3.2.2 (page 45). A TPC-C database with 50 warehouses was created with a size of about 5GB. One dedicated disk of the machine is used for the log file of DB2. The buffer cache of DB2 can be configured up to 440MB, which is large compared to the size of the database used. The database can be created across from 3 to 11 physical disks. To get the best performance, software RAID-0 managed by Windows NT instead of RAID-5 was used to organize multiple disks. In the experiments, the number of disks used by the TPC-C database was 11 and the size of the buffer cache was 380MB, unless otherwise stated.

When the TPC-C benchmark is running, new data are appended to the database. Therefore, the database becomes larger which impacts the system throughput adversely. In order to fairly compare the throughput under different configurations, the database was backed up after the TPC-C database is first populated, and was restored to the initial state before each measurement session. Each measurement session lasted for 30 minutes. Only the throughput values when the system enters stable state were used to calculate the average throughput. The coefficient of variance of the throughput values, i.e., the ratio of the standard deviation over the average, is less than 0.03.

Since the focus of this study is server performance, remote terminal emulators required by TPC-C were not used to generate the transactions. Instead, all transactions were generated on the DBMS server by a TPC-C driver program. The think time between transactions was removed to test the maximum throughput that the DBMS can achieve. A TPC-C driver program developed by IBM's Toronto Software Laboratory was used. When the TPC-C benchmark is running, 60 users send OLTP transactions to the DBMS.

4.2.2 The Buffer Cache Simulator

A buffer cache simulator was written to simulate DB2's buffer cache management algorithm and the disk subsystem [133]. An event-driven architecture was used in the simulator. Different components of the simulator communicate through events. Figure 4.2 shows the components and the main event types. The four basic components are:

- The *Buffer Cache Manager* contains the basic buffer cache management algorithm. It manages the placement and replacement of the buffer cache pages. It accepts the fix and unfix events, and sends out the read and synchronous write (SyncWrite) I/O events. It also notifies Page Cleaners to start cleaning by sending StartCleaning events.
- The *Page Cleaner* manages the page cleaning of the buffer cache. It accepts StartCleaning events and performs page cleaning on the dirty pages of the buffer cache. There can be more than one page cleaner in the simulator. Both the Buffer Cache Manager and the Page Cleaner can access *Buffer Cache Pages* which is the data structure holding all the pages of the buffer cache.
- An *Clerk* represents a client that sends requests to the Buffer Cache. Requests belonging to each client are organized into a separate *Trace File*. Each Clerk simply reads a record (either fix or unfix) from its trace file and sends it to the Buffer Cache Manager. The number of Clerks is equal to the number of clients when running TPC-C.
- The *Disk* accepts disk I/O events (Read, SyncWrite, and AsyncWrite) and returns IOFinish events. Because the I/O requests in TPC-C are random reads and writes of one page, a simple disk model with a fixed disk access time is used. Unprocessed requests to a disk are queued and served in a first-in-first-out order. When there are multiple disks in the simulator, different disks can perform reads and writes simultaneously.

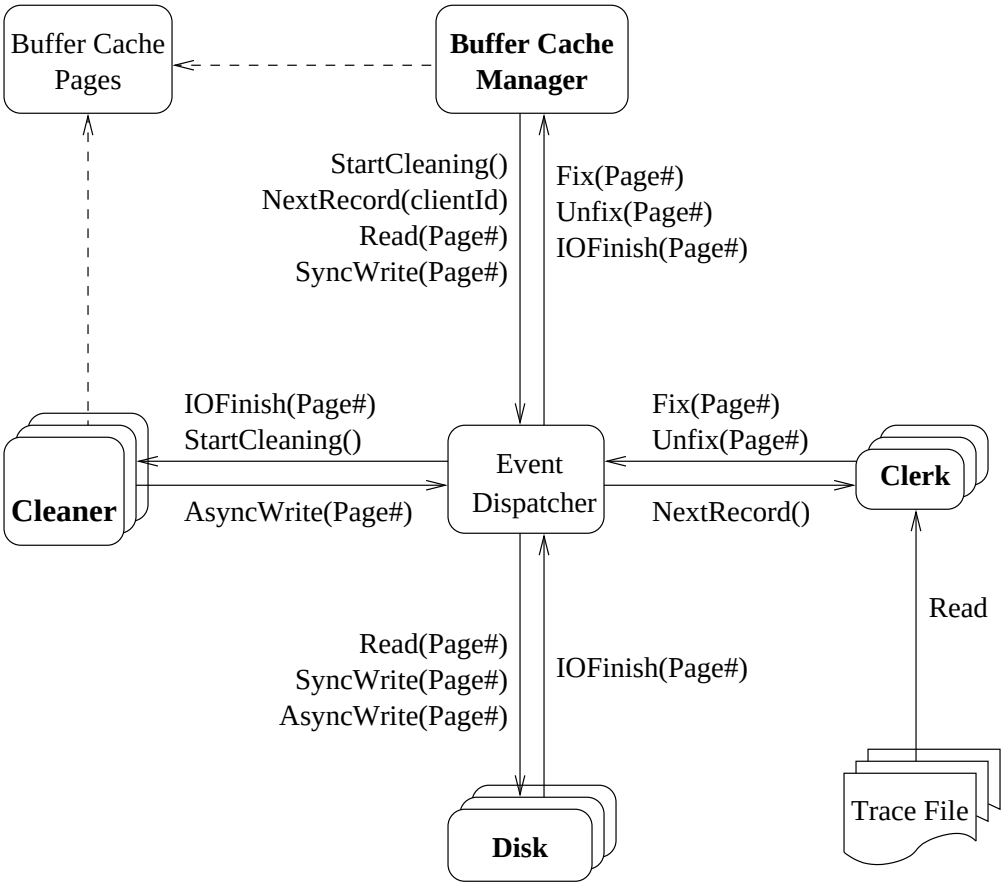


Figure 4.2: The Structure of the Simulator

A timestamp is associated with each event. All events are sent to the event queue and sorted by their timestamps. The *Event Dispatcher* selects the event with the minimum timestamp and sends it to the corresponding component. The typical event flow when fixing a page is as follows: a fix event is read from the Clerk and sent to the Buffer Cache Manager. If the page is in the buffer cache, the fix finishes, and the Buffer Cache Manager sends a NextRecord event to the Clerk for the next record. If the page is not in the buffer cache, and a clean page is found for the replacement, a Read event is sent to the Disk, and when the Buffer Cache Manager receives an IOFinish event, it sends a NextRecord event to the Clerk asking for the next record.

Some initial experiments confirmed that neither the log disk nor the CPU is the bottleneck under the hardware configuration used. Instead, the performance bottleneck is the disks storing the TPC-C database. Therefore, only the disk access time was modeled in the simulator. The disk access time was set to 9ms, which is close to the disk access time of the real disks used when many requests are queued at each disk.

Since DB2 is the particular DBMS studied in this thesis, special attention was taken to implement the DB2 replacement algorithm. The source code of DB2 was studied during the summer of 1999 in the IBM Toronto Software Laboratory to understand its buffer cache management algorithm. The DB2 source code related to buffer cache management is about 100,000 lines of C code. There are about 2,500 lines of C++ code related to the DB2 buffer cache management algorithm after it is implemented in the simulator, since only the key parts of the algorithm were simulated. The components that do not significantly affect the performance of the system under the OLTP workload, such as error handling, prefetch, and logging, were not simulated.

The simulator reports the number of transactions finished per minute as the throughput of the system. Other quantities related to page activities, I/O channel, and the buffer cache management algorithms are also reported. Because TPC prohibits the disclosure of TPC-C performance results that have not been audited

by independent auditing agencies, the absolute values of any simulation or experimental results are withheld, and only normalized values are presented in this chapter. This does not compromise the performance comparisons.

4.2.3 Simulator Validation

A number of measurement and simulation experiments were performed to validate the simulator. Some internal statistics from the measurements are also compared with that from the simulator.

Figure 4.3 shows the system throughput measured in both the simulator and DB2 under the untuned configuration. The simulation results are quite close to those obtained from measurement. Both begin with an empty cache. Throughput improves as pages are cached until the cache is full, at which time replacement decisions must be made. The throughput spike marks the point when the page cleaners of the buffer cache begin to write dirty pages back to the disk. After that, the system performance stabilizes.

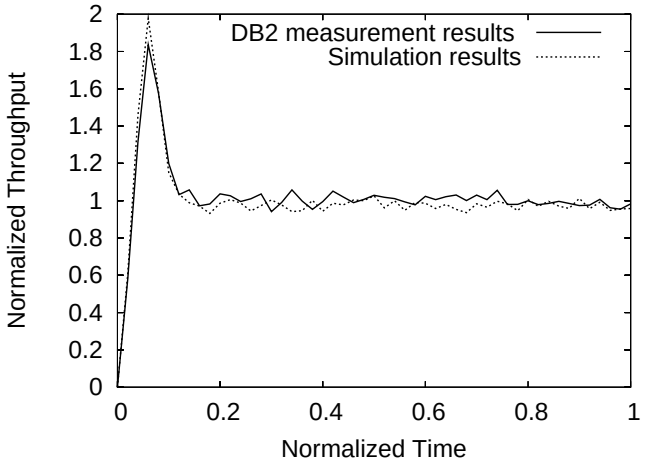


Figure 4.3: Simulation vs. Measurement
[The configuration is untuned, with 2 page cleaners. The database spans over 11 disks. The throughput is normalized relative to the average throughput of the measurement results after the system enters the stable state. The running time is normalized relative to the total running time of the measurement experiment.]

Some other outputs of the simulator were also compared to measurement

results as shown in Table 4.1. After the system throughput becomes stable, the percentage of dirty pages in the buffer cache from simulation (87%) closely matches the measurement result (85%). The buffer cache hit ratio from simulation (96.6%) is also very close to the measurement result (96.5%). Later results on the impact of the number of page cleaners (see Section 4.3.2 in page 66) also show a close match between the simulator and the real system.

Table 4.1: Comparison of Measurement and Simulation Results

	Measurement	Simulation	Relative difference
Percentage of Dirty Pages	85%	87%	2.4%
Buffer Cache Hit Ratio	96.5%	96.6%	0.1%

These comparisons confirm that the simulator has behaviour very similar to the real system, indicating that conclusions drawn from simulation results are valid for the real system. New algorithms designed in the simulator should have similar effects if applied to the real system.

4.3 Experiments with the Page Cleaning Algorithm

4.3.1 I/O Activities in the Buffer Cache

A number of simulation and measurement experiments were conducted to investigate I/O behaviour in the buffer cache and the impact of the page cleaning algorithm on performance.

In initial experiments, it was found that about 90% of the pages in the buffer cache were dirty, which seemed high. This motivated more experiments to investigate the distribution of pages in the buffer cache. Figure 4.4 shows the evolution of pages in the buffer cache over the first 30% of the simulation. At the beginning, all pages in the buffer cache are free pages. Both the number of dirty pages and the number of clean pages increase as time goes on. After the buffer

cache is full, the number of dirty pages continues to increase, but the number of clean pages drops. At the same time, the throughput drops. When 90% of the buffer cache pages are dirty, the system enters steady state. At this point the number of clean pages is much lower than it is when the buffer cache is just full, implying that there are too many dirty pages in the buffer cache in steady state.

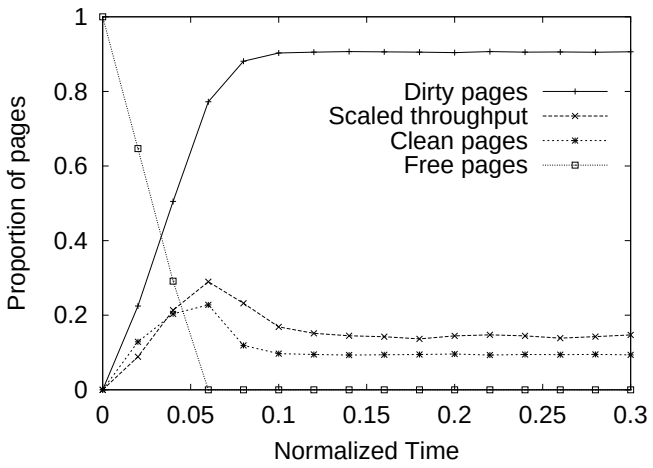


Figure 4.4: Pages in the Buffer Cache in the Untuned Configuration
[The untuned configuration has 2 page cleaners. The throughput is scaled so that its shape can be compared easily with the other curves plotted. The number of pages is normalized relative to the buffer cache size.]

To investigate the reason for this, the I/O activities of the buffer cache were examined in more detail, and the results are shown in Figure 4.5. When the buffer cache is almost full, the page cleaners begin to clean out dirty pages by asynchronous writes. However, asynchronous writes cannot clean out pages fast enough in this untuned configuration, so dirty pages must be selected for replacement. This means that synchronous writes occur. The synchronous writes not only delay the reads directly (since a read cannot proceed before the synchronous write finishes), but also compete for I/O bandwidth with other activities. Therefore, the read speed is slowed down (and read latency is increased) by the need to write in order to create space for the incoming pages. When the read speed becomes slower, the throughput drops and dirty pages are generated more slowly (i.e., fewer pages in the buffer cache are changed per unit time).

When the number of dirty pages generated by the TPC-C requests equals the number of dirty pages cleaned by writes in the same time interval, the system enters steady state.

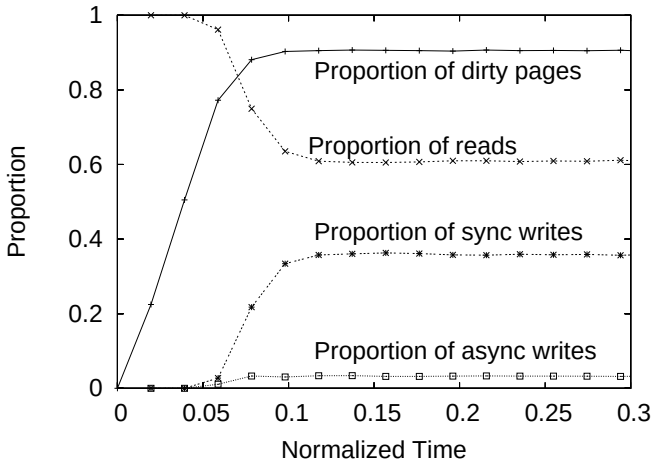


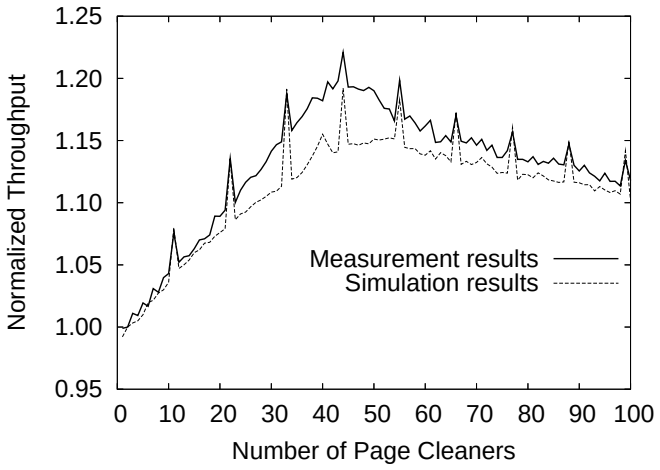
Figure 4.5: I/O Activities of the Buffer Cache in the Untuned Configuration
[The untuned configuration has 2 page cleaners.]

As Figure 4.5 shows, the proportion of synchronous writes is high (close to 40% of all I/O activity) in this configuration, which implies that the page cleaning speed is too low. The number of asynchronous writes should be increased in order to decrease the number of synchronous writes. To do this, the aggregate page cleaning speed must be increased, and this can be done by using more page cleaners.

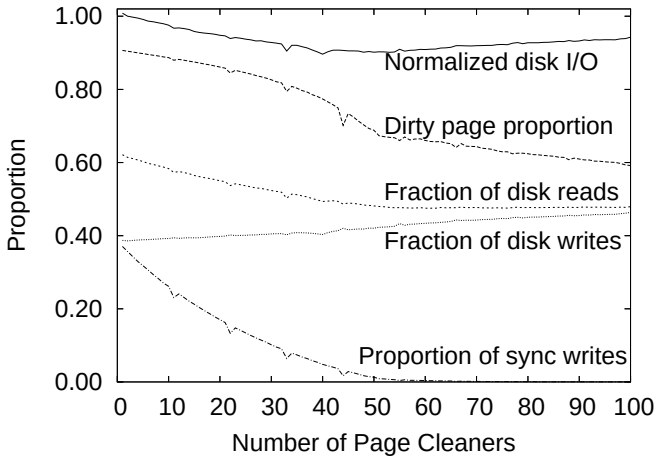
4.3.2 The Impact of the Number of Page Cleaners

Figure 4.6 shows the effect of varying the number of page cleaners from 1 to 100. Figure 4.6(a) shows the effect of the number of page cleaners on throughput. The simulation results match the measurement results quite closely. When the number of page cleaners is an integral multiple of the number of physical disks, which is 11 in this case, better load balance across disks can be achieved. Therefore, the performance spikes occur on these specific points.

When the number of page cleaners is less than 44, the throughput generally



(a) Impact on Throughput



(b) Impact on Dirty Pages, Synchronous Writes, and Disk I/Os

Figure 4.6: Impact of Multiple Page Cleaners

[In Figure 4.6(a), all throughput values are normalized relative to the throughput under the untuned configuration. In Figure 4.6(b), the number of disk I/Os is normalized relative to the disk I/Os under the untuned configuration.]

increases with more page cleaners. After that point, however, putting more page cleaners to work does not improve performance any further. More is not always better. The selection of the appropriate number of page cleaners is clearly important in tuning such a system.

Figure 4.6(b) shows the effect of the number of page cleaners on various buffer cache characteristics: dirty pages, synchronous writes, disk reads, disk writes, and total I/Os. When the number of page cleaners increases, the number of read misses drops and the number of write misses increases. The number of disk I/Os first decreases then increases. Increasing the number of page cleaners reduces both the proportion of dirty pages and the proportion of synchronous writes. Further increasing the number of page cleaners after the proportion of synchronous writes is close to 0 brings no additional benefits: the number of read misses becomes almost flat; the decrease of dirty pages slows down; the number of disk I/Os starts to increase; and throughput drops. Figure 4.6(b) shows a criterion for tuning the page cleaning activity: the number of page cleaners should be tuned to the minimum number so that the synchronous writes are just eliminated. A self-tuning algorithm for changing the page cleaning speed based on this principle is described in the next section.

The number of page cleaners in the untuned configuration is 2. The system achieves peak throughput when the number of page cleaners is 44. Figure 4.7 shows the peak throughput and the throughput under the untuned configuration. Figure 4.8 shows the I/O activities and the proportion of dirty pages with 44 page cleaners: there are very few synchronous writes left and the proportion of dirty pages drops significantly.

Other simulation experiments were performed on systems with different numbers of disks and different buffer cache sizes, and the effect of the number of page cleaners on throughput is very similar to Figure 4.6(a), although the locations of the spikes are different because the number of disks is different. The number of page cleaners that achieve peak throughput for each system configuration is shown in Table 4.2. This value is always an integral multiple of the

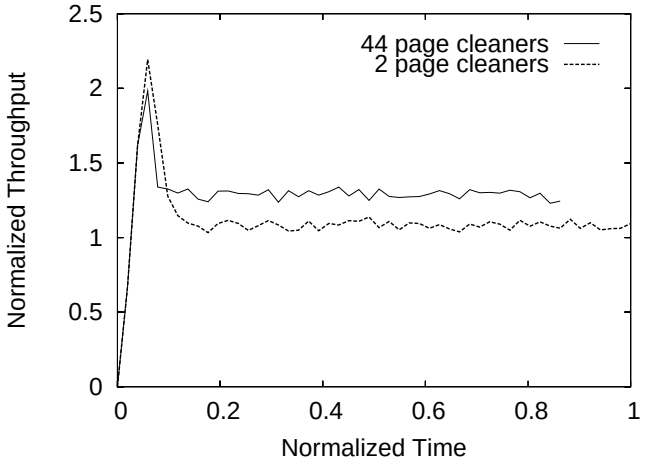


Figure 4.7: Effect of Number of Page Cleaners

[The run time on the x-axis and the throughput on the y-axis are normalized relative to the run time and average throughput with 2 page cleaners. The line with 44 page cleaners finishes earlier since it has higher throughput and finishes processing the trace in shorter simulated time.]

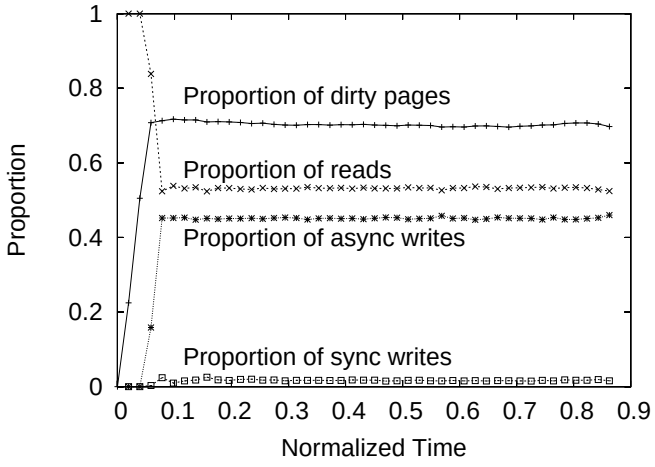


Figure 4.8: I/O Activities with 44 Page Cleaners

number of physical disks, but it is different under different configurations, which increases the difficulty of tuning the number of page cleaners.

Table 4.2: Number of Page Cleaners That Achieves Peak Throughput

Number of Disks	7	8	11	15
Buffer Cache=380MB	49	48	44	45
Buffer Cache=440MB	49	56	44	45

4.4 A Self-tuning Algorithm for Page Cleaning

The number of page cleaners that maximizes system performance is different for different workloads or different configurations. Tuning this parameter manually is difficult and time-consuming. A new self-tuning page cleaning algorithm was designed to overcome this problem [134], the objective of which is to maximize throughput by dynamically changing the page cleaning speed.

In the page cleaning algorithm used in DB2 7.1.0 for Windows, each page cleaner collects many pages and sends out one page at a time for cleaning. One more page is sent out after the previous write is done.

The number of page cleaners is fixed in the self-tuning algorithm to keep the algorithm simple. In order to change the page cleaning speed without changing the number of page cleaners, each page cleaner keeps more than one outstanding asynchronous write. A parameter N is introduced for this purpose – a real number whose integral part $\lfloor N \rfloor$ indicates the number of outstanding asynchronous writes kept by each page cleaner. The page cleaner compares the number of outstanding asynchronous writes with $\lfloor N \rfloor$ whenever an asynchronous write sent by this page cleaner finishes. If there are more than $\lfloor N \rfloor$ asynchronous writes outstanding, the page cleaner stops sending new writes to the disks; otherwise, more writes are sent to the disks until the number of outstanding asynchronous writes sent by this cleaner equals $\lfloor N \rfloor$. N thus has the same effect as the number of page cleaners in the current algorithm: the bigger the N value, the faster the page cleaning speed.

The initial value of N is its minimum value 1. N is adjusted periodically in order to dynamically tune the page cleaning speed to its desired value. An *adjustment interval* is defined for this purpose. Some statistics of the buffer cache and the disk activities are collected during each adjustment interval. N is adjusted at the end of each adjustment interval based on the data collected.

An adjustment goal must be defined so that N can be adjusted to make the system achieve the goal. The results presented in Section 4.3.2 show that the page cleaning speed should be increased to the point where the number of synchronous writes just reaches zero. It is easy to determine the number of synchronous writes that occur in any adjustment interval, but it is hard to tell whether the page cleaning speed is too high if the observed number of synchronous writes is zero. As Figure 4.6(b) shows, the number of synchronous writes is zero when the number of page cleaners is more than necessary. Therefore, adjusting the number of synchronous writes to zero may cause unnoticed high cleaning speed which impacts performance adversely. Instead, the self-tuning algorithm seeks to keep the *proportion* of synchronous writes small (say, 5%).

The notation used in describing the adjustment operation performed in each adjustment interval is summarized in Table 4.3.

Table 4.3: Notation for the Self-tuning Algorithm

Symbol	Definition
N	Number of outstanding asynchronous writes kept by each page cleaner
w_o	Proportion of synchronous writes observed in an adjustment interval
w_d	The desired proportion of synchronous writes
Δ	The scale parameter

At the end of each adjustment interval, the following adjustment is performed:

$$N \leftarrow \max(1, N + \Delta \cdot (w_o - w_d)) \tag{4.1}$$

During each adjustment interval, the number of synchronous writes and total number of disk I/Os are observed. The ratio between them is the observed

proportion of synchronous writes, w_o. At the end of every adjustment interval, w_o is compared with the desired proportion of synchronous writes w_d. The greater the difference between w_o and w_d, the more N needs to be changed. The change to N should be proportional to $|w_o - w_d|$. The value of $\Delta \cdot (w_o - w_d)$ in Equation 4.1 shows the amount that N needs to be changed. The scale parameter Δ is used to amplify the difference between w_o and w_d. If w_o equals w_d, the current page cleaning speed is the desired value and N can remain unchanged. If $w_o > w_d$, the proportion of synchronous writes is more than desired and so N needs to be increased to clean pages faster. If $w_o < w_d$, the proportion of synchronous writes is less than desired, which indicates that the page cleaning speed is too high. Thus $\Delta \cdot (w_o - w_d)$ is negative and its absolute value indicates the amount that N should be decreased. Since the minimum value of N is 1, the use of the max function guarantees that $N \geq 1$ after the adjustment. The use of a real number N instead of an integer N is important for this algorithm to work. The minimum possible value of w_o is 0, and w_d is close to 0, so it is easy to have $\Delta w_d < 1$, which indicates that the maximum amount of adjustment to N is less than 1. If an integer N is used and N is too large at some point, N would be impossible to decrease and defeat the purpose of the self-tuning.

4.5 Simulation Results

The results of simulation experiments with the self-tuning algorithm are presented in this section. The algorithm uses three parameters – *Adjustment Interval*, Δ, and w_d. The parameter values were arbitrarily choose and listed in Table 4.4. These values were used to generate the simulation results presented in this section. Experiments in the latter part of this section indicate that the performance of this self-tuning algorithm is not sensitive to particular parameter values.

Figure 4.9 shows the throughput of the system under the untuned configuration (2 page cleaners), the best manually tuned configuration (44 page cleaners), and the self-tuning algorithm. The performance of the self-tuning algorithm is close to

Table 4.4: The Parameter Values

Parameter	Value
Adjustment Interval	1 second
w_d	5%
Δ	7.5

that of the best manually tuned system. The throughput of the best manually tuned system is 19.2% higher than that of the untuned configuration, and the throughput of the self-tuning algorithm is 16.3% higher. This result shows that the self-tuning algorithm performs comparably to the best manually tuned system.

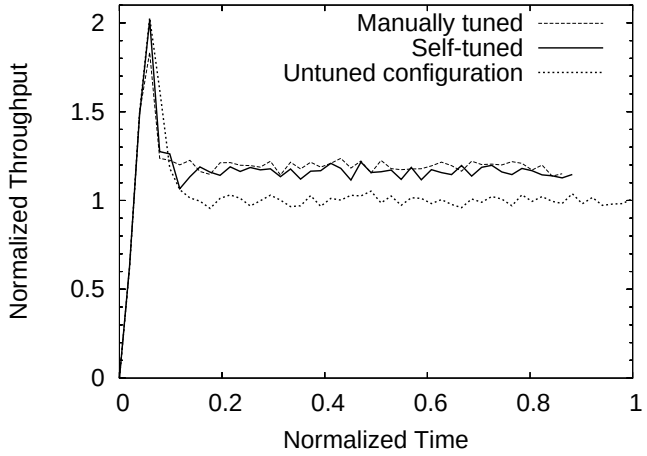


Figure 4.9: Throughput Comparison
[The throughput values are normalized relative to the average throughput of the untuned configuration after the system enters the stable state.]

Figure 4.10 shows the system I/O activities when running the self-tuning algorithm. The proportion of synchronous writes is kept very close to 5%, which is the value of w_d, indicating that the self-tuning algorithm can effectively control the proportion of synchronous writes. Because of the higher page cleaning speed, the proportion of dirty pages is lower than that of the untuned configuration.

Figure 4.11 shows how the parameter N is adjusted over a ten-minute period. The value of N fluctuates in a small range (between 3 and 5), because the characteristics of the TPC-C workload do not change.

This self-tuning algorithm was tested in several other system configurations

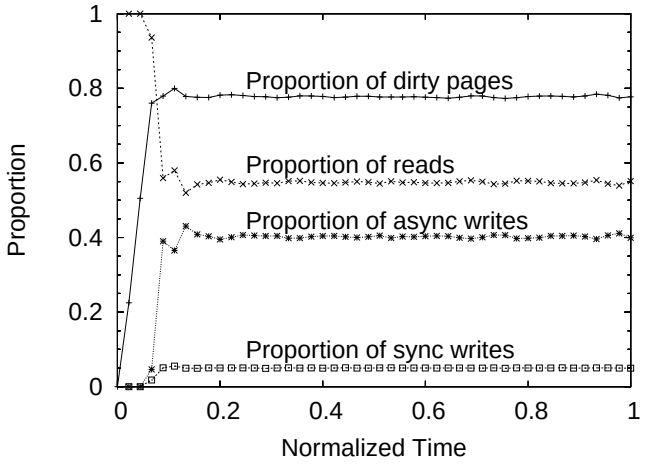


Figure 4.10: I/O Activities with the Self-tuning Algorithm

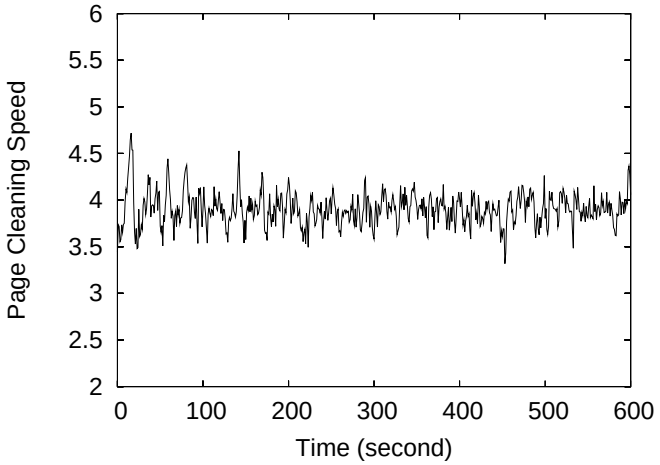
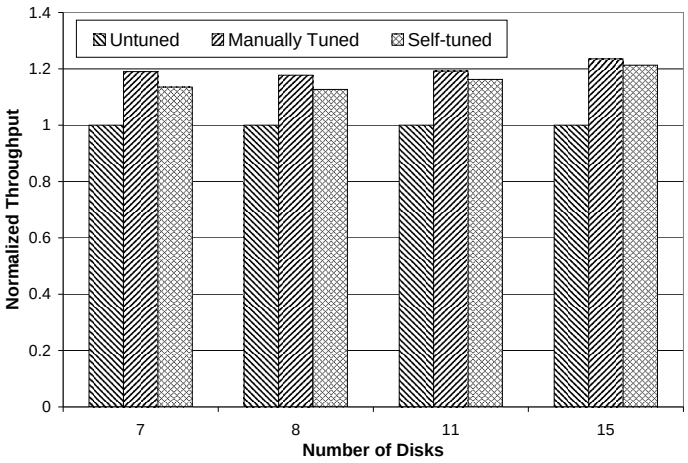
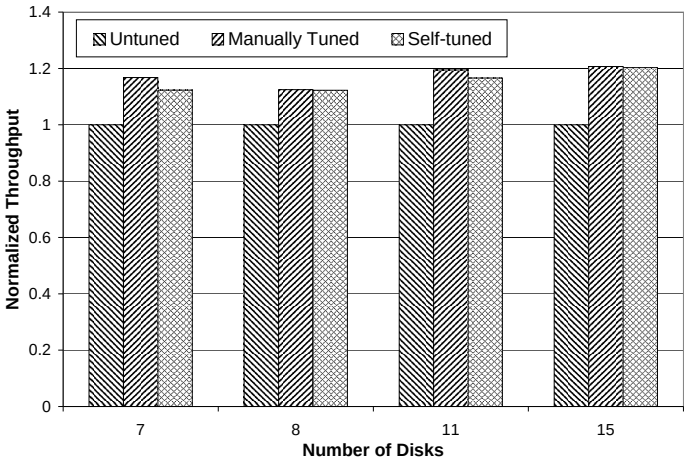


Figure 4.11: How Page Cleaning Speed is Adjusted

(different numbers of physical disks and different buffer cache sizes). The results are summarized in Figure 4.12. These results confirm that the self-tuning algorithm performs close to the best manually tuned algorithm.



(a) Buffer Cache Size = 380MB



(b) Buffer Cache Size = 440MB

Figure 4.12: Comparison of Self-tuned and Manually Tuned Algorithms
[All throughput values are normalized relative to the throughput under the untuned configuration.]

In order for this algorithm to be robust, the performance must not be unduly sensitive to the selection of values for the three parameters (Adjustment Interval, w_d, and Δ). More simulation experiments were performed to determine the sensitivity of the results to the values of these parameters. All throughput values in the following figures are normalized relative to the average throughput under

the parameter values shown in Table 4.4.

Figure 4.13 shows the impact of the adjustment interval on performance. Even though the adjustment interval is varied from 0.1 seconds to 120 seconds (three orders of magnitude), the system throughput changes by less than 1%. Figure 4.14 shows the system throughput with very small adjustment interval. The system throughput drops when the adjustment interval is close to the average disk access time. These results show that as long as the adjustment interval is several times longer than the average disk access time (10-20ms for typical hard drives), there is no significant difference in performance. A small interval permits the system to respond promptly to a workload change, while a large interval can reduce system overhead. Since the workload of TPC-C does not change in the simulation experiments performed, the adjustment interval is not important to throughput.

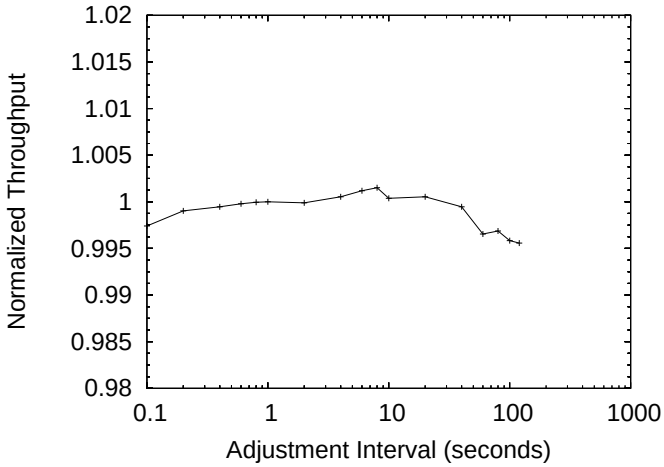


Figure 4.13: Impact of the Adjustment Interval

Figure 4.15 shows that when the desired synchronous write proportion w_d changes from 0.2% to 10%, the throughput also varies by less than 1%. This indicates that as long as w_d is a small value, performance does not change significantly.

Figure 4.16 shows the impact of the scale parameter Δ under two different adjustment intervals. Again the performance difference is within 1%. The results of these experiments indicate that the performance of the self-tuning algorithm is

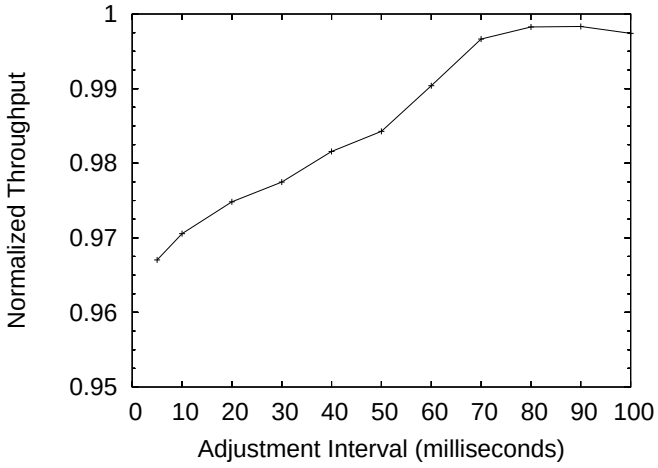


Figure 4.14: Impact of Small Adjustment Interval

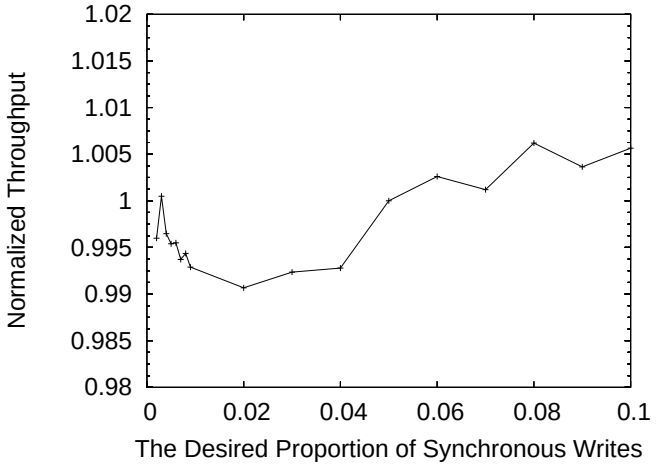


Figure 4.15: The Effect of Parameter w_d

not sensitive to the parameter values.

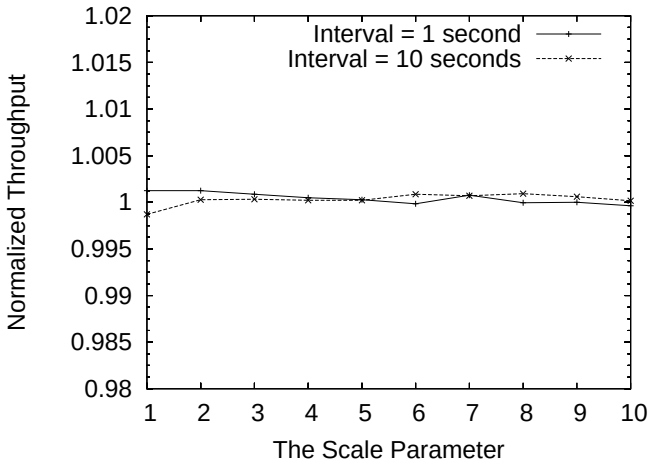


Figure 4.16: Impact of the Scale Parameter Δ

4.6 Summary

Tuning resource management algorithms for best performance is often difficult in a large scale system. The analysis of the I/O activities of the buffer cache of DB2 running the TPC-C benchmark shows that properly tuning the number of page cleaners is important to performance. A new self-tuning algorithm was developed to automate this tuning task. By monitoring the I/O activities of the buffer cache, the self-tuning algorithm can achieve throughput comparable to that of the best manually tuned algorithm. The performance of this algorithm is not sensitive to the value of its parameters.

Chapter 5

Lock Contention in Buffer Cache Management

Most buffer cache replacement algorithms, such as LRU, LRFU [66], EELRU [112], AFC [20], LIRS [59], ARC [76], and Eviction-Based Replacement [18] have a global data structure to manage the buffer cache. This data structure is modified on every cache access, and the accesses must be serialized by a lock to protect its integrity. In systems with multiple processors and a large number of threads, lock contention may become an issue when multiple threads access the global data structure, and this could significantly impact system performance. Lock contention has been observed in some real systems [10, 105], and no general solution has been proposed. Some systems use a CLOCK-based algorithm to reduce contention [25], since this kind of algorithm does not modify the global data structure on buffer cache hits. However, a CLOCK-based algorithm typically has a lower hit ratio than other replacement algorithms and may cause poor system performance. Some systems use a variation of LRU without global data structures [10, 105], but these solutions either have high overhead or cannot be applied to other replacement algorithms.

This chapter investigates the performance impact of lock contention in buffer cache replacement algorithms and proposes a new management approach that eliminates contention without sacrificing hit ratio or increasing the overhead of a replacement algorithm. Section 5.1 discusses the motivation for this study. Section 5.2 defines the context and definitions used. Section 5.3 describes the methodology used to conduct this study. Section 5.4 analyzes the impact of contention on performance and various design tradeoffs involved. Section 5.5 proposes a multi-region approach to managing the buffer cache. Section 5.6 evaluates the performance of the multi-region approach. Section 5.7 discusses the

design tradeoffs in large scale systems. Section 5.8 concludes this study.

5.1 Motivation

A large scale computer system may contain many disks, multiple processors, and lots of memory. Depending on the configuration and the workload running on it, different components of the system may become the performance bottleneck. This study focuses on heavy system load situations because this is where system performance really matters.

When a system performs simple operations on large amounts of data (e.g., a file server or a storage server), disks are busy while processors have much idle time. Disk access is the performance bottleneck in this kind of system, and such a system is called an *I/O-bound* system. On the other hand, when a system performs complex operations on a small amount of data (e.g., 3D image rendering), the processors are fully utilized while the disks have much idle time. Such a system is called a *CPU-bound* system.

When a system has multiple processors and many concurrent threads accessing the global data structure of the buffer cache frequently, accesses to the global data structure must be serialized by a lock. This lock may become a contention point which threads frequently wait for. When lock contention happens, both processors and disks have much idle time and system resources are wasted. Such a system is called a *contention-bound* system.

A large scale system needs to support different kinds of workloads and may experience different performance bottlenecks. The buffer cache management algorithm in a large scale system should perform well for all these workloads, no matter where the performance bottleneck is. Different performance bottlenecks pose challenges on different aspects of a buffer cache replacement algorithm. Tradeoffs of these aspects must be considered when designing a robust buffer cache replacement algorithm.

In an I/O-bound system, disk access is crucial to system performance. Two

approaches are often used to improve performance:

- *Increasing the buffer cache hit ratio.* This can be done by using a buffer cache management algorithm which has a higher hit ratio, or by using more memory for the buffer cache. Typically a replacement algorithm with a higher hit ratio has higher overhead, e.g., LRU-2 has a higher hit ratio and higher overhead than LRU. If the system has CPU idle time, the higher overhead of the new replacement algorithm can overlap with disk I/Os, and thus has no impact on the throughput of the system.

- *Increasing disk throughput.* This can be done by using faster disks or more disks.

In a CPU-bound system, the system can replace current processors with faster processors or use more processors. A buffer cache replacement algorithm with lower overhead (such as replacing LRU-2 with LRU, or replacing LRU with CLOCK) is preferred, but these low-overhead algorithms often have lower hit ratio, which causes more disk I/Os. If these extra disk I/Os cannot be overlapped with CPU computation, the overall system performance can decrease.

In a contention-bound system, the CPU and disk resources are wasted since many threads are waiting for the right to access the global data structure managing the buffer cache. Simply adding more hardware cannot resolve the bottleneck, while the replacement algorithm needs to be changed. Lock contention can be avoided by using algorithms without global data structures, such as Random, but this could severely decrease the hit ratio of the buffer cache, resulting in poor system performance when the system is I/O bound. The solutions used in real systems are situational (case-by-case) and ad hoc. SQL Server 7.0 uses a CLOCK-based algorithm to reduce contention [25], but CLOCK-based algorithms have lower hit ratio than many other replacement algorithms, and such algorithms could decrease system performance when the system is I/O-bound. BerkeleyDB uses a chained hash table to approximate LRU with low contention [10], but this approach works only for LRU and has high overhead when a page is put back to

the hash table, which happens frequently. ADABAS [105] uses many LRU lists to reduce contention. It accesses each list in a round-robin manner, and so a page can be on different lists at different times. This approach cannot maintain page histories for each list, which is a data structure commonly used in many replacement algorithms, such as LRU-2 [84], 2Q [60], LIRS [59], ARC [76], etc.

To improve cost-effectiveness, many large-scale systems are configured as *balanced* systems for the peak workload, where the utilization of processors and disks is high. Using a contention-free algorithm with lower hit ratio may require lots of additional memory and/or disks to keep the system balanced.

A buffer cache replacement algorithm with no contention, high hit ratio, and low overhead is important for large scale systems to achieve good performance at low cost.

5.2 Context and Definitions

A system with an in-memory buffer cache and an array of disks (RAID) is considered in this study. Both the cache and the disks are managed in units of fixed-size pages. *Write back* is assumed to get good performance.

Because the lock contention of the buffer cache happens only on systems with multiple users, it is assumed that the cache receives many streams of logical requests simultaneously and that each stream of requests is sent by one user – there is no think time between consecutive requests sent by each user. The disks have the ability to process multiple physical reads and writes concurrently, because different disks can serve different requests at the same time and each disk can queue several requests to achieve better disk I/O throughput[1]. To achieve the maximum system throughput, it is assumed that different users do not work on the same page at any time so that there are no data dependencies among users. Although this assumption is largely true for data pages, it is not true for pages

[1]Only SCSI disk can queue multiple requests, while most IDE disks can only handle one request at a time.

storing metadata. Special optimizations can often be made in real systems, however, by using a much smaller lock granularity on the metadata pages. Without this assumption, the workload would be intrinsically limited by the data dependencies among concurrent users and would not be able to benefit from improvement in buffer cache replacement algorithms. The system throughput, which is measured as the total number of logical requests performed by all users per unit time, is used to reflect the performance of the system.

It is assumed that a user performs some computation on a page between logical requests. The behaviour of a "typical" user is illustrated in Figure 5.1. The time to execute lines 2–4 is called the *lock interval* T_l, and the time to execute line 5 is called the *work interval* T_w. The *lock proportion*, r, is defined as: $r = \frac{T_l}{T_l + T_w}$. If a disk I/O or several disk I/Os are required to obtain this page from the buffer cache, only the pages performing disk I/Os are locked and the global buffer cache lock is not held during the disk I/O operations. Therefore, the disk I/O time is considered as part of the work interval instead of the lock interval.

```
1.  while (true) {
2.      lock;
3.      send a logical request to cache;
4.      unlock;
5.      work on this page;
6.  }
```

Figure 5.1: Workload Model of a "Typical" User

5.3 Methodology

Analytical modeling, simulation and emulation were used in this study. The performance of buffer cache management is first investigated in controlled environments, using a simulator and a micro-benchmark. The proposed approach is then analyzed by modeling and evaluated by simulation. Finally, the new approach is tested in a more realistic environment, using a buffer cache emulator.

5.3.1 Buffer Cache Simulator

LRU, Random, CLOCK, LRU-2 [84], 2Q [60], LIRS [59], and ARC [76] were implemented in the buffer cache simulator. These algorithms were selected because they are either popular (e.g., LRU and CLOCK), simple (e.g., Random, LRU, CLOCK, and 2Q), or recent research efforts (e.g., LIRS and ARC). LRU-2 was selected because it is representative of replacement algorithms with $O(\log n)$ overhead.

The implementations of these algorithms were first verified manually using small artificial traces. The LIRS implementation was verified further by comparing against Jiang's LIRS simulator [59]. Two different implementations of each of LRU, Random, CLOCK, and 2Q were developed independently and compared against each other. LRU-2 was implemented using two different approaches: LRU2-F (LRU-2 fast miss) uses a heap, which has $O(\log n)$ overhead on both page hits and page misses; LRU2-S (LRU-2 slow miss) uses an array, which has $O(1)$ overhead on cache hits and $O(n)$ overhead on cache misses because the whole cache must be searched to find the victim page. These two LRU-2 implementations were verified against each other.

Some of these algorithms use tunable parameters. All experiments in this study used the same set of parameter values, as suggested by the authors of the algorithms. For simplicity, the same parameter values as those of 2Q were used in LRU-2, because 2Q is a low-overhead approximation of LRU-2. In 2Q, the size of the short term queue is $0.25c$ and the size of the history queue is $0.5c$, where c is the cache size; in LRU-2, the correlated reference period is $0.25c$ and the size of the history list is $0.5c$; in LIRS, the size of the free list is $0.01c$ and the maximum size of the LRU stack is unlimited.

5.3.2 Contention Micro-benchmark

A contention micro-benchmark was used to emulate the contention caused by cache replacement algorithms. This micro-benchmark was designed to make lock

contention the bottleneck so that lock contention can be studied by measuring system throughput. In the micro-benchmark, a controller thread creates one worker thread for each user to access the cache. Each worker thread performs the computation shown in Figure 5.2. The *mutex* is used to emulate the contention point when accessing the global cache data structure. *Simulated overhead* and *simulated work* represent some computation operations (specifically, string comparison and copying), the length of which can be controlled. The throughput of the micro-benchmark is expressed as the total amount of simulated work performed by all users in a unit time. The contention micro-benchmark can be configured so that each user accesses a different mutex, in which case the maximum throughput can be achieved since there is no contention.

```
1.  while (true) {
2.      lock(mutex);
3.      simulated overhead;
4.      unlock(mutex);
5.      simulated work;
6.  }
```

Figure 5.2: Thread Computation Model

The micro-benchmark was implemented on Windows 2000 using native threads. The mutex implementation was ported from BerkeleyDB [11] with a change to make it work well under high contention situations: the Windows Event object of the mutex is created during initialization.

The test machine for the contention micro-benchmark was an IBM x255 server in the DISCUS laboratory at the University of Saskatchewan, with 4 Xeon 1.5 GHz processors with HyperThreading (i.e., each physical processor has two logical processors), 8GB of memory, and 12 IBM 36.4GB 15k hot swap disks connected to two UltraSCSI 160 controllers.

5.3.3 Buffer Cache Emulator

The simulator can compute only read and write hit ratios, but not overhead. The contention micro-benchmark can measure the impact of contention on throughput, but not overhead or hit ratios. Therefore, an emulator was used to provide a more realistic environment for studying all three aspects. The emulator manages a real buffer cache in memory and performs real disk I/Os on cache misses. In the emulator, a trace generator thread can either read a real trace file or generate a synthetic trace on-the-fly. One or more emulated users (threads) keep reading requests from the trace generator and sending them to the cache. Each thread performs some simulated work on the acquired page (specifically, string comparison and copying). Other than the simulated work, there is no overhead or think time in between consecutive requests. The 32-bit Xeon processor where the emulator runs allows a maximum of 4GB of virtual address, where 3GB can be used by user applications in Windows 2000 Advanced Server. In many real systems, a typical page size is 4KB or 8KB. In the emulator, a smaller page size of 1KB was used so that a buffer cache with many more pages could be emulated.

In the emulator, the cache data structure is protected using the lock mechanism, while the integrity of page data is not preserved. This relaxation does not compromise the performance of the buffer cache replacement algorithm and its lock contention effect, but greatly simplifies the implementation of the cache emulator.

5.3.4 Workloads

Traces of different types of workloads were used in this study, including NFS file servers, email servers, OLTP, decision support, e-commerce, and web search engines.

A *TPC-C* trace was collected when running the TPC-C benchmark on IBM DB2 on Windows NT Server 4.0. As was mentioned previously, TPC-C is an OLTP benchmark. Three *TPC-W* traces were collected when running the TPC-W

benchmark implementation [125] with the shopping, browsing, and ordering configurations on IBM DB2 8.1 Open Beta 2 on Windows NT Server 4.0. Dynamic caching was not used on the web server side. TPC-W is an e-commerce benchmark. These two categories of traces are not filtered by the upper level cache.

Twenty-six *TPC-H* traces were collected when running a 300 GB TPC-H benchmark on Informix Extended Parallel Server 8.30FC2 on HP-UX 11.00 64-bit, including all tests required by the TPC-H benchmark [124]: 24 single-query power tests (22 queries and 2 update queries) and 2 multi-query throughput tests. TPC-H is a decision support benchmark. The six *Openmail* traces were one-hour traces collected in 1999 from six EMC 3700 servers running HP's OpenMail, collected during the servers' busy periods. The two *Financial* traces [113] were disk I/O traces collected from OLTP applications running in two large financial institutions. The three *WebSearch* traces [113] were disk I/O traces collected from a popular web search engine. These four categories of traces were collected at the I/O controller level and have been filtered by first level caches.

The *NFSEmail* trace [32] is an one-day trace collected in October 2001 from an NFS server at Harvard University. This trace is dominated by email activities. Since the size of directories is not known from the trace, the metadata operations (15% of the requests) are discarded and only data read and write requests (85% of the requests) are used. Requests in this trace have been filtered by the NFS client caches, but not the NFS server cache.

A representative set of experimental results for these traces is presented. The characteristics of the traces involved are listed in Table 5.1.

5.4 Analysis of Contention

5.4.1 Spin Lock and Contention

Several locking algorithms can be used to protect the access to global data. A lock variable shared among threads is used to indicate whether the lock is already held.

In the busy wait algorithm, the thread requesting the lock keeps polling the lock variable value in a tight loop. This algorithm is useful only when there are multiple processors and the worst case wait time is much shorter than the context switch overhead. In the typical wait algorithm provided by most operating systems, the thread requesting the lock goes to sleep if the lock is not available, and will be woken up by the operating system once the lock is available. At least two context switches are involved in this procedure. This algorithm is useful when the expected waiting time is much larger than the overhead of a context switch. The spin lock algorithm is a combination of the above two algorithms. In spin lock, the thread requesting the lock first polls the lock variable repeatedly for a short period of time, then goes to sleep if the lock is still not available. If the lock becomes available during the polling, the context switch overhead can be avoided. Spin lock is useful when the average waiting time is much shorter than the overhead of a context-switch and the worst case waiting time is much longer.

Typically spin lock is used to protect the global data structures of the buffer cache replacement algorithm in multi-processor systems. When used in

Table 5.1: Trace Characteristics

Trace	Number of Requests ($\times 10^6$)	Number of Reads ($\times 10^6$)	Number of Writes ($\times 10^6$)	Number of Unique Pages ($\times 10^6$)	Page Size (Bytes)
NFSEmail	28.69	21.05	7.64	1.18	8K
Financial1	36.11	5.56	30.55	7.69	512
Financial2	17.69	13.88	3.81	2.47	512
OpenMail 1	10.98	4.16	6.82	3.01	1K
OpenMail 2	8.66	2.47	6.19	2.69	1K
OpenMail 3	7.97	1.98	5.99	2.34	1K
OpenMail 6	19.91	11.86	8.05	5.37	1K
TPC-C	209.23	176.46	32.77	0.98	4K
TPC-W Shopping	60.08	60.06	0.02	0.14	4K
TPC-H Query 11	0.57	0.57	0.002	0.30	128K
WebSearch1	2.00	2.00	0.0002	0.80	8K
WebSearch2	8.63	8.63	0.001	1.11	8K
WebSearch3	8.21	8.20	0.004	1.11	8K

single-processor systems, the busy wait part of the spin lock is disabled. In a system supporting many threads, if the lock is not available when a thread attempts to acquire it, the thread spins a certain time and may go to sleep and be woken up later. The overhead spent on the spin and the context switches is called *lock contention*, which can significantly decrease system throughput.

5.4.2 Factors Impacting Contention

Some replacement algorithms, such as Random, do not need global data structures to manage the pages in the cache. Some algorithms, such as CLOCK and LRU2-S, need to update the global data structures only on cache misses. Since cache misses are often much fewer than cache hits and much higher latency is allowed for misses (because of the slow disk I/Os required), the lock contention in these algorithms can be ignored. These algorithms are called *contention-free* algorithms. Most other algorithms (such as LRU, LIRS, etc) must update their global data structures on both hits and misses. The lock contention in these algorithms can become a performance problem.

For algorithms that update their global data structures on both cache hits and cache misses, lock contention can be affected by several factors, including the number of processors, the number of threads, and the lock proportion. Since higher lock contention causes lower throughput, throughput is used to indicate the extent of lock contention. The contention micro-benchmark is used to examine the impact of various factors on contention. Figure 5.3 compares the throughput with and without lock contention. The figure shows that the lock proportion is the most significant factor affecting contention, and the number of processors is also an important factor. Contention happens in two situations: one is that multiple running threads attempt to acquire the lock simultaneously, which happens more frequently with more processors and larger lock proportion; another is that a thread acquires the lock and is preempted by the operating system before releasing the lock so that all other threads must perform the busy wait and sleep, which

causes more contention with more threads and larger lock proportion. In typical time-sharing systems, threads are preempted every tens to hundreds of milliseconds, and the time a thread holds the lock is normally hundreds of thousands of times shorter. The chance that a thread is preempted while holding the lock is small. Therefore, the number of threads affects contention only slightly.

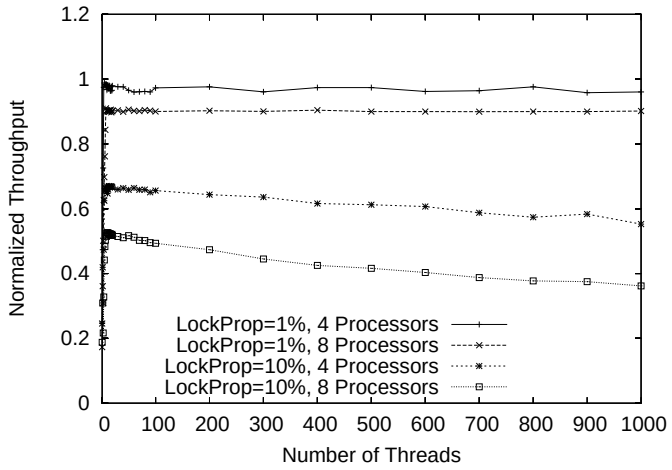


Figure 5.3: Impact of Contention on Throughput
[The throughput values are normalized to the maximum throughput obtained by letting each user access a different mutex. Lower throughput indicates higher contention.]

5.4.3 Tradeoffs Among Contention, Hit Ratio, and Overhead

5.4.3.1 Hit Ratio

Using a simple contention-free replacement algorithm such as CLOCK can eliminate contention while sacrificing the hit ratio. In I/O-bound systems, system throughput is a monotonously increasing function of the buffer cache hit ratio. When the buffer cache is large relative to the data size, the hit ratio differences among different cache replacement algorithms become very small, as shown in Figure 5.4. It is tempting to conclude from this figure that the selection of replacement algorithms is not important when the cache is large. It might also be

concluded that CLOCK is a good replacement algorithm, since it does not cause contention on cache hit while the difference of its hit ratio compared to other algorithms is small.

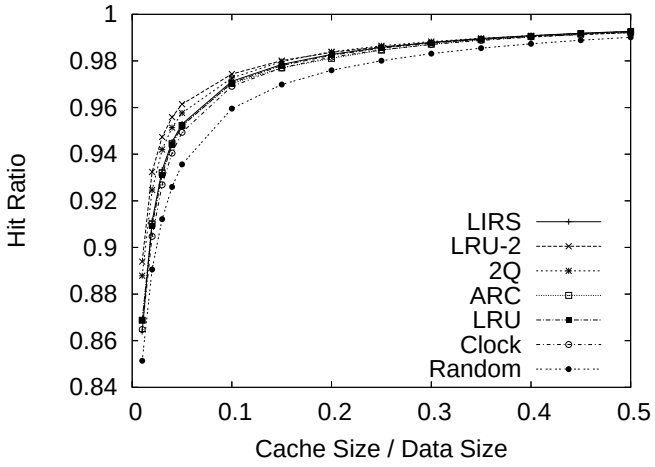


Figure 5.4: Hit Ratio of Replacement Algorithms with the TPC-C Trace

A small difference in hit ratio, however, does not necessarily result in a small difference in system throughput [66, 141]. In an I/O-bound system, CPU computation time is overlapped by disk I/O time, and the system throughput is the number of logical I/Os finished per unit time. The execution time is proportional to the disk I/O time. Therefore, the system throughput can be defined as

$$\frac{1}{T_{pg} \cdot R},$$

where T_{pg} is the disk I/O time of one page, and R is the buffer cache miss ratio. $\frac{1}{R}$ is called *cache speedup*, which represents the performance improvement the system can achieve using the buffer cache as compared to directly accessing the disks without using buffer cache. The more effective the algorithm, the higher the cache speedup. The minimum possible cache speedup is 1, which happens when the overall miss ratio is 1 (i.e., no cache). Cache speedup goes towards infinity and loses its meaning when the buffer is large and the miss ratio is very close to 0. This is not a problem since the system is not likely to be I/O-bound when the miss ratio is very low, where factors other than cache speedup, such as CPU

computation or lock contention, determine the system throughput. Emulation experiments were conducted to confirm that the cache speedup computed from measured miss ratio varies in proportion to the measured throughput, as shown in Table 5.2. The system is I/O bound in these emulation experiments. Therefore, the amount of emulated work per page access only affects the CPU utilization other than the throughput of the system.

Table 5.2: Comparison of Cache Speedup and Measured Throughput
[The results were measured in the emulator. The numbers in the last two columns are normalized relative to Random. The trace is TPC-C with 60 users, the number of processors is 8, the data are on a 10-disk RAID-10 array, the ratio of cache size to data size is 5%.]

Algorithm	Cache Speedup	Measured Throughput (requests / second)	Normalized Cache Speedup	Normalized Throughput
Random	10.57	2990.05	1	1
CLOCK	12.89	3627.68	1.22	1.21
LRU	13.71	3858.84	1.30	1.29
LRU2-F	13.72	3929.32	1.30	1.31
2Q	14.85	4150.44	1.41	1.39
LIRS	14.02	3957.59	1.33	1.32
ARC	13.48	3823.02	1.28	1.28

Figure 5.5 shows the cache speedup of the different replacement algorithms on the same trace as Figure 5.4. The results were obtained from the buffer cache simulator. The figure shows that the absolute differences in throughput among different replacement algorithms become larger when larger caches are used, which is contrary to the trend shown in Figure 5.4. Simulation results on other traces had similar trends and are not shown here.

Figure 5.6 shows the cache speedups of a number of replacement algorithms as a function of cache size under various traces. The y-axis is the cache speedup normalized to that of the Random algorithm under the same cache size. As shown in the figure, when the cache size is less than 30% of the data size, the normalized cache speedups show larger differences on larger caches, while the practical cache size in most large systems is less than 25% of data size. Since the WebSearch1 and Openmail6 traces are already filtered by the first level cache, the performance of

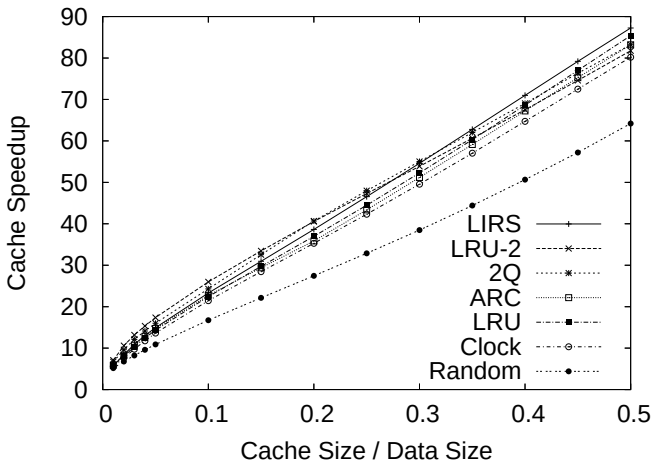
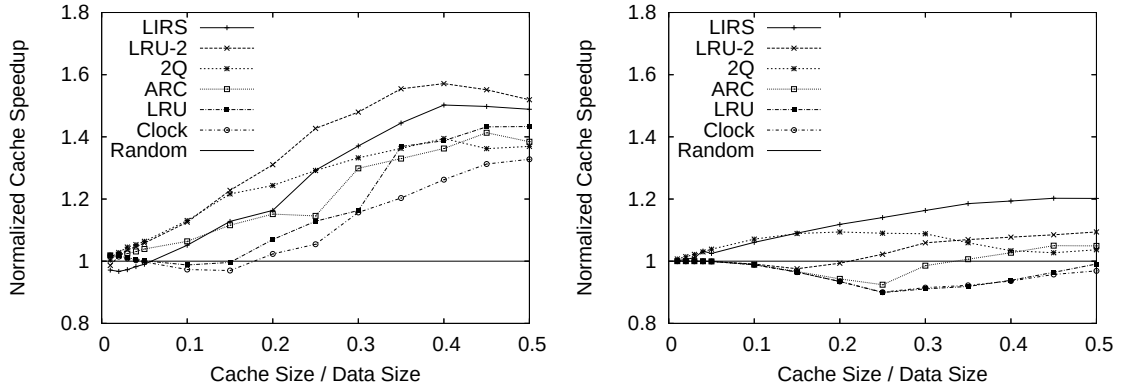


Figure 5.5: Cache Speedup of Replacement Algorithms with the TPC-C Trace

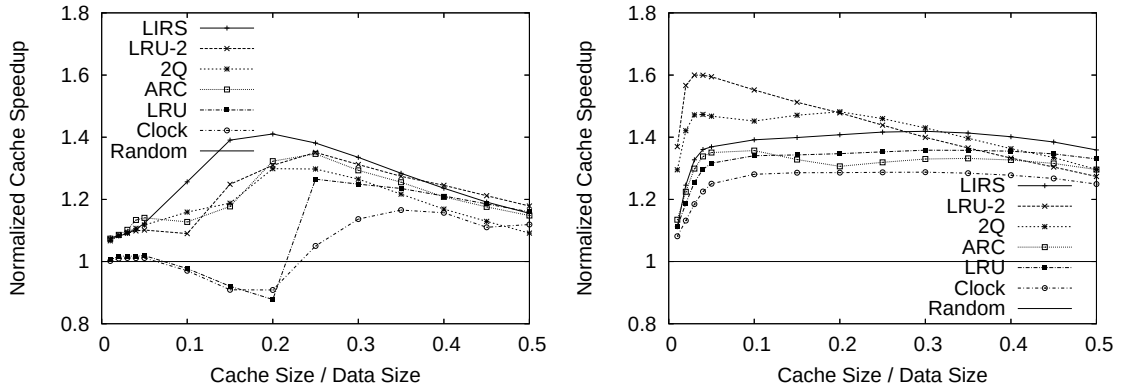
LRU and CLOCK are worse than Random at small caches because of poor temporal locality [137]. Based on the points where the cache speedup quickly increases, the size of the first level cache is about 25% of the data size for the WebSearch1 trace (Figure 5.6(b)) and 20% for the Openmail6 trace (Figure 5.6(c)). The results shown in Figure 5.6 imply that the selection of replacement algorithms has a larger impact on system throughput under large cache sizes than under small cache sizes, which runs contrary to the common belief that replacement algorithms perform similarly under large cache sizes. The figures show that CLOCK performs worse than all algorithms except Random, and so using CLOCK to reduce lock contention may significantly decrease the system throughput when the system is I/O-bound.

5.4.3.2 Overhead

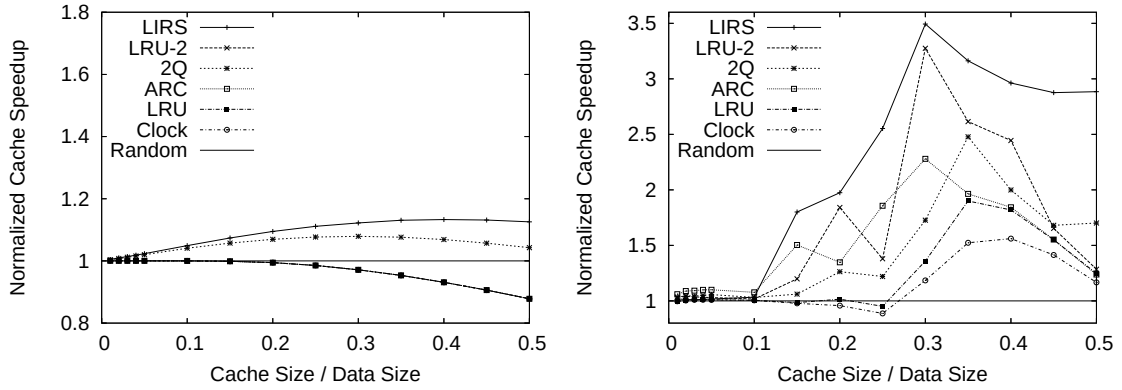
Every replacement algorithm suffers some CPU overhead when managing buffer cache. The overhead to handle a buffer cache hit is called the *hit overhead*, and the overhead to handle a buffer cache miss is called the *miss overhead*. Since the miss overhead is insignificant by comparison with the time consumed by the one or more disk I/Os following a miss, the algorithm overhead means simply *hit overhead* in the rest of this chapter unless otherwise stated.



(a) NFSEmail (b) WebSearch1



(c) Openmail6 (d) TPC-C



(e) TPC-H Power Query 11 (f) TPC-W Shopping

Figure 5.6: Normalized Cache Speedups

[In Figure 5.6(e), LRU-2, ARC, LRU, and CLOCK overlap. Figure 5.6(f) uses a different scale on y-axis because of its much larger range of normalized cache speedups.]

Most existing replacement algorithms (such as CLOCK, LRU, 2Q [60], LIRS [59] and ARC [76]) have constant overhead. Some algorithms (such as LRU-K [84], FBR [93] and LRFU [66]) have $O(\log n)$ overhead, where n is the size of the buffer cache. The common belief is that $O(\log n)$ overhead is too large to be useful. Although some designers think $O(1)$ overhead is small enough to be negligible, others struggle to reduce the overhead further, such as by using CLOCK to approximate LRU.

In a multi-user workload, overhead can affect the system throughput only when the system is CPU-bound. File servers, storage servers and web servers typically copy the pages to the network after they are read into the buffer cache. In these systems, disks are often the bottleneck, and the overhead of cache replacement algorithms does not usually affect system throughput. Database workloads vary. OLTP workloads are often I/O-bound and decision support workloads are often CPU-bound. Real database workloads are typically a mix of both, such as many production workloads [51, 52] and e-commerce workloads [33]. Since "it is extremely rare to have access to real production workloads" [52], OLTP and decision support benchmarks were used to represent the real workloads. The analysis of overhead for these benchmarks can be used to speculate on the algorithm overhead in real database workloads.

In CPU-bound systems, the impact of algorithm overhead on overall performance is affected mainly by *lock proportion*, which is the proportion of the time that the system spends on "overhead" operations[2].

The *pgbench* program of PostgreSQL was used as the OLTP workload. Pgbench is an implementation of the TPC-B benchmark [124]. Although TPC-B is simpler than TPC-C, previous studies [9] found that its memory behaviour is

[2]In order to get realistic lock proportion values in database systems, PostgreSQL [90] version 7.3.4 compiled on cygwin [83] was instrumented to record the lock proportion on the test machine described in Section 5.3.2 (page 84). The instrumentation of PostgreSQL is non-intrusive since there were no observed performance differences. To remove the effect of disk I/O, the database size was set to be much smaller than system memory, and all disk data required by the testing benchmarks were pre-loaded into the buffer cache of the operating system by running the query more than once. PostgreSQL manages its buffer cache by LRU [91].

similar to that of TPC-C. The measured average lock proportion of pgbench is 7%.

The DBT3 benchmark developed by the Open Source Development Lab [26] is a simplified implementation of the TPC-H benchmark, which is a popular decision support benchmark. Similar to TPC-H, DBT3 has 22 queries. Figure 5.7 shows the measured average lock proportion of these queries. The lock proportions vary between 0.5%-8.5%, but most are around 2%. These results show that decision support workloads have smaller lock proportion than OLTP workloads. This is because OLTP applications often perform relatively simple work on each page, e.g., search in an index page or update one record of a page, while a decision support application often needs to perform more complex computations such as scan, sort, join, etc. To be conservative, larger lock proportion is assumed. 1% and 10% are used as the lower and upper bounds of lock proportion in subsequent experiments. The lock proportion of systems such as storage servers and I/O controllers may be higher than 10%, since they simply copy the data from the buffer cache to the network. These systems are likely to be I/O-bound, however, and so the contention and overhead of replacement algorithms do not affect the system throughput.

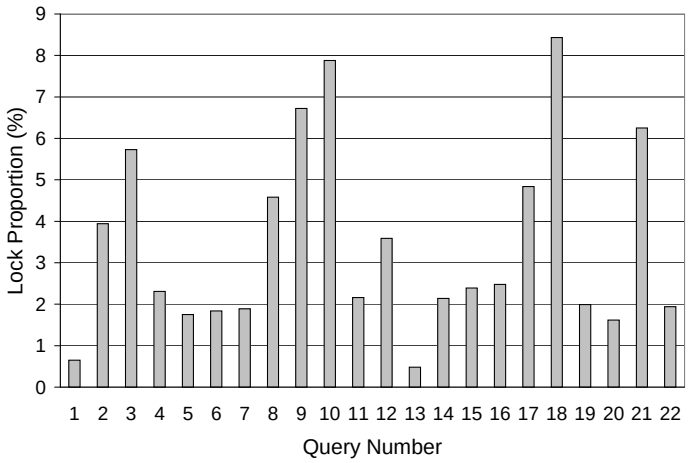


Figure 5.7: Lock Proportion in a Decision Support Workload
[The workload is the DBT3 Benchmark. The DBMS is PostgreSQL 7.3.4 running cygwin.]

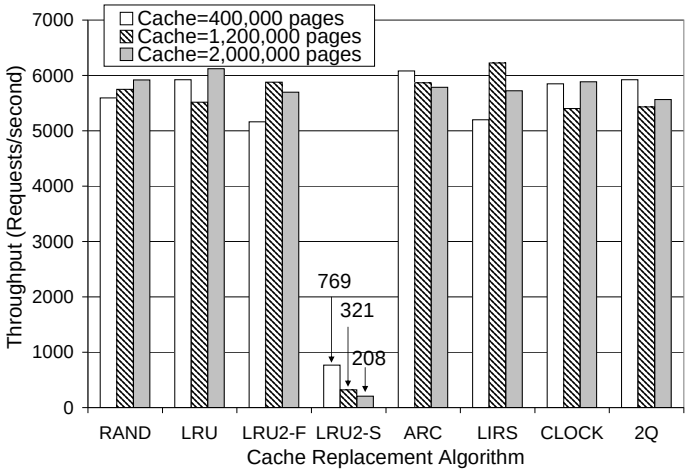
To demonstrate the impact of overhead on system throughput, the cache emulator was configured to be CPU-bound. Figure 5.8 shows the measured

throughput of different replacement algorithms under different cache sizes and lock proportions. All disk I/Os are bypassed, because otherwise the system is I/O-bound and no performance difference can be observed. With a small lock proportion (1%), all $O(1)$ algorithms and the $O(\log n)$ algorithm (LRU2-F) perform similarly. With a large lock proportion (10%), the throughput of LIRS is about 12% lower than that of other $O(1)$ algorithms, and the throughput of LRU2-F is 25% lower than that of the best performing $O(1)$ algorithms. One may be tempted to conclude that the overhead of replacement algorithms is crucial to performance, but the overhead of a replacement algorithm matters only when the system is CPU-bound, which typically happens when the lock proportion is small. Systems with large lock proportion, such as file servers, storage servers and OLTP workloads, are often I/O-bound, in which case the overhead of replacement algorithms does not impact the overall throughput. These results show that a higher replacement algorithm overhead (even as high as $O(\log n)$) has negligible impact on system throughput. Therefore, it is worthwhile to design new approaches for buffer cache management to improve the hit ratio and/or reduce lock contention even if slightly higher overhead is introduced.

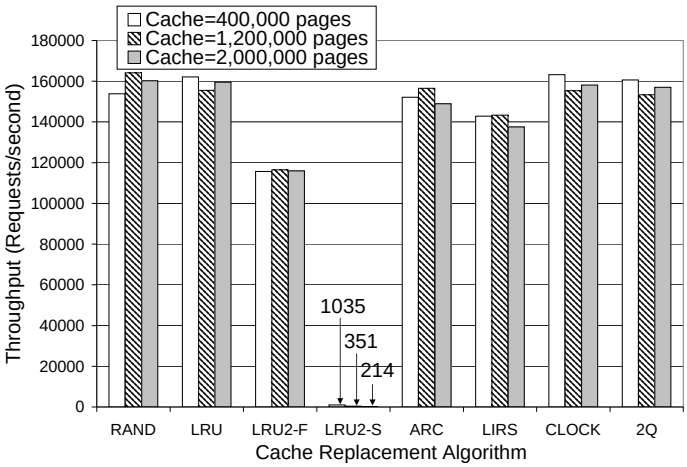
A first glance at the throughput reduction of LRU2-F under a 10% lock proportion would attribute this reduction to the $O(\log n)$ time complexity of the algorithm. However, its throughput under large caches is similar to that under small cache, implying that this decrease is not caused by the time complexity. This degradation is likely to be caused mainly by more processor cache misses for $O(\log n)$ algorithms. Similar results were reported by Megiddo and Modha [76], where the overheads of $O(\log n)$ algorithms (LRU-2 and LRFU) do not increase with cache sizes (but are larger than $O(1)$ algorithms).

5.5 The Multi-region Cache Approach

A new approach was developed for cache management, called *multi-region cache* (or *Mr. cache* for short). Multi-region cache can eliminate contention with little



(a) Lock Proportion for LRU = 1%



(b) Lock Proportion for LRU = 10%

Figure 5.8: Impact of Replacement Algorithm Overhead on Throughput

[The two figures have different scales on the y-axis. All throughput values are measured from the cache emulator. The random synthetic trace is used. The cache miss ratio is 5%. All disk I/O calls are bypassed to make the workload CPU-bound. The number of users is 1. If the page size is 8KB, the three cache sizes are equivalent to 3.2GB, 9.6GB, and 16GB.]

overhead without compromising the hit ratio. The basic idea of multi-region cache is to remove the global data structure of the replacement algorithm by splitting the cache into n fixed-size regions, numbered 1, 2, ..., n. Each region is managed by an instance of the replacement algorithm. $Mr.Alg(RgNum, RgSize)$ is used to denote a multi-region cache using replacement algorithm Alg with $RgNum$ regions

of $RgSize$ pages each[3]. The buffer cache without using multi-region is the same as the multi-region cache with one region, and is called the *one-region* cache for convenience.

In a multi-region cache, every page is hashed uniquely into a region. Typically a replacement algorithm employs a hash table to maintain a directory of all pages in the buffer cache. To distinguish between these two hash functions, the one used by the multi-region cache is called the *region-hash-function*, and the one used to look for pages in the buffer cache is called the *page-hash-function*. Of course, the same hash function can be used for both. A hash function designed by Jenkins [57] is used as the region-hash-function in these experiments unless otherwise stated.

One way to implement the multi-region approach is to manage each region using an unchanged instance of the replacement algorithm. On each buffer cache access, the index of the region that contains this page is calculated using the region-hash-function. This page is then handed over to the instance of the replacement algorithm managing that region. This approach is simple to implement since it requires no change to the existing replacement algorithm. The region-hash-function must be evaluated on every buffer cache access (hits and misses), however, which increases the overhead of the algorithm. Moreover, because consecutive pages are scattered into different regions by the region-hash-function, the detection of prefetch within a region is not possible.

The overhead of evaluating the region-hash-function can be minimized by using one global hash table instead of a hash table for every region. On a cache miss, the region index of the page is computed and stored in the page header. On a cache hit, the page is located in the global hash table and the region index is retrieved from the page header. A reference to this page is then passed to the instance of the replacement algorithm managing this region. Many replacement algorithms use history buffers to record the pages that have been evicted recently. The headers of these pages are also maintained in the global hash table, with an extra

[3]For simplicity, Alg or $RgSize$ is sometimes omitted when presenting results. $RgNum$ is always included.

flag indicating that the page data are not in the buffer cache. The region-hash-function is evaluated only on each cache miss. This introduces a negligible additional delay, since a cache miss must perform one or more slow disk accesses anyway. Since only a hash bucket needs to be locked when accessing a page of the hash table, and each hash bucket has less than one page on average, the global hash table does not cause lock contention.

Because contiguous pages are scattered into different regions by the region-hash-function, the detection and execution of prefetch must be conducted on the whole buffer cache. The page data with contiguous disk addresses should be placed in contiguous memory space, while their page headers are managed by different instances of replacement algorithms from different regions. This arrangement enables efficient DMA (Direct Memory Access) transfers when prefetching large amount of data. Some algorithms, such as SEQ [41], detect the patterns of page addresses to make replacement decisions. These algorithms cannot be directly used in the multi-region cache, since pages with consecutive physical addresses are often hashed into different regions. However, alternative algorithms can be used that do not rely on physical addresses to detect such patterns. For example, EELRU [112] has properties similar to SEQ without detecting page addresses.

Partitioning buffer cache into regions is an old idea, but one that was proposed for different objectives. Multi-region cache is analogous to the *set associative cache* used in memory hierarchies [47]. A set associative cache is a fast cache between processor and memory used to reduce the memory access latency by caching popular memory blocks in a small fast cache. In an m-way set associative cache, the cache is partitioned into many *sets*, each of which can hold m different memory blocks. Each memory block is hashed into a set based on the lower bits of the block address. If m equals the cache size, the cache is called a *fully associative* cache. The design goal of set associative cache is to reduce the cost of the hardware to search a certain block in the cache. A multi-region cache where each region contains m pages is similar to a m-way set associative cache, and the

traditional one-region approach is similar to a fully associative cache.

Buffer cache partitioning is also used for other purposes: some approaches aim at achieving hit ratios higher than a global cache [116, 121, 123]; some approaches employ partitioning as a way of admission control [34]; some approaches seek to perform goal-oriented tuning [14]. All these partitioning approaches strategically place objects into partitions and adjust the sizes of partitions dynamically. The objective of multi-region cache is to eliminate contention without paying a price for the hit ratio. Therefore, the multi-region cache has fixed-size regions and does not require any tuning.

5.6 Evaluation of Multi-region Cache

5.6.1 Contention

The goal of multi-region cache is to eliminate contention, without sacrificing performance. Lock contention happens when two or more threads access pages in the same region at the same time. Therefore, the fewer pages a region has, the lower the probability of lock contention. When the region is small (i.e., the number of regions is large), the chance of lock contention becomes small.

The contention micro-benchmark was used to study the effectiveness of multi-region cache in reducing contention. Multiple mutexes are used to protect the data structure for the replacement algorithm of each region. Each user randomly selects a mutex to wait on, which simulates the scenario that a page in the corresponding region is required. Contention happens only when multiple users request the same mutex at the same time, the probability of which is low when the number of mutexes is large. Figure 5.9 shows the system throughput normalized to the maximum throughput, which is achieved by configuring the micro-benchmark to have no contention. Since contention is the major factor that could decrease throughput in the micro-benchmark, higher throughput indicates lower contention. In the figure, the normalized throughput quickly increases to a point close to the

maximum throughput, implying that the contention is almost eliminated. This figure demonstrates that contention can be effectively eliminated by using a moderate number of regions.

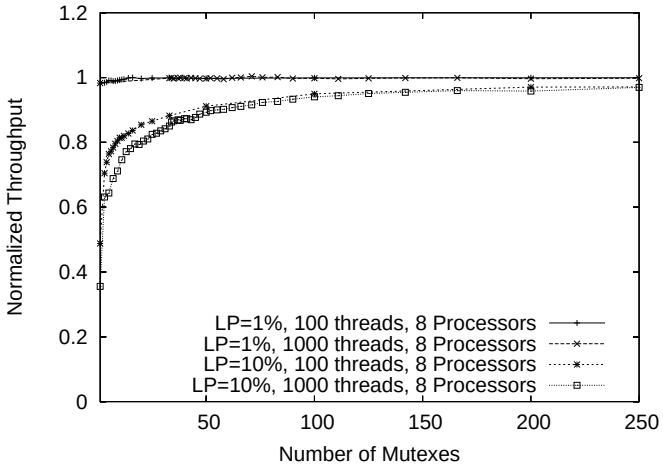


Figure 5.9: Effect of Number of Mutexes on Reducing Contention

The above experiments assume that each region has the same probability to be accessed. In real workloads, the page access probability has a skewed distribution [52]. When each region has a random set of pages, the distribution of access probability for each region has much less skewness than that of the page access probability, since hot pages are likely to be scattered in different regions. The results shown in Figure 5.10 confirm that multi-region cache can effectively eliminate contention in real workloads with skewed accesses. The results in the figure were obtained from the buffer cache emulator. Disk I/O calls were bypassed to make the system contention-bound. Two processors were used in this experiment, because the trace-reading thread cannot keep up with the worker threads when more processors are used. The throughput of Mr.LRU(524, 40000) is 21% higher than that of LRU. The average time to finish a cache hit in Mr.LRU(524) is $5.0\mu s$[4], whereas it is $82.1\mu s$ in LRU because of contention. The CLOCK algorithm has contention only on cache misses. Therefore, its throughput is higher than that of LRU, and Mr.CLOCK performs similarly to CLOCK.

[4] $1\mu s = 10^{-6}$ second

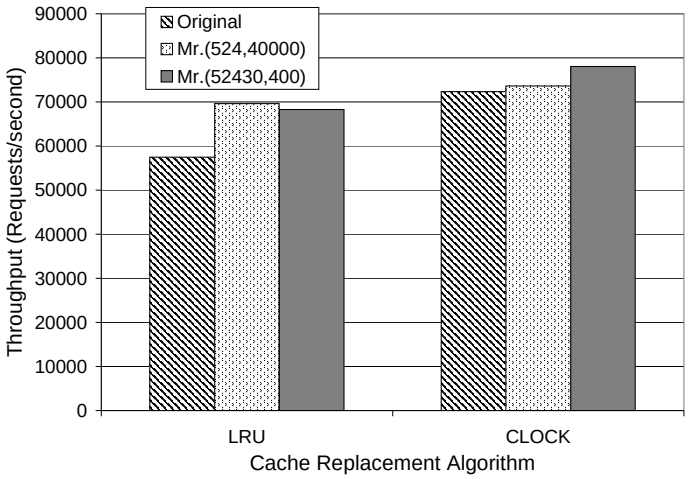


Figure 5.10: Effect of Number of Regions

[The trace is OpenMail6. The page size is 1KB. All I/O calls are bypassed. The cache size is 2GB (The first level cache of this trace is 1GB). The number of processors is 2 and the number of users is 30.]

5.6.2 Miss Ratio

Miss ratio is a crucial metric when evaluating the performance of multi-region cache. The overall miss ratio of a multi-region cache can be affected by many factors: the number of pages mapped into a region, the access frequency of a region (i.e., the sum of access frequencies of all pages in this region), the replacement algorithm used and the miss ratio of the region. A detailed mathematical model is difficult to develop, however, because these factors, especially the miss ratio of each region, are hard to model.

Since the multi-region cache with region size m is similar to an m-way set associative cache, previous studies on set associative cache can be used as an indication of how multi-region cache performs. Normally, m-way set associative cache has higher miss ratio than fully associative cache. This difference decreases as m increases. The miss ratio of an 8-way set associative cache is not observably different from that of a fully associative cache [47]. Because the buffer cache is large, each region of a multi-region cache typically is much larger than 8 pages. Analogous to the set associative cache, it can be expected that multi-region cache has almost the same miss ratio as that of the one-region cache.

Several analyses of the miss ratio of the multi-region cache with simplified assumptions are first presented to provide insight into the miss ratio of the multi-region cache. Simulation studies on different traces are then presented.

5.6.2.1 Modeling Results

The notation used in this section is summarized in Table 5.3.

Table 5.3: Notation for the Analysis of the Multi-region Cache

Symbol	Definition
D_i	Number of distinct pages hashed into region i as a random variable
L_i	Proportion of logical references to pages in region i as a random variable
M_i	The miss ratio of region i as a random variable
$\mathscr{M}$	The miss ratio of the whole multi-region cache as a random variable
D	Data size in pages (the total number of distinct pages accessed)
n	Number of regions
R	Size of a region in pages
C	Total cache size in pages $C = nR$
$miss(r)$	Miss ratio function of the buffer cache given the proportion of the cache size to the data size

In the one-region cache, the miss ratio of the buffer cache is $miss(C/D)$. In multi-region cache, the set of pages in any region is a random sample of all pages. The probability that this sample represents the behaviour of all pages is high as long as the region size is not too small. Therefore, it is assumed that each region has the same miss ratio function as the whole buffer cache. Since approximately $\frac{D}{n}$ data pages are hashed into each region, the proportion of the cache size to the data size in each region is close to

$$\frac{\left(\frac{C}{n}\right)}{\left(\frac{D}{n}\right)} = \frac{C}{D},$$

indicating that the miss ratio of each region, and the overall miss ratio of the multi-region cache, is the same as that of the one-region cache.

In multi-region cache, the regions need not have exactly the same number of pages. The probability that a page is put into a region is $1/n$. Thus the number of

pages in region i is a binomial random variable D_i, whose probability function is:

$$P_{D_i}(k) = \binom{n}{k} \left(\frac{1}{n}\right)^k \left(\frac{n-1}{n}\right)^{n-k}$$

The distribution of the proportion of page references in each region L_i depends on the distribution of page references to all pages, which is a skewed distribution [52]. A distribution commonly used to represent the skewness of data accesses is defined by Knuth [65, p. 400]:

$$p_i = \frac{i^\theta - (i-1)^\theta}{N^\theta},$$

where $i = 1 \ldots N$ and $0 < \theta \leq 1$.

When $\theta = 1$, this is the uniform distribution. When

$$\theta = \frac{\log 0.80}{\log 0.20} = 0.1386,$$

this is the popular "80-20" distribution where 80% of the references go to 20% of the pages. This distribution is called $Knuth(a, b)$, where

$$\theta = \frac{\log 0.01a}{\log 0.01b}.$$

A region with more distinct pages hashed into it has more average logical references to this region:

$$E[L_i] = \frac{D_i}{D},$$

where $E[L_i]$ is the mean of L_i. Therefore, $D_i = D \cdot E[L_i]$, and the correlation between L_i and D_i is:

$$E[L_i \cdot D \cdot E[L_i]] - E[L_i] \cdot E[D \cdot E[L_i]] = 0.$$

Preliminary experiments on many traces and different configurations confirmed that the correlation between L_i and D_i is small ($< 10^{-5}$). The following analysis

assumes that L_i and D_i have no correlation.

The miss ratio of the overall buffer cache $\mathscr{M}$ is:

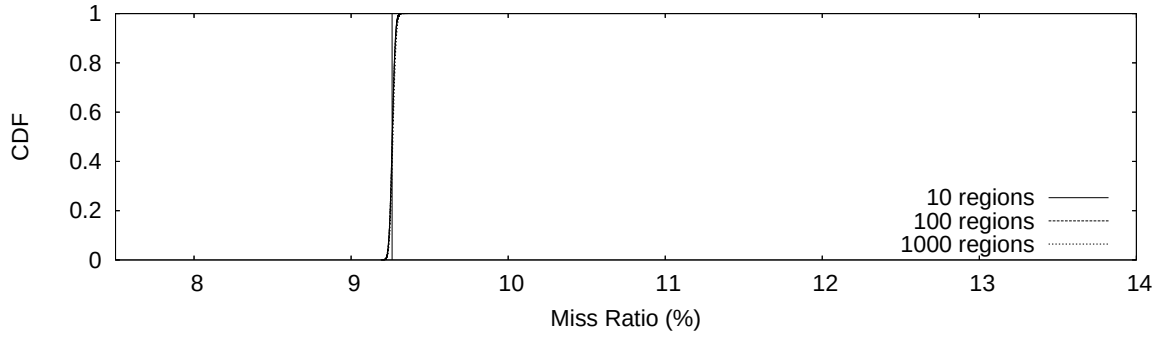
$$\mathscr{M} = \sum_{i=1}^{n} L_i M_i,$$

where M_i is the miss ratio of region i: $M_i = miss(\frac{C}{nD_i})$.

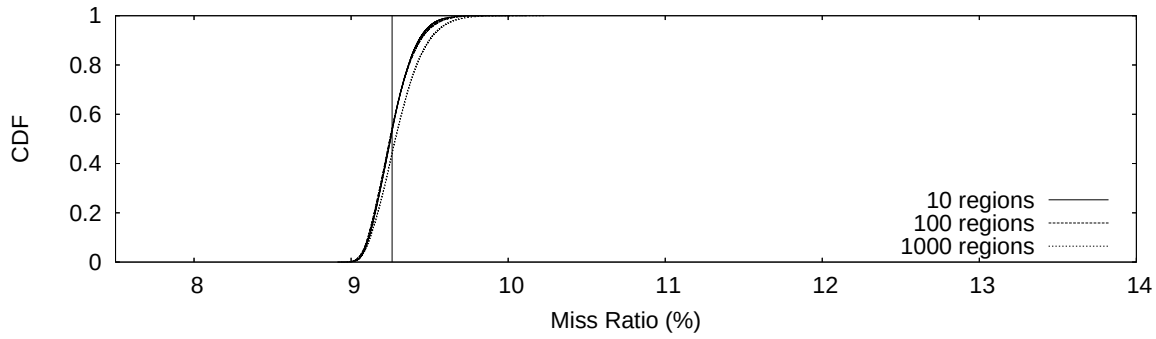
L_1, L_2, ..., and L_n have negative correlation, since when one region gets more logical references, other regions tend to get fewer logical references. Preliminary experiments indicate that the correlation between any two of them is very close to zero as long as the number of regions is not too small. Therefore, the correlation among them is not considered. The small negative correlation among D_1, D_2, ..., D_n are ignored for the same reason. Since the L_i $(i = 1, 2, \ldots, n)$ are considered independent, L is used to represent all of them. Similarly, R is used to represent R_i $(i = 1, 2, \ldots, n)$. The overall miss ratio $\mathscr{M}$ can be written as:

$$\mathscr{M} = \sum_{i=1}^{n} L \cdot miss(\frac{C}{nD_i}). \tag{5.1}$$

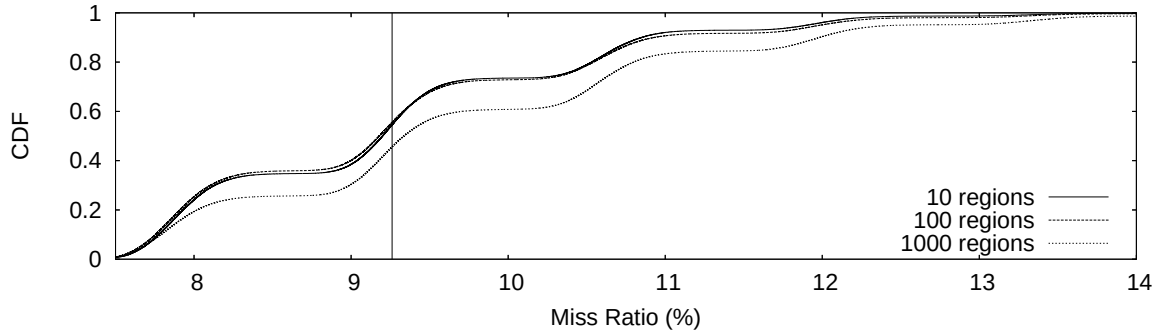
The study of production database workloads indicates that the miss ratio function can be well fitted by [52]: $miss(r) = 0.102(100r + 0.208)^{-0.511}$, which is used as the miss ratio function when solving expression 5.1. The distribution of the overall miss ratio $\mathscr{M}$ with different skewness is simulated numerically and shown in Figure 5.11; its mean and standard deviation are listed in Table 5.4. The figure and table show that the distribution of miss ratio is affected mainly by the skewness of page accesses. With a more skewed distribution, the miss ratio has a larger standard deviation. The number of regions has no obvious effect on the average miss ratio. In all cases, the average miss ratio is within 1% of that of the one-region cache.



(a) Knuth(40,20)



(b) Knuth(60,20)



(c) Knuth(80,20)

Figure 5.11: Distribution of Miss Ratio

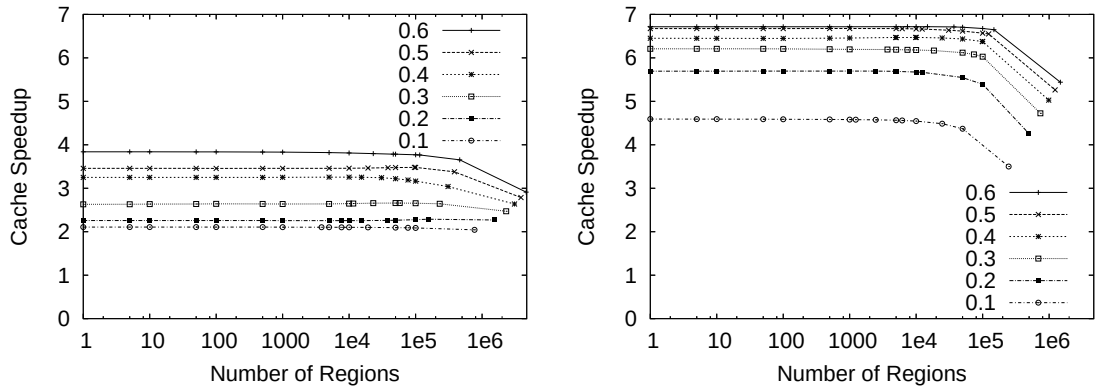
[The vertical line is the miss ratio of the one-region cache.]

Table 5.4: Miss Ratio of Multi-region Cache

Skewness	Number of Regions	Miss Ratio	
		Mean	Standard Deviation
-	1	9.261%	0
$Knuth(40, 20)$	10	9.259%	0.000494
	100	9.260%	0.000217
	1000	9.263%	0.000171
$Knuth(60, 20)$	10	9.260%	0.00132
	100	9.258%	0.00131
	1000	9.268%	0.00136
$Knuth(80, 20)$	10	9.263%	0.0131
	100	9.250%	0.0137
	1000	9.373%	0.0145

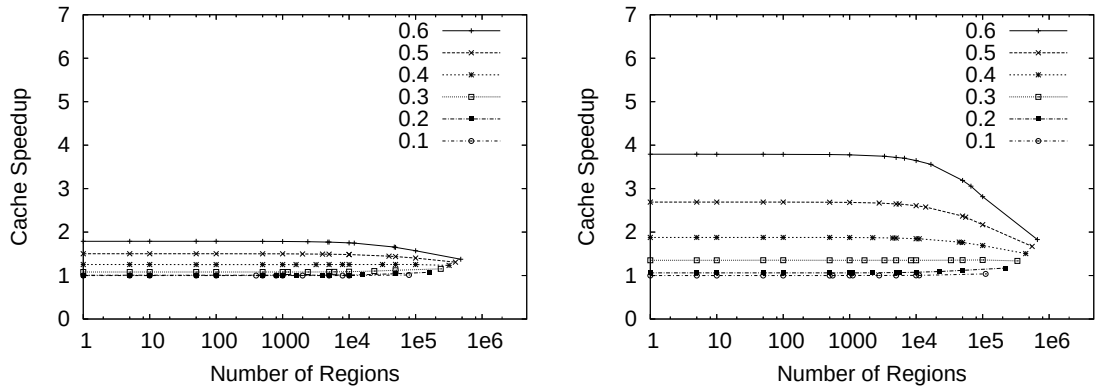
5.6.2.2 Simulation Results

Figures 5.12 and 5.13 show the cache speedups of different replacement algorithms as a function of region size under different cache sizes and different workloads. These results were obtained from simulation. Only results from the LRU algorithm are presented, since all other algorithms have the same trend as LRU. On the rightmost data point of each line, the region size is 1. The figure shows that the system throughput is stable across a wide range of region sizes for a wide range of workloads. In all traces except the TPC-W trace, the cache speedup only starts to drop when each region has less than about 100 pages, at which point the number of regions is several hundreds of thousands. Based on Figure 5.9 (page 102), the number of regions required to eliminate contention is far less than the points where the cache speedup starts to drop. The cache speedup for the TPC-W trace drops earlier when the buffer cache is large. This trace has good temporal locality [68], and large TPC-W configurations typically use a buffer size which is less than 5% of the data size [126]. Therefore, the drop of cache speedup in Figure 5.13(f) is not likely to happen in practice.



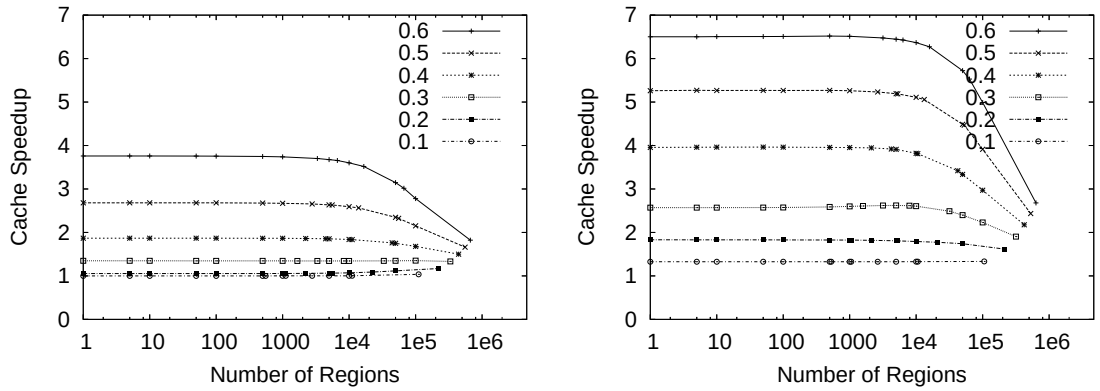
(a) Financial1

(b) Financial2



(c) WebSearch1

(d) WebSearch2

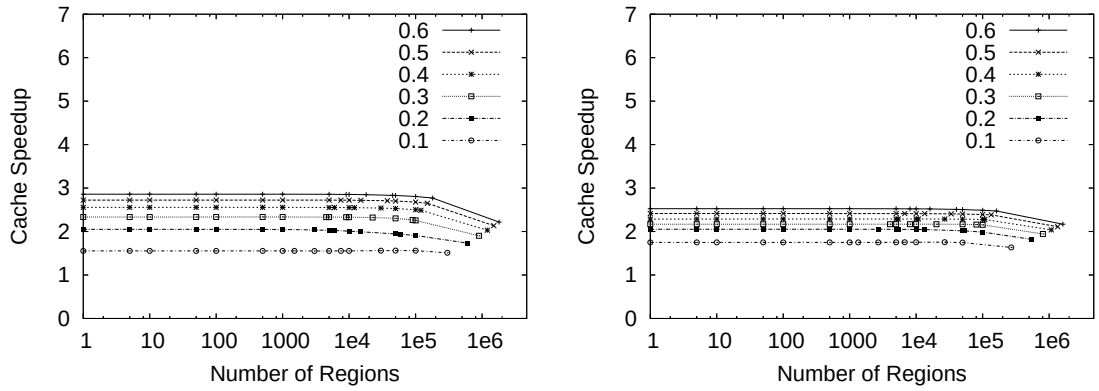


(e) WebSearch3

(f) NFSEmail

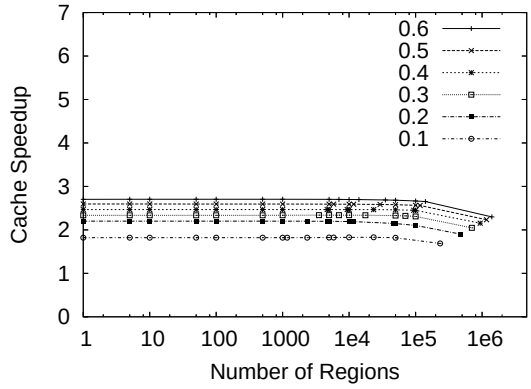
Figure 5.12: Effect of Number of Regions

[Each line represents a cache size ranging between 0.1-0.6 of the data size. LRU is the replacement policy. On the rightmost point of each plotted line, the size of each region is one page.]

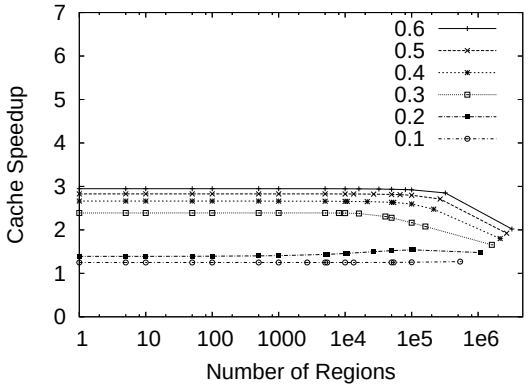


(a) Openmail1

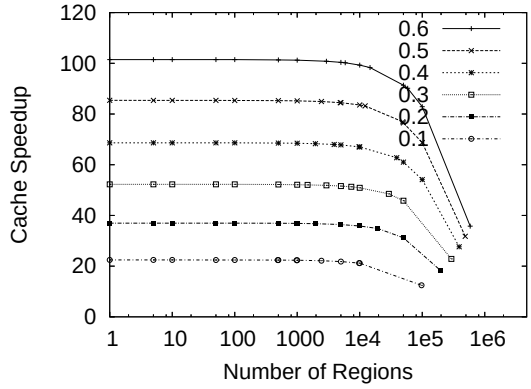
(b) Openmail2



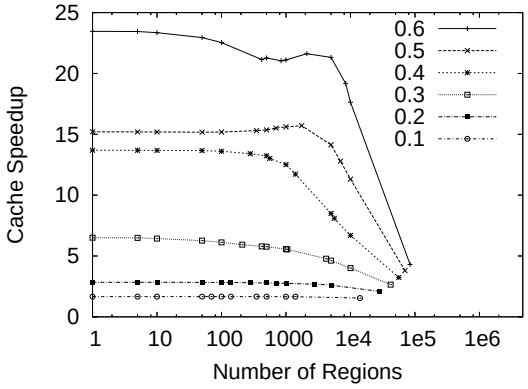
(c) Openmail3



(d) Openmail6



(e) TPC-C



(f) TPC-W Shopping

Figure 5.13: Effect of Number of Regions – More Workloads

[Each line represents a cache size ranging between 0.1-0.6 of the data size. LRU is the replacement policy. On the rightmost point of each plotted line, the size of each region is one page. Figure 5.13(e) and 5.13(f) have different scales on the y-axis than all other figures.]

5.6.3 Overhead

The overhead that multi-region cache adds to a one-region cache comes from the need to evaluate the region-hash-function on every cache miss. This cost should be small because of the low frequency and long latency of cache misses. The overhead of multi-region cache was measured in the cache emulator to confirm this.

The cache emulator is configured to bypass all disk I/O calls to measure the overhead of multi-region cache. This omission makes the results conservative, since the overhead of evaluating region-hash-functions can otherwise be dominated by disk I/Os in cache misses. The random synthetic trace was used since the type of the trace does not affect the overhead of the replacement policy. Three replacement algorithms with different time complexities were selected. The lock proportion was set to the upper bound of 10% to show the worst case of overhead. Figure 5.14 shows the results obtained.

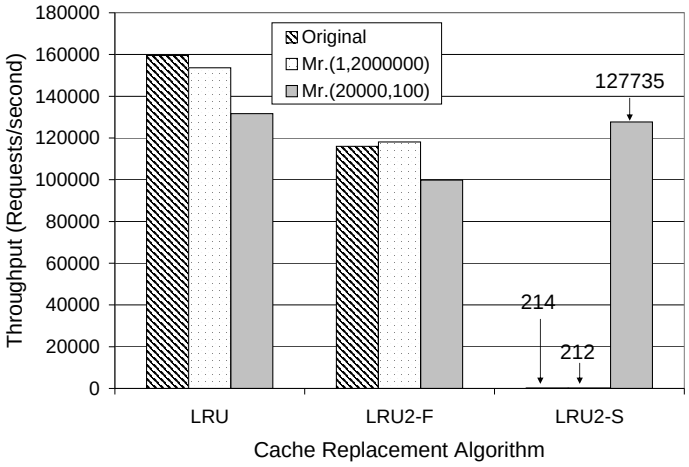


Figure 5.14: Overhead of Multi-region Cache
[All throughput values were measured from the cache emulator, with the random synthetic trace. The cache miss ratio is 5%. All disk I/O calls were bypassed. The page size was 4096 bytes, and the cache size was 1600MB. The number of users was 1. The work interval was configured so that the lock proportion is 10% when using the LRU replacement policy.]

The throughput of Mr.LRU(1) is similar to that of LRU, implying that the overhead introduced by multi-region cache is small. Mr.LRU(20000,100) has 19% lower throughput than LRU, because the program manipulating 20,000 linked lists

generates more processor cache misses than the program manipulating one linked list. The same trend can be seen for the Mr.LRU2-F and LRU2-F algorithms. These results are conservative since (1) the emulated workload does not compete for processor cache with LRU as much as a real workload does; (2) when the lock proportion is 10%, the system is likely to be I/O-bound, where the overhead does not matter. In more realistic environments, the overhead of the multi-region cache is expected to be close to the overhead of the one-region cache.

Mr.LRU2-S(20000,100) outperforms LRU2-S by about 600 times. Moreover, it has throughput comparable to Mr.LRU and outperforms LRU2-F. The reason for this speedup is that, although the miss overhead of LRU2-S is $O(n)$, it performs comparably to an algorithm with $O(1)$ overhead when n is small. As an extreme case, when the miss ratio is decreased from 5% to 20% by increasing the data size, the throughput of Mr.LRU2-S(20000,100) drops 21% due to the $O(n)$ miss overhead. The system is unlikely to be CPU-bound under such high miss ratios, however, and so this extra miss overhead can be overlapped by disk requests.

The multi-region cache introduces some space overhead. Every instance of the cache replacement algorithm has to maintain some bookkeeping information. When there are hundreds of thousands of regions, the total space occupied by these bookkeeping data structures could be a concern. One way to reduce this space overhead would be to extract the read-only or infrequently updated data from these instances and store them in global data structures.

5.7 Discussion

An ideal cache replacement algorithm should have high hit ratio, low overhead and low lock contention, but an algorithm which is superior in one aspect is often inferior in others. Designers often need to make tradeoffs between these aspects.

Some systems, such as storage servers, disk caches and I/O controller caches, are likely to be I/O-bound because of the large lock proportion in the workload. Normally only one processor (or two for high availability) is needed in these

systems, and so lock contention can be safely ignored. The overhead of the replacement algorithm can also be overlapped by disk I/Os. As a result, a replacement algorithm with high hit ratio is the preferred choice.

Some systems, such as database systems, must work well under both I/O-bound workloads and CPU-bound workloads. In such environments, systems with many disks and multiple processors are often used to support many concurrent users, and lock contention has the largest impact on system throughput. Without using the multi-region cache, a CLOCK-based algorithm is a good choice, since it does not cause lock contention on cache hits and its lower hit ratio can be offset by adding more memory or disks. With the multi-region cache, the replacement algorithm with the highest hit ratio can be used.

Emulation results presented in Figure 5.14 suggest that when using multi-region cache, a replacement algorithm with $O(1)$ hit overhead and $O(n)$ miss overhead can perform well, even in a CPU-bound system. This relaxation on cache miss overhead enables a new family of replacement algorithms that perform simple bookkeeping on a cache hit (e.g., recording a timestamp in the page header) and a full region scan for the victim page on a cache miss.

Another area that can benefit from the multi-region cache approach is virtual memory management, one of the most crucial components in operating systems. The design of replacement algorithms for virtual memory management is constrained by the requirement that cache hits must be performed by hardware. The virtual memory management functions in existing hardware only allow CLOCK-based algorithms to be used, since the hardware support for more advanced algorithms is too expensive. Using multi-region cache, it is possible to accurately implement sophisticated algorithms (such as LRU, LRU-2, LIRS, etc.) in virtual memory with small additional hardware support. For example, Mr.LRU could be implemented by letting the processor write the current timestamp into the page table entry whenever a page is referenced. On a cache miss on page p, the page table entries of all pages in the region that page p belongs to are scanned and the page with the oldest timestamp is evicted.

5.8 Summary

In this chapter, lock contention in buffer cache replacement algorithms is analyzed. The design tradeoffs in various aspects of a replacement algorithm are discussed and approaches that can eliminate lock contention are investigated. It is found that the impact of lock contention on system throughput is mostly affected by the lock proportion of the workload and the number of processors in the system. The number of threads affects the extent of contention only weakly.

One of the current solutions for reducing contention is to use a CLOCK-based replacement algorithm. Simulations found that in large buffer caches, the CLOCK algorithm has an unnoticeably smaller hit ratio than other advanced algorithms, while this small hit ratio difference results in significantly worse system throughput compared to other replacement algorithms.

Further study of the overhead of the replacement algorithm indicates that algorithms with $O(\log n)$ overhead only slightly impact system throughput compared to algorithms with $O(1)$ overhead, even in large buffer caches. Moreover, the larger overhead of the $O(\log n)$ algorithm is not caused by the higher time complexity but by more processor cache misses. This result suggests that it is worthwhile to deploy replacement algorithms with higher hit ratio and/or lower lock contention at the expense of some CPU overhead.

A new approach called multi-region cache is proposed to reduce lock contention. The multi-region cache splits the buffer cache into many fixed-size regions, each of which is managed by an instance of a replacement algorithm. Any replacement algorithms can be used together with the multi-region cache. Since the multi-region cache removes the need for a global data structure, lock contention is reduced to a negligible level. Analysis and simulation show that use of the multi-region cache does not noticeably decrease either the overall hit ratio or the cache speedup of the buffer cache with hundreds of thousands of regions. The low overhead of the multi-region cache makes it practical in real systems.

Chapter 6

Disk Layout Management

Disk layout management deals with how the data are placed on the disks. Its main design objective is to reduce the time that the disk head must wait when reading and writing data. This is often done by aggregating disk I/O requests so that a few large I/Os are performed instead of many small I/Os; this means that the slow disk arm movement and rotational latency can be reduced when accessing data on different locations of the disk. Disk layout management can significantly affect disk I/O performance of the system [74, 75].

Some disk layout approaches seek to optimize read performance [74] while some approaches mainly optimize write performance [96]. With large in-memory buffers, most disk reads can be resolved in memory [88]. As a result, in write-intensive systems, such as database servers running OLTP applications, email servers, file servers and storage servers, write requests make up a large portion of the total disk traffic [32, 113]. This makes the optimization of write performance crucial to overall system performance.

The typical approach to disk layout for writes is *Overwrite*, which means that new data are overwritten on top of old copies. Data of one file are normally placed contiguously on disks. If the entire file is written as a whole, write performance is good. But some workloads, such as OLTP and email workloads, have small random writes. Moreover, a large system often supports many users. Interleaved requests from multiple concurrent users destroy the locality of the disk request stream and result in poor write performance. LFS (Log-structured File System) [88, 96] uses a non-overwrite approach in which data are accumulated and written to new places in large chunks. It has the potential to achieve superior

write performance while maintaining comparable read performance [96, 108], but LFS has to perform segment cleaning to reclaim large contiguous free space for further writes. Previous studies [108] on a 1991 disk under OLTP workloads have found that this cleaning overhead significantly degrades system performance when the disk space utilization is higher than 50%. Disk technology has improved dramatically since these studies were published. Using 1991's DEC RZ26 and today's Cheetah X15 36LP as examples, the disk positioning time has decreased from 15ms to 5.6ms (a 2.7x times improvement), while the disk bandwidth has increased from 2.3MB/s to 61MB/s (a 27x times improvement). The disk bandwidth improved 10 times more than the positioning time for these two disks, and this trend is expected to continue [43]. Since LFS was designed to utilize the disk bandwidth effectively, this trend speaks favourably to the performance of LFS. Whether or not the cleaning cost of LFS is still prohibitively high on modern and future disks is an unaddressed issue.

This chapter investigates the performance of LFS and Overwrite under modern disk technologies in large configurations. A new disk layout approach is proposed that further improves performance [135]. The remainder of this chapter is organized as follows. Section 6.1 develops a performance model for LFS and Overwrite and analyzes their performance. Section 6.2 describes the design of the new disk layout approach, called HyLog. Section 6.3 discusses the methodology used to evaluate HyLog and Section 6.4 presents simulation results of HyLog. Section 6.5 summarizes this chapter.

6.1 Disk Layout Write Cost Model

The extensive use of client and server caching on read traffic makes write performance an important factor in many systems [88]. In fact, write traffic was found to exceed read traffic on some recent file systems [32] and OLTP workloads [113]. Two popular approaches for managing disk layouts are *Overwrite*, where updated data are overwritten to their old addresses, and LFS [96], where

updated data are written to new locations. Previous studies have found that the read performance of these two approaches is comparable while their write performance has large differences [96, 108]. An analytical model that quantifies the write performance of these two approaches is important for understanding their overall performance, as well as for designing and evaluating new disk layout approaches. A new write cost model is developed that gives the average time the disk takes to write one page of data under Overwrite or LFS.

6.1.1 Assumptions and Definitions

To simplify the modelling, it is assumed that read performance is not affected by different disk layouts. The notation used for the model is summarized in Table 6.1.

Table 6.1: Notation for the Disk Layout Write Cost Model

Category	Symbol	Definition
Disk Parameters	T_{pos}	Disk positioning time
	B	Average sustained bandwidth of the disk
	N_d	Number of disks in a disk array (RAID)
	T_{pg}	Average disk I/O time to read/write a disk page
	T_{seg}	Average disk I/O time to read/write a disk segment
	η	Segment I/O efficiency
System Config.	P	Size of a disk page in bytes
	S	Size of a disk segment in number of pages
System Statistics	u_d	Disk space utilization
	u	Space utilization of the segments to be cleaned
	P_{idle}	Proportion of idle time in a disk array (RAID)
	h	Proportion of pages in the hot partition (hot pages)
	w	Proportion of writes to the hot partition (hot writes)
Write Costs	C_{ow}	Write cost of Overwrite
	C'_{ow}	Scaled write cost of Overwrite
	$C_{lfscleaning}$	Write cost of LFS using cleaning
	$C'_{lfscleaning}$	Scaled write cost of LFS using cleaning
	$C_{lfsplugging}$	Write cost of LFS using hole-plugging
	$C'_{lfsplugging}$	Scaled write cost of LFS using hole-plugging
	C'_{hylog}	Scaled write cost of HyLog

A simple disk model with seek time, rotational latency and transfer bandwidth is used. The positioning time T_{pos} is the sum of the average seek time and the

average rotational latency, i.e., the time for the disk to rotate half a rotation. The transfer bandwidth B is the average sustained bandwidth at which the disk can read or write data. It is assumed that the read bandwidth is the same as the write bandwidth.

It is also assumed that data are stored on the disk in fixed-size pages of P bytes. In LFS, the disk is separated into fixed-size segments, each of which has S pages. The time to read or write a page is T_{pg} and the time to read or write a segment is T_{seg}. Therefore,

$$T_{pg} = T_{pos} + \frac{P}{B},$$

and

$$T_{seg} = T_{pos} + \frac{SP}{B}.$$

The *disk space utilization* u_d represents the proportion of the disk space occupied by user data: $0 < u_d < 1$.

LFS writes data in units of segments instead of pages in an attempt to achieve better write performance than Overwrite. The *segment I/O efficiency* η represents the saving of disk I/O time for writing one segment over writing S pages of the segment individually. η is defined as

$$\eta = \frac{ST_{pg}}{T_{seg}} = \frac{S(P + T_{pos}B)}{SP + T_{pos}B}. \tag{6.1}$$

The higher the η, the better the performance of LFS, if other factors are constant. η is a monotonously increasing function of the segment size S and a monotonously increasing function of $T_{pos}B$, called the *disk performance product*. $T_{pos}B$ represents the amount of data the disk can transfer during the time required to position the disk head. The parameters of three high-end SCSI disks of different years are listed in Table 6.2. Their segment I/O efficiency is shown in Figure 6.1. Modern disks have much larger η than old disks, implying LFS should perform much better on modern disks than on old disks.

When a disk has multiple pending requests from several users, a disk

Table 6.2: Disk Parameters

Brand Name	Year	Positioning Time (ms)	Bandwidth(MB/s)
Cheetah X15 36LP	2004	5.6	61.0
Quantum atlas10k	1999	8.6	20.4
DEC RZ26	1991	15.0	2.3

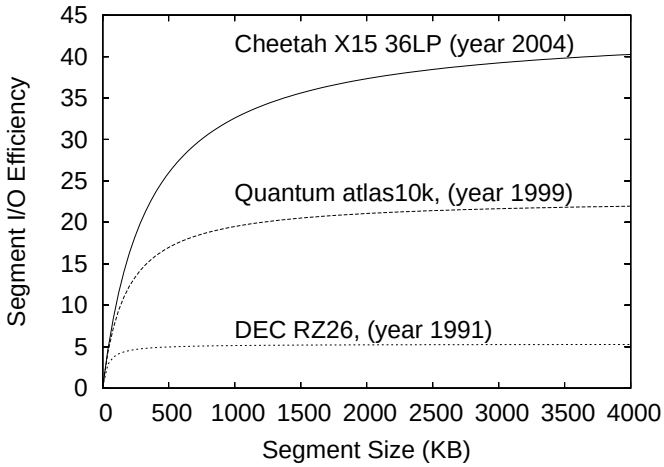


Figure 6.1: Segment I/O Efficiency of Different Disks
[Page size is 8KB.]

scheduling algorithm is often used to reorder the requests, which could reduce the average disk positioning time. As a result, η decreases as the number of pending requests increases.

6.1.2 Modelling LFS and Overwrite

6.1.2.1 The Write Cost Model

The Write Cost of Overwrite

In Overwrite, each write takes T_{pg} time. Thus the write cost of Overwrite C_{ow} is

$$C_{ow} = T_{pg}.$$

The Write Cost of LFS

To model the write cost of LFS, the segment cleaning overhead must be considered. The cost of segment cleaning is directly affected by the space utilization of the segments selected for cleaning, which is defined as the *cleaning space utilization* u. There are two segment cleaning methods: *cleaning* [96] and *hole-plugging* [72]. These variants of LFS are called *LFS-cleaning* and *LFS-plugging*, respectively.

In LFS-cleaning, some candidate segments for cleaning are selected and read into memory. The pages that have been written again after this segment was written are called *dead* pages and other pages are called *live* pages. The live pages in these segments are written out to new segments, while the dead pages are discarded. After this cleaning procedure, the old copies of these segments are considered free and the space occupied by the dead pages in these segments is reclaimed. After 1 segment is read, segment space u is written and segment space $1-u$ is freed. Therefore $\frac{1+u}{1-u}$ segment I/O operations are required to free 1 segment space. For the system to be balanced, whenever a segment of user data is written to the disk, a segment of free space is reclaimed by cleaning. Thus LFS requires

$$1 + \frac{1+u}{1-u} = \frac{2}{1-u}$$

segment I/O operations to write one segment of user data. The average time required to write one page in LFS is defined as the write cost $C_{lfscleaning}$:

$$C_{lfscleaning} = \frac{T_{seg}}{S} \cdot \frac{2}{1-u}.$$

In LFS-plugging, some candidate segments are read into memory, and the alive pages of these candidate segments are written out to holes found in other segments so that the space occupied by these candidate segments becomes free. To reclaim one segment of free space requires 1 segment read and uS page writes. Therefore, the write cost of LFS-plugging $C_{lfsplugging}$ is defined as the average time required

to write one page:

$$C_{lfsplugging} = \frac{1}{S} \cdot (2T_{seg} + uST_{pg}).$$

Calculation of Cleaning Space Utilization

If the workload is uniform random update, the segment with the lowest space utilization should be selected for cleaning in both LFS-cleaning and LFS-plugging. u increases with u_d and $u \leq u_d$. The relation between the two can be approximated by [77]:

$$u_d = (u - 1)/\ln u. \tag{6.2}$$

Figure 6.2 shows that simulation results match this formula well. If the page update frequencies have a skewed distribution, as seen in real workloads [52, 94], the cleaning space utilization is lower than that under the uniform random update workload [72]. Expression 6.2 can be used as an upper bound estimation of u, given the disk space utilization u_d.

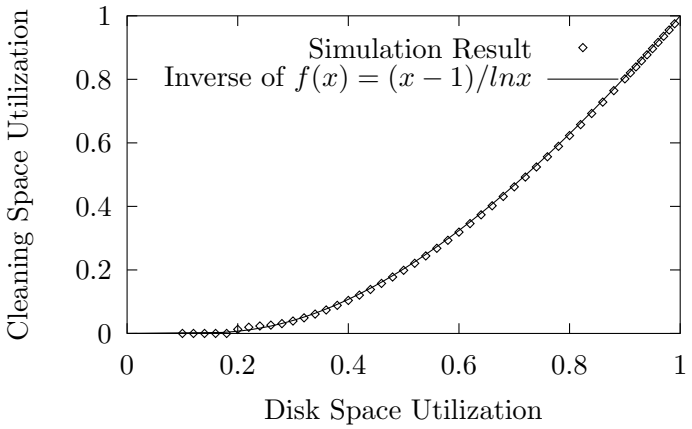


Figure 6.2: Disk Space and Cleaning Space Utilization
[Assuming $f(x) = (x-1)/\ln x$, Equation 6.2 can be written as $u_d = f(u)$. Therefore, $u = f^{-1}(u_d)$. The function $f^{-1}(u)$ cannot be represented in a closed form. The $f^{-1}(x)$ line shown in the figure was plotted from numerical solutions.]

Impact of Segment Size in LFS

Previous simulation studies [72] showed that the segment size is an important contributor to the performance of LFS. By experimenting on different disks, the

following rules of thumb were found [72]:

1. The optimal segment sizes are different for different disks. Only the disk performance product (the product of the positioning time and the transfer bandwidth) matters.
2. Larger segments are required for faster disks. The optimal segment size is approximately 4 times the disk performance product.

Equation (6.1) shows that $T_{pos}B$ is the only disk characteristic that affects η, which is consistent with the first rule of thumb. The scaled write costs (Equations (6.4), (6.5) and (6.6)) indicate that the higher the η and the lower the u, the more advantage LFS can achieve over Overwrite. Figure 6.1 shows that the larger the segment, the higher the η. On one hand, however, the increase of η is slower with larger segment sizes, while on the other hand, the cleaning space utilization becomes higher with larger segments [72]. Therefore, there is an optimal segment size to achieve the best performance. From Equation (6.1), the limit of η is:

$$\lim_{S\to\infty} \eta = \lim_{S\to\infty} \frac{S(P + T_{pos}B)}{SP + T_{pos}B} = \frac{T_{pos}B + P}{P}.$$

Assume that a segment size should be selected so that proportion α of this limit is achieved ($0 < \alpha < 1$). Then

$$\frac{S(T_{pos}B + P)}{T_{pos}B + SP} = \alpha \frac{T_{pos}B + P}{P}.$$

Thus

$$S \cdot P = \frac{\alpha}{1-\alpha} T_{pos}B,$$

where $S \cdot P$ is equal to the segment size. If $\alpha = 80\%$, the segment size is

$$S \cdot P = 4T_{pos}B, \tag{6.3}$$

which is consistent with the second rule of thumb. The preferred segment sizes suggested by Equation (6.3) are marked by small crosses in Figure 6.3. The crosses

are close to the "knee" of the curve, which means a reasonably high η value is achieved with a relatively small segment size. In this study, the segment size is calculated from this formula and then rounded to the closest size in powers of two.

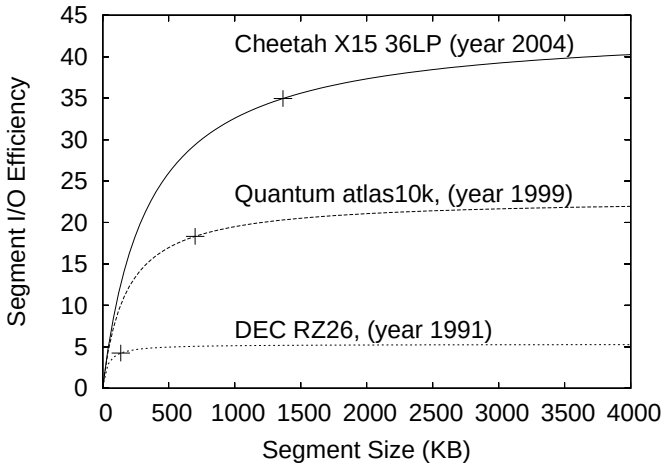


Figure 6.3: Segment Sizes of Different Disks
[Page size is 8KB. The small crosses indicate the segment size suggested by Equation (6.3).]

6.1.2.2 Performance Comparisons

The performance of these disk layouts can be compared in terms of their write costs. To simplify the write costs, the *scaled write cost* is defined by scaling all write costs by $\frac{S}{T_{seg}}$:

$$C'_{ow} = \frac{S}{T_{seg}} C_{ow} = \eta \tag{6.4}$$

$$C'_{lfscleaning} = \frac{S}{T_{seg}} C_{lfscleaning} = \frac{2}{1-u} \tag{6.5}$$

$$C'_{lfsplugging} = \frac{S}{T_{seg}} C_{lfsplugging} = 2 + u\eta \tag{6.6}$$

Note that $C'_{lfscleaning}$ is the same as the traditional write cost of LFS [96]. The write cost of Overwrite was defined as the reciprocal of the utilized disk bandwidth (i.e., $\frac{T_{pos}B + P}{P}$) [96], which ignores the effect of segment size. Segment size is important to the performance of LFS [72] and is taken into account by C'_{ow}.

Figure 6.4 shows the scaled write cost of the disks listed in Table 6.2. The relationship between LFS-cleaning and LFS-plugging is consistent with previous studies [72]. Overwrite, LFS-cleaning and LFS-plugging always cross at the same point when $u = 1 - 2/\eta$. Since faster disks have larger η, this cross point happens at higher disk space utilization for faster disks (e.g., $u = 94\%$ or $u_d = 97\%$ for a year 2004 disk), which means that the performance advantage of LFS over Overwrite increases as disk technologies improve. Figure 6.4 indicates that LFS outperforms Overwrite under such workloads when the cleaning space utilization is below 94% under modern disks. Under real workloads other than a uniform random update workload, LFS should perform better than what is shown in Figure 6.4 since data accesses in real workloads have skewness [51] which could significantly decrease the cleaning space utilization [72]. Therefore, under modern and future disk technologies, the cleaning cost of LFS is much less important than the common belief derived from studies with thirteen-year-old disks [108].

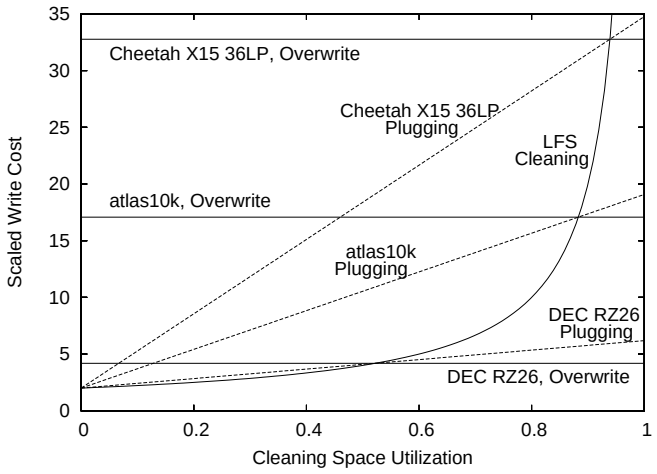


Figure 6.4: Write Costs of Different Layouts
[Smaller values indicate better performance. The segment size for Cheetah X15 36LP (year 2004 disk) is 1MB, for atlas10k (year 1999 disk) is 512KB, and for DEC RZ26 (year 1991 disk) is 128KB. The selection of segment sizes is discussed in Section 6.1.2.1. The scaled write cost of LFS-cleaning is the same for all disks.]

Many systems use disk arrays and have multiple concurrent users. The number of disks is defined as N_d. It is assumed that users send out requests without think

time. When RAID is used, all disks are viewed as one large logical disk. The stripe size of RAID is set to S pages, where S is computed based on Equation (6.3). In RAID-0, the segment size of the logical disk is $N_d S$; in RAID-5, the segment size of the logical disk is $(N_d - 1)S$, because one disk worth of space is used to store parity data. This organization allows segment I/O to utilize all available disk bandwidth and eliminates the small write penalty [89] in RAID-5.

6.1.3 The HyLog Model and Performance Potential

Figure 6.4 indicates that a small reduction in disk space utilization can significantly reduce segment cleaning cost and improve the performance of LFS. Because of the skewness in the page access distribution [51], most writes are to a small portion of data pages (called *hot* pages), while the other pages (called *cold* pages) are updated infrequently. In LFS, hot pages rarely need to be cleaned because their current copies on the disk are often invalidated by further writes to these pages before the space they occupy is reclaimed by the cleaner. Therefore, most of the cleaning cost comes from cold pages, while most of the high write performance comes from accumulating the writes to hot pages. If only these hot pages are managed by LFS while cold pages are managed by Overwrite, all free space can be dedicated to the hot pages, since Overwrite does not need extra free space. The resulting space utilization for the hot pages would be lower, which implies higher performance for the hot pages. Therefore, the overall performance could exceed both LFS and Overwrite. This leads to a new approach, called the *Hybrid Log-structured* (HyLog) layout. The basic idea underlying HyLog was first mentioned in the conclusions of Lomet's work [69]: "it is not impossible to envision an LFS in which some segments are managed using in-place updating", but no further analyses or experiments were conducted.

In HyLog, the disk is divided into fixed-size segments, similar to LFS. A segment is a hot segment (containing hot pages and free pages), a cold segment (containing cold pages and free pages), or a free segment (containing only free

pages). The hot segments and the free segments form the *hot partition*, while the cold segments form the *cold partition*.

Since LFS-plugging performs worse than LFS-cleaning under low space utilization and worse than Overwrite under high space utilization, including LFS-plugging in HyLog brings no performance benefit. Therefore, LFS-plugging is not considered when modelling HyLog. Assume the proportion of hot pages is h ($0 < h < 1$) and the proportion of writes to the hot pages (called *hot writes*) is w ($0 < w < 1$). If all free space is in the hot partition, the disk space utilization of the hot partition is

$$\frac{uh}{1-u+uh}. \tag{6.7}$$

The space utilization of segments to be cleaned, u, can be calculated from expression 6.7 based on Equation (6.2). The scaled write cost of HyLog, C'_{hylog}, is

$$\begin{aligned} C'_{hylog} &= (1-w)C'_{ow} + wC'_{lfs} \\ &= (1-w)\eta + \frac{2w}{1-u}. \end{aligned} \tag{6.8}$$

When h is 0 and 1, the cost of HyLog degrades to Overwrite and LFS, respectively. The proportion of hot writes w is a function of h, which is the CDF of the write frequencies.

For uniformly distributed random access, $w = h$. It was found that the CDF of the page update frequency in production database workloads follows the $Hill(f_{max}, k, n)$ distribution $Hill(105, 0.528, 0.546)$ [51], which is defined by $f(x) = \frac{f_{max} \cdot x^n}{k + x^n}$. Note that these distributions are for page updates before being filtered by the buffer cache. When write through is used (as in an NFS server), these distributions can also describe the page writes to disks. When write back is used (as in a database server), the page writes to disks are less skewed (closer to the uniform distribution).

The $Knuth(a, b)$ distribution (see Section 5.6.2 in page 103) is used to

represent the skewness of data accesses. $Knuth(a, b)$ means that $a\%$ of the references go to $b\%$ of the pages. Figure 6.5 shows the CDF of these distributions with different parameters.

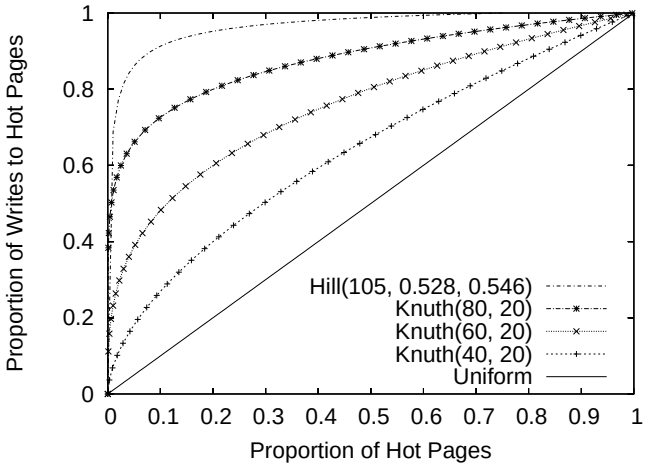
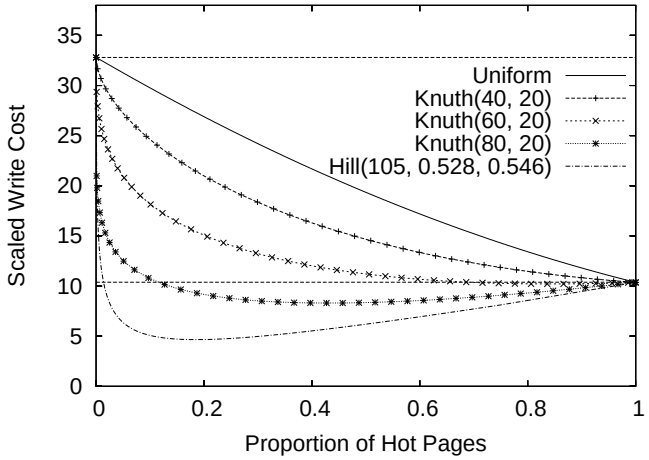
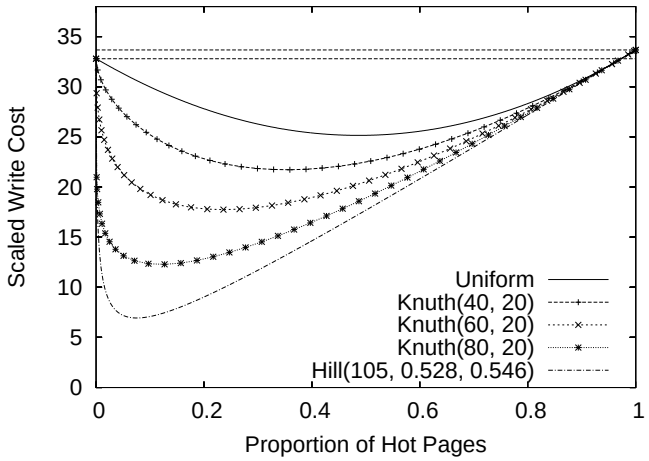


Figure 6.5: Cumulative Distribution Functions
[$Knuth(a, b)$ means that $a\%$ of the references go to $b\%$ of the pages.]

Figures 6.6(a) and 6.6(b) show the scaled write cost of HyLog under these distributions. Equation (6.2) is used to convert disk space utilization to cleaning space utilization. Since this equation works only for uniform random workloads, the results shown in Figures 6.6(a) and 6.6(b) are conservative for skewed distributions. With the right number of hot pages, HyLog outperforms both Overwrite and LFS. The higher the skewness of the distribution, the fewer hot pages are required and the more benefit can be achieved. In other words, HyLog has greater performance potential than LFS and Overwrite under high disk space utilization. When the disk space utilization is low, HyLog has limited benefit over LFS.



(a) Disk Space Utilization 90%



(b) Disk Space Utilization 97%

Figure 6.6: Performance Potential of HyLog

[The two horizontal lines in Figure 6.6(a) and 6.6(b) represent the write cost of Overwrite and LFS, respectively. η is 32.8, representing the Cheetah X15 36LP disk with 1MB segment size and 8KB page size. $Knuth(a, b)$ means that $a\%$ of the references go to $b\%$ of the pages.]

6.2 The Design of HyLog

6.2.1 Design Assumptions

It is assumed that the disk layouts under study (Overwrite, LFS, WOLF and HyLog) are at the storage level rather than the file system level. It is assumed that NVRAM is used by these disk layouts so that small synchronous writes caused by

metadata operations are not necessary, and so the metadata operations are not considered in this study. This omission greatly simplified the design and implementation of the disk layout simulator. Because LFS performs much better than Overwrite on metadata operations [96, 108], the omission of metadata operations makes the results presented for LFS, WOLF and HyLog conservative compared to Overwrite.

These assumptions, however, do not mean that HyLog can be used only at the storage level with NVRAM. The metadata integrity and fast recovery from crash must be supported without NVRAM if HyLog is used at the file system level. One approach is to apply technologies such as Soft-updates [37] and journaling [140] to HyLog. Another approach is to treat all metadata as hot pages so that fast recovery can be achieved in a way similar to LFS.

WOLF [131] reduces the segment cleaning cost of LFS by sorting the pages to be written based on their update frequencies and writing to multiple segments at a time. This idea can be easily applied to HyLog to reduce its cleaning cost further, but, to isolate the benefit realized from the design of HyLog and from the idea of WOLF, this optimization is not performed in this study.

6.2.2 Separating Algorithm

Before a page is written to the disk, HyLog runs a *separating algorithm* to determine if this page is hot. If it is, the write is delayed and the page is stored temporarily in an in-memory segment buffer. Otherwise, it is overwritten immediately to its original place on the disk. When hot pages fill up the segment buffer, they are written out to a free disk segment, freeing the disk space occupied by their old copies.

As time goes on, some hot pages may become cold. These pages are written to the cold partition rather than to their current locations in the hot partition to avoid extra cleaning overhead. As some cold pages become hot and are written to the hot partition, free space may appear in the cold partition. To reclaim this free

space more effectively, HyLog uses an adaptive cleaning algorithm to select segments with the highest cleaning benefit from both hot and cold partitions.

Separating hot pages from cold pages accurately is the key to the design of HyLog, as shown in Figure 6.6. The basic idea of the separating algorithm is simple. First, the write frequencies of recently updated pages are collected. These write frequencies are used to get the relationship between w and h. Then Equation (6.8) is used to calculate C'_{hylog} for all h. The hot page proportion h with the lowest C'_{hylog} is used as the expected hot page proportion.

Measuring η accurately is important for HyLog to make correct decisions. The service time of page I/O and segment I/O of each request is collected at the disk level. The average of the most recent 10,000 requests is used to compute η. Since a segment I/O always keeps all disks busy, while a page I/O keeps only one disk busy, page I/O is less efficient in disk arrays. If the proportion of the disk idle time is P_{idle}, η is adjusted to $\frac{\eta}{1 - P_{idle}}$.

The write frequencies of all disk pages are collected in real time. A frequency counter is associated with each page. This counter is initialized to 0, and reset to 0 after every *measurement interval*. Whenever a page is written to the disk, its frequency counter is incremented. At the end of each measurement interval, all frequency counters are sorted in a descending order and stored in an array, which is used to calculate hot writes given the hot page proportion. The separating algorithm is invoked every measurement interval. After the expected hot page proportion is obtained, a page separating threshold can be determined so that all pages with write frequencies no less than the threshold are considered hot pages.

Preliminary experiments were conducted to study the sensitivity of system performance to the value of the measurement interval. When the measurement interval is smaller than 20 minutes, the throughput is not sensitive to the measurement interval. However, the throughput starts dropping with larger measurement intervals. 20 minutes is used as the measurement interval to reduce the frequency that the separating algorithm is invoked.

6.2.3 Segment Cleaning Algorithm

HyLog's segment cleaning algorithm is adapted from the adaptive cleaning algorithm of LFS [72], which dynamically selects between cleaning and hole-plugging based on their write cost.

In Hylog, the cleaner is invoked whenever the number of free segments falls below a threshold (set to 10). In every cleaning pass, the cleaner processes up to 20MB of data. It first calculates the cost-benefit values of the following four possible cleaning choices: (1) cleaning in the hot partition, (2) hole-plugging in the hot partition, (3) cleaning in the cold partition, and (4) hole-plugging in the cold partition. It then performs cleaning using the scheme with the lowest cost-benefit value.

6.3 Methodology

6.3.1 The Simulator, Verification, and Validation

Trace-driven simulations were used to compare the throughput of different disk layouts. The simulator consists of a disk component, a disk layout component, and a buffer cache component. The disk component was ported from DiskSim 2.0 [38]. The disk layout component simulates disk layouts for Overwrite, LFS, WOLF, and HyLog. The implementation of LFS is based on the algorithm descriptions of two previous LFS simulators [72, 96] and the source code of the Sprite operating system [114]. The implementation of WOLF is based on the algorithm description of a previous WOLF simulator [131]. The buffer cache component uses the LRU algorithm. Together, this component and the disk layout component comprise over 31,000 lines of C++ code. The three components communicate through an event-driven mechanism. Overwrite, LFS and WOLF are implemented as special cases of HyLog. Overwrite is obtained by considering all pages as cold, and LFS and WOLF are obtained by treating all pages as hot. Therefore, the only difference between these disk layouts is the page separating algorithm. This design

guarantees that HyLog does not perform any "shortcut" operations that the other three approaches do not have and thus improves the fairness of performance comparisons. Since HyLog performs more work than Overwrite, LFS, and WOLF, this design may increase the execution time of the simulation, which does not affect the simulation result in anyway.

In order to verify the disk layout component, a simple disk layout simulator called *TinySim* was developed independently. TinySim simulates LFS and WOLF, and supports a single user and single disk environment. TinySim and the disk layout component were run under uniformly distributed random update and hot-cold (10% of the pages are referenced 90% of the time) synthetic workloads [96], respectively. The overall write cost [72, 131], which is the key performance metric of LFS and WOLF, was obtained from both simulators. In most cases, the differences between the results of the two simulators were within 5%.

After verification, the cleaning algorithms in the disk layout component were validated against results presented by Matthews *et al.* [72]. Figure 6.7 shows the overall write costs of the cost-age, hole-plugging, and adaptive cleaning algorithms under a uniformly distributed random update workload. These cleaning algorithms show trends very similar to those in Figure 6 of the study by Matthews *et al.* [72].

The implementations of Overwrite and LFS were validated further by comparing their performance with that of FFS (uses overwrite) and BSD LFS done by Seltzer *et al.* [108] under the TPC-B benchmark. As was done by Matthews *et al.* [72], the uniformly distributed random update workload is used to simulate the TPC-B workload. Since the DSP 3105 disk used by Seltzer *et al.* [108] is not available in DiskSim, a similar disk, the DEC RZ26, was used in the validation experiments. Table 6.3 lists the specifications of the two disks. The DEC RZ26 has slightly slower average seek time and slightly higher transfer rate because it has one more sector per track than the DSP 3105, but it is very similar otherwise.

Table 6.4 shows the throughput of Overwrite and LFS in this study and by

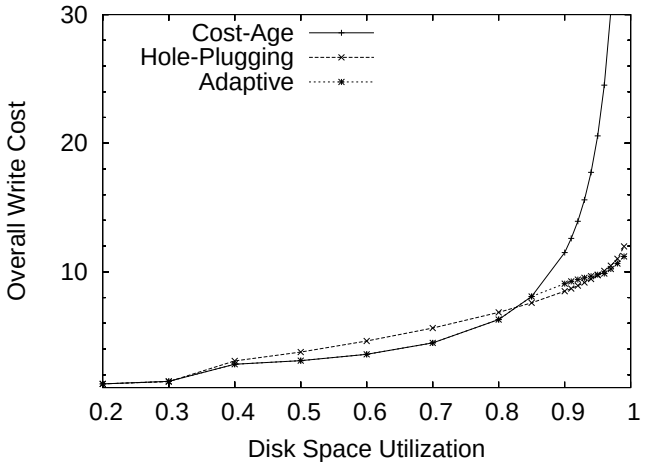


Figure 6.7: Validation of Overall Write Cost of LFS
[The workload is uniform random update. Page size is 8KB, segment size is 256KB, $T_{pos} = 15ms$, $B = 5MB/s$, and cache size is 128MB.]

Table 6.3: Disk Comparison for Simulator Validation

Parameters	DSP 3105	DEC RZ26
RPM	5400	5400
Sectors/Track	57	58
Cylinders	2568	2599
Platters	14	14
Track Buffer	256KB	285KB
Avg. Seek Time	9.5ms	9.8ms
Transfer Rate	2.3MB/s	2.3MB/s

Seltzer *et al.* [108]. Although the reported throughput of LFS with cleaning in the study of Seltzer *et al.* was 27.0, it has been argued [85] that 34.4 should be a more reasonable value. Therefore, 34.4 is used here when calculating the difference. The 4.8% lower throughput observed for Overwrite in the experiments may be due to the 3.2% slower seek time of the DEC RZ26. The LFS implementation used by Seltzer *et al.* can achieve only 1.7MB/s write throughput, 26% slower than the maximum hardware bandwidth, because of "missing a rotation between every 64 KB transfer" [108]. Since the number of segment reads is equal to the number of segment writes (for every segment read, there is always u segment write for cleaning and $1 - u$ segment write for new data), this slowdown of segment write should cause 13% performance difference, which matches the difference in

Table 6.4. Since the differences in all results are within a reasonable range, it is reasonable to believe that the implementations of Overwrite and LFS used in this study are valid.

Table 6.4: Throughput Validation (Disk Space Utilization is 50%)

Layout	Seltzer *et al.* [108]	This study	Diff.
FFS/Overwrite	27.0	25.7	-4.8%
LFS w/o cleaning	41.0	43.3	5.6%
LFS w cleaning	**27.0** (**34.4**)	39.0	13.4%

6.3.2 The Workloads

Three traces were used in the performance experiments for this study: TPC-C, Sub-Financial1, and NFSEmail. Their characteristics are summarized in Table 6.5.

Table 6.5: Trace Characteristics

	TPC-C	Sub-Financial1	NFSEmail
Data size(MB)	5088	10645	9416
Page size(KB)	4	4	8
#reads($\times 10^6$)	176.46	0.97	21.05
#writes($\times 10^6$)	32.77	3.41	7.64
Logical reads writes ratio	5.38	0.28	2.76
Physical reads writes ratio	1.37	0.13	2.56

The TPC-C and the NFSEmail trace are the same as the ones described in Section 5.3.4 (page 86). The NFSEmail trace contains 85% reads and writes, and 15% directory operations. Since the sizes of the directories are unknown from the trace, it is difficult to replay the directory operations in the simulator, but because it is assumed that NVRAM is used, these directory operations do not cause expensive synchronized writes. So their impact on performance is small. The directory operations were discarded and only the reads and writes of the trace were used in this study.

The Sub-Financial1 trace is part of the Financial1 [113] trace described in Section 5.3.4. Sub-Financial1 contains all I/O requests of the Financial1 trace

except those from three of the largest volumes. This trace reduction significantly reduces the computing resources required by the simulator. Since these three volumes have the fewest requests relative to their sizes, omitting them is expected to have little impact on the results.

6.3.3 Experimental Setup

Since the interest of this study is the performance of various disk layouts on busy systems, the simulator is configured as a closed system without think time. That means the next trace record is issued as soon as the processing of the previous one finishes. Using this method, a small number of users in the trace can represent the workloads imposed on a system by many more users with think time. For example, the weighted average think time plus keying time defined in Clause 5.2.5.7 of TPC-C benchmark version 5.0 [124] is 21 seconds. The simulation results indicate that the system with 30 users without think time has an average response time of 1.28 seconds if one disk is present in the system. Assuming that the number of users with think time in the system is n, the average arrival rate of users is $\frac{n}{21+1.28} = \frac{n}{22.28}$. From Little's Law: $30 = \frac{n}{22.28} \times 1.28$. Therefore, $n = 522$, which indicates that the workload generated by 30 users without think time presents equivalent workload to that generated by 522 users with 21 seconds think time between requests.

The Quantum atlas10k 1999 disk model, which is the latest disk model provided by DiskSim 2.0, was used in this study. Its specifications are given in Table 6.6. Write-back caching is disabled to protect data loss from power failure. The disk scheduling algorithm is SCAN based on logical page numbers.

To study the performance of disk layouts on today's disks and future disks, disk models for a high-end disk of year 2004 and a high-end disk of the sort that customers might expect to see in year 2008 were also designed. Looking back over 15 years history of disk technology evolution, it seems reasonable to make the following assumptions: every 5 years, transfer rate increases by 242% [43], average

Table 6.6: Disk Specifications

Parameters	Atlas10k (Year 1999)	Year 2004 Disk	Year 2008 Disk
RPM	10025	15000	24000
Sectors/Track	229-334	476	967
Cylinders	10042	10042	10042
Platters	6	8	8
Size(GB)	9.1	18	36
Seek Time(ms)	5.6	3.6	2.0
Bandwidth(MB/s)	20.4	61	198

seek time decreases by 76% [101], and RPM (Rotations Per Minute) increases by 61% [3]. It is also assumed that all cylinders have the same number of tracks, the number of platters is 8, and the disk size for the year 2004 disk is 18GB and for the year 2008 disk is 36GB. The specifications of these two disks calculated on the basis of these assumptions are given in the two rightmost columns of Table 6.6. The seek time distribution data were created by linearly scaling the seek time distribution of the atlas10k disk defined in DiskSim.

RAID-0 and RAID-5 were used as the multi-disk models in this study. The stripe size for both RAID-0 and RAID-5 is computed based on Equation (6.3) and then rounded to the closest powers of two. For RAID-0 arrays with n disks, the segment size is $n \times StripeSize$. For RAID-5 arrays, the segment size is $(n-1) \times StripeSize$, since $1/n$ of the total disk space is dedicated to parity data.

In order to vary the disk space utilization, only part of the disk is accessed, independent of the actual size of the disk. For example, if the data size is 6GB and the disk space utilization is 60%, the total disk space required is $\frac{6GB}{60\%} = 10GB$. If there are 5 disks, the first 2GB of each disk is used. Since the disk layout approaches do not handle page allocation and deallocation, all data are stored on the former part of the disk initially. As a result, the seek time (particularly for Overwrite) is very short, which makes η smaller. Thus this data layout makes the performance results for LFS, WOLF, and HyLog conservative compared to Overwrite than in real workloads where data are often placed far apart. For LFS,

WOLF, and HyLog, the data will eventually spread across the whole disk as data are written, which is considered as the warmup period. Only the performance data collected after the warmup period is measured.

The performance metric used in this study is throughput, defined as the number of I/O requests finished per second.

Table 6.7 summarizes the parameters and values used in the experiments. Since these parameters can be easily controlled in the TPC-C trace, this trace is used to study the impacts of various parameters on throughput. When evaluating the throughput of RAID-5, a 9-disk RAID-5 array is compared with an 8-disk RAID-0 array so that they have the same segment size. The buffer cache used for HyLog, LFS and WOLF are the total buffer cache size minus one segment (for LFS and HyLog) or two segments (for WOLF) worth of space required for segment writing. As a result, all approaches use the same amount of memory to allow a fair performance comparison.

Table 6.7: Experimental Parameters

Configuration	Values	Default
Disk layout	Overwrite, LFS, WOLF, HyLog	—
Number of users	1–30	20
Number of disks	1–15	4
Disk utilization	0.5–0.98	—
Disk type	atlas10k, year 2004 disk, year 2008 disk	atlas10k
Disk array type	RAID-0, RAID-5	RAID-0
Workload	TPC-C, Sub-Financial1, NFSEmail	TPC-C
Buffer cache size	—	400MB

6.4 Performance Evaluation

6.4.1 Validation of the Cost Model

Since the cost model was developed for the uniform random update workload, the simulation results for the same workload were used to validate the cost model. In particular, the previous results for TPC-B [85, 108], a random update workload,

were used. Since the write cost is the average time required to write a page and a transaction requires a page read and a page write, the throughput X is computed as

$$X = \frac{1}{T_{pg} + C + T_{cpu}},$$

where C is the write cost of the disk layout, and T_{cpu} is the CPU overhead for processing each page, which is $0.9ms$ for Overwrite and $1.8ms$ for LFS [108]. Table 6.8 shows that the modelling results match the measurement results well. The modelling results are also close to the simulation results shown in Table 6.4 (page 134).

Table 6.8: Cost Model Validation

Layout	Previous	Model	Difference
Overwrite	27.0 [108]	28.6	6.0%
LFS-cleaning	34.4 [85]	37.3	8.4%

6.4.2 Impact of Disk Space Utilization and Disk Type

Figure 6.8 shows the throughput of different layouts under various disk space utilization and different disks. Since the throughput curves of LFS, WOLF and HyLog almost overlap for the year 2004 and year 2008 disks, only one line is shown for each of these disks. The throughput of all layouts improves with faster disks. The throughput of Overwrite is not affected by the disk space utilization, while the throughput of other layout approaches decreases when the space utilization is high. The faster the disk, the more LFS and WOLF can tolerate the high space utilization because faster disks have higher η as shown in Figure 6.1 (page 119). Figure 6.9 gives a closer look at the throughput of the atlas10k disk. The throughput of WOLF overlaps that of LFS for most configurations and outperforms LFS by 5% when the disk space utilization u_d is very high (98%). The throughput of HyLog overlaps that of LFS when $u_d \leq 95\%$. This is because HyLog considers all pages as hot based on its cost model Equation (6.8) (see Figure 6.6(a) on page 128). The throughput of HyLog is comparable to Overwrite when the disk

space utilization is higher. HyLog outperforms Overwrite by 7.4% when the disk space utilization is 97%.

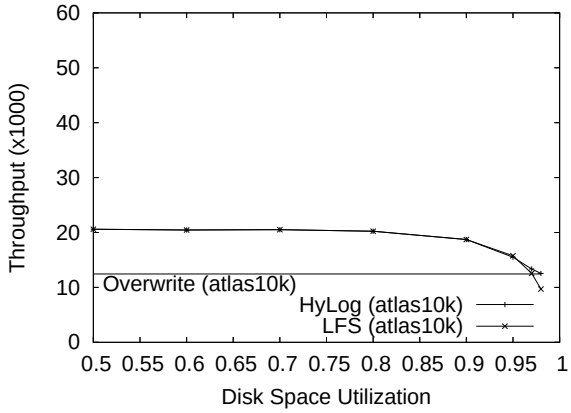
To provide some insights into the performance that LFS and HyLog show above, two example points in the figure were analyzed further: LFS running on an atlas10k disk with 95% disk space utilization and HyLog running on an atlas10k disk with 98% disk space utilization.

In the LFS example, the cleaning space utilization u obtained from the simulator is 88.4%. This is lower than the 90.2% computed from Equation (6.2) because of the skewness of accesses in TPC-C. Therefore, to write one segment of new data, $\frac{1+u}{1-u} = 16.3$ segment I/Os need to be performed for cleaning. So the cleaning traffic contributes 94.2% to the total segment I/O traffic. The T_{pg} and T_{seg} values obtained from simulation are 5.6ms and 27.3ms, respectively. Therefore, $\eta = 26.3$. The proportion of disk idle time obtained from the simulator is 30%, so η should be adjusted to $\eta/(1-30\%) = 37.6$. Based on the scaled write cost model,

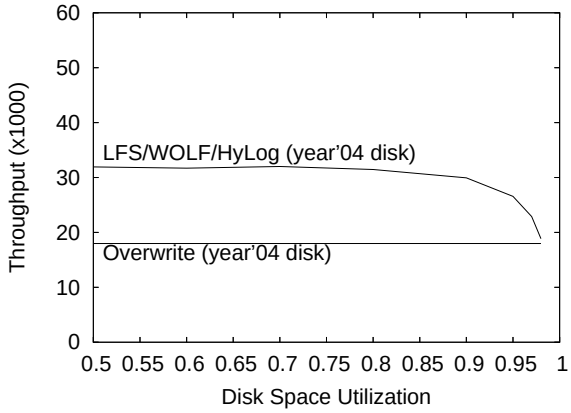
$$C'_{ow}/C'_{lfscleaning} = \eta(1-u)/2 = 2.2,$$

which means that the write throughput of LFS is 2.2 times the write throughput of Overwrite. Since the write traffic contributes 42% to the total traffic after being filtered by the buffer cache (Table 6.5 in page 134), LFS outperforms Overwrite by 30%, which is close to the simulation result of 27%.

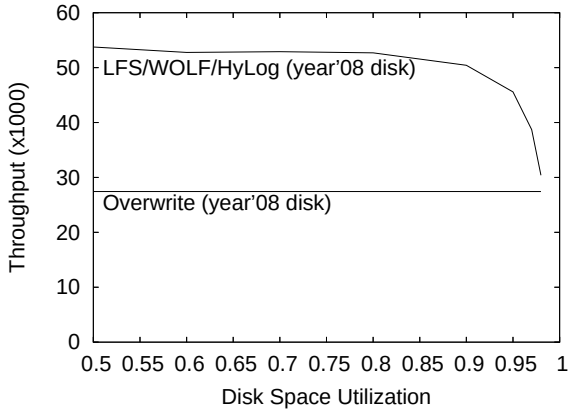
In the HyLog example, the hot page proportion selected by the page separating algorithm during the run is 35%-45%. The data collected at the first measurement interval after warmup were used as the example. The proportion of hot pages is 42.2%, and the proportion of hot writes is 58.2%. The space utilization of segments being cleaned is 93.4%, which is lower than that in LFS for the same configuration (96.2%). The proportion of disk idle time is 22.5%, the T_{pg} and T_{seg} are 5.8ms and 27.2ms, respectively, and the adjusted η is $\frac{T_{pg}S}{T_{seg}(1-P_{idle})} = 35.2$. Therefore, the write cost model indicates that the write throughput of the hot partition is 16% higher than Overwrite. Thus the overall weighted write throughput is 9% higher



(a) Year 1999 Disk (Atlas10k Disk)



(b) Year 2004 Disk



(c) Year 2008 Disk

Figure 6.8: The Impact of Disk Space Utilization on System Throughput
[The number of users is 20, the number of disks is 4, the trace is TPC-C, and the buffer cache size is 400MB.]

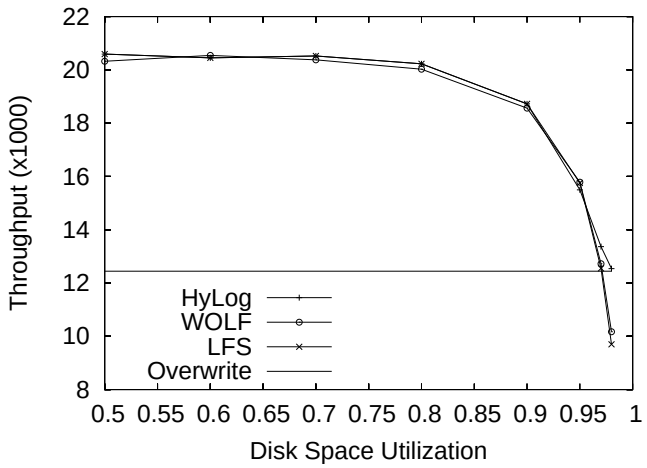


Figure 6.9: The System Throughput using the Atlas10k Disk
[The number of users is 20, the number of disks is 4, the trace is TPC-C, and the buffer cache size is 400MB.]

than Overwrite. Taking the read traffic into account, the throughput of HyLog is 1.036 that of Overwrite, which is close to the simulation result of 1.008. The write throughput of LFS computed from the cost model under 98% disk space utilization is 66.9% of Overwrite, and the overall throughput of LFS including read and write traffic is 82.8% of Overwrite, which is close to the simulation result of 78.0%.

Figure 6.10 shows how well the separating algorithm works. The hot page proportion found by the separating algorithm (35%-45%) is close to the range of values that results in good throughput when a fixed hot page proportion is used, and the achieved throughput is 96.4% of the maximum possible throughput.

6.4.3 Impact of Number of Users and Number of Disks

Figures 6.11 and 6.12 show the throughput normalized to Overwrite under different numbers of users and disks. The throughput of WOLF is not shown since it is virtually identical to LFS. Two trends can be observed in the relative throughput of LFS, WOLF and HyLog: (1) throughput drops with more users; (2) throughput drops with more disks.

With more users, the average disk seek time is reduced because of the use of

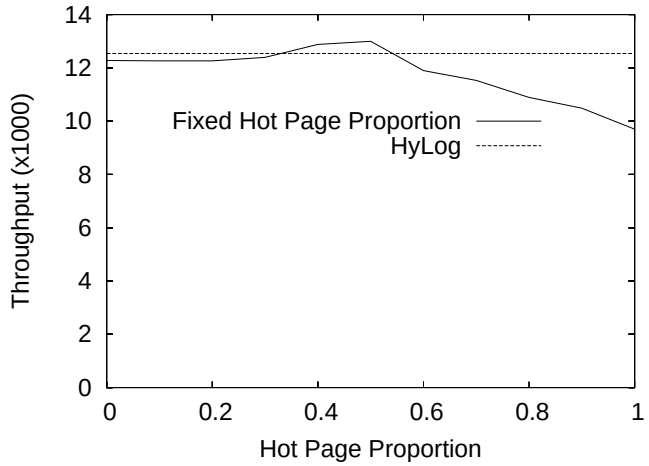


Figure 6.10: Sensitivity to Separating Criteria

[The number of users is 20, the number of disks is 4, the trace is TPC-C, the disk space utilization is 98%, and the buffer cache size is 400MB.]

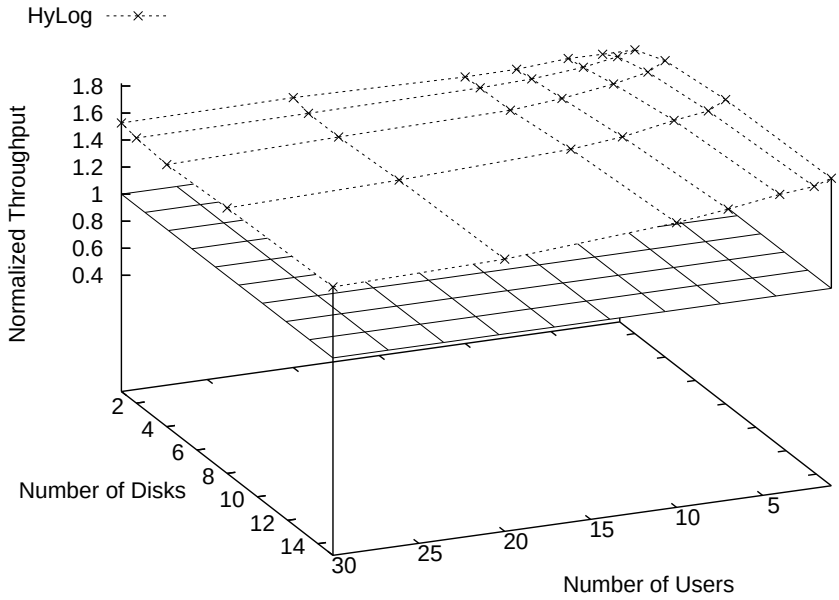
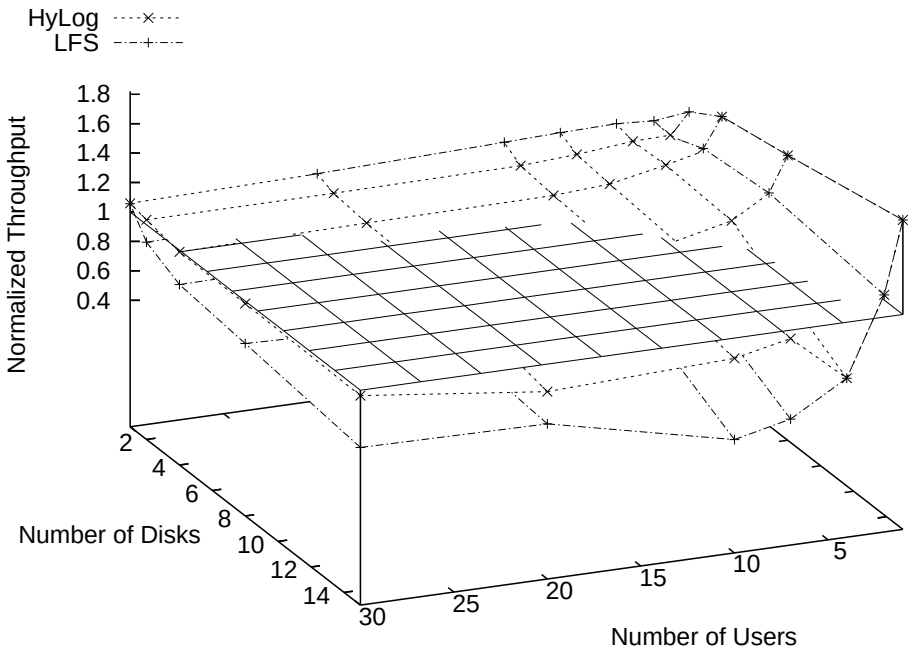


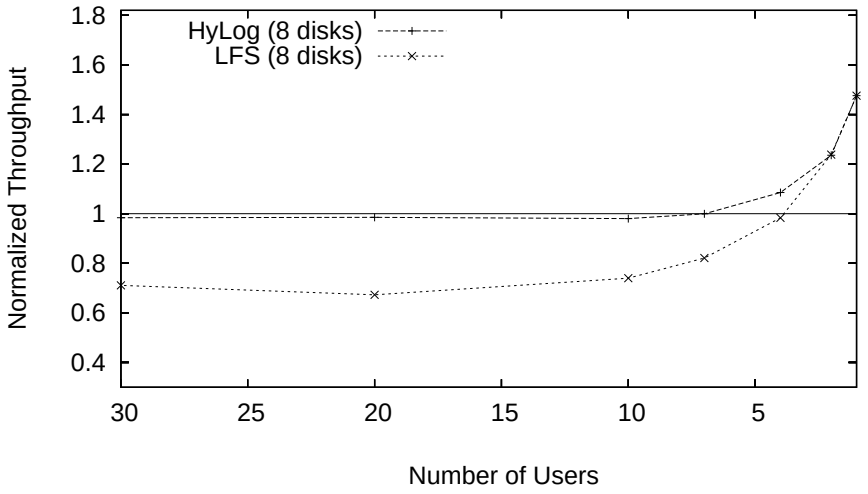
Figure 6.11: Impact of Number of Users and Number of Disks (Disk Space Utilization is 90%)

[The disk is atlas10k, trace is TPC-C. The throughput curves of LFS and HyLog are virtually identical, and so only the throughput of HyLog is drawn.]

the disk scheduling algorithm, which reduces η. The disk idle time in Overwrite is



(a) Disk Space Utilization 98% – All Data



(b) Disk Space Utilization 98% – The Hidden Points

Figure 6.12: Impact of Number of Users and Number of Disks (Disk Space Utilization is 98%)

[The disk is atlas10k, and the trace is TPC-C. Figure 6.12(b) shows the hidden data points of Figure 6.12(a).]

reduced with more users, which also reduced the adjusted η, i.e., $\frac{\eta}{1-P_{idle}}$. This decrease is not affected by the disk space utilization. Therefore, the first trend happens in both low disk space utilization (Figure 6.11) and high disk space utilization (Figures 6.12(a) and 6.12(b)).

With more disks, the segment size is larger, and so the cleaning cost is higher [72], which reduces the benefit of the log-structured layout. This happens only when cleaning cost plays an important role, which is true when the disk space utilization is high. Therefore, the second trend is apparent only when the disk space utilization is high (Figure 6.12(a)).

In Figures 6.12(a) and 6.12(b), the throughput of HyLog is virtually identical to that of LFS when LFS outperforms Overwrite, and HyLog becomes comparable to Overwrite when Overwrite outperforms LFS. HyLog incorrectly follows LFS when there are 4 users and 15 disks, because at this configuration a very small error in the estimation of η can cause HyLog to make the wrong decision, while HyLog can tolerate some error in the estimation of η in other configurations.

6.4.4 Impact of Disk Array Type

Figure 6.13 shows the throughput of the four disk layouts (Overwrite, LFS, WOLF, and HyLog) on RAID-0 and RAID-5. For Overwrite, the throughput on RAID-5 is about 50% of that on RAID-0. This performance degradation is caused by the slower page update of RAID-5. For LFS and WOLF, the use of RAID-5 increases throughput by 6.5%-10%, because the segment I/O performance is not affected by small write penalty of RAID-5, while the one more disk in RAID-5 increases the page read throughput. When the disk space utilization is high, the throughput of HyLog on RAID-0 is comparable to Overwrite. The throughput of HyLog on RAID-5 is comparable to LFS because the slower page I/O in RAID-5 makes η higher, which makes HyLog treat most pages as hot pages.

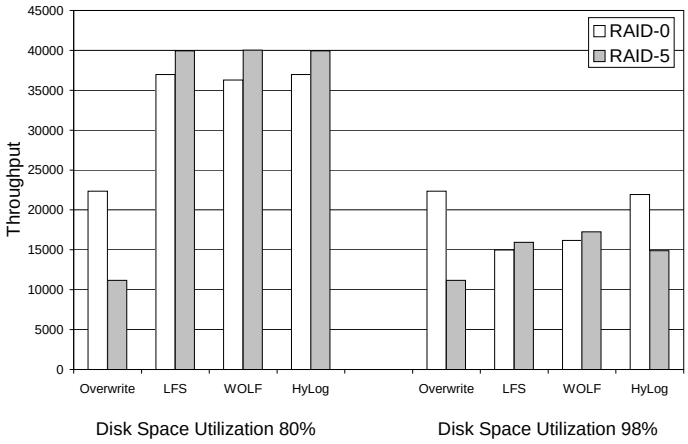


Figure 6.13: Throughput under RAID-0 and RAID-5 Arrays
[The RAID-0 array contains 8 disks, the RAID-5 array contains 9 disks, the disk is the atlas10k, the trace is TPC-C, the number of users is 20, the segment size is 512KB per disk, and the buffer cache size is 400MB.]

6.4.5 Impact of Workload

Figure 6.14 shows the throughput of the four disk layouts using the Sub-Financial1 and NFSEmail traces. The throughput is normalized relative to that of Overwrite. For both traces, the performance advantage of LFS, WOLF, and HyLog is much higher than that observed with the TPC-C trace. This difference is attributed to two facts. First, the distribution of data updates in the Sub-Financial1 and NFSEmail traces is more skewed than it is in the TPC-C trace, leading to lower segment cleaning cost. Second, the proportion of writes in these two traces is much higher than in the TPC-C trace, since many reads have already been filtered out by client-side buffers (in the NFSEmail trace) or in-memory buffers (in the Sub-Financial1 trace). Because the Sub-Financial1 trace is more skewed and has lower read-to-write ratio than the NFSEmail trace, the advantage of log-structured layouts in the Sub-Financial1 trace is higher than in the NFSEmail trace. The performance results under other configurations have similar trends and thus are not shown.

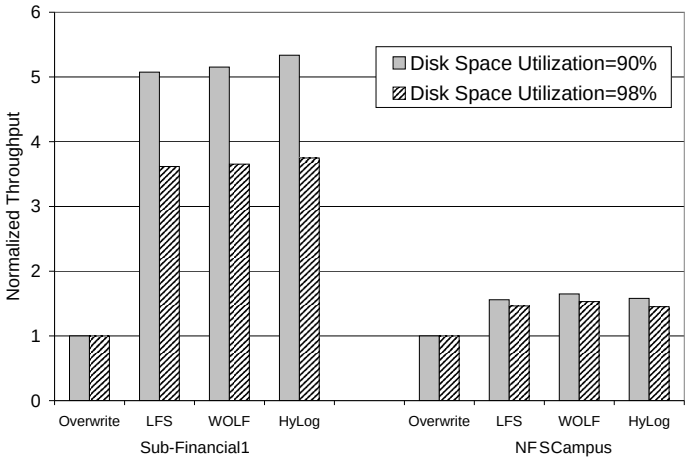


Figure 6.14: Normalized Throughput under Real Workloads
[The throughput is normalized relative to that using Overwrite. The disk is atlas10k, the number of disks is 1, the segment size is 512KB, and the buffer cache size is 400MB.]

6.5 Summary

The write performance of the Overwrite and LFS disk layouts was investigated. A write cost model was developed to compare the performance of these disk layouts. Contrary to the common belief that its high segment cleaning cost disadvantages LFS, it is found that because of advances in disk technologies, the performance of LFS is significantly better than Overwrite on modern and future disks, even under the most pathological workload for LFS (uniform random update), unless the disk space utilization is very high.

Since LFS still performs worse than Overwrite under certain conditions such as high disk space utilization, a new disk layout approach called HyLog is proposed. HyLog uses a log-structured approach for hot pages to achieve high write performance and overwrite for cold pages to reduce the segment cleaning cost. The page separating algorithm of HyLog is based on the write cost model and can separate hot pages from cold pages dynamically. Simulation results on a wide range of system and workload configurations show that HyLog performs comparably to the best of Overwrite, LFS and WOLF in most configurations.

Chapter 7

Conclusions

Storage is an important component of any large scale computer system. Storage management is important to the overall system performance. This thesis studies performance issues in two important components of storage management in large scale systems, namely the buffer cache management and disk layout management.

The thesis contains three parts: self-tuning in buffer cache management, lock contention in buffer cache management, and disk layout management. Section 7.1 summarizes each part of the thesis work, Section 7.2 states the main contributions, and Section 7.3 briefly outlines further research directions.

7.1 Thesis Summary

Self-tuning of Buffer Cache Management

The buffer cache layer in storage management caches popular disk pages in memory in an attempt to speed up accesses to the disk pages. The buffer cache management algorithm used in real systems often has many parameters that require careful tuning to get good performance. This thesis studies the buffer cache management algorithm in a real DBMS running an I/O-intensive workload. The I/O activities and impact of various parameters on performance are investigated through measurements and simulation. A new self-tuning algorithm is proposed to automatically tune an important parameter in the buffer cache management algorithm by monitoring the I/O activities of the buffer cache. Results from the simulator show that this algorithm achieves performance comparable to that of a manually tuned system under various system configurations.

Lock Contention of Buffer Cache Management

The buffer cache replacement algorithm is the most important element of buffer cache management. Most advanced replacement algorithms employ a global data structure to manage the pages in the buffer cache. Since every access to the buffer cache needs to change the global data structure, it is protected by a lock. In large scale systems with multiple processors, this lock can easily become a contention point and significantly limit system throughput. There are no existing universal approaches to solve lock contention. Some approaches are valid only for a specific algorithm, some approaches have high overhead, and some approaches decrease the hit ratio of the buffer cache, which significantly decreases system throughput when the system is I/O-bound. Various factors affecting lock contention are studied and a new approach called the multi-region cache is proposed. The multi-region cache can be applied to most buffer cache replacement algorithms. Analysis and evaluation results indicate that the multi-region cache reduces lock contention to an insignificant level, affects the overall hit ratio of the buffer cache only marginally, and incurs little overhead.

Disk Layout Management

Work at the disk layout layer tries to improve the I/O efficiency of the storage management subsystem by appropriately arranging the layout of disk pages. The typical disk layout approach, called Overwrite, is optimized for sequential reads and writes of a single file or files in the same directory. In large scale systems with many concurrent users and large buffer caches, most reads are absorbed by the buffer cache, and the interleaved writes from different users make writes randomly scattered over the disks. Although the Log-structured File System (LFS) is optimized for such workloads, previous studies have found that its expensive garbage collection overhead offsets its benefit to Overwrite in OLTP workloads. Analytical models are developed to investigate the write performance of Overwrite and LFS. It is found that because of the much faster improvement of disk transfer

bandwidth over disk positioning time, LFS performs much better than Overwrite on modern and future disks in most workloads, including OLTP, unless the disk space utilization is very high. A new approach is proposed, called HyLog, which is a hybrid of LFS and Overwrite. Simulation results show that HyLog achieves performance comparable to the best of existing disk layout approaches in most workloads and configurations.

7.2 Thesis Contributions

In summary, the main contributions of this thesis are:

- A self-tuning algorithm is proposed to automatically tune parameters of buffer cache management. This algorithm achieves performance close to the best manually tuned system.
- The problem of lock contention in buffer cache management is investigated. A new approach, called *multi-region* is proposed to eliminate the lock contention of buffer cache. This approach can work together with most buffer cache replacement algorithms. It affects the overall hit ratio of the buffer cache only marginally and incurs little overhead.
- Different disk layout management approaches are modeled and their performance characteristics are analyzed. A new approach called *HyLog* is proposed and is found to achieve performance comparable to the best of existing disk layout approaches in most workloads and configurations.

7.3 Future Work

There are many open research issues related to this thesis work. Some possible future research directions may include:

1. Studying whether the self-tuning page cleaning algorithm can respond well to workload changes in the system.

2. Implementing the self-tuning page cleaning algorithm in a real DBMS system and evaluating its performance under real workloads.

3. Designing better hash functions for the multi-region cache so that its performance is not compromised, even in highly skewed workloads. Modelling analysis indicates that when the distribution of logical requests to pages is highly skewed, the hit ratio of the multi-region cache has larger variance. A better hash function that scatters hot pages into different regions could reduce this variance.

4. Improving the multi-region cache so that its performance is more stable when a large number of regions are used. Simulation results indicate that the hit ratio of the multi-region cache decreases when there are too many regions. This is because some regions have too low hit ratios. If these regions can be identified and their pages can be rearranged, multi-region cache could achieve good performance even when many regions are used.

5. Implementing and evaluating the multi-region cache approach in real systems.

6. Using different replacement algorithms in different regions of the multi-region cache to adaptively select the best performing algorithm. This is similar to the caching using multiple experts approach [5, 42], but should have much lower overhead. Each region of the multi-region cache has almost the same performance. When different replacement algorithms are used in each region, the replacement algorithm performs the best on one region is likely to perform the best on all other regions. Therefore, the best replacement algorithm can be adaptively selected and applied to most regions without simulating all replacement algorithms on each buffer cache access.

7. Designing new cache replacement algorithms which can take advantage of full region scan at cache misses using multi-region cache. Scanning the whole region on a cache miss incurs little overhead when each region is small. This

relaxes the time complexity from $O(1)$ to $O(n)$, and thus enables more design choices when designing replacement algorithms.

8. Designing approaches to implement advanced replacement algorithms in virtual memory management with small additional hardware support. By recording the last reference time in the page table by hardware, LRU can be implemented by scanning the whole region for the candidate to evict. With slightly more additional hardware, a large class of replacement algorithms can be implemented in virtual memory in a similar way. The performance benefits and hardware costs of such design could provide valuable guidance to computer architecture designers.

9. Studying the read performance of HyLog, especially in workloads with random updates and sequential reads, since the sequential layout of data are damaged by the random updates and could decrease the read performance.

10. Stabilizing the LFS implementation in NetBSD and evaluating its performance under various benchmarks. The LFS implementation in NetBSD is the most close to complete implementation of LFS available in modern open source operating systems. After making the NetBSD LFS implementation more stable, performance tests can be conducted on modern hardware to verify its performance against that predicted by the write cost model proposed in Section 6.1 (page 116).

11. Implementing HyLog in NetBSD and comparing its performance to LFS and other file systems.

12. Designing approaches to add snapshot support in LFS and HyLog. Implementing it in real systems, and comparing its performance to existing systems supporting snapshots, such as FreeBSD FFS [35] and WAFL [48].

References

[1] Anurag Acharya, Mustafa Uysal, and Joel H. Saltz. Active disks: Programming model, algorithms and evaluation. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS 1998)*, pages 81–91, San Jose, CA, October 1998.

[2] Nick Allen. Don't waste your storage dollars: What you need to know. Research note, Gartner Group, March 2001. `http://www.gartner.com/DisplayDocument?doc_cd=96816`.

[3] Dave Anderson, Jim Dykes, and Erik Riedel. More than an interface — SCSI vs. ATA. In *Proceedings of the 2^{nd} USENIX Conference on File and Storage Technologies (FAST 2003)*, pages 245–257, San Francisco, CA, March-April 2003.

[4] Eric Anderson, Ram Swaminathan, Alistair Veitch, Guillermo A. Alvarez, and John Wilkes. Selecting RAID levels for disk arrays. In *Proceedings of the 1^{st} USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 189–201, Monterey, CA, January 2002.

[5] Ismail Ari, Ahmed Amer, Ethan Miller, Scott Brandt, and Darrell Long. Who is more adaptive? ACME: Adaptive caching using multiple experts. In *4^{th} Workshop on Distributed Data and Structures (WDAS 2002)*, pages 143–158, Paris, France, March 2002.

[6] Martin F. Arlitt. A performance study of Internet web servers. Master's thesis, Department of Computer Science, University of Saskatchewan, 1996.

[7] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the 13^{th} ACM symposium on Operating systems principles (SOSP 1991)*, pages 198–212, 1991.

[8] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with adaptive replacement. In *Proceedings of the 3^{rd} USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 187–200, San Francisco, CA, March-April 2004.

[9] Luiz Andr Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA 1998)*, pages 3–14, Barcelona, Spain, 1998.

[10] BerkeleyDB 4.1.25. `http://www.sleepycat.com`.

[11] BerkeleyDB 4.1.25 Win32 mutex implementation. Source code `mutex/mut_win32.c`. `http://www.sleepycat.com/download/index.shtml`.

[12] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the 1995 USENIX Annual Technical Conference (USENIX 1995)*, pages 277–288, New Orleans, LA, January 1995.

[13] Kurt P. Brown. *Goal-oriented Memory Allocation in Database Management Systems*. PhD thesis, Computer Sciences Department, University of Wisconsin, Madison, WI, 1995.

[14] Kurt P. Brown, Michael J. Carey, and Miron Livny. Goal-oriented buffer management revisited. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD 1996)*, pages 353–364, Montreal, PQ, 1996.

[15] Fang-Fang Cai, M Elizabeth C Hull, and David A. Bell. Buffer management for high performance database systems. In *Proceedings of the High-Performance Computing on the Information Superhighway (HPC-Asia 1997)*, pages 633–638, Seoul, Korea, April-May 1997.

[16] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled caching, prefetching and disk scheduling. *ACM Transactions on Computer Systems (TOCS)*, 14(4):311–343, 1996.

[17] Valeria Cardellini, Emiliano Casalicchio, Michele Colajanni, and Philip S. Yu. The state of the art in locally distributed web-server systems. *ACM Computing Surveys (CSUR)*, 34(2):263–311, 2002.

[18] Zhifeng Chen and Yuanyuan Zhou. Eviction-based cache placement for storage caches. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX 2003)*, pages 269–282, San Antonio, TX, June 2003.

[19] Jongmoo Choi, Sam H. Noh, Sang Lyul Min, and Yookun Cho. An adaptive block management scheme using on-line detection of block reference patterns. In *Proceedings of the 1998 International Workshop on Multimedia Database Management Systems (IW-MMDBMS 1998)*, pages 172–179, Dayton, OH, August 1998.

[20] Jongmoo Choi, Sam H. Noh, Sang Lyul Min, and Yookun Cho. Towards application/file-level characterization of block references: A case for fine-grained buffer management. In *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2000)*, pages 286–295, Santa Clara, CA, June 2000.

[21] Hong-Tai Chou and David J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the* 11^{th} *International Conference on Very Large Data Bases (VLDB 1985)*, pages 174–188, Stockholm, Sweden, August 1985.

[22] Wesley W. Chu and Holger Opderbeck. Program behavior and the page-fault-frequency replacement algorithm. *Computer*, 9(11):29–38, November 1976.

[23] Jen-Yao Chung, Donald Ferguson, George Wang, Christos Nikolaou, and Jim Teng. Goal oriented dynamic buffer pool management for data base systems. Technical Report TR94-0125, ICS/FORTH, Heraklion, Crete, Greece, October 1994.

[24] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the* 3^{rd} *USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 1–14, San Francisco, CA, March-April 2004.

[25] Microsoft Corporation. Microsoft SQL Server 7.0 storage engine capacity planning tips. MSDN Library, March 1999. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsql7%/html/storageeng.asp`.

[26] OSDL Database test 3. `http://www.osdl.org/lab_activities/kernel_testing/osdl_database_test_su%ite/osdl_dbt-3/`.

[27] Timothy Denehy, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Bridging the information gap in storage protocol stacks. In *Proceedings of the 2002 USENIX Annual Technical Conference (USENIX 2002)*, pages 177–190, Monterey, CA, June 2002.

[28] Peter J. Denning. Effects of scheduling on file memory operations. In *AFIPS Spring Joining Computer Conference*, pages 9–21, Washington, D.C., April 1967.

[29] Peter J. Denning. The working set model for program behavior. *Communications of the ACM (CACM)*, 11(5):323–333, May 1968.

[30] Rohit Dube. A comparison of the memory management sub-system in FreeBSD and Linux. Technical report, Department of Computer Science, University of Maryland, September 1998.

[31] Wolfgang Effelsberg and Theo Haerder. Principles of database buffer management. *ACM Transactions on Database Systems (TODS)*, 9(4):560–595, December 1984.

[32] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive NFS tracing of email and research workloads. In *Proceedings of the* 2^{nd} *USENIX Conference on File and Storage Technologies (FAST 2003)*, pages 203–216, San Francisco, CA, March-April 2003.

[33] Said S. Elnaffar. A methodology for auto-recognizing DBMS workloads. In *Proceedings of CASCON 2002*, pages 74–88, Toronto, ON, September 2002.

[34] Christos Faloutsos, Raymond T. Ng, and Timos K. Sellis. Flexible and adaptable buffer management techniques for database management systems. *IEEE Transactions on Computers (TC)*, 44(4):546–560, April 1995.

[35] *FreeBSD Developers' Handbook*. `http://www.freebsd.org/doc/en_US.ISO8859-1/books/developers-handbook/`.

[36] Kevin W. Froese and Richard B. Bunt. The effect of client caching on file server workloads. In *Proceedings of the* 29^{th} *Hawaii International Conference on System Sciences (HICSS 1996)*, pages 150–159, Kihei, HI, January 1996.

[37] Gregory R. Ganger, Marshall K. McKusick, Craig A. N. Soules, and Yale N. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems (TOCS)*, 18(2):127–153, 2000.

[38] Gregory R. Ganger, Bruce L. Worthington, and Yale N. Patt. *The DiskSim Simulation Environment Version 2.0 Reference Manual*, December 1999. `http://www.ece.cmu.edu/~ganger/disksim/`.

[39] Robert Geist and Stephen Daniel. A continuum of disk scheduling algorithms. *ACM Transactions on Computer Systems (TOCS)*, 5(1):77–92, 1987.

[40] Dominic Giampaolo. *Practical File System Design*. Morgan Kaufmann, 1999.

[41] Gideon Glass and Pei Cao. Adaptive page replacement based on memory reference behavior. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1997)*, pages 115–126, Seattle, WA, June 1997.

[42] Robert B. Gramacy, Manfred K.Warmuth, Scott A. Brandt, and Ismail Ari. Adaptive caching by refetching. In *15th Annual Conference on Neural Information Processing Systems (NIPS 2002)*, pages 1465–1472, Vancouver, BC, December 2002.

[43] Jim Gray and Prashant J. Shenoy. Rules of thumb in data engineering. In *Proceedings of the 16th International Conference on Data Engineering (ICDE 2000)*, pages 3–12, San Diego, CA, February – March 2000.

[44] John Linwood Griffin, Steven W. Schlosser, Gregory R. Ganger, and David Nagle. Modeling and performance of MEMS-based storage devices. In *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2000)*, pages 56–65, Santa Clara, CA, June 2000.

[45] John Linwood Griffin, Steven W. Schlosser, Gregory R. Ganger, and David Nagle. Operating system management of MEMS-based storage devices. In *Proceedings of the 4^{th} ACM Symposium on Operating System Design and Implementation (OSDI 2000)*, pages 227–242, San Diego, CA, October 2000. `http://www.usenix.org/events/osdi2000/griffin.html`.

[46] Laura M. Haas, Walter Chang, Guy M. Lohman, John McPherson, Paul F. Wilms, George Lapis, Bruce G. Lindsay, Hamid Pirahesh, Michael J. Carey, and Eugene J. Shekita. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2(1):143–160, March 1990.

[47] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, third edition, 2003.

[48] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference (USENIX Winter 1994)*, pages 235–246, San Francisco, CA, January 1994.

[49] Bo Hong, Scott A. Brandt, Darrell D. E. Long, Ethan L. Miller, Karen A. Glocer, and Zachary N. J. Peterson. Zone-based shortest positioning time first scheduling for MEMS-based storage devices. In *Proceedings of the 11th International Symposium on Modeling, Analysis and Simulation on Computer and Telecommunication Systems (MASCOTS 2003)*, pages 104–113, Orlando, FL, October 2003.

[50] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young. Projecting the performance of decision support workloads on systems with smart storage (SmartSTOR). In *Proceedings of the 2000 International Conference on Parallel and Distributed Systems (ICPADS 2000)*, pages 417–425, Iwate, Japan, July 2000. Also available as Technical Report CSD-99-1057, UC Berkeley, Berkeley, CA, August 1999.

[51] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young. Characteristics of production database workloads and the TPC benchmarks. *IBM Systems Journal*, 40(3):781–802, 2001.

[52] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young. I/O reference behavior of production database workloads and the TPC benchmarks – an analysis at the logical level. *ACM Transactions on Database Systems (TODS)*, 26(1):96–143, 2001.

[53] Yiming Hu and Qing Yang. DCD – disk caching disk: A new approach for boosting I/O performance. In *Proceedings of the* 23^{rd} *Annual International Symposium on Computer Architecture (ISCA 1996)*, pages 169–178, Philadelphia, PA, May 1996.

[54] Lan Huang and Tzi-cker Chiueh. Experiences in building a software-based SATF disk scheduler. Technical Report ECSL-TR81, State University of New York, Stony Brook, March 2000. Revised in July, 2001.

[55] Larry Huston, Rahul Sukthankar, Rajiv Wickremesinghe, M. Satyanarayanan, Gregory R. Ganger, Erik Riedel, and Anastassia Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *Proceedings of the* 3^{rd} *USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 73–86, San Francisco, CA, March-April 2004.

[56] David Jacobson and John Wilkes. Disk scheduling algorithms based on rotational position. Technical Report HPL-CSP-91-7, Hewlett-Packard Technical Report, February 1991.

[57] Bob Jenkins. Hash functions for hash table lookup. Dr. Dobb's, September 1997. Available at `http://burtleburtle.net/bob/hash/evahash.html`.

[58] The IBM JFS open source project. `http://oss.software.ibm.com/developerworks/opensource/jfs`.

[59] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2002)*, pages 31–42, Marina Del Rey, CA, 2002.

[60] Theodore Johnson and Dennis Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of* 20^{th} *International Conference on Very Large Data Bases (VLDB 1994)*, pages 439–450, Santiago de Chile, Chile, September 1994.

[61] John P. Kearns and Samuel DeFazio. Locality of reference in hierarchical database systems. *IEEE Transactions on Software Engineering*, SE-9(2):128–134, March 1983.

[62] John P. Kearns and Samuel DeFazio. Diversity in database reference behavior. In *Proceedings of the 1989 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1989)*, pages 11–19, Berkeley, CA, May 1989.

[63] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (IDISKs). *SIGMOD Record*, 27(3):42–52, 1998.

[64] Jong Min Kim, Jongmoo Choi, Jesung Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. Unified buffer management scheme that exploits sequential and looping references. In *Proceedings of the* 4^{th} *USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, CA, October 2000.

[65] Donald E. Knuth. *The Art of Computer Programming – Sorting and Searching*, volume 3. Addison-Wesley, second edition, 1998.

[66] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1999)*, pages 134–143, Atlanta, GA, May 1999.

[67] Sangdon Lee and Sukho Lee. Applying dynamic buffer allocation to predictive load control. In *Proceedings of the* 13^{th} *International Conference on Technology and Education (ICTE 1995)*, pages 150–155, Orlando, FL, March 1995.

[68] Fujian Liu, Yanping Zhao, Wenguang Wang, and Dwight Makaroff. Database server workload characterization in an e-commerce environment. In *Proceedings of the* 12^{th} *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2004)*, pages 475–483, Volendam, Netherlands, October 2004.

[69] David Lomet. The case for log structuring in database systems. In *Proceedings of the* 6^{th} *International Workshop on High Performance Transaction Systems (HPTS 1995)*, September 1995.

[70] Christopher R. Lumb, Jiri Schindler, and Gregory R. Ganger. Freeblock scheduling outside of disk firmware. In *Proceedings of the* 1^{st} *USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 275–288, Monterey, CA, January 2002.

[71] Christopher R. Lumb, Jiri Schindler, Gregory R. Ganger, David F. Nagle, and Eric Riedel. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In *Proceedings of the* 4^{th} *USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, CA, October 2000.

[72] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file systems with adaptive methods. In *Proceedings of the* 16^{th} *ACM Symposium on Operating Systems Principles (SOSP 1997)*, pages 238–251, Saint-Malo, France, October 1997.

[73] Marshall K. McKusick. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Longman, Reading, MA, 1996.

[74] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems (TOCS)*, 2(3):181–197, 1984.

[75] Larry W. McVoy and Steve R. Kleiman. Extent-like performance from a UNIX file system. In *Proceedings of the USENIX Winter 1991 Technical Conference (USENIX Winter 1991)*, pages 33–44, Dallas, TX, January 1991.

[76] Nimrod Megiddo and Dharmendra S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the* 2^{nd} *USENIX Conference on File and Storage Technologies (FAST 2003)*, pages 115–130, San Francisco, CA, March-April 2003.

[77] Jai Menon. A performance comparison of RAID-5 and log-structured arrays. In *Fourth IEEE Symposium on High-Performance Distributed Computing (HPDC 1995)*, pages 167–178, Charlottesville, VI, August 1995.

[78] Jai Menon and Larry Stockmeyer. An age-threshold algorithm for garbage collection in log-structured arrays and file systems. IBM Research Report RJ 10120, IBM Research Division, San Jose, CA, 1998.

[79] Alan Gilbert Merten. *Some quantitative techniques for file organization*. PhD thesis, Computer Sciences Department, University of Wisconsin, Madison, WI, 1970.

[80] MySQL 4.0.4 beta buffer pool replacement algorithm. `innobase/buf/buf0buf.c`. `http://www.mysql.com`.

[81] Network Appliance Inc. `http://www.netapp.com`.

[82] Victor F. Nicola, Asit Dan, and Daniel M. Dias. Analysis of the generalized clock buffer replacement scheme for database transaction processing. In *Proceedings of the 1992 ACM SIGMETRICS and PERFORMANCE 1992 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1992 / PERFORMANCE 1992)*, pages 35–46, Newport, RI, June 1992.

[83] Geoffrey J. Noer. Cygwin32: A free Win32 porting layer for UNIX applications. In *2nd USENIX Windows NT Symposium*, pages 31–38, Seattle, WA, August 1998.

[84] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD 1993)*, pages 297–306, Washington, DC, May 1993.

[85] John K. Ousterhout. The second critique of Seltzer's LFS measurements. `http://www.eecs.harvard.edu/~margo/usenix.195/ouster_critique2.html`.

[86] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the USENIX Summer 1994 Technical Conference (USENIX Summer 1994)*, pages 247–256, Anaheim, CA, June 1990.

[87] John K. Ousterhout, Hervè 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. In *Proceedings of the* 10^{th} *ACM Symposium on Operating Systems Principles (SOSP 1985)*, pages 15–24, 1985.

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

[89] David Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data (SIGMOD 1988)*, pages 109–116, Chicago, IL, June 1988.

[90] PostgreSQL. `http://www.postgresql.org`.

[91] PostgreSQL 7.2.2 buffer cache replacement algorithm. `postgresql-7.2.2-1/src/backend/storage/buffer/freelist.c`. `http://www.postgresql.org`.

[92] ReiserFS. `http://www.namesys.com/`.

[93] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1990)*, pages 134–142, Boulder, CO, May 1990.

[94] Drew Roselli, Jacob Lorch, and Thomas Anderson. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX 2000)*, pages 41–54, San Diego, CA, June 2000.

[95] Mendel Rosenblum. *The Design and Implementation of a Log-Structured File System*. PhD thesis, Department of Compute Science, University of California, Berkeley, June 1992. Also available as Technical Report UCB/CSD 92/696.

[96] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[97] Mark Russinovich. Inside NTFS. Windows 2000 Magazine, January 1998. http://www.winntmag.com/Articles/Index.cfm?ArticleID=3455.

[98] Giovanni Maria Sacco. Index access with a finite buffer. In *Proceedings of the* 13^{th} *International Conference on Very Large Data Bases (VLDB 1987)*, pages 301–309, Brighton, England, September 1987.

[99] Giovanni Maria Sacco and Mario Schkolnick. Buffer management in relational database systems. *ACM Transactions on Database Systems (TODS)*, 11(4):473–498, December 1986.

[100] Jiri Schindler and Gregory R. Ganger. Automated disk drive characterization. Technical Report CMU-CS-99-196, School of Computer Science, Carnegie Mellon University, December 1999.

[101] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger. Track-aligned extents: Matching access patterns to disk drive characteristics. In *Proceedings of the* 1^{st} *USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 259–274, Monterey, CA, January 2002.

[102] Jiri Schindler, Steven W. Schlosser, Minglong Shao, Anastassia Ailamaki, and Gregory R. Ganger. Atropos: A disk array volume manager for orchestrated use of disks. In *Proceedings of the* 3^{rd} *USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 159–172, San Francisco, CA, March-April 2004.

[103] Steven W. Schlosser and Gregory R. Ganger. MEMS-based storage devices and standard disk interfaces: A square peg in a round hole? In *Proceedings of the* 3^{rd} *USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 87–100, San Francisco, CA, March-April 2004.

[104] Frank Schmuck and Roger Haskin. GPFS: a shared-disk file system for large computing clusters. In *Proceedings of the* 1^{st} *USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 231–244, Monterey, CA, January 2002.

[105] Harald Schoening. The ADABAS buffer pool manager. In *Proceedings of the* 24^{th} *International Conference on Very Large Databases (VLDB 1998)*, pages 675–679, New York City, NY, August 1998.

[106] Margo Seltzer, Keith Bostic, Marshall K. McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of the USENIX Winter 1993 Technical Conference (USENIX Winter 1993)*, pages 307–326, San Diego, CA, January 1993.

[107] Margo Seltzer, Peter Chen, and John K. Ousterhout. Disk scheduling revisited. In *Proceedings of the USENIX Winter 1990 Technical Conference (USENIX Winter 1990)*, pages 313–324, Berkeley, CA, 1990.

[108] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File system logging versus clustering: A performance comparison. In *Proceedings of the 1995 USENIX Annual Technical Conference (USENIX 1995)*, pages 249–264, New Orleans, LA, January 1995.

[109] Chuck Silvers. UBC: An efficient unified I/O and memory caching subsystem for NetBSD. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX 2000), FREENIX Track*, pages 285–290, San Diego, CA, June 2000.

[110] Muthian Sivathanu, Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving storage system availability with D-GRAID. In *Proceedings of the* 3^{rd} *USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 15–30, San Francisco, CA, March-April 2004.

[111] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proceedings of the* 2^{nd} *USENIX Conference on File and Storage Technologies (FAST 2003)*, pages 73–88, San Francisco, CA, March-April 2003.

[112] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. EELRU: Simple and effective adaptive page replacement. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1999)*, pages 122–133, Atlanta, GA, May 1999.

[113] Storage performance council I/O traces. `http://traces.cs.umass.edu/storage`.

[114] The Sprite operating system. `http://www.cs.berkeley.edu/projects/sprite/sprite.html`.

[115] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, 4^{th} edition, 2000.

[116] Harold S. Stone, John Turek, and Joel L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers (TC)*, 41(9):1054–1068, 1992.

[117] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *Proceedings of the 1996 USENIX Annual Technical Conference (USENIX 1996)*, pages 1–14, San Diego, CA, January 1996.

[118] EMC Corporation. `http://www.emc.com`.

[119] James Z. Teng and Robert A. Gumaer. Managing IBM database 2 buffers to maximize performance. *IBM Systems Journal*, 23(2):211–218, 1984.

[120] Eno Thereska, Jiri Schindler, John Bucy, Brandon Salmon, Christopher R. Lumb, and Gregory R. Ganger. A framework for building unobtrusive disk maintenance applications. In *Proceedings of the* 3^{rd} *USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 213–226, San Francisco, CA, March-April 2004.

[121] Dominique Thiébaut and Harold S. Stone. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers (TC)*, 41(6):665–676, 1992.

[122] Ken Thompson. UNIX implementation. *Bell Systems Technical Journal*, 57(6):1931–1946, July-August 1978.

[123] Wenhu Tian, Pat Martin, and Wendy Powley. Techniques for automatically sizing multiple buffer pools in DB2. In *Proceedings of CASCON 2003*, pages 237–245, Toronto, ON, October 2003.

[124] Transaction processing performance council. `http://www.tpc.org/`.

[125] Java TPC-W implementation distribution. `http://www.ece.wisc.edu/~pharm/tpcw.shtml`.

[126] TPC-W results list. `http://www.tpc.org/tpcw/results/tpcw_results.asp`.

[127] Theodore Tso and Stephen Tweedie. Planned extensions to the Linux ext2/ext3 filesystem. In *Proceedings of the 2002 USENIX Annual Technical Conference (USENIX 2002), FREENIX Track*, pages 235–244, Monterey, CA, June 2002.

[128] Mustafa Uysal, Arif Merchant, and Guillermo A. Alvarez. Using MEMS-based storage in disk arrays. In *Proceedings of the* 2^{nd} *USENIX Conference on File and Storage Technologies (FAST 2003)*, pages 89–101, San Francisco, CA, March-April 2003.

[129] Werner Vogels. File system usage in Windows NT. In *Proceedings of the* 17^{th} *ACM Symposium on Operating Systems Principles (SOSP 1999)*, pages 93–109, Kiawah Island, SC, December 1999.

[130] Jun Wang and Yiming Hu. PROFS–performance-oriented data reorganization for log-structured file systems on multi-zone disks. In *Proceedings of the 9th International Symposium on Modeling, Analysis and Simulation on Computer and Telecommunication Systems (MASCOTS 2001)*, pages 285–292, Cincinnati, OH, August 2001. `http://www.ececs.uc.edu/~oscar/papers/PROFS.html`.

[131] Jun Wang and Yiming Hu. WOLF – a novel reordering write buffer to boost the performance of log-structured file systems. In *Proceedings of the* 1^{st} *USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 46–60, Monterey, CA, January 2002.

[132] Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Virtual log based file systems for a programmable disk. In *Proceedings of the* 3^{rd} *USENIX Symposium on Operating Systems Design and Implementation (OSDI 1999)*, pages 29–43, New Orleans, LA, February 1999.

[133] Wenguang Wang and Richard B. Bunt. Simulating DB2 buffer pool management. In *Proceedings of CASCON 2000*, pages 88–97, Toronto, ON, November 2000.

[134] Wenguang Wang and Richard B. Bunt. A self-tuning page cleaner for DB2. In *Proceedings of the* 10^{th} *International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2002)*, pages 81–89, Fort Worth, TX, October 2002.

[135] Wenguang Wang, Yanping Zhao, and Richard B. Bunt. HyLog: A high performance approach to managing disk layout. In *Proceedings of the* 3^{rd} *USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 145–158, San Francisco, CA, March-April 2004.

[136] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. In *Proceedings of the* 15^{th} *ACM Symposium on Operating Systems Principles (SOSP 1995)*, pages 96–108, Copper Mountain, CO, 1995.

[137] Darryl L. Willick, Derek L. Eager, and Richard B. Bunt. Disk cache replacement policies for network fileservers. In *Proceedings of the* 13^{th} *International Conference on Distributed Computing Systems(ICDCS 1993)*, pages 2–11, Pittsburgh, PA, May 1993.

[138] Theodore Wong and John Wilkes. My cache or yours? making storage more exclusive. In *Proceedings of the 2002 USENIX Annual Technical Conference (USENIX 2002)*, pages 161–175, Monterey, CA, June 2002.

[139] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. Scheduling algorithms for modern disk drives. In *Proceedings of the 1994 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1994)*, pages 241–251, Nashville, TN, 1994.

[140] Zhihui Zhang and Kanad Ghose. yFS: A journaling file system design for handling large data sets with reduced seeking. In *Proceedings of the* 2^{nd} *USENIX Conference on File and Storage Technologies (FAST 2003)*, pages 59–72, San Francisco, CA, March-April 2003.

[141] Yuanyuan Zhou, James F. Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the 2001 USENIX Annual Technical Conference (USENIX 2001)*, pages 91–104, Boston, MA, June 2001.

[142] Yingwu Zhu and Yiming Hu. Can large disk built-in caches really improve system performance? In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2002)*, pages 284–285, Marina Del Rey, CA, 2002.