Second-tier Cache Management to Support DBMS Workloads

by

Xuhui Li

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2011

© Xuhui Li 2011

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Enterprise Database Management Systems (DBMS) often run on computers with dedicated storage systems. Their data access requests need to go through two tiers of cache, i.e., a database bufferpool and a storage server cache, before reaching the storage media, e.g., disk platters. A tremendous amount of work has been done to improve the performance of the first-tier cache, i.e., the database bufferpool. However, the amount of work focusing on second-tier cache management to support DBMS workloads is comparably small. In this thesis we propose several novel techniques for managing second-tier caches to boost DBMS performance in terms of query throughput and query response time.

The main purpose of second-tier cache management is to reduce the I/O latency endured by database query executions. This goal can be achieved by minimizing the number of reads and writes issued from second-tier caches to storage devices. The first part of our research focuses on reducing the number of read I/Os issued by second-tier caches. We observe that DBMSs issue I/O requests for various reasons. The rationales behind these I/O requests provide useful information to second-tier caches because they can be used to estimate the temporal locality of the data blocks being requested. A second-tier cache can exploit this information when making replacement decisions. In this thesis we propose a technique to pass this information from DBMSs to second-tier caches and to use it in guiding cache replacements.

The second part of this thesis focuses on reducing the number of writes issued by second-tier caches. Our work is two fold. First, we observe that although there are second-tier caches within computer systems, today's DBMS cannot take full advantage of them. For example, most commercial DBMSs use *forced writes* to propagate bufferpool updates to permanent storage for data durability reasons. We notice that enforcing such a practice is more conservative than necessary. Some of the writes can be issued as *unforced* requests and can be cached in the second-tier cache without immediate synchronization. This will give the second-tier cache opportunities to cache and consolidate multiple writes into one request. However, unfortunately, the current POSIX compliant file system interfaces provided by mainstream operating systems (e.g., Unix and Windows) are not flexible enough to support such dynamic synchronization. We propose to extend such interfaces to let DBMSs take advantage of using *unforced writes* whenever possible.

Additionally, we observe that the existing cache replacement algorithms are designed solely to maximize read cache hits (i.e., to minimize read I/Os). The purpose is to minimize the read latency, which is on the critical path of query executions. We argue that minimizing read requests is not the only objective of cache replacement. When I/O bandwidth becomes a bottleneck the objective should be to minimize the total number of I/Os, including both reads and writes, to achieve the best performance. We propose to associate a new type of replacement cost, i.e., the total number of I/Os caused by the replacement, with each cache page; and we also present a partial characterization of an optimal algorithm which minimizes

the total number of I/Os generated by caches. Based on this knowledge, we extend several existing replacement algorithms, which are write-oblivious (focus only on reducing reads), to be write-aware and observe promising performance gains in the evaluations.

Acknowledgements

I would like to thank all the people who made this possible. At the very first I would like to thank my supervisor, Professor Kenneth Salem, for his many years of support, patience, advanced insights, thoughtful advice, and precise guidance. Without his constant support and help this work could not be accomplished.

I would also like to give many thanks to my thesis committee members, Professor Ashraf Aboulnaga and Professor Tim Brecht, for their continuous support during my study. Their important suggestions and comments greatly helped me in the in depth comprehension of my research topics and they have always gladly lent me a hand in solving the research problems. In addition, I would like to use this opportunity to appreciate Prfessor Sebastian Fischmeister and Professor Patrick Martin for being (respectively) the Internal-External member and the External Examiner of my thesis committee. I would also thank them in advance for their valuable comments and suggestions.

Mrs.Nicole Keshev, who is my English tutor and helped me with my thesis writing, made a tremendous amount of effort into this work. The thesis presented today would not be in its current shape without her help. I would like to give her a very special thanks and to show my appreciation.

IBM Canada supported me financially for years of my study and provided me with intern opportunities in their Toronto Lab. The first project of my research was initiated there with the help from Matt Huras, Aamer Sachedina, and Calisto Zuzarte. I would like to use this opportunity to say thank you to all these wonderful people and to IBM.

Dedication

This thesis is dedicated to my family who support me infinitely in this lengthy academic odyssey.

Contents

List of Tables x

List of Figures xii

1 Introduction 1

1.1 Motivation . . . 1

1.2 Contributions . . . 4

1.3 Thesis Organization . . . 5

2 Background 6

2.1 DBMS Bufferpool Management . . . 6

2.1.1 Bufferpool Management Architecture . . . 6

2.2 Bufferpool I/O Categorization . . . 10

3 Second-tier Cache Management Using Write Hints 12

3.1 Introduction . . . 12

3.2 Write Hints . . . 13

3.3 Managing the Storage Server Cache . . . 15

3.3.1 Using Hints for Cache Management . . . 15

3.3.2 LRU+Hints . . . 16

3.3.3 MQ+Hints . . . 17

3.3.4 The TQ Algorithm . . . 19

3.4 Evaluations . . . 21

3.4.1 Methodology . . . 22

3.4.2 Results: Baseline Case . . . 24

3.4.3 Results: Sensitivity Analysis . . . 25

3.5 Summary . . . 27

4 **Deferred Synchronization of Write Requests** **29**
4.1 Introduction . . . 29
4.2 Research Problem . . . 30
4.2.1 Data Integrity in Current Systems . . . 31
4.2.2 DBMS Bufferpool Write Synchronization Requirements . . . 34
4.3 Proposed New I/O Operation . . . 39
4.3.1 Introducing New I/O Operation . . . 39
4.3.2 Exploiting the New I/O Operation . . . 40
4.4 Evaluations . . . 44
4.4.1 Evaluations Using Synthetic Workloads . . . 44
4.4.2 Evaluations Using TPC-C Workload . . . 49
4.5 Summary . . . 52

5 **Write-Aware Replacement For Second-Tier Cache** **53**
5.1 Introduction . . . 53
5.2 Read-Write Workload Model and Cache Model . . . 55
5.2.1 Read-Write Workload Model . . . 55
5.2.2 Read-Write Cache Model . . . 56
5.3 Research Problems . . . 58
5.4 Partial Characterization of an Optimal Algorithm . . . 60
5.4.1 Preliminary Definitions . . . 61
5.4.2 Partial Characterization of an Optimal Algorithm . . . 65
5.5 Write-aware Replacement Algorithms . . . 67
5.5.1 A Write-aware Off-line Algorithm . . . 67
5.5.2 Write-aware On-line Algorithms . . . 68
5.6 Evaluations . . . 72
5.6.1 Evaluations by Using MySQL Bufferpool I/O Traces . . . 74
5.6.2 Evaluations by Using DB2 Bufferpool I/O Traces . . . 79
5.7 Summary . . . 85

6 **Related Work** **87**
6.1 Increasing the Read Cache Hit Rate . . . 87
6.2 Deferring Write Synchronization . . . 90
6.3 Reducing Total Number of I/Os . . . 92

7 Conclusions and Future Work **96**
7.1 Summary of Contributions 96
7.2 Future Work 98

Appendices **98**

A Proof of the Correctness of Lemma 5.1 **99**

B Proof of the Correctness of Lemma 5.2 **102**

C Proof of the Correctness of Lemma 5.3 **105**

D Proof of the Correctness of Theorem 5.1 **113**

E Proof of the Correctness of Lemma 5.5 **114**

References **117**

List of Tables

3.1 DB2 Parameter Settings 22
3.2 I/O Request Traces 22

4.1 TPC-C Workload, MySQL Database on Single Disk 50
4.2 TPC-C Workload, MySQL Database on RAID-5 Disk Array 51

5.1 MySQL Database Configuration Parameters 74
5.2 MySQL Bufferpool I/O Request Traces 75
5.3 MySQL Trace Analysis 77
5.4 DB2 Bufferpool I/O Request Traces 78
5.5 DB2 Trace Analysis 78
5.6 DB2 540K_4000 Trace Case, Decomposed I/O Numbers 84

List of Figures

1.1 A DBMS Running With Multiple Cache Tiers 2

2.1 Architecture of Bufferpool Management 7

3.1 Structures used by TQ. Arrows show possible movements between queues in response to cache requests 19

3.2 Read Hit Ratios in Storage Server Cache. Baseline (300_400) trace. DBMS bufferpool size is 300K pages (1.1 G bytes). Storage server cache size is 120K pages (469 MB). 24

3.3 Read Hit Ratios in the Storage Server Cache. Baseline (300_400) trace. DBMS bufferpool size is 300K blocks, storage server cache size varies from 60K blocks to 300K blocks. 25

3.4 Read Hit Ratios in the Storage Server Cache. Traces 60_400, 300_400, and 540_400. DBMS bufferpool size varies from 60K blocks (234 MB) to 540K blocks (2.1 GB). Storage server cache size is 120K blocks (469 MB). 26

3.5 Read Hit Ratios in the Storage Server Cache as `softmax` is varied. Traces 300_50, 300_400, and 300_4000. DBMS bufferpool size is 300K blocks (1.1 GB), storage server cache size is 120K blocks (469 MB). 27

4.1 Bufferpool REPLACE Write 35

4.2 Bufferpool RECOV Write 36

4.3 Another REPLACE Write 37

4.4 Example of Log Entries and Bufferpool Writes/Synchronizations . . 42

4.5 Synthetic Workload Program Issuing Writes 45

4.6 Synthetic Workload Program Issuing SQL UPDATE Queries, Sectors Physically written 47

4.7 Synthetic Workload Program Issuing SQL UPDATE Queries, Program Execution Time 48

4.8 Synthetic Workload Program Issuing SQL UPDATE Queries, I/O Service Time 49

5.1 States, Transitions, and I/Os of Cache Pages 58
5.2 Using the MIN-RW Algorithm, Total Number of I/Os: 3 59
5.3 Using Another Replacement Algorithm, Total Number of I/Os: 1 . 60
5.4 The Baseline Case, Vary Second-tier Cache Size 75
5.5 Sensitivity Analysis, Vary MySQL Bufferpool Size 76
5.6 The Baseline Case, Vary Second-tier Cache Size 81
5.7 The Baseline Case, Vary Second-tier Cache Size 82
5.8 Sensitivity Analysis, Vary Bufferpool Size 83
5.9 Sensitivity Analysis, Vary LSN Gap Threshold 85

A.1 States, Transitions, and I/Os of Cache Pages 100

B.1 States, Transitions, and I/Os of Cache Pages 103

C.1 Total I/O Costs of Page Replacements in Policy P and P' 106
C.2 Total I/O Costs of Page Replacements in Policy P and P' 107
C.3 Total I/O Costs of Page Replacements in Policy P and P' 108
C.4 Total I/O Costs of Page Replacements in Policy P and P' 108
C.5 Total I/O Costs of Page Replacements in Policy P and P' 109
C.6 Total I/O Costs of Page Replacements in Policy P and P' 109
C.7 Total I/O Costs of Page Replacements in Policy P and P' 110
C.8 Total I/O Costs of Page Replacements in Policy P and P' 110
C.9 Total I/O Costs of Page Replacements in Policy P and P' 111
C.10 Total I/O Costs of Page Replacements in Policy P and P' 111
C.11 Total I/O Costs of Page Replacements in Policy P and P' 112

E.1 Total I/O Costs of Page Replacements in Policy P and P' 115
E.2 Total I/O Costs of Page Replacements in Policy P and P' 115
E.3 Total I/O Costs of Page Replacements in Policy P and P' 116
E.4 Total I/O Costs of Page Replacements in Policy P and P' 116
E.5 Total I/O Costs of Page Replacements in Policy P and P' 117
E.6 Total I/O Costs of Page Replacements in Policy P and P' 118
E.7 Total I/O Costs of Page Replacements in Policy P and P' 118

Chapter 1

Introduction

Enterprise computers usually have dedicated, network connected storage systems. In such a computing environment there exist multiple tiers of cache. These cache tiers are designed to improve the input/output (I/O) efficiency of running applications and provide several benefits. First, cache hits generated in these cache tiers can save physical I/O operations on storage devices. For example, an input request, which we refer to as *read*, can be handled by the first cache tier where the requested data block is found. For another example, an output request, which we refer to as *write*, can stop at any of the cache tiers if it is an un-forced write request (a forced write requires the written content to be sent to persistent media immediately, where an unforced write does not have such a requirement). These I/O requests need not be propagated to permanent storage devices, whose operation is usually slower than data transfer between cache tiers. Second, the presence of multiple cache tiers provides file systems and storage systems with opportunities to do I/O scheduling. For example, in the Linux 2.6 kernel [1], the file pages buffered in the file system cache are always organized into batches during I/O operations if they have adjacent file offsets. By using these cache tiers, the running application can quickly resume work instead of waiting for I/O operations to be completed by the storage devices. However, before a database management system (DBMS) can take full advantage of them, some obstacles need to be addressed. In the next section we will give a more detailed discussion.

1.1 Motivation

In Figure 1.1 we illustrate an example in which a DBMS is running in a computer system containing two tiers of cache, namely a database bufferpool and a storage server cache. The DBMS manages the first-tier cache, i.e., the database bufferpool. It issues I/O requests to the second cache tier. The requests are issued for different reasons. For example, some read requests are issued because the data blocks being accessed by the database queries cannot be found in the bufferpool. For another example, some read requests are issued to prefetch data blocks that probably (based

on the DBMS's prediction) will be accessed in the near future. In this thesis we refer to the I/Os generated by the DBMS for the purpose of bufferpool management as *bufferpool I/Os*.

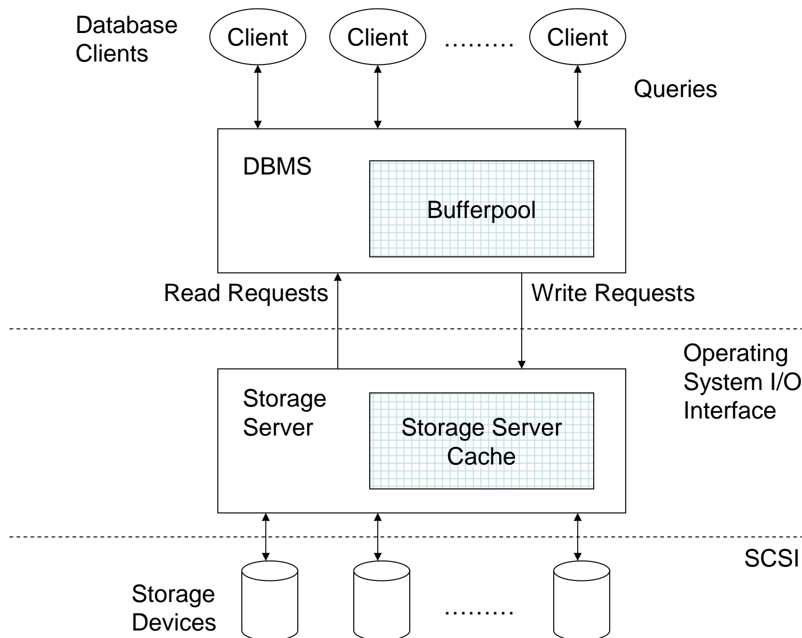


Figure 1.1: A DBMS Running With Multiple Cache Tiers

We observe that data blocks carried by bufferpool I/Os issued for different reasons have different access patterns in the second-tier cache. For example, the data blocks carried by one type of write request may be re-referenced sooner than the data blocks carried by another type of write request. This information about temporal locality is useful to the second-tier cache in helping it making replacement decisions. In this thesis we present a novel technique to identify different types of I/O requests and to pass this information to the second tier, where it can be used for cache management.

We also notice that different types of bufferpool writes have different synchronization requirements. In particular, some types of writes need to be synchronized immediately to permanent storage, while other types of writes do not need immediate synchronization. Their synchronization can be deferred to a later time, and the deferral gives second-tier cache opportunities to absorb and to consolidate write requests. Unfortunately, the current I/O interface provided by mainstream operating systems (OS), e.g., Unix and Windows, is not flexible enough to support such dynamic synchronization requirements. Consequently, today's DBMS systems pursue data durability by using forced requests for all bufferpool writes. We observe that this practice is too conservative. It prevents DBMS systems from exploiting

the optimization opportunities in the second-tier cache.

DBMS systems use dedicated threads to issue write requests asynchronously with respect to query execution. This technique can effectively hide write latency when the I/O bandwidth is not heavily utilized. In such a circumstance the write requests are not on the critical path of query executions. On the contrary, the on-demand read requests are typically synchronous with respect to query executions. Therefore, in today's second-tier cache management, the objective is to maximize read cache hits and to reduce read latency. We observe that the above scenario does not always prove to be true in a computer system. The storage devices can be saturated and can become the performance bottleneck. When this happens, the write requests, although they are issued asynchronously, will compete with read requests for I/O bandwidth. Both read and write requests become synchronous with respect to query executions. Consequently, maximizing read cache hits no longer leads to optimal performance. We argue that in such a scenario the objective of cache replacement should be minimizing the total number of I/Os, including both reads and writes, issued by the cache.

In this thesis we focus on second-tier cache management to support DBMS workloads. Enterprise DBMS servers usually have dedicated storage systems attached to them. The DBMS can be seen as a client of the storage server. This architecture includes two tiers of cache, i.e., the DBMS bufferpool and the storage server cache. Managing a second-tier cache is different from managing a first-tier cache and is more difficult in the following ways:

- Because first-tier cache management is usually based on recency and frequency, it captures the application's accesses to the hot pages. Consequently, the temporal locality of page accesses is weak and difficult to track in the the second-tier cache. Therefore, the cache replacement algorithms based purely on recency, e.g., the Least Recently Used (LRU) algorithm, usually underperform in second-tier cache management.

- Since there are two tiers of cache serving for one data set (e.g., a database), a single page may have duplicate copies in both cache tiers. This redundancy hurts the performance of running applications. Due to the lack of information or collaboration, it is difficult for a second-tier cache to exclude duplicated pages and to save cache space.

- A first-tier cache is usually located in (or close to) an application's (e.g., a DBMS) own memory space and is sometimes managed by the application itself. It is easier for the first-tier cache to know or to infer the application's data access patterns. For example, based on a query's execution plan, the DBMS can predict which pages will soon be accessed. However, this type of information is not easily available to second-tier cache management.

These difficulties encountered by second-tier cache management have been recognized in the literature [69, 29, 20, 110, 22]. Various second-tier cache man-

agement techniques have been proposed by researchers. For example, some researchers suggest the use of a unified, centralized, manager for both cache tiers [81, 16, 18, 96, 47, 38]. For another example, some work suggests the use of hints, such as the likelihood that the pages will be evicted, from the first-tier cache to the second-tier cache [88, 90, 91, 11, 22, 87].

These proposed techniques may create burdens on different cache tiers. For instance, some techniques need modifications to be made to the first-tier cache so that it can manage the second-tier cache as well. Other techniques may need the second-tier cache to spend extra CPU time and I/O bandwidth to infer the content or the management of the first-tier cache. With these observations in mind, our approaches proposed in this thesis are light-weight (in terms of resource consumption) and easy to implement.

1.2 Contributions

Our research focuses on second-tier cache management to support DBMS workloads. We explore the topic from two perspectives. Our first goal is to increase read hits (i.e., to reduce read I/Os) in the second-tier cache. We observe that a DBMS issues bufferpool writes for different reasons and this information can be used to improve the read hit ratio in the second-tier cache. We propose a novel technique to pass the information as write hints from a DBMS to a second-tier cache. We also propose techniques to use these hints in cache management. The proposed techniques are evaluated by cache simulations and the results demonstrate significant improvement of read hit ratios.

Our second goal is to reduce the number of write I/Os issued by second-tier caches. Our approach consists of two parts. First, we notice that different types of bufferpool writes have different synchronization requirements. The existing I/O interface provided by mainstream operating systems is not flexible enough to support such dynamic requirements. Consequently, DBMS systems always issue forced bufferpool writes to ensure data durability. We argue that this write synchronization practice is more conservative than necessary and can waste I/O bandwidth. We propose to extend the current I/O interface to support finer-grained and more flexible synchronizations. We implement the proposed I/O interface extension in an open source operating system and modify a DBMS to use it.

With the introduction of the unforced bufferpool writes, second-tier caches now face a different workload and need to re-think their replacement decisions. Typically, second-tier cache replacement algorithms are designed to achieve a sole objective, which is to maximize the read cache hits, because all the cached pages are clean pages and the write requests issued by DBMSs are asynchronous with respect to query executions. However, with the introduction of unforced bufferpool writes, a second-tier cache may contain both clean and dirty pages. Additionally, the write requests become synchronous when the I/O bandwidth is saturated and becomes

a performance bottleneck. We argue that under such a circumstance the objective of cache replacement should be to minimize the total number of I/Os, including both reads and writes. We propose novel techniques that can be used in such a scenario. Simulation based evaluations show that by using the proposed techniques the number of total I/Os issued by a second-tier cache can be reduced significantly.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. In Chapter 2 we introduce background knowledge of DBMS bufferpool management. In Chapter 3 we discuss how write hints can be passed to second-tier caches and can be used in cache management. In chapter 4 we observe that different types of bufferpool writes have different synchronization requirements. Also in this chapter we propose a novel technique to let a DBMS take advantage of deferred synchronization of bufferpool writes. In Chapter 5 we present work to minimize the total number of I/Os generated by second-tier caches. We give a survey of related work in Chapter 6 and in Chapter 7 we conclude and discuss possible extensions to the work presented in this thesis.

Chapter 2

Background

In this chapter we introduce some background about DBMS bufferpool management. Our research in this thesis focuses on second-tier cache management to support DBMS workloads. The work presented in different chapters is based on the behavior of the DBMS bufferpool manager. To fully understand the approaches proposed in this thesis, the following background is needed as preliminary knowledge.

2.1 DBMS Bufferpool Management

Bufferpool management is one of the key components of a DBMS. It provides a data access service to running queries and affects both query throughput and query response time. In this thesis we refer to the DBMS component working on bufferpool management as the *bufferpool manager*. In bufferpool management the database objects, e.g., tables and indexes, are organized into data blocks, which we refer to as *pages* in this thesis. One page may contain one or multiple table records or index entries. A page is also the smallest unit carried by an I/O request. Due to space limitations and other reasons, a bufferpool manager needs to issue I/O requests to the lower tier of the storage hierarchy. I/O efficiency is important to the performance of the bufferpool and to the performance of the DBMS as a whole.

2.1.1 Bufferpool Management Architecture

We illustrate the architecture and the key components of a bufferpool manager in Figure 2.1. Although this example does not represent all DBMSs, it is consistent with some mainstream commercial and open source systems, e.g., DB2 [42], Oracle [77], SQL Server [65], and MySQL [76].

There are three types of threads involved in bufferpool management. The first type is database *agents*. Agents are threads working on both query executions and

bufferpool management. A pool of agent threads is initiated when the database starts. Agent threads wait to serve queries from database clients. Multiple agents can work concurrently. While a query is being executed, the agents executing it may need to access database objects, e.g., table records and index entries. Depending on the queries, the agents may also need to update the objects. To access data objects, an agent makes data access requests to the bufferpool manager. The bufferpool manager then locates the bufferpool pages containing the requested data objects and returns the pages' memory addresses to the agent.

If a requested page is not found in the bufferpool, then the agent must issue a read request to bring it from the storage. While this I/O operation is in progress, the agent will be blocked by the operating system and will wait for I/O completion. It is worth noting that the agent is now working on bufferpool management (instead of query execution) and that the read request is synchronous with respect to the query execution. After the page is brought into the bufferpool, it will be pinned and will remain there until the query execution is completed. We refer to a page whose content has not been updated by any agent since it was last read into the bufferpool (or since the page has been written to lower level storage) as a *clean* page. On the contrary, a page whose content has been updated by an agent and the update has not been written to storage is referred to as a *dirty* page.

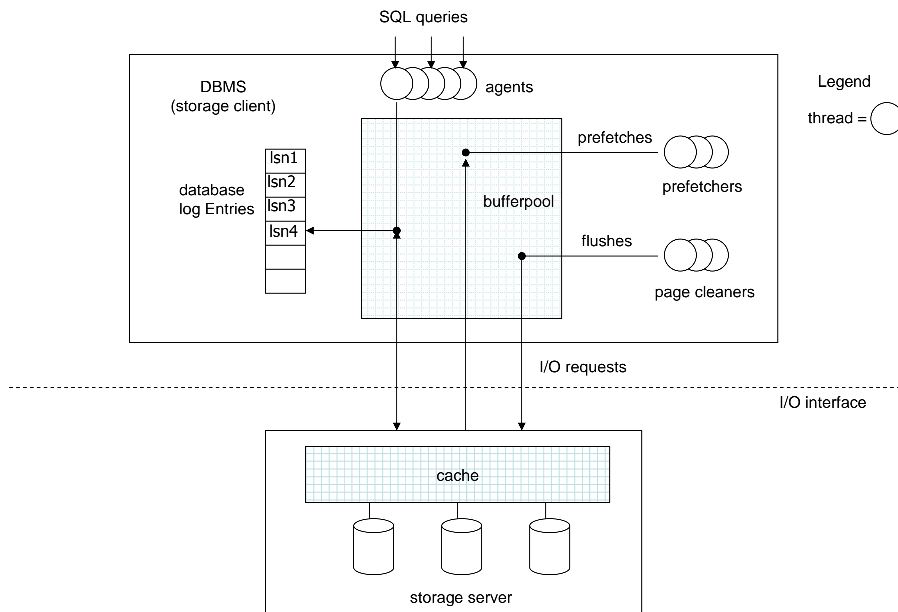


Figure 2.1: Architecture of Bufferpool Management

The second type of thread involved in bufferpool management is prefetcher threads. Under certain circumstances, usually based on a query's execution plan, a

DBMS's query optimization component can predict which data objects are likely to be accessed in the near future. It issues data prefetching requests to the bufferpool manager. Upon receiving these requests, the bufferpool manager puts them into a waiting list, called a *prefetching list*. The requests will wait for their turn to be served. A group of dedicated threads, named *prefetchers*, will take read requests from the prefetching list and will bring the requested data into the bufferpool in advance of their use. This prefetching technique is designed to reduce the latency caused by read I/Os and thus to speed up the query execution.

There is a third type of thread, *page cleaners*, involved in bufferpool management. Page cleaners are dedicated to cleaning dirty pages. Page cleaning includes two phases of work:

- First, determine which dirty pages should be written to disks and organize them into a list, named the *flush list*
- Second, issue write requests to the operating system to write the pages on the flush list to persistent storage

Page cleaning can be triggered for two different reasons. The first reason is for page replacement. A database bufferpool has a limited size and it is usually smaller than the database. When the bufferpool is full, to bring a new page into it, an existing page must be evicted to make room for the new page. If the victim page is dirty, then before its eviction the dirty page must be written to persistent storage to preserve data durability. In this thesis we refer to the write requests issued by the bufferpool manager for this reason as *REPLACE* writes. When making replacement decisions, i.e., when deciding which page should be evicted, most DBMSs rely on the recency or frequency (or both) of page references. Different replacement algorithms, such as LRU and Clock, are used by various DBMSs.

The second reason to trigger page cleaning is to limit database recovery time. Many DBMS systems adopt the ARIES algorithm [67] to manage crash recovery. To comply with the ARIES algorithm, the Write Ahead Logging (WAL) protocol [10, 30] is widely implemented to preserve data durability. Each update made to a bufferpool page by an agent will be recorded in a log entry and will be assigned a Log Sequence Number (LSN). The LSN numbers are increased monotonically by the DBMS and the successive updates will see larger LSNs. The log entries are written into log files and are forced to disks when the transaction which performed those updates is committed. In case of a system crash and recovery, the log entries of committed transactions are safe on disks and can be used to bring the database back to a consistent state.

This WAL logging technique also relieves the bufferpool manager from the burden of synchronizing each page update to disks. Because the DBMS's log management component takes the responsibility to ensure that the updates made by committed transactions can survive failures, a bufferpool page can absorb multiple updates without being written to disk.

However, there are limitations on the number of updates that can be kept in a bufferpool without synchronization. The first limitation is the duration of recovery. The more updates that are kept in the bufferpool without synchronization, the more log entries need to be replayed during a recovery. Some DBMS applications cannot tolerate a lengthy recovery time. Therefore, the number of unsynchronized updates that can be kept within the bufferpool must be limited.

Log space limitation is another reason to limit the number of unsynchronized updates in a bufferpool. As the database runs, the amount of disk space consumed by log entries increases. Since the amount of disk space that can be assigned to log files is limited, the DBMS must recycle and reuse the space occupied by the log entries. Before a log entry's space can be reused, the corresponding bufferpool update must be synchronized to disks. Otherwise a failure may cause some unsynchronized updates to be irrecoverably lost because the space used by their log entries has been reused by other updates.

The above limitations determine that a DBMS must write updates to disks if the amount of unsynchronized updates exceeds a certain threshold. Usually the threshold is a configurable database parameter which imposes an upper bound on the gap between the following two LSN numbers:

- the LSN number assigned to the newest update made to the bufferpool
- the LSN number assigned to the oldest update that has not been synchronized to disk

The difference between the two LSN numbers is referred to as the *LSN Gap*. Once the LSN Gap threshold is reached, the bufferpool manager will trigger a page cleaning session and write the dirty pages containing the oldest updates to disks. In this thesis we refer to the bufferpool writes issued for recovery reasons as *RECOV* writes. LSN Gap threshold is a key DBMS parameter and should be configured carefully. On one hand, if the threshold is set to be larger than the desirable value then the recovery time, or the log space limitation, guaranteed to the users cannot be met. On the other hand, if the parameter is set be smaller than the desirable value then the bufferpool manager will trigger more page cleaning sessions than necessary and will waste I/O bandwidth.

Page cleaning, no matter when it is triggered, is performed by the bufferpool manager asynchronously with respect to query executions. The bufferpool manager continuously monitors the percentage of dirty pages in the bufferpool (for replacement reasons) and the LSN Gap (for recovery reasons). The dirty pages whose states triggered page cleaning will be put into the flush list and wait for their turn to be written to disks.

A bufferpool manager maintains another auxiliary data structure, named the *free list*, for page replacement. The free list keeps a number of clean pages as replacement candidates. The pages are selected by the bufferpool manager based

on their past references and the cache replacement algorithm. If a selected page is dirty, then it will be put into the flush list to be written to disk. After the dirty page is written and becomes clean it will be put into the free list. When an agent issues a read request to bring a new page into the bufferpool, it will use the space occupied by a clean page in the free list.

Although the bufferpool manager tries its best to keep enough clean pages in the free list, in some circumstances it may not catch up with the consumption by the agents. For example, when a storage device is busy and becomes a bottleneck, the page cleaners may get stuck in the I/O traffic and therefore unable to provide enough clean pages into the free list. In such cases an agent that needs a page space but cannot find one in the free list may need to flush a dirty page by itself. This is not an ideal scenario in bufferpool management because now the write request becomes synchronous with respect to the query execution. We refer to the writes issued synchronously by agents themselves as *SYNCH writes*.

2.2 Bufferpool I/O Categorization

In this section we define some fundamental concepts that are related to DBMS bufferpool I/Os and that will be used frequently in the following parts of this thesis. As discussed in the previous section, bufferpool I/Os can be categorized into two different types based on how their operations are performed with respect to query executions:

- Synchronous I/O: The I/Os whose operation is synchronous with respect to the query executions. The DBMS agents, who are working on query executions, must wait (e.g., be blocked by the OS) for such I/O operations to complete before continuing with their work. It is worth noting that the I/O completion provided by the OS can have different meanings and effects. For example, when the OS returns a write completion to the DBMS, it can have two different meanings:

  - The written content has been copied from the DBMS's user space to the OS's kernel space, but has not been propagated to disk
  - The written content has been copied from the DBMS's user space to the OS's kernel space and has been propagate to disk (and is persistent)

  The SYNCH bufferpool writes and the bufferpool reads issued by agents themselves, are typically synchronous I/Os.

- Asynchronous I/O: The I/Os whose operation is asynchronous with respect to query executions. The DBMS agents do not need to wait for these I/O operations to be completed before continuing with their query execution work. The REPLACE and RECOV bufferpool writes are typically implemented as asynchronous I/Os.

In addition to the above categorization, the bufferpool writes can also be categorized into two types based on their synchronization methods:

- Synchronized Write: The writes for which operation completion guarantees that the written data are persistent on storage media and can survive failures such as system crashes. For data durability reasons, some mainstream DBMS, such as DB2 and Oracle, implement all bufferpool writes as synchronized writes.
- Unsynchronized Write: The writes for which operation completion does not guarantee that the written data are persistent on the storage media. The written data may be propagated to underlying cache tiers that are volatile and nonpersistent.

Please note that the above two bufferpool I/O categorizations are orthogonal. A synchronous write can be unsynchronized and, similarly, an asynchronous write can be a synchronized write.

Please also note that these terms (Synchronous, Synchronized, Asynchronous, and Unsynchronized) are similar therefore they may cause confusion under some circumstances. The reason for risking confusion and using them in this thesis is that these particular terms, and their semantic interpretations, have been widely accepted and used in the research and documentation of DBMS and OS (as we will see in the following parts of this thesis).

Chapter 3

Second-tier Cache Management Using Write Hints

In this chapter we present our work on improving the read hit rate in second-tier caches by using write type hints from first-tier caches.

3.1 Introduction

In Figure 2.1 we illustrate an example of two cache tiers coexisting in a computer system. The storage server manages its own cache (i.e., the second-tier cache) and provides storage service to the DBMS. In Chapter 2 we showed that the DBMS bufferpool manager issues write requests for different reasons and that based on these reasons the writes can be categorized into different types. We argue that this type information can be used by second-tier caches to predict future access patterns of the written pages.

In this chapter we are proposing a technique to improve the read hit rate in second-tier caches. It requires cooperation between the two cache tiers. The first-tier cache identifies the reason for which it issues each write request, and then pass the reason as a hint to the second-tier cache. The second-tier cache exploits the hints to manage its buffer and improve its read hit rate. To use our approach the DBMS needs to be modified. However, the changes are simple, because the DBMS has full knowledge of the reasons why it issues each write request. The only changes involved are to categorize these writes into different types and to attach a hint to each write request.

We focus our study on a common scenario in which the first-tier cache is a database bufferpool managed by the DBMS. However, this is not the only circumstance where our approach can be used. Other systems with their own managed caches, such as web servers or file systems, can potentially take advantage of our techniques.

In our research, we modify two existing hint-oblivious cache replacement algorithms, namely LRU and MQ [111], to be hint-aware and we observe some significant performance gains in the resulting read hit rates. In addition, we present a new, primarily hint-based replacement algorithm, *Type Queue (TQ)*, which can perform twice as well as MQ.

Compared with existing second-tier cache management techniques, our approach has the following advantages:

- It is simple to implement at the DBMS and the storage server. As we have discussed above, the DBMS needs to be modified to categorize the writes into different types based on the reasons for which the writes are issued and to attach a hint to each write. The storage server can then use a hint-aware algorithm for cache replacement and does not need to simulate or to track the content of the DBMS cache.
- It is purely opportunistic and does not require extra I/O bandwidth or device operations. The hints are only a few bits in size and can be passed as a parameter along with the write request to the second-tier cache. In addition, because the data block carried by a write must be transfered from the DBMS to the storage server, the second-tier cache does not need to fetch the block from the device if it decides to cache it. If the cache decides not to cache the block and to immediately propagate it to the device, it still consumes no extra bandwidth because the block would eventually have been flushed to the device anyways.
- Because read requests are primarily absorbed by the first-tier cache, there is a higher percentage of writes among the second-tier cache accesses. This provides more opportunities for the second-tier cache to exploit write hints.

The remainder of this chapter is organized as follows. In Section 3.2 we explain how we categorize DBMS bufferpool writes into different types and how we use type hints in second-tier cache management. In Section 3.3 we present three hint-aware replacement algorithms. We show our evaluation methodology and experimental results in Section 3.4, and conclude in Section 3.5.

3.2 Write Hints

In Chapter 2 we showed that a DBMS bufferpool manager issues write requests for three different reasons. We propose to categorize the bufferpool writes into three different types based on these reasons. The write types and the information that can be inferred from them are:

- **SYNCH writes**: A SYNCH write is a write issued (synchronously with respect to the query execution) by an agent in a situation in which the agent

cannot find a clean page in the *free list* to replace, so that it must write out a dirty page by itself to make room for a new page. It is certain that the page being written will be evicted from the bufferpool immediately after the write completion, and that the DBMS's next request for this page will be a read.

- **REPLACE writes**: A REPLACE write is a write issued asynchronously by a *page cleaner* thread for cache replacement reasons. The page being written is selected based on the bufferpool's replacement algorithm. After the page is written, it will be placed into the *free-list* and it is likely to be evicted in the near future. However, because an on-line replacement algorithm's predictions cannot be perfect, the page may be accessed again (by the agents) before its eviction and may be removed from the free list. For the second-tier cache, the page's next request is *probably* a read. The probability is based on the accuracy of the first-tier cache replacement algorithm's predictions.

- **RECOV writes**: A RECOV write is a write issued asynchronously by a page cleaner for recovery reasons. The page is selected to be written because it contains an update which is old enough to cause the bufferpool's LSN Gap to reach its threshold. Depending on the LSN Gap threshold, which is a configurable database parameter, the pages being accessed by RECOV writes may have remained in the bufferpool for various periods of time. If the threshold is set to a large value, then usually the pages carried by RECOV writes have been in the bufferpool for a long time (compared with other bufferpool pages), and usually these pages are *hot* pages because otherwise they would have been evicted and would not be able to contain old updates. On the other hand, if the threshold is set to a small value, then every page, regardless of its access pattern, can contain an update old enough to trigger a page cleaning session. In this work we focus on the databases with *typical* configurations (i.e., without extremely small LSN Gap threshold) and, therefore, a RECOV write indicates that the carried page will probably stay in the first-tier cache. Thus, this page is unlikely to be read soon by the DBMS.

From the above descriptions we can see that the types of different writes can be used to infer future access patterns of the written pages. This type information can be useful to second-tier cache management.

Although the write types described above are categorized based on a certain type of bufferpool manager implemented by DBMSs (e.g., DB2 [42], Oracle [77], SQL Server [65], and MySQL [76]), similar write types can also be found in the cache management of other software systems, such as file systems [66, 8]. Our write type categorization method can potentially also be used in those systems.

We propose to pass write types as hints to the storage server. In our study, we modify the DBMS so that it tags each write request with one of the following three types of hints: SYNCH, REPLACE, or RECOV. The required modifications in the DBMS are simple and natural. For example, we easily instrumented DB2 Universal

Database to label each write with one of these three types of hints. Because the tags add only a few bits per write request, the overhead on the I/O bandwidth is negligible. It is worth noting that when the DBMS categorizes its write types, it does not need to understand how the storage server will use the categorizations. It is up to the storage server to decide whether and how to use the write hints for cache management.

3.3 Managing the Storage Server Cache

In this section we propose techniques to use write hints for storage server cache management. We extend two existing cache replacement algorithms, LRU and MQ, to be hint-aware and to take advantage of the write hints. We also propose a primarily hint-based algorithm, TQ, for cache replacement.

3.3.1 Using Hints for Cache Management

The goal of second-tier cache management is to maximize the cache read hit rate. However, as we have introduced in Chapter 2, managing a second-tier cache is more difficult than managing a first-tier cache. For example, as some researchers have pointed out [69, 29, 20, 110, 22], temporal locality is more difficult to exploit at the second-tier cache. In this section we argue that the following two factors should be taken into account by the second-tier cache manager:

- **data exclusivity**: the storage server needs to cache pages that are not in the DBMS bufferpool, and to evict pages that are already in the DBMS bufferpool. The DBMS does not issue read requests for a page already in its cache, so caching such pages in the second-tier cache is pointless. Similar observations have been made by other researchers [104, 23, 107, 34].
- **type of the next page reference**: only the *read* cache hits matter to the storage server. If a page's next reference is a write then it should not be cached because doing so can only harm the cache performance (by consuming cache space) and results in no benefit. If there is a read request following the write, then the page can still be cached at the time when it is written. In reality, a page's next reference type is unknown to the on-line replacement algorithms. However, the algorithms can use the write type hints passed from the first tier to infer this information.

In the scenario that we study, where the storage client is a DBMS, these two factors are closely related. The DBMS will only issue *read* requests, not *write* requests, for pages that are not in its bufferpool; and it will only issue *write* requests, not *read* requests, for pages that are already in its bufferpool. We argue that the

bufferpool write type hints, such as those introduced in the previous section, can help storage servers to maintain data exclusivity.

Now we discuss in detail how different write hints can help the storage server. Let us consider the SYNCH writes first. When a DBMS issues a SYNCH write, we are sure that the page being written will be evicted from the DBMS's bufferpool immediately after the write is completed. Caching the page in the second-tier cache will not impair data exclusiveness. We also know that the next reference of the page will be a read. We consider a SYNCH write to be a positive hint, which means we should keep the page being written in the storage server cache.

From a REPLACE write, the storage server can infer that the page is being written by the DBMS for replacement reasons. It is probable (depending on the accuracy of the bufferpool replacement algorithm's predictions) that the page will be evicted from the bufferpool in the near future, to make room for a new page. If the eviction occurs, then the page's next reference will be a read. We consider a REPLACE write to be a positive hint, which means we should keep the written page in the storage server cache.

As for a RECOV write, the written page probably has been in the DBMS's bufferpool for a period of time so that it contains an update old enough to trigger page cleaning for recovery reasons. As we have discussed earlier, depending on the LSN Gap threshold configuration, that period of time may vary. However, since we focus on databases with *typical* configurations, usually the pages will not be requested again to the storage server cache soon after they are written by the RECOV writes. In our work we treat RECOV writes as negative hints, which means we should not keep the written pages in the storage server cache.

We can also infer useful information from read requests issued by DBMSs. A read request is issued by the DBMS because the requested page is not in its bufferpool. Once the page is brought into the bufferpool, it may be cached there for a period of time, the length of which depends on a number of factors considered by the DBMS. However, we can make the following observations:

- If the page is a *hot* page in the DBMS bufferpool, then probably it will stay there and will not be read again in the near future. It is of no benefit for the storage server to cache it.
- If the page is not *hot* in the DBMS bufferpool, then probably it also will not be read again in the near future

In this work we consider a read request as a negative type hint to the storage server cache.

3.3.2 LRU+Hints

We extended the existing LRU cache replacement algorithm, which is hint-oblivious, to use type hints in managing the LRU list. Our extension is simple: we cache the

pages carried by SYNCH and REPLACE writes because they are positive hints and the pages being accessed are likely to be evicted from the DBMS bufferpool in the near future. We do not cache READ or RECOV pages because we treat these two request types as negative hints and the pages are not likely to be read soon.

More specifically, if page p is accessed by a SYNCH or REPLACE request, then we add p into the cache if it is not there, and we move p to the MRU end of the LRU list. If the cache is full then the page at the LRU end will be replaced. On the contrary, if page p is accessed by a READ or RECOV request, we do not change the cache content (by serving the requested page directly from the storage device to the DBMS), except if this happens during a cold start. When the cache is cold, we will cache the page but will put it at the LRU end of the LRU list. The hint-aware LRU algorithm, *LRU+Hint*, is summarized in Algorithm 3.1.

Algorithm 3.1 LRU+Hints

```
LRUWithHints(p : page access)
 1  if the cache is full
 2     then if p is already in the cache /* cache hit */
 3             then if type(p) = Synch or type(p) = Replace
 4                     then move p to the MRU end of the LRU list;
 5                     else  do not change the cache content; /* this is a READ or RECOV */
 6
 7             else  /* cache miss */
 8                   if type(p) = Synch or type(p) = Replace
 9                      then evict the page at the LRU end of the LRU queue;
10                           append p to the MRU end of the LRU queue;
11                      else  do not change the cache content; /* this is a READ or RECOV */
12
13
14     else  /* the cache is not full */
15           if p is already in the cache /* cache hit */
16             then if type(p) = Synch or type(p) = Replace
17                     then move p to the MRU end of the LRU queue;
18                     else  do not change the cache content; /* this is a READ or RECOV */
19
20             else  /* cache miss */
21                   if type(p) = Synch or type(p) = Replace
22                      then append p to the MRU end of the LRU queue;
23                      else  append p to the LRU end of the LRU queue; /* this is a READ or RECOV */
24
25
26
```

3.3.3 MQ+Hints

The Multi-Queue (MQ) algorithm [111] was designed and proposed specifically for second-tier cache management. It takes both recency and frequency into account while making replacement decisions. The authors claim that MQ can outperform purely recency-based or purely frequency-based replacement algorithms, such as LRU or LFU. The MQ algorithm uses multiple LRU queues and each LRU queue contains a group of pages with similar reference frequencies. A page is promoted to the LRU queue with higher reference frequencies if it is referenced more frequently.

A page that has not been referenced within a certain period of time will be demoted to a LRU queue with a lower frequency range. When a replacement is required, the page at the LRU end of the lowest-frequency queue will be replaced. To help track the weak temporal locality in second-tier caches, the MQ algorithm also uses an *out queue* to record the reference frequencies of the pages evicted from the cache.

We extend the MQ algorithm to be hint-aware in a way similar to our extension of the LRU algorithm. We consider the SYNCH and REPLACE write types as positive hints and will treat their carried pages exactly as they would be treated under the original MQ algorithm. If the request is a READ or RECOV, then we do not change the cache except during cold start, where we treat it the same way as it would be treated in the original MQ. Of course, if a READ request is a cache hit then the page will be served from the cache and will be returned to the DBMS.

It is worth noting that to use MQ algorithm a few parameters need to be preconfigured by the user. The parameters include:

- The number of LRU queues in the algorithm. Each LRU queue corresponds to a frequency range and will contain the pages whose access frequencies fall into the range. In our evaluations we set the number of LRU queues to 80.
- The frequency range corresponding to each LRU queue. If a page is accessed more frequently and its access frequency reaches a threshold, it will be promoted to the LRU queue with the higher frequency range. In our evaluations we set each of the 80 LRU queues to correspond to only one access frequency (with the bottom LRU queue corresponding to frequency 1 and the top LRU queue corresponding to frequency 80 and above).
- The *expireTime* of each queue. If a page in a LRU queue is not accessed for a period of time (i.e. the expireTime, in terms of page accesses), it will be demoted to the LRU queue with the lower frequency range. In our evaluations the expireTime is set to be 20 millions of page accesses, which is longer than the workload length.
- The size of the *out queue*, which keeps statistics for the pages evicted from the cache. We set the out queue size to be the same as the cache size in our evaluations. At maximum, it keeps statistics about as many additional pages as the cache does.

Before our evaluations, we ran some cache simulation trials to select the parameters for MQ algorithm. The parameters were varied in different runs and the settings which gave the best performance were chosen to run the reported evaluations. To make a fair comparison, the MQ+Hints algorithm uses the same parameter settings.

3.3.4 The TQ Algorithm

In addition to the hint-aware algorithms introduced above, we also propose a new algorithm that relies primarily on request types to make replacement decisions. We refer to this algorithm as the *Type Queue (TQ)* algorithm. Among our hint-aware algorithms, TQ places the most emphasis on using request types (or hints) for replacement. Our evaluation results presented in Section 3.4 show that TQ outperforms other algorithms in most cases. The algorithm is summarized in Figure 3.1 and Algorithm 3.2.

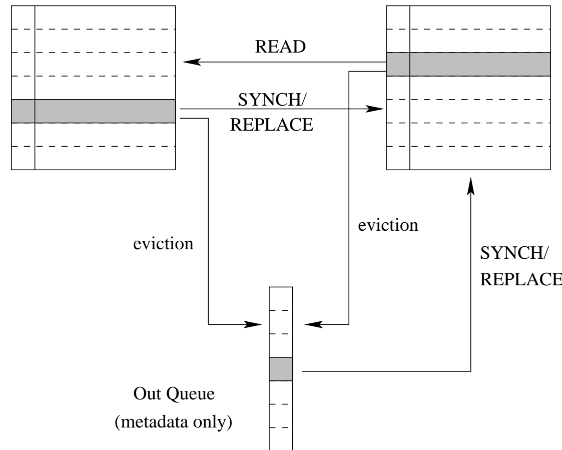


Figure 3.1: Structures used by TQ. Arrows show possible movements between queues in response to cache requests

In the TQ algorithm we organize cached pages into two queues. A *high priority queue* contains pages whose most recent reference is a SYNCH or REPLACE write. These two write types are considered as positive hints because the carried pages are likely to be evicted by the DBMS bufferpool in the near future. A *low priority queue* contains pages whose last reference is a READ. We ignore the RECOV requests except during a cold start, when the page is placed into the low priority queue if it is not in the cache. We treat READ and RECOV hints differently in TQ because we find that the RECOV writes do not provide negative hints as strongly as the READ requests do. This observation is especially true in the cases when the LSN Gap threshold (a database parameter) is set to small values, which means that every page, regardless of its access pattern, can be written by a RECOV write for recovery reasons.

If there is a SYNCH or a REPLACE request and the page being accessed is in the low priority queue, then it will be moved to the high priority queue. Similarly, if there is a READ request and the page is in the high priority queue, it will be moved to the low priority queue. Thus the sizes of the two queues can vary from time to time. In case of a replacement the algorithm will try to evict a page from the low priority queue unless it is empty, in which case the eviction will happen from the high priority queue.

Algorithm 3.2 The TQ Algorithm

```
TQAccess(b : block access)
1  /* for the sake of simplicity, this assumes that the cache and the out queue are already full */
2  if type(b) = Read
3     then if b is in Q_high /* b is in high priority queue */
4             then move b to the MRU end of Q_low; /* move b to low priority queue */
5          /* if this Read follows a Synch or Replace, update write-to-read distance */
6          if b is in cache or Q_out and ∞ > lastWritePosition(b) > 0
7             then update avgWriteReadDist(b) using (currentPosition − lastWritePosition(b));
8                  lastWritePosition(b) = ∞;
9  elseif type(b) = Synch or type(b) = Replace
10    then if b is in Q_low
11            then nextReadPosition(b) = currentPosition + avgWriteReadDist(b);
12                 move b to Q_high; /* move b to high priority queue */
13                 lastWritePosition(b) = currentPosition; /* remember when this write happened */
14         elseif b in in Q_out and lastWritePosition = ∞
15            then nextReadPosition(b) = currentPosition + avgWriteReadDist(b);
16                 move victim from cache to Q_out;
17                 /* victim is LRU in Q_low, or latest nextReadPosition in Q_high if Q_low is empty */
18                 move b to Q_high; /* put b into high priority queue */
19                 lastWritePosition(b) = currentPosition; /* remember when this write happened */
20         elseif b is not in cache and b is not in Q_out
21            then remove Q_out entry with largest avgWriteReadDist;
22                 move victim from cache to Q_out;
23                 /* victim is LRU in Q_low, or latest nextReadPosition in Q_high if Q_low is empty */
24                 put b into Q_high; /* put b into high priority queue */
25                 lastWritePosition(b) = currentPosition; /* remember when this write happened */
26                 avgWriteReadDist(b) = ∞;
27                 nextReadPosition(b) = ∞;
```

The high priority queue is managed based on a replacement algorithm that we refer to as the *Latest Predicted Read (LPR)*. When a page p is placed into the high priority queue, the TQ algorithm makes a prediction of the time, *nextReadPosition(p)*, that the page will next be READ. The pages in the high priority queue are sorted based on their *nextReadPosition* values. In case the replacement happens in the high priority queue, the page with the largest *nextReadPosition()*, which means it is predicted to be read again furthest in the future, will be evicted. Several variables used in the LPR and in Algorithm 3.2 are shown as follows:

- *lastWritePosition(p)*: The most recent time (in terms of page accesses) when page p was written. The initial value of this variable is infinity.
- *avgWriteReadDist(p)*: The average distance (in terms of page accesses) between a write of page p and the next following read of p. If there are multiple writes before a read then only the first write is counted. The value of this variable is calculated based on the page's access history, including *lastWritePosition(p)*. The initial value of this variable is infinity.
- *nextReadPosition(p)*: The predicted (by using the LPR) time of the next *read* of page p. It is the sum of *lastWritePosition(p)* and *avgWriteReadDist(p)*. The initial value is infinity.
- *currentPosition*: The current time (in terms of page accesses).

The LPR is similar to the off-line optimal (MIN) algorithm. However, as it is an on-line algorithm, it relies on the observation of past, not the knowledge of the future, to make the page access predictions. The TQ algorithm maintains a running average value, *avgWriteReadDist()*, for each page in the cache. The value is the average distance, in terms of page accesses, between a REPLACE or SYNCH write and the next READ request to the same page. To calculate the running average *avgWriteReadDist()*, TQ also needs to keep the time of the most recent SYNC or REPLACE request for each page. In case of a READ request, TQ will update the value of the average *avgWriteReadDist()* and reset the time of the last SYNC or REPLACE to infinity, which means it will wait for the next reference of these request types to re-start another round of calculation.

The pages in the low priority queue are most recently referenced by READ requests, which means two of their statistics, *lastWritePosition()* and *nextReadPosition()*, have been reset. The low priority queue is managed by using the LRU algorithm, not LPR. Like the MQ algorithm, TQ also maintains an auxiliary data structure to keep the statistics for the pages that were evicted from the cache. We refer to this data structure as the *out queue*. When a page is evicted from the cache, its statistics will be saved into the out queue. If the out queue is full, a replacement is required. The pages' statistics in out queue are sorted based on their *avgWriteReadDist* values. The entry with the largest *avgWriteReadDist()* will be replaced. When a page is accessed and is not cached, TQ will first search the out queue for the page. If it is found there then the stored statistics will be used to calculate the page's *nextReadPosition()*, which is the sum of *currentPosition()* and *avgWriteReadDist()*.

It worth noting that the TQ algorithm lacks an aging mechanism for the pages in the high priority queue. More specifically, if a page is associated with a relatively small *nextReadPosition()* attribute, then the page may stay in cache for a lengthy period even though the current time has passed the page's predicted *nextReadPosition()* time. The lack of an aging mechanism may result in an undesired situation in which the pages that will never be *read* again may remain in cache forever. This shortcoming should be addressed and be solved in our future work.

3.4 Evaluations

We use trace-driven simulations to evaluate the performance of the cache management techniques described in Section 3.3. The goal of our evaluation is to determine whether the hint-aware algorithms outperform (in terms of read cache hits) the hint-oblivious algorithms. An on-line random replacement algorithm is also evaluated to provide a point of comparison. We also study the performance of an off-line optimal cache replacement algorithm to determine how much room remains for improvement.

Parameter	Our Default Value	Other Values	Description
bufferpool size	300000 4KB blocks	60000, 540000 blocks	size of the DBMS bufferpool
softmax	400	50, 4000	recovery effort threshold
chngpgs_thresh	50%	-	bufferpool dirtiness threshold
maxagents	1000	-	maximum number of agent threads
num_iocleaners	50	-	number of page cleaner threads

Table 3.1: DB2 Parameter Settings

Trace Name	Bufferpool Size in blocks	softmax	Number of Requests	SYNCH Writes	REPLACE Writes	RECOV Writes	Reads
300_400	300K (1.1 GB)	400	13269706	0.00%	62.57%	3.60%	33.83%
60_400	60K (234 MB)	400	15792519	0.08%	48.89%	0.18%	50.85%
540_400	540K (2.1 GB)	400	12238848	0.00%	35.78%	49.89%	14.33%
300_4000	300K (1.1 GB)	4000	13226138	0.01%	65.37%	0.11%	34.51%
300_50	300K (1.1 GB)	50	15175377	0.00%	0.03%	74.33%	25.64%

Table 3.2: I/O Request Traces

3.4.1 Methodology

In our evaluation we use DB2 Universal Database (version 8.2) as the storage client. We instrumented DB2 so that it would record traces of its I/O requests. We also modified DB2 to associate an appropriate type hint with each I/O request. Each record in the I/O request trace describes an I/O and its associated hint.

To collect I/O traces, we drive the instrumented DB2 with a TPC-C [101] workload, with a scale factor of 25. The initial size of the database, including all tables and indexes, is 606,317 4KB pages, or approximately 2.3 GB. The database grows slowly during the trace collection. The I/O request trace generated by DB2 depends on the settings of a variety of database parameters. Table 3.1 shows the settings for the most significant parameters. We study DB2 bufferpools ranging from 10% to 90% of the (initial) database size. The softmax and chngpgs_thresh parameters are important because they control the mix of write types in the request stream. The chngpgs_thresh determines the percentage of bufferpool pages that must be dirty to trigger the page cleaners to generate REPLACE writes to clean them. The softmax parameter defines the threshold of LSN Gap. Larger values of softmax allow longer recovery times and result in fewer RECOV writes. By setting chngpgs_thresh at 50% (near DB2's default value) and varying softmax, we are able to control the mix of REPLACE and RECOV writes generated by the page cleaners. In addition, we set the agent thread pool size to be 1000, which is large enough to meet the requirements of the running transactions. The number of page cleaner threads is set to be 50, which is also adequate for the bufferpool to perform page cleaning jobs and to issue REPLACE and RECOV write requests.

In Table 3.2 we summarize the traces collected and used in our evaluation. The 300_400 trace is the baseline trace, collected by using the default DB2 parameters.

The remaining traces are collected using alternative bufferpool sizes and `softmax` settings. Not surprisingly, increasing the size of the DB2 bufferpool decreases the percentage of READ requests in the trace, because more data accesses can be served by the DB2 bufferpool. Large bufferpools also tend to increase the percentage of RECOV writes, since fewer REPLACE writes are needed for cache replacement. As discussed above, smaller values of `softmax` increase the prevalence of RECOV writes. The 300_50 trace represents a fairly extreme scenario with a low `softmax` setting. This causes DB2 to issue a RECOV write soon after a page has been updated, so that recovery will be extremely fast. Although these settings are unlikely to be used in practice, we include this trace for the sake of completeness.

We use these traces to drive simulations of a storage server cache and evaluate the algorithms described in Section 3.3. We also implement a variation of the off-line MIN algorithm, which we refer to as OPT, to establish an upper bound on the read hit ratio in the storage server's cache. Suppose that a storage server cache with capacity C receives a request for page p, the OPT algorithm will work as follows:

- If the cache is not full, then put p into the cache.
- If the cache is full and p is in cache, then leave the cache contents unchanged.
- If the cache is full and p is not in cache, then among all the pages, including the C pages in the cache and p, eliminate the one that will not be *read* for the longest time and keep the C remaining pages in the cache.

The OPT algorithm may choose *not* to buffer p at all if the pages currently in cache will be *read* sooner. The on-line random replacement algorithm, which we refer to as RAND, behaves the same as the OPT algorithm when the cache is not full or when the requested page p is already in cache. However, when the RAND algorithm needs to replace a victim page it randomly chooses one cached page to evict.

For the MQ, MQ+Hints, and TQ algorithms, we set the maximum size of their out queues to be equal to the size of the cache being simulated. Thus, each of these algorithms tracks statistics for the pages that are currently buffered, plus an equal number of pages that have been previously evicted. We subtracted the space required for the out queue from the available cache space for each of these algorithms so that the comparisons with other algorithms that do not require an out queue would be on level ground.

On each simulation run, we first allow the storage server's cache to warm up. Once the cache is warmed up, we then measure the *read hit ratio* for the cache being simulated. The ratio is the percentage of read requests that can be served by the cache.

To ensure the integrity of our evaluations, we conduct the following sanity checks throughout our evaluations presented in this thesis (including the implementations

of the simulation programs and the modifications of the operating system and the database systems):

- Using assertions in the code. We insert assertions in the code to make sure that the programs behave as expected.
- Additional statistics. We design and implement the experiments to output more statistics in addition to those required by the evaluations. These additional statistics are collected to help us understand whether the programs behave as expected. For example, in the simulation of TQ algorithm we also collect the sizes (in maximum and at the end of simulation) and the read hits of the two priority queues to verify whether they are consistent with the reported results.

By using the above techniques, we are confident that the evaluation results presented in this thesis are integral and trustworthy.

3.4.2 Results: Baseline Case

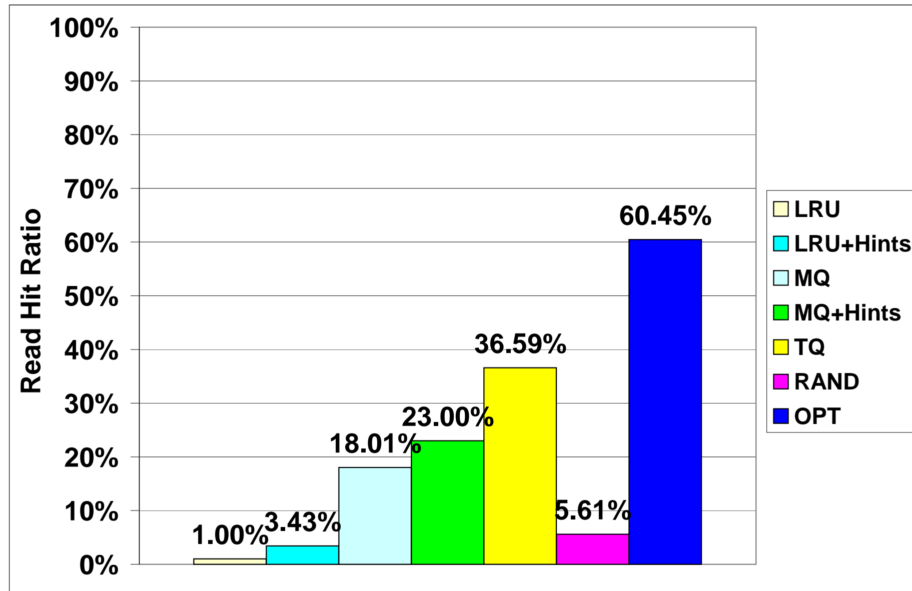


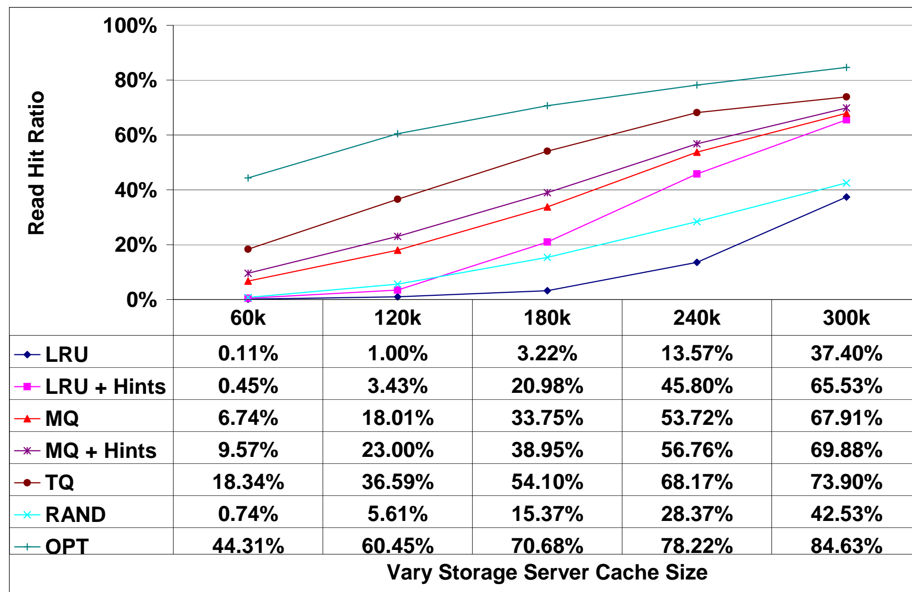
Figure 3.2: Read Hit Ratios in Storage Server Cache. Baseline (300_400) trace. DBMS bufferpool size is 300K pages (1.1 G bytes). Storage server cache size is 120K pages (469 MB).

Figure 3.2 shows the read hit ratios of the simulated storage server cache under each of the techniques discussed in Section 3.3, for the baseline 300_400 trace and a storage server cache size of 120K pages (469 MB). The results show that the LRU

algorithm has the poorest performance, which is consistent with other previous evaluations of LRU in second-tier caches [69, 29, 110, 22]. The LRU+Hints algorithm, which takes advantage of write hints, results in a hit ratio more than three times that of LRU, but it is still low compared with other algorithms, including the RAND. The MQ algorithm, which considers frequency as well as recency, performs significantly better than LRU and RAND, and MQ+Hints further improves the performance. The primarily hint-based TQ algorithm provides the best performance, with a hit ratio nearly double that of MQ. TQ achieves more than half of the hit ratio of the off-line OPT algorithm.

3.4.3 Results: Sensitivity Analysis

We evaluate the sensitivity of the baseline results in Figure 3.2 to changes in three significant parameters: the size of the storage server cache, the size of the DBMS bufferpool, and the value of the `softmax` parameter, which controls the mix of write types among the I/O requests.

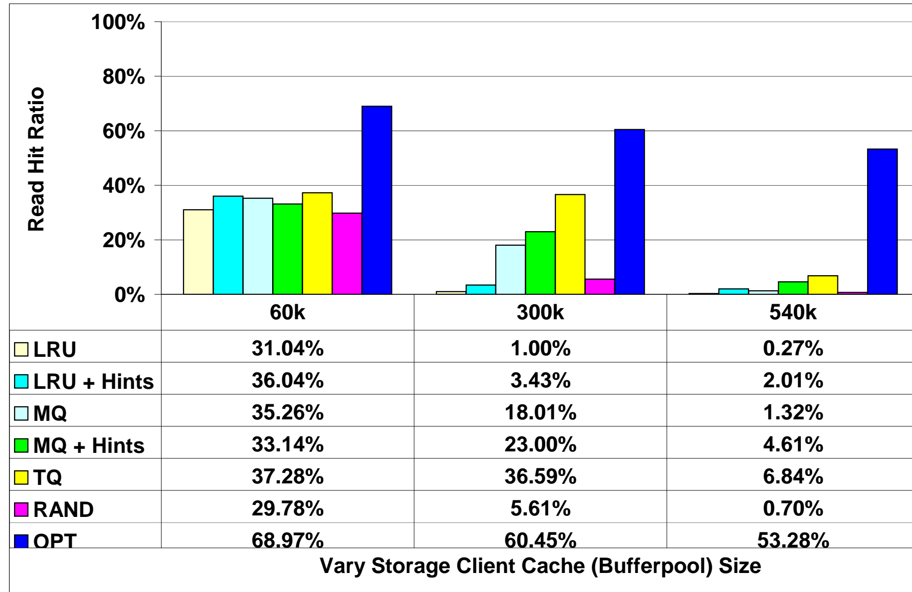


	60k	120k	180k	240k	300k
LRU	0.11%	1.00%	3.22%	13.57%	37.40%
LRU + Hints	0.45%	3.43%	20.98%	45.80%	65.53%
MQ	6.74%	18.01%	33.75%	53.72%	67.91%
MQ + Hints	9.57%	23.00%	38.95%	56.76%	69.88%
TQ	18.34%	36.59%	54.10%	68.17%	73.90%
RAND	0.74%	5.61%	15.37%	28.37%	42.53%
OPT	44.31%	60.45%	70.68%	78.22%	84.63%

Figure 3.3: Read Hit Ratios in the Storage Server Cache. Baseline (300_400) trace. DBMS bufferpool size is 300K blocks, storage server cache size varies from 60K blocks to 300K blocks.

Figure 3.3 shows the read hit ratios of the storage server cache as its size varies from 60K pages (234 MB) to 300K blocks (1.1 GB), which is the size of the first-tier cache in the baseline case. Several observations can be made from these results. First, the relative advantage of the TQ algorithm persists until the server's cache reaches the largest size (300K blocks, 1.1 GB), at which point the advantage of TQ begins to diminish. For this large second-tier cache size case, the improvement

obtained by adding hints to MQ also becomes negligible. However, for large cache sizes the performance of the simple LRU+Hints algorithm is much better than that of the plain LRU and RAND, and is comparable to that of TQ and the MQ policies. As the storage server cache decreases, the performance of LRU+Hints (and plain LRU) drops off quickly.



	60k	300k	540k
LRU	31.04%	1.00%	0.27%
LRU + Hints	36.04%	3.43%	2.01%
MQ	35.26%	18.01%	1.32%
MQ + Hints	33.14%	23.00%	4.61%
TQ	37.28%	36.59%	6.84%
RAND	29.78%	5.61%	0.70%
OPT	68.97%	60.45%	53.28%

Figure 3.4: Read Hit Ratios in the Storage Server Cache. Traces 60_400, 300_400, and 540_400. DBMS bufferpool size varies from 60K blocks (234 MB) to 540K blocks (2.1 GB). Storage server cache size is 120K blocks (469 MB).

Figure 3.4 illustrates the impact of changing the DBMS bufferpool size, with the storage server cache size fixed at 120K pages (469 MB). These results show that management of the storage server cache becomes more difficult as the DBMS bufferpool becomes larger. A large DBMS bufferpool absorbs most of the temporal locality available in the request stream, leaving little for the storage server cache to exploit. Larger DBMS bufferpools also make it more difficult to maintain exclusiveness between the client and server caches. For very large DBMS bufferpools, the TQ algorithm performs more than five times better than the best hint-oblivious algorithm. However, *all* of the algorithms, including TQ, have poor performance in absolute terms, with read hit ratios far below that of the off-line OPT algorithm. When the DBMS bufferpool is very small (60K pages), all of the algorithms provide similar performance. This is because the temporal locality in the workload is relatively good at the the storage server cache (due to the small DBMS bufferpool size). Some of the cached pages will be *read* again even though the RAND algorithm is used to manage the cache.

Finally, Figure 3.5 shows the server cache read hit ratios as the `softmax` parameter increases from 50 to 4000. When the `softmax` parameter is set to be large

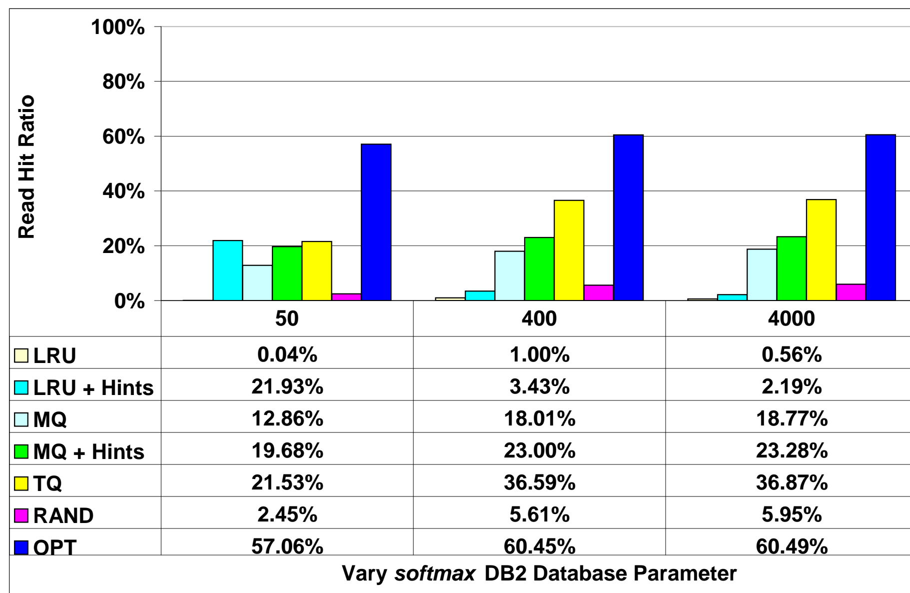


Figure 3.5: Read Hit Ratios in the Storage Server Cache as `softmax` is varied. Traces 300_50, 300_400, and 300_4000. DBMS bufferpool size is 300K blocks (1.1 GB), storage server cache size is 120K blocks (469 MB).

(4000), the database administrator states that a long recovery times is acceptable. Under those conditions (trace 300_4000), the DBMS generates almost no RECOV writes; this is the primary difference between the baseline 300_400 trace and the 300_4000 trace. This difference has little impact on the performance of any of the algorithms.

When the `softmax` is set to be 50, all of the hint-aware algorithms have similar performance, which is better than that of MQ and much better than LRU and RAND. When `softmax` is 50, almost three quarters of the I/O requests are RECOV writes, and there are no REPLACE writes. As we discussed earlier, this represents an extreme scenario in which bufferpool updates are flushed to the storage server almost immediately. As a result, this `softmax` setting generally gives poor overall system performance because the RECOV write consumes substantial I/O bandwidth, and so it is rarely used in practice.

3.5 Summary

We observe that information about different types of writes issued by a storage client can be useful to the storage server. We propose to pass these write types as hints to the storage server and to use them in cache management. We focus on a common scenario in which the storage client is a DBMS, which operates its own bufferpool (first-tier cache) and issues I/O requests to a dedicated storage server

managing its own cache (second-tier cache). We categorize DBMS bufferpool writes into three different types, namely SYNCH, REPLACE, and RECOV, and conduct an analysis of the reasons for which these types of writes are issued. We find that the write types indicate the future access patterns of the bufferpool pages being written.

To use the write type hints, we propose to extend two hint-oblivious replacement algorithms, LRU and MQ, to be hint-aware. We also propose a new, primarily hint-based algorithm, Type Queue, for second-tier cache management. Trace-driven simulations show that the hint-aware algorithms significantly outperform their original hint-oblivious counterparts in terms of read hit ratios. In most of our evaluations, the new algorithm, Type Queue, performs best among the on-line algorithms studied. During our evaluations, we also find that the relative performance of the algorithms is sensitive to a few system configuration parameters, such as the DBMS bufferpool size and the LSN Gap threshold (e.g., the *softmax* parameter in DB2 configuration). When the DBMS bufferpool size is set to be small, or when the RECOV writes dominate the I/O trace, the benefits of using write hints are limited.

We argue that although using our approach requires minor modifications to the DBMS, it is simple and natural to implement. The DBMS typically has full knowledge of the reasons why it issues write requests. This fact provides us with good opportunities to categorize the write requests into different types. In addition, we observe that the overhead put on the CPU time and the I/O bandwidth is negligible. The hints consist of only a few bits to be carried by each write request.

Chapter 4

Deferred Synchronization of Write Requests

In Chapter 3 we propose techniques to improve second-tier cache performance by reducing the number of read requests issued to the storage devices (i.e., by increasing the read cache hit rate). In this chapter we propose a technique to reduce the number of write requests issued by second-tier caches.

4.1 Introduction

As previously discussed, modern computer systems usually contain multiple tiers of cache lying between applications and storage devices. For example, the file system may have its own managed cache. At a lower level, the storage system may also maintain its own cache. From an application's point of view, these cache tiers lie beneath its user level memory space. These multiple cache tiers are designed to facilitate data input/output operations, and they provide several benefits. First, hits in these caches save on expensive physical I/O operations performed by the storage devices, e.g., the disks. Second, the presence of these caches provides file systems and storage systems with opportunities to do I/O scheduling. By these means, these caches improve I/O efficiency and hence system performance.

Although the existence of multiple cache tiers can boost performance, using them improperly can jeopardize data integrity. One major reason is that multiple copies of one piece of data may coexist in different cache tiers and the content of those copies can be inconsistent. Each I/O request made by applications must go through these cache tiers sequentially and sometimes asynchronously with regard to the application's execution. As a result, the copies of a single piece of data may be inconsistent in different cache tiers. This inconsistency may ruin data integrity during accidents, e.g., system crashes or power failures, because some cache tiers may reside in volatile storage media, e.g., DRAM, and their content cannot persist across such failures.

In this study we focus on write synchronization. Applications, e.g., DBMSs, issue write requests to update their data stored on storage devices. The data carried by a write request is propagated through the cache tiers below the application's user space. The update may not be made to the bottom of the storage hierarchy, i.e, the storage devices. Even worse, the application itself may not be aware of the progress of its write synchronizations. To some applications, e.g., DBMS systems, this uncertainty is unacceptable. We focus our study on a common scenario where the application is a DBMS. The DBMS manages a bufferpool as the first-tier cache, and the DBMS issues write requests to a lower level system, such as a file system or a storage system, which operates its own cache as the second-tier cache.

We make several contributions in this research work. First, we observe that different types of DBMS bufferpool writes have different synchronization requirements. For example, RECOV writes need to be synchronized immediately to permanent storage devices, while synchronizations of REPLACE writes can be deferred to a later time. The deferral of REPLACE write synchronizations can give the second-tier cache opportunities to consolidate several write requests into one physical write and to do better I/O scheduling. Second, we observe that the current I/O interface provided by mainstream operating systems, e.g., Linux and Windows, is not flexible enough to support these dynamic write synchronization requirements. Consequently, DBMSs choose to trade off I/O efficiency for data integrity. Third, based on the above observations we propose to extend the existing I/O interface to include a novel I/O operation, which provides applications with a means to synchronize specific data blocks to storage devices at any time deemed appropriate. Fourth, to evaluate our approach we implement the proposed new I/O operation in an open source operating system, namely Linux. We also modify an open source DBMS, MySQL, to use the new operation. Experimental results show that by using our techniques the DBMS can increase the transaction throughput by up to 25 percent for an Online Transaction Processing (OLTP) workload.

The remainder of this chapter is organized as follows. In Section 4.2 we give a more detailed discussion about the problems that we find in the current DBMS bufferpool write synchronization scheme. In Section 4.3 we propose our approach to solve the problems identified in the previous section. We evaluate the proposed techniques in Section 4.4 and conclude in Section 4.5.

4.2 Research Problem

In this section we give a more detailed discussion about what the synchronization requirements are for different types of DBMS bufferpool writes, and we explain why the existing I/O interface does not satisfy these requirements. First, we begin our discussion with the data integrity issue in computer systems with multiple cache tiers.

4.2.1 Data Integrity in Current Systems

The *Portable Operating System Interface for Unix (POSIX)* [41] is a set of related standards specified by the *Institute of Electrical and Electronics Engineers (IEEE)* to define and regulate the Application Programming Interface (API) between applications and Unix (and Unix-like) operating systems. The POSIX standards take I/O synchronization into account and try to provide applications with some mechanisms (through the APIs) to preserve data integrity. The following concepts are defined in the Base Definitions volume of the current POSIX standards (IEEE Std 1003.1, 2004 Edition):

- 3.373 Synchronized Input and Output: A determinism and robustness improvement mechanism to enhance the data input and output mechanisms, so that an application can ensure that the data being manipulated is physically present on secondary mass storage devices.

- 3.374 Synchronized I/O Completion: The state of an I/O operation that has either been successfully transferred or diagnosed as unsuccessful.

- 3.375 Synchronized I/O Data Integrity Completion: For read, when the operation has been completed or diagnosed if unsuccessful. The read is complete only when an image of the data has been successfully transferred to the requesting process. If there were any pending write requests affecting the data to be read at the time that the synchronized read operation was requested, these write requests are successfully transferred prior to reading the data. For write, when the operation has been completed or diagnosed if unsuccessful. The write is complete only when the data specified in the write request is successfully transferred and all file system information required to retrieve the data is successfully transferred. File attributes that are not necessary for data retrieval (access time, modification time, status change time) need not be successfully transferred prior to returning to the calling process.

- 3.376 Synchronized I/O File Integrity Completion: Identical to a synchronized I/O data integrity completion with the addition that all file attributes relative to the I/O operation (including access time, modification time, status change time) are successfully transferred prior to returning to the calling process.

- 3.377 Synchronized I/O Operation: An I/O operation performed on a file that provides the application assurance of the integrity of its data and files.

Definitions 3.373 and 3.377 require the OS implementation to include an I/O mechanism to ensure applications that the manipulated data should be present in persistent storage. These requirements help to preserve data integrity in the presence of caches during failures, such as system crashes or power failures.

Furthermore, in definitions 3.374 , 3.375, and 3.376 POSIX provides hints on how the related mechanism can be implemented, i.e., through I/O completion notification. By the time the application gets notice from the OS about the I/O completion, the data carried by the I/O should have been successfully *transfered*. This is an adequate requirement for read I/O as the term *transfered* means that the application has already acquired the data in its address space. However, we find that this term is ambiguous for write I/Os. In the presence of cache tiers below the application's user space, the meaning of the term, *transfered*, becomes critical. To guarantee the requirements stated in definition 3.373 and 3.377, the word *transfered* in definition 3.374 through 3.375 should be interpreted to mean *transfered to persistent storage media*.

The POSIX System Interfaces also defines two mechanisms that can be used by applications to ensure data integrity:

- File open flags: If a file is opened with some specific flags, i.e., O_SYNC and O_DSYNC, then for each I/O operation on the opened file, the I/O completion is guaranteed to indicate:
  - Synchronized I/O File Integrity Completion, if the file is opened with the O_SYNC flag.
  - Synchronized I/O Data Integrity Completion, if the file is opened with the O_DSYNC flag.
- Explicit system calls: An application can explicitly synchronize a file to disk by using system calls, e.g., *fsync()* and *fdatasync()*. By the time it returns, a fsync() system call guarantees Synchronized I/O File Integrity Completion for all previously issued I/O requests on that file. Similarly, the return of a fdatasync() system call guarantees Synchronized I/O Data Integrity Completion.

Although the POSIX standards and the interfaces that comply with them can be used to preserve data integrity in systems with caches, we argue that they are not flexible enough to meet some applications' requirements for write synchronization. We will discuss their shortcomings in detail in the following sections.

Most Unix or Unix-like operating systems are implemented to comply with POSIX standards. Applications can use either of the two mechanisms, file open flags or synchronization system calls, to achieve the goal of data integrity in a computer system with multiple cache tiers. In addition, most Unix operating systems allow applications to decide whether to use the underlying caches at all. A running application can make the decision by setting the O_DIRECT flag when it opens a file. Although this functionality is not defined in POSIX, it is widely implemented in the I/O interface of most Unix (and related) OS. Data pages carried by I/O requests can be buffered in the underlying cache tiers only if the pages' file was opened without the O_DIRECT flag. Based on this open flag, the OS can also indicate

to lower tier systems, e.g. to a storage server, whether to buffer the file's pages. This is done through the lower level I/O interfaces, e.g., SCSI [3] or SATA [95] . In this study we define two categories of I/O request and show their relationships with I/O synchronization as follows:

- Unbuffered I/O, if the specified data is not to be buffered in file system cache or other underlying cache tiers. Unbuffered I/Os must be synchronized I/Os because all the data being read (or written) must be fetched from (or propagated to) the bottom of the storage hierarchy, and cannot by placed into the any lower level cache tiers.

- Buffered I/O, if the specified data is to be buffered in file system cache or other underlying cache tiers. Buffered I/Os can be either synchronized I/Os or unsynchronized I/Os, depending on whether or not (respectively) the carried data must be fetched from or propagated to the persistent storage devices. A buffered and unsynchronized I/O can be served by a underlying cache tier rather than the persistent storage devices.

Using buffered and unsynchronized I/O can be more efficient than using unbuffered I/O or synchronized I/O because only the former approach can benefit from using underlying cache tiers. It can generate cache hits and save physical I/O operations performed by storage devices. In addition, the underlying cache tiers can provide some lower level systems, e.g., the file system and the storage system, with opportunities to do I/O scheduling. For example, adjacent data blocks can be accessed sequentially on a hard disk. However, as previously discussed, using buffered and unsynchronized I/O also jeopardizes data integrity because the caches may reside in volatile storage media (RAM, for example) and are not durable in case of failures. In DBMS environments, data durability is a necessary condition to ensure data integrity.

To control the tradeoff between data durability and I/O efficiency, applications can use the synchronization system calls, e.g., fsync(), to explicitly flush buffered data pages from underlying cache tiers to storage devices. For example, an application may issue buffered and unsynchronized I/O requests against a file and so benefit from underlying cache tiers. At a later time it may issue a fsync() system call against the same file to ensure data durability. All buffered but unsynchronized file pages will be synchronized to the storage devices. By the time the fsync() call returns, Synchronized I/O File Integrity Completion is guaranteed.

We have described how the current POSIX-compliant I/O interface provides applications with I/O mechanisms that allow them to utilize underlying cache tiers and to control data durability. Now, we argue that this I/O interface is not flexible enough to meet the requirements of some applications. The POSIX-compliant interfaces have several weaknesses. First, the file open flags are declared at a file's open time. The application cannot change these flags unless it closes and opens the file again. Therefore, all of the I/O requests issued against the same file will have the

same buffer and synchronization requirements. It is not possible to control these requirements on a per I/O basis. Second, the fsync() and other synchronization system calls synchronize the data pages for the entire file. Once the call is issued, all of the file's buffered but unsynchronized pages will be synchronized. There is no way to synchronize some specific data pages without synchronizing the others.

It is worth noting that some of the I/O interfaces [3, 95] underlying file systems do provide mechanisms to support per-operation or per-block based write synchronization. For example, by using the *SYNCHRONIZE CACHE* SCSI command and defining the logical block address, the storage client can indicate to the SCSI device which specific block must be synchronized. For another example, the SATA standards allow storage clients to use command *WRITE BUFFER* or *WRITE SECTOR* to decide whether the written blocks should be placed into the cache or on the persistent media, or both. However, we observe that these flexibilities are not provided to the applications at the file system level. In the following section we discuss the different synchronization requirements of DBMS bufferpool writes and we argue that they are not well-served by the current POSIX-compliant file system interface.

4.2.2 DBMS Bufferpool Write Synchronization Requirements

We observe that different types of DBMS bufferpool writes have different synchronization requirements. First, we give a detailed discussion of transaction durability in DBMS. Transaction management is one of the core components of DBMSs. To maintain database integrity and reliability, a DBMS needs to preserve the durability property for the transactions:

- Transaction durability: After a transaction is completed successfully, its effects on the database should be persistent and should survive failures, such as system crashes or power outages

Most DBMSs use the write ahead logging (WAL) protocol to preserve transaction durability:

- write ahead logging: Log any updates made to database objects and always propagate the log entries to disk before the changes to the data objects are propagated

The WAL protocol ensures that in case of crash and recovery a DBMS can redo or undo all of the updates made by the transactions based on the log entries. Based on the above WAL definition the DBMS must write and synchronize the log entries to disk before the corresponding pages can be written from the bufferpool. We can see that these Write Ahead Logging activities (i.e., the synchronized writes of the log entries) are triggered by bufferpool writes.

When write ahead logging is used, log entries will accumulate on disk. However, the log cannot grow forever and eventually old log entries must be deleted or archived. Before the log entries can be deleted, the corresponding dirty bufferpool pages must be synchronized to disks to preserve data durability. Otherwise, a failure may cause those updates to be lost and irrecoverable. Furthermore, forcing the dirty bufferpool pages to disk can also reduce the time spent during recovery, because the number of log entries that need to be replayed will be reduced. As introduced in Chapter 2, we refer to these bufferpool writes, which are issued for recovery reasons, as RECOV writes. We observe that RECOV writes need immediate synchronization because otherwise data durability and/or recovery time guarantee cannot be preserved in case of a crash.

We make another key observation about synchronization of REPLACE writes. REPLACE writes, which are issued for replacement reasons, do not need immediate synchronization because their corresponding log entries are still present in the log files, and can be used during recovery. The synchronization of REPLACE writes can be deferred to a later time when the corresponding log entries, which describe the updates made to the written pages, are deleted. The same observation can also be made on SYNCH writes.

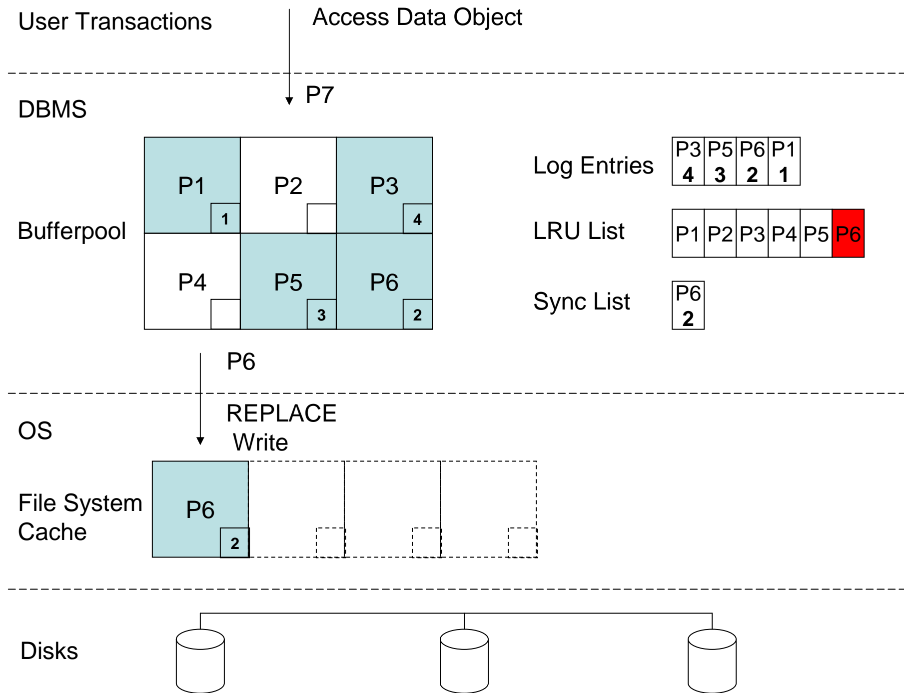


Figure 4.1: Bufferpool REPLACE Write

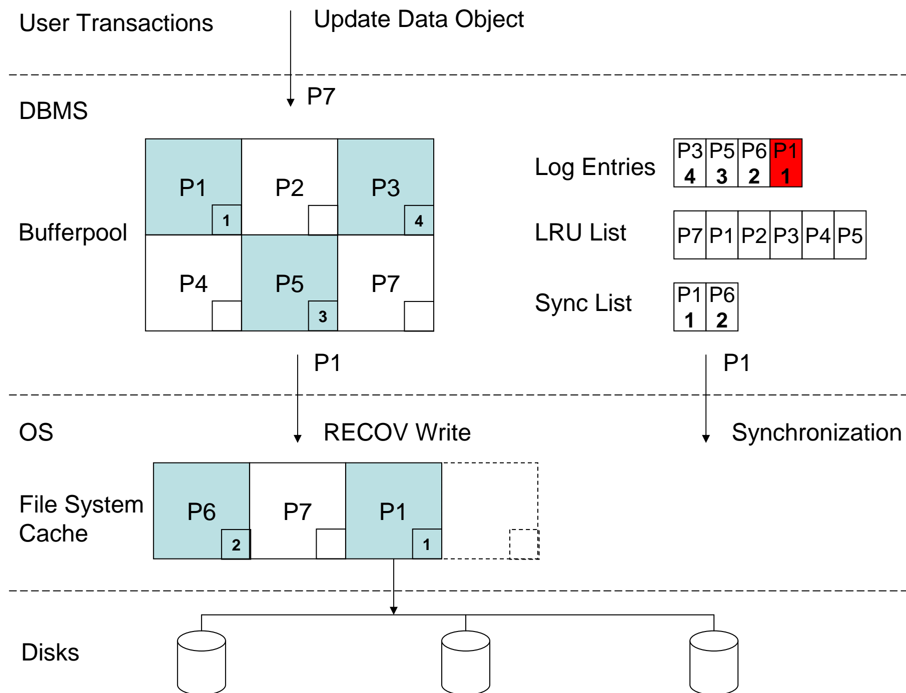


Figure 4.2: Bufferpool RECOV Write

In Figures 4.1 to Figure 4.3 we give an example of when and why a DBMS issues different types of bufferpool writes and why the writes have different synchronization requirements. At the top of the diagrams are the user transactions which access data objects stored in the database. At the lower level is a DBMS system which includes its own bufferpool and other auxiliary data structures. Below the DBMS is the operating system which manages the file system cache. At the bottom level are the storage devices, e.g., disks. The database bufferpool illustrated in the example can hold six pages. The clean pages in the bufferpool and file system cache are marked in white while the dirty pages are marked in blue.

To track the updates made to bufferpool pages and to decide which page should be synchronized to disk, some DBMS systems, e.g., DB2, Oracle, SQL Server, and MySQL, record the oldest update made to each dirty page that has not been synchronized. This is done by keeping the Log Sequence Number (LSN) of the oldest unsynchronized update to the page in the page's metadata. This LSN number is called recLSN in these DBMSs and we use the same term to refer to it here. In the diagrams the recLSN is indicated at the bottom right of each dirty bufferpool page.

In Figure 4.1 a user transaction tries to access a data object in page P7, which is not in the bufferpool. The bufferpool is full, therefore, an existing page must be

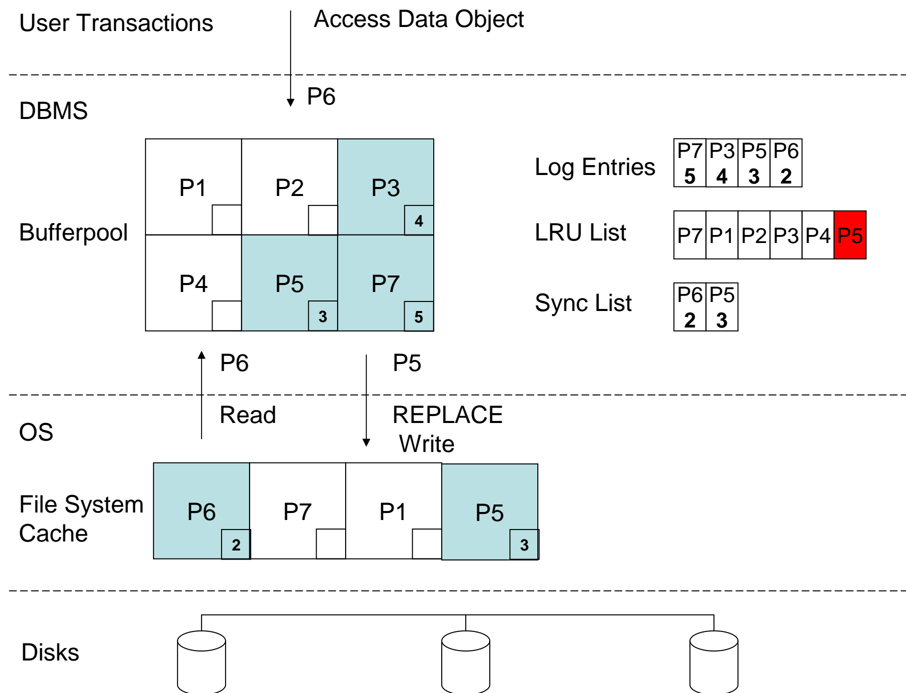


Figure 4.3: Another REPLACE Write

evicted to make room for P7. Assume that the cache replacement policy is LRU, and according to the LRU algorithm, the victim page should be P6. Because P6 is dirty, its content must be written to the disk before it can be evicted. This is a REPLACE write of P6.

In Figure 4.2 a user transaction tries to update a data object in page P7. Assume that the log space is full, or the recovery time threshold (i.e., the *LSN Gap threshold* introduced in Chapter 2) is reached. In order to record the update in a new log entry and to commit the transaction, an existing log entry, i.e., the one containing the update made to P1, should be dropped. However, since page P1 is still dirty and its updated contents have not been propagated to permanent storage, P1 must be written out before the log entry is dropped. This is a RECOV write.

One observation that can be made from the above descriptions is that REPLACE and RECOV writes have different synchronization requirements. RECOV writes, such as the one illustrated in Figure 4.2, are issued due to log space pressure or recovery time pressure. The corresponding log entries must be deleted soon otherwise the recovery time threshold or the log space threshold will have been reached. In order to preserve data integrity and durability, or to provide the recovery time guarantee, RECOV writes need immediate synchronization to ensure that the dirty pages, which contain the updates described by the victim log entries, are

persistent on storage devices.

On the other hand, REPLACE writes, as the one illustrated in Figure 4.1, can be issued as unsynchronized I/Os and the written pages can be buffered in underlying cache tiers without immediate synchronization. This will not impair data durability and integrity because the log entries corresponding to the pages being written are still valid and have been propagated to permanent media (based on the WAL protocol). The synchronization of REPLACE writes can be deferred to a later time, when their corresponding log entries are deleted. We also argue that the deferral does not increase the recovery time because these log entries (corresponding to the unsynchronized REPLACE writes) will certainly be replayed during a recovery.

The deferral of REPLACE writes synchronization can lead to performance gains in two ways:

- It can save I/O bandwidth because the future write requests may access the same pages and may generate cache hits in the underlying cache tiers. The deferred synchronization gives file systems and storage systems opportunities to combine several logical write requests made to the same page into one physical write operation on the disk. For example, in Figure 4.3 the page P6 is accessed by user transactions again. If it is updated and written by the bufferpool again then the file system can combine the two logical writes of P6 into one physical write to disk. Therefore one physical write operation can be saved.
- It provides underlying systems, e.g., file systems and storage systems, with opportunities to do I/O scheduling. Sequential or adjacent writes can be organized into batches, to potentially reduce I/O service time. For example, in Figure 4.3 both P5 and P6 are dirty pages in the file system cache. Assuming they are in adjacent blocks on a storage device, the file system can schedule a physical write operation to synchronize both of the pages at once.

Current POSIX-complaint I/O interfaces do not have the flexibility to support the dynamic synchronization requirements introduced above, i.e., immediate synchronizations for RECOV writes and deferred synchronizations for REPLACE writes. As a result, some DBMSs trade I/O efficiency and performance for data durability and integrity. For example, the DB2 Administration Guide [42] suggests that users use raw devices or unbuffered I/Os (by setting O_DIRECT flag) to bypass underlying cache tiers. As another example, the MySQL [76] InnoDB storage engine issues a fsync() system call after each write request to explicitly enforce synchronization. Although these approaches can guarantee data durability, they ignore or under-utilize the underlying cache tiers and thus miss an opportunity to improve system performance. In this work we propose a novel I/O operation to better control the tradeoff between data durability and DBMS performance.

4.3 Proposed New I/O Operation

In the previous section we argue that the existing POSIX-compliant I/O interface is not flexible enough to meet the write synchronization requirements of some applications, such as DBMSs. Now we propose to extend the existing I/O interface to include a new I/O operation, so that it can satisfy these requirements.

4.3.1 Introducing New I/O Operation

We propose to extend the existing I/O interface and provide the applications with the following functionalities:

- Applications can choose a synchronization method, either synchronized or unsynchronized, for each individual write request without opening and closing the file again
- An application can explicitly synchronize specific file pages to disk at any time it wishes

To achieve the above functionalities, we propose to add a new I/O operation to the current I/O interface. This I/O operation is used to synchronize a specific set of pages from file system cache to disks, if the pages are still in the cache and unsynchronized. The application is responsible for providing the OS with the following arguments when it invokes the new I/O operation:

- The descriptor of the file that it wishes to synchronize
- An array of offsets and ranges that defines the data blocks to be synchronized

After the application submits the new I/O request it will be blocked by the OS until the specified portions of the file have been synchronized. Upon completion, the OS guarantees that the specified ranges and file metadata are propagated to disk, as defined by the POSIX Synchronized I/O File Integrity Completion.

We implemented the proposed I/O operation in Linux and added a new system call for applications to invoke it. The new system call is named *dsync* and the prototype is:

```
int dsync(int fd, int nr, struct offsete *offsetp)
```

The new system call carries three arguments from the application:

- fd, the file descriptor
- nr, number of entries in the file offset array

- offsetp, a pointer to the address of the offset array in the application's user space

Each entry in the offset array is a data structure defined as follows:

```
struct offsete
{
   long long offset; /* offset of the data */
   long long length; /* length of the data range */
}
```

Algorithm 4.1 dsync I/O operation

for each file offset range in the offset array **do**
  if the offset range is valid for the file **then**
    search the file system cache for pages that overlap with the specified offset range
    if such pages are found in the file system cache and are marked dirty **then**
      write the pages to the device
    end if
  end if
end for
synchronize the file metadata to the device
wait for completion of all writes issued to the device
return to the calling application

Upon receiving the system call, the OS conducts the operations shown in Algorithm 4.1. The algorithm is conceptual and the real implementations may be different. However, once the I/O request returns successfully, the calling application knows that the file ranges and the file metadata have been synchronized to disk and are persistent.

4.3.2 Exploiting the New I/O Operation

Applications can use the proposed I/O operation, *dsync*, along with the existing operations, to achieve flexible write synchronizations. The applications can:

- Open the file without the un-buffer and synchronization flags. Therefore their write requests can be buffered in the file system cache and in other underlying cache tiers without synchronization.

- Writes that require no immediate synchronization, such as the REPLACE write illustrated in Figure 4.1, can be issued by using the existing unsynchronized I/O operations, e.g., by using write() or pwrite() system calls.

- Writes that require immediate synchronization, such as the RECOV write illustrated in Figure 4.2, can also be issued using the existing unsynchronized I/O operations, but are followed immediately by our newly proposed I/O system call, dsync(). The latter system call explicitly synchronizes the pages written by the write() or pwrite() call to disk. After the dsync() call returns, it is guaranteed that the pages have been propagated to disk and are persistent.
- For writes that require deferred synchronizations, the application must take the responsibility to keep some information, i.e., the file descriptors, the offsets, and the lengths of the data blocks that have been written. That information can be stored in data structures in user space. At the time of the deferred write synchronization, the application can use the recorded information to issue the dsync() system call.

By using the proposed I/O operation, applications can utilize the underlying caches more effectively and improve I/O performance. At the same time, the synchronization of write I/O requests can be done whenever necessary to preserve data durability.

To evaluate the effects of the proposed approach, we modified the MySQL InnoDB storage engine to use the new I/O operation. InnoDB uses the WAL protocol and asynchronous page cleaning techniques (as introduced in Chapter 2) in its log and bufferpool management. It regularly writes a batch of dirty pages that carry the oldest updates (made to the data objects by user transactions) to disks. These writes are RECOV writes. Each time such a page cleaning session is triggered for recovery reasons, the InnoDB issues RECOV writes to flush bufferpool pages with small recLSNs to disks so as to reduce the LSN Gap. It ensures that those updates are safe on disk. After the page cleaning session is completed, the log entries describing those updates can be deleted and their space can be reused by new updates.

Because the existing POSIX-compliant I/O interfaces are not flexible enough to support the synchronization requirements of both REPLACE and RECOV writes, the InnoDB storage engine was implemented to be conservative in doing write synchronizations. It issues a fsync() system call after each bufferpool write to synchronize the pages explicitly. As discussed earlier, this approach is safe but less efficient, and we will show an example in Figure 4.4.

In Figure 4.4 we illustrate a small log with 16 log entries and each entry contains its own LSN. From the legend we can see that there three types of entries in the log:

- the entries whose corresponding updates carried by bufferpool pages have been written and synchronized to disk. To guarantee recovery time and/or log space limitation, the DBMS needs to issue (RECOV) writes and to enforce synchronization when the Log Gap reaches its threshold. These log entries, with their LSNs being from 11 to 15, are illustrated in blue color. They can

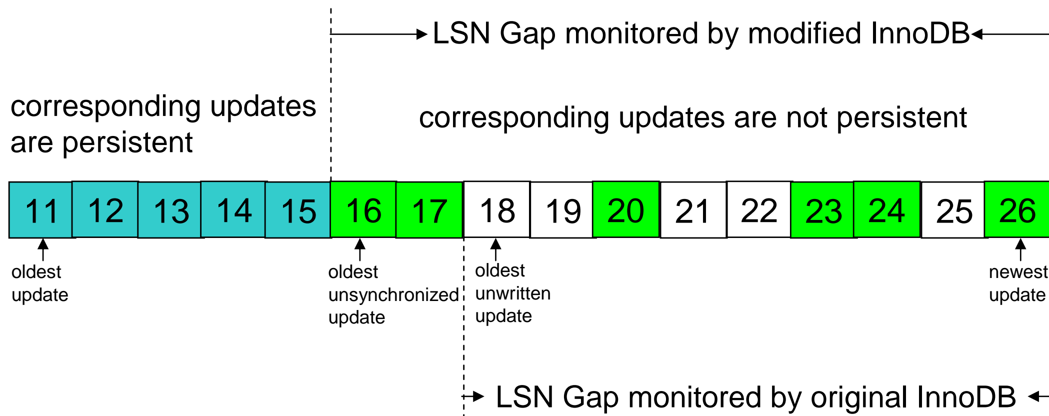


Figure 4.4: Example of Log Entries and Bufferpool Writes/Synchronizations

be deleted without jeopardizing data durability. In addition, these log entries do not need to be replayed in case of a crash and recovery.

- the entries whose corresponding updates carried by bufferpool pages have been written, but have not been synchronized to disk. Under bufferpool space pressure, the DBMS issued write requests (REPLACE writes) for these pages and may have evicted them from its bufferpool. However, it need not ensure that these writes are synchronized, since the updates can be recovered from the DBMS log in case of a failure. These type of entries (with their LSNs being 16, 17, 20, 23, 24, and 26) are illustrated in green color. They cannot be deleted from the log because their corresponding updates may not be persistent. In case of a crash and recovery, these log entries will be used to bring the database to a consistent state.

- the entries whose corresponding updates carried by bufferpool pages have not been written to disk. These entries (with their LSNs being 18, 19, 21, 22, and 25) are illustrated in white color. These entries also cannot be deleted and will be used during a recovery.

Under the current InnoDB implementation, each bufferpool write is followed by a fsync() system call to explicitly enforce synchronization. However, from the above example we can see that this approach is more conservative than necessary.

Some of the writes, i.e., to write the updates described by the *green* log entries, do not need to be synchronized immediately, because their corresponding log entries are still present in the log. Their synchronization can be deferred to a later time, when their corresponding log entries are deleted.

With the above observations in mind, we made the following changes to the InnoDB code to let it use our approach:

- Eliminate the fsync() system call after each bufferpool write request. Thus the data pages being written can be buffered in the file system cache without immediate synchronizations to disk.
- Create a new data structure, called *synchronization list*, in InnoDB The elements in the list contain the metadata of buffpool writes, e.g., the page number, the file id, the file offset, and the recLSN number of the written pages. The list is sorted by the pages' recLSN values. We illustrated this synchronization list as *Sync List* in Figure 4.1. The metadata stored in this list will be used to synchronize the written pages to disk.
- For each bufferpool write, either a REPLACE or a RECOV, we check whether the bufferpool page's metadata exists in the Sync List. If not, then we add the metadata as a new element into the list. If yes, then we only issue the write and do not update the Sync List.
- As originally implemented, the InnoDB bufferpool manager keeps monitoring the LSN Gap. If it reaches the LSN Gap threshold, which is a database parameter, then InnoDB triggers a page cleaning session for recovery reasons. However, because of our approach, there may be unsynchronized updates (such as the ones described by LSNs 16 and 17 in Figure 4.4) carried by bufferpool pages in the lower cache tiers. We changed InnoDB to let the bufferpool manager also monitor the Sync List to determine the LSN number which describes the oldest unsynchronized updates (described by LSN 16 in Figure 4.4) made to the database objects.
- Each time InnoDB triggers a page cleaning session for recovery reasons, it issues a batch of RECOV writes. We modify InnoDB so that, after issuing the RECOV write batch, it checks the newly implemented Sync List to search for the elements with their recLSN earlier than the LSN Gap threshold. It then issues a dsync() system call to ensure that these updates are synchronized.
- Upon the completion of the dsync() system call we notify InnoDB that the the synchronizations (of the pages in the RECOV batch and the Sync List) have been completed and that the corresponding log entries can be deleted or reused. We also delete the elements synchronized by the dsync() system call from the Sync List.

By changing the InnoDB we make it stop doing explicit write synchronizations, i.e., issuing fsync() after each bufferpool write. We change it to take advantage of the new I/O operation. The experimental results show significant performance gains and we will discuss them in the following section.

4.4 Evaluations

We implemented the proposed new I/O operation in a Linux kernel and evaluated its effects by running synthetic workloads. The goal of our evaluations is to determine whether, and to what extent, the number of physical writes can be reduced because of the additional flexibility provided to the second-tier cache by dsync(), which allows applications to be less conservative with write synchronization. We expect to see a significant drop on the numbers of physical writes required to support the applications, and a significant improvement on applications' performance, e.g., DBMS transaction throughput.

A Dell Poweredge 2850 server was used to run our experiments. The server has two 3.0GHZ Intel Xeon CPUs (each with two Hyper Threading cores) and 4GB physical memory. There are six SCSI hard drives attached through a disk controller. The installed operating system is the Ubuntu Linux distribution with kernel 2.6.26 (with SMP support), updated to include the proposed dsync() I/O operation. During the evaluations we created the data files on an ext2 [1] file system. The file system page size is the default 4KB.

In these experiments, the Linux file system buffer cache acts as the second-tier cache. Other than the addition of dsync(), we did not modify the file system's buffer manager in any way. Thus, any reductions in the number of physical writes are the result of the existing buffer manager's ability to exploit weaker application synchronization requirements. In Chapter 5, we will consider whether any additional performance can be gained by using a second-tier cache manager that is explicitly designed to exploit the weaker synchronization requirements that are enabled by dsync().

4.4.1 Evaluations Using Synthetic Workloads

We run two different types of workloads against the extended I/O interface. The first one is generated by a small program which issues write requests with different synchronization requirements against a file in the ext2 file system. The second workload is generated by another program which issues SQL UPDATE commands against a MySQL database.

4.4.1.1 Synthetic Workload Program Issuing Write Requests

In this evaluation the synthetic workload program issues two million write requests against an ext2 file, which is composed of 1 million 4KB pages, on a single SCSI disk. The file system cache is configured to be 200MB in size and the data accesses are exponentially distributed across the entire file with the distribution parameter set to 1.0.

The program generates two kinds of writes: immediately synchronized and deferred-synchronized. Immediately synchronized writes are followed immediately by a dsync() operation. Deferred synchronized writes are followed by a dsync() operation that occurs after a delay. We measure the delay in terms of the number of intervening writes that occur between the write and its dsync() operation. We vary two important parameters in the evaluations:

- The percentage of immediately synchronized writes in the I/O workload. We vary the parameter to be 10, 50, and 90 percent in different runs.
- The delay, measured as the number of intervening I/Os, between the deferred-synchronized writes and the corresponding dsync() operations. We vary the parameter from 5K, 10K, and to 200K in different runs.

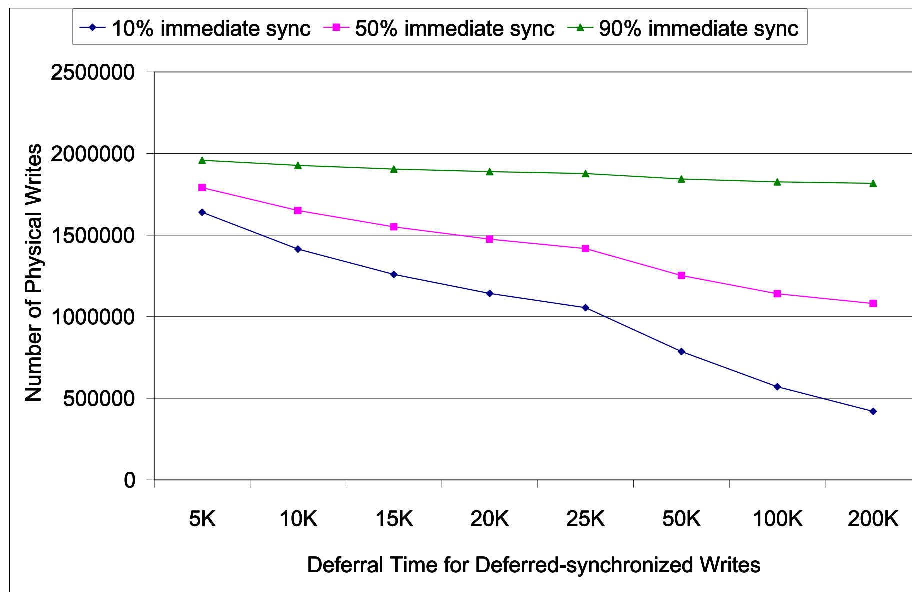


Figure 4.5: Synthetic Workload Program Issuing Writes

While the workload program is running we measure the number of physical writes issued from the OS to the disk. The results are shown in Figure 4.5. Since the program issues 2 million logical writes to the OS, we should observe the same

number of physical writes if they are issued as synchronized writes using the existing I/O interface. As expected, by using the extended I/O interface, some of the physical writes are saved. We observe two different trends from the results:

- The more deferred-synchronized writes in the workload the more physical writes can be saved. This is reasonable because a synchronized write needs immediate synchronization (by issuing the dsync() call after the write) and will definitely generate a physical write. Fewer synchronized writes means there is a higher probability to have write cache hits in the file system cache and to consolidate multiple logical writes into one physical write.
- The longer the deferral between an unsynchronized write and the corresponding dsync() operation, the more physical writes can be saved. This is also a reasonable result. When the file page carried by an unsynchronized write stays longer in the file system cache without synchronization, the probability that the same file page will be written again becomes higher. This is especially true for the runs with less synchronized writes (e.g., the 10% cases). The number of physical writes is reduced to less than one third of the number of logical writes when the deferral length is increased from 50K to 200K operations. However, if there is a high percentage of synchronized writes in the workload (e.g., the 90% cases), then the benefit of using our proposed approach becomes insignificant.

4.4.1.2 Synthetic Workload Program Issuing SQL UPDATEs

As introduced in Section 4.3, we modified MySQL InnoDB storage engine to use our proposed approach. After the modification the InnoDB no longer relies on fsync() calls to explicitly enforce write synchronizations and it issues REPLACE writes as unsynchronized requests. To determine the effect of this change we ran a synthetic workload program against a MySQL database. The database contains only one table holding 240K records. Each record is about 4KB size and the table is stored in a 1GB ext2 file on a single SCSI disk. The log files of the database is located on another dedicated SCSI disk. We configured both the database bufferpool and the file system cache to be 240MB in size. In addition, the log file size, which controls the LSN Gap threshold, is set to be 200MB. This is a reasonable value corresponding to the database size and the workloads. The synthetic workload program issues 2.4 million single record SQL UPDATE commands against the table. The access to the table records is exponentially distributed with the distribution parameter set to 1.0

The MySQL InnoDB storage engine originally (without our modification) issues a fsync() system call after each bufferpool write, but we have changed it to use the proposed dsync() operation only when necessary (i.e., when it needs to synchronize writes). To fully evaluate the effects of our proposed approach, we also include an additional version of InnoDB in our experiments. Unlike the original InnoDB, this

version is changed to use the fsync() calls *cleverly*, i.e., not after each bufferpool write, but only when it is necessary (when InnoDB decides to trigger page cleaning for recovery reasons and issues a batch of RECOV writes). Therefore we have three versions of MySQL code to compare in our evaluations:

- org: The InnoDB code using the original approach to do write synchronization. It issues a fsync() system call after each bufferpool write.
- rec: The InnoDB code is changed to issue fsync() call only when necessary. When InnoDB decides to do page cleaning for recovery reasons, it issues a fsync() system call after each RECOV write.
- nio: The code is changed to use the proposed new I/O operations, dsync(), when necessary, as described in Section 4.3.

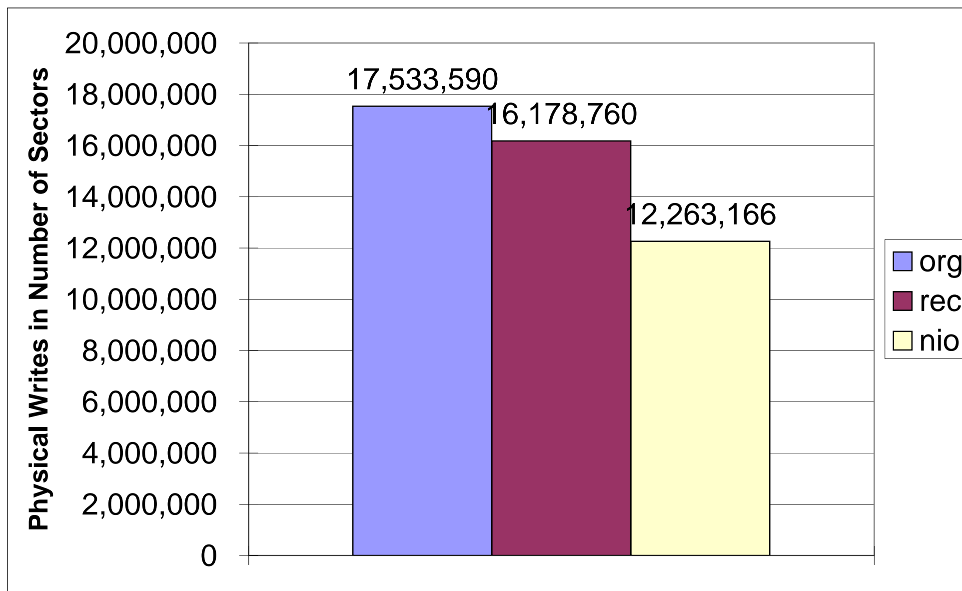


Figure 4.6: Synthetic Workload Program Issuing SQL UPDATE Queries, Sectors Physically written

Figure 4.6 shows the number of sectors physically written to the disk (which contains the database) by MySQL. From the results we can see that in terms of sectors physically written the nio version performs the best. It eliminates 30 percent of the written sectors compared with the org version. The rec version also saves some written sectors but not as many. Under the rec version, the REPLACE writes can be cached in the file system cache. However, because it uses fsync() for

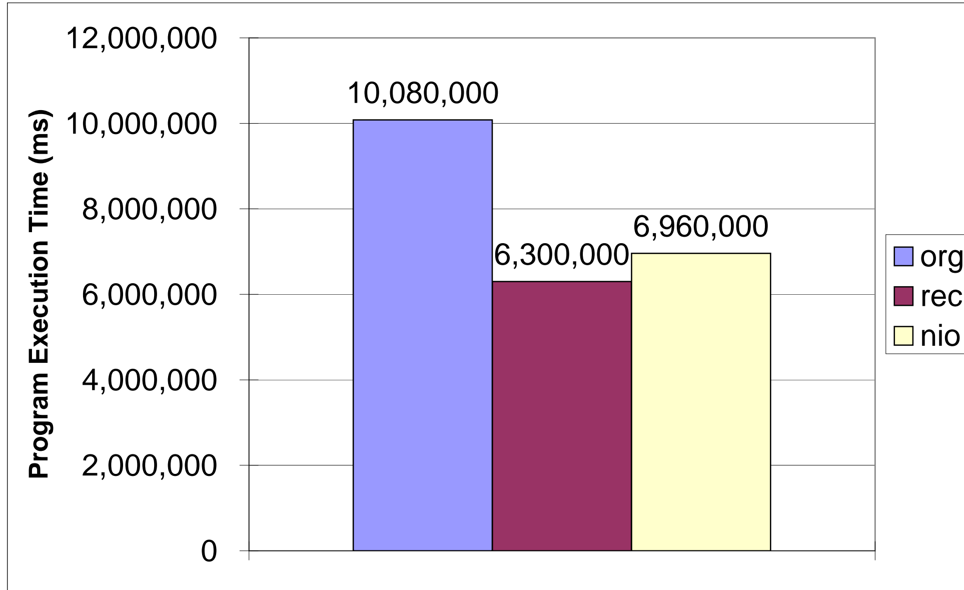


Figure 4.7: Synthetic Workload Program Issuing SQL UPDATE Queries, Program Execution Time

synchronization, it synchronizes pages of the entire file to disk, and is less efficient than the nio version, which only synchronizes the necessary pages to disk.

In terms of workload execution time, which is shown in Figure 4.7, the rec version is slightly faster than the nio version, even though the latter writes fewer sectors to disk. To explain this, we monitored the average I/O service time. The result is shown in Figure 4.8. The I/O service time shown includes both read and write requests. We can see that the rec version has shorter I/O service time than that of the proposed nio version.

We reviewed the Linux implementation of the fsync() I/O operation, which is used by the rec version to do write synchronizations when necessary. First, the fsync() searches the file system cache for the dirty pages from the target file. The offsets of the dirty pages are recorded in a sorted list based on their offset values. After the search is completed Linux issues a batch of writes to the device according to the sorted offset list. In our evaluations the file offsets are in the same order in which the data blocks are physically located on the disk. Since on average each fsync() call flushes a large amount of data (more than 100MB) from the file system cache, some of the physical writes are performed sequentially on the disk.

On the other hand, in the nio case (which uses the proposed dsync() operation) there are far fewer sectors being written by each dsync() operation (less than 10MB on average). The probability to have sequential I/Os on the disk is much lower.

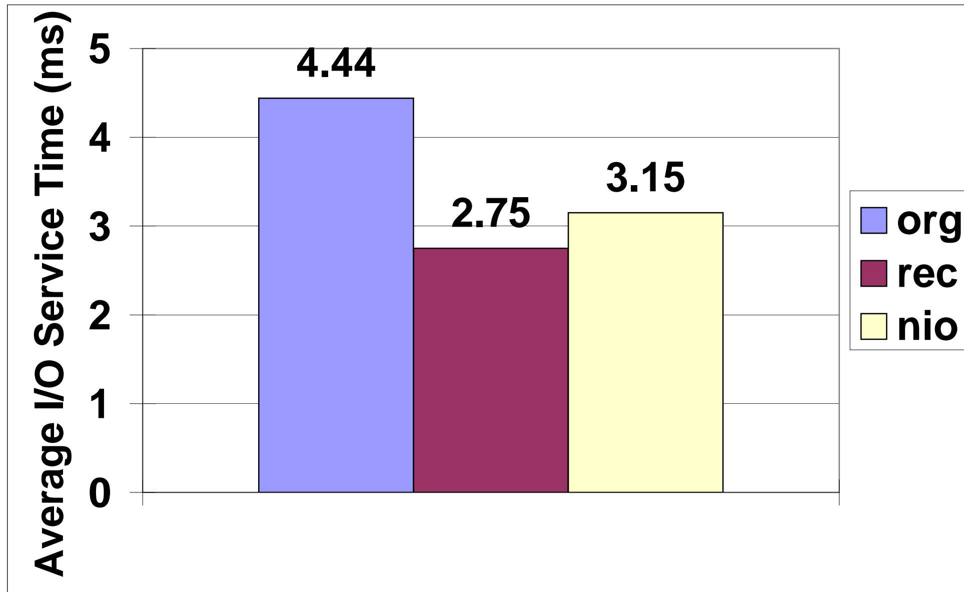


Figure 4.8: Synthetic Workload Program Issuing SQL UPDATE Queries, I/O Service Time

As discussed earlier, the performance gains obtained by deferring write synchronizations (by using either fsync() or dsync()) come from two factors: better I/O scheduling and more write cache hits. However, these two factors may contradict with each other. Our evaluation shows that writing larger batches of sectors may result in better I/O scheduling. For the same reason, in the org version case we monitored the longest I/O service time. This version of InnoDB issues a fsync() system call after each bufferpool write to explicitly synchronize the write to disk and, therefore, cannot benefit from either of the two factors.

4.4.2 Evaluations Using TPC-C Workload

In addition to the synthetic workload programs, we also use a TPC-C workload kit to conduct our evaluations. The goal is to determine how the proposed techniques can effect a DBMS application's performance, in terms of transaction throughput. It gives us insight on the effectiveness of the proposed approach in a more realistic scenario. The TPC-C workload kit issues a pre-generated workload of SQL queries against the three versions (i.e., org, rec, and nio) of MySQL. The Transactions Per Minute Type C (tpmC), which is the standard throughput metric for TPC-C workloads [101], and other I/O statistics are collected after the warmup of the database bufferpool and the file system cache.

Because the results may vary in different runs, we repeat each evaluation five

InnoDB version	read sector per second	write sector per second	I/O service time	% disk utilization	transaction per minute type C
org	2756.03	2178.24	3.57	96.77	1134.20
rec	4395.03	3740.54	2.23	98.35	1936.87
nio	4161.58	2762.89	2.41	92.60	1822.17

Table 4.1: TPC-C Workload, MySQL Database on Single Disk

times and report the median value collected. Before each run, the computer, the operating system, and the MySQL database server, are rebooted. We observe that the variance in the results collected in different runs is trivial, with the differences (including the ones between the maximum/minimum and the median) in less than 0.5% of the median value. This is because in each run the workload is exactly the same and the computational environment remains almost the same. The trivial variance in the results is caused by different thread scheduling (among the multiple CPUs) in different runs. That results in different I/O scheduling at the file system, the disk controller, and the disk levels in different runs.

4.4.2.1 TPC-C Workload, MySQL Database on Single Disk

In this experiment we place the database file (ext2) on a single SCSI disk as we did in the synthetic workload evaluations. The TPC-C workload is configured to include 10 warehouses (SF=10) and 10 connections running against the database. The database size is about 1GB and both the bufferpool size and the file system cache size are set to be the same at 240MB. We keep the log file size at 200MB, which is same as that in the previous synthetic workload evaluations. The results are shown in Table 4.1.

From Table 4.1 we can make following observations:

- In terms of Transactions Per Minute Type C (tpmC), the relative performance of the three MySQL versions is similar to that observed in the synthetic workload evaluations. The rec and the nio versions perform much better than the org version. At the same time the rec version still performs somewhat (6.3%) better than the nio.

- In all the runs of the three versions the disk utilization is more than 90 percent, which means the system is I/O bound.

- The nio version, which uses our proposed approach, eliminates a significant number of physical writes. In the nio case the ratio of the average written sectors per second to the average read sectors per second ratio is 1.51, where the same ratios for the org and rec cases are 1.27 and 1.17, respectively. This means to complete the same amount of transactions, which issues a fixed number of read sectors, the nio version generates the fewest written sectors.

InnoDB version	read sector per second	write sector per second	I/O service time	% disk utilization	transaction per minute type C
org	5324.32	4738.52	1.63	93.62	2511.20
rec	5649.03	5164.61	1.60	96.94	2698.83
nio	7007.84	4641.97	1.49	97.37	3194.03

Table 4.2: TPC-C Workload, MySQL Database on RAID-5 Disk Array

- The rec version has the shortest average I/O service time (2.23 ms). It also writes many more sectors (3740.54) per second than either the org or the nio version does. This is consistent with our previous observations made in the synthetic workload evaluations. Once again, we argue that this is because in the rec version, the fsync() system calls flush large chunks of dirty pages of the entire database file to the single disk, and therefore, it benefits from sequential disk operations.

4.4.2.2 TPC-C Workload, MySQL Database on RAID-5 Disk Array

As discussed earlier, sequential write operations on a single disk favor the rec version which uses fsync() calls when synchronizations are necessary. However, this is not the only physical design of database systems. In some other circumstances the reduction of physical writes can become more important. We set up another experiment in which we place the database file on a RAID-5 disk array, which is composed of 4 SCSI disks. The other experimental settings remain the same as those of the previous TPC-C evaluations and the configurations of the RAID-5 array is shown as follows:

- The stripe size is the default 64KB
- The battery-backed cache on the disk controller is 256MB
- The write policy is write-back, the default

Under this physical design the database file is striped across the 4 SCSI disks. We assume that sequential writes operations will no longer be the dominant factor to impact the database performance, and that the reduction of physical writes will become more important. We expect to see that the nio version can take advantages of using the proposed approach and can outperform the rec version. The experimental results are shown in Table 4.2.

We can make the following observations from the results:

- The disk array remains more than 90 percent busy on average in all the three cases, which means the system is I/O bound.

- The average I/O service time values are much shorter than those of the previous TPC-C evaluations on the single disk (shown in Table 4.1). With the presence of the 4 disks and the controller cache, this is a reasonable result. We can also observe that the rec version no longer achieves shorter I/O service time compared the nio version.
- The nio version still eliminates a significant number of physical writes. The reads sectors to write sectors ratios for the nio, the rec, and the org versions are 1.51, 1.10, and 1.12, respectively.
- The nio version outperforms the rec version in terms of tpmC (with 18.3% more transactions per minute on average), which is consistent with our expectation. With the striped database file and the the controller cache, sequential I/O operations are no longer the dominant factor to impact the database performance. Reducing the number of physical writes is more important.

4.5 Summary

In this research we make the following contributions:

- We observe that DBMS bufferpool writes have different synchronization requirements. REPLACE writes, unlike RECOV writes, do not need immediate synchronization. Synchronization can be deferred to a later time and this deferral will generate several benefits, such as write cache hits and better I/O scheduling.
- We also observe that the current POSIX-compliant I/O interface is not flexible enough to support the synchronization requirements of DBMSs. We propose to extend the current interface to include a novel I/O operation, *dsync*, which synchronizes only specific parts of a file. It gives applications more flexibility and more control over their write synchronizations, and it will bring several benefits, such as reduced numbers of physical writes and better I/O scheduling.
- To evaluate the effects of our approach, we implement the proposed new I/O operation in Linux and modify MySQL InnoDB storage engine to use it. By running synthetic workloads and a TPC-C workload against the modified MySQL, which works on the extended I/O interface, we observe that the number of physical writes caused by bufferpool writes are significantly reduced (compared with the cases that use original MySQL working on the POSIX-compliant I/O interface). The transaction throughput of MySQL is also significantly improved.

Based on the evaluations we conclude that by using the proposed approach, some applications, such as a DBMS, can reduce their physical write rates and can achieve better I/O scheduling.

Chapter 5

Write-Aware Replacement For Second-Tier Cache

In Chapter 3 we propose a technique to use write hints in second-tier cache management to reduce the number of read requests issued by the cache. In Chapter 4 we propose another technique to defer synchronization of bufferpool writes so that the cache can reduce the number of writes. In this chapter we propose that under some circumstances, the second-tier cache manager should take the total I/O cost associated with a cache replacement (i.e., the cost to synchronize the victim page to disk and the cost to bring it back into the cache when the page is referenced next time) into account while making replacement decisions. We argue that under these circumstances, reducing the total number of I/Os, including both reads and writes, leads to better DBMS performance than reducing only the read requests.

5.1 Introduction

The replacement algorithm is the core of second-tier cache management. In today's DBMS environments, where there exist both read and write requests and multiple cache tiers, various second-tier cache replacement algorithms, such as ARC [64], MQ [111], and Type Queue (introduced in Chapter 3), have the sole objective of maximizing the number of *read* hits. The reasons behind the decision to ignore *write* cache hits are:

- The write requests issued by database bufferpools are asynchronous with regard to query executions (except the SYNCH writes). The latency of the write requests can be hidden from the DBMS clients by using dedicated threads, e.g., the page cleaners. In contrast, many read requests are issued on demand and are on the critical path of query executions. Thus, for a second-tier cache, serving the read requests, which are synchronous, is more important than serving the write requests, which are asynchronous.

- The write requests issued by database bufferpools are synchronized writes, which means that the second-tier cache must issue a write to the storage device for each write request that it receives. Therefore, having write hits is of no benefit to the cache or to DBMS performance.

Similar observations can also be made for other types of storage clients, such as file systems [66, 8], in which the reads are often issued to lower level storage on demand from the running applications and the writes are issued asynchronously with respect to the applications' executions. These observations lead second-tier cache replacement algorithms, such as ARC, MQ, and TQ to focus only on maximizing read cache hit rates and to ignore the writes.

However, these observations do not always hold. First, with the introduction of the deferred synchronization mechanism (by using the *dsync* request), which is introduced in Chapter 4, the REPLACE bufferpool writes become unsynchronized. This technique gives second-tier caches opportunities to optimize write performance (through write cache hits and better I/O scheduling). Second, it is not always true that writes are asynchronous with respect to query executions. We observe that this assumption does not hold when the storage devices are saturated by the I/O loads. In particular, when the storage devices become bottlenecked, both read and write requests are blocked in I/O traffic and compete for I/O bandwidth. In such a scenario, the writes become synchronous with respect to the query executions even though they are issued asynchronously.

We argue that under such circumstances, maximizing read cache hits is no longer in the best interest of DBMS performance. The objective of cache replacement should be to minimize the total number of I/Os, including both reads and writes, issued from the second-tier cache to the storage devices. Reducing the total number of I/Os saves I/O bandwidth and allows more read requests, which are on the critical path of query executions, to be served within the same amount of time. It also helps the read requests (waiting in the queue) to be served sooner and faster.

In this work we propose new cache replacement algorithms which focus on reducing the total number of I/Os (including both reads and writes). We refer to them as *write-aware* algorithms. We use the term *write-oblivious* to describe algorithms designed to minimize only read requests.

Our contributions in this research are threefold. First, we present a partial characterization of an optimal, off-line, and write-aware replacement algorithm. In doing so, we state several lemmas which describe the characteristics of the optimal algorithm. Second, based on the above knowledge, we propose a write-aware off-line cache replacement algorithm and extend several write-oblivious on-line algorithms to be write-aware. Third, we evaluate the effects of the proposed write-aware algorithms by trace-driven cache simulations. The experimental results show that by using the extended write-aware algorithms the second-tier cache can reduce the total number of I/Os by up to 50%, compared with using the write-oblivious algorithms to manage the cache.

The remainder of this chapter is organized as follows. In Section 5.2 we give several preliminary definitions required in our research. In Section 5.3 we give a detailed definition of the research problem on which we focus. In Section 5.4 we present the partial characterization of an optimal cache replacement algorithm. We propose several write-aware algorithms in Section 5.5. Our evaluations are shown in Section 5.6 and we conclude in Section 5.7.

5.2 Read-Write Workload Model and Cache Model

In this section we give some preliminary definitions. First, we make the following assumptions about the second-tier caches in our study:

- The pages are brought into the cache on-demand, which means that a page is brought into the cache *only* when the page is requested (read or write) by the first-tier
- All the pages being requested (read or write) by the first-tier must be placed into the cache

These assumptions are made for simplicity reasons. The second-tier cache content is determined only by the page requests received by the cache and by the replacement algorithm used to manage the cache. Other cache management related policies, such as pre-fetching policies and policies to decide whether to bypass the cache (by serving the requested page directly from the storage device to the first-tier), are not taken into account in this research.

5.2.1 Read-Write Workload Model

In this work we focus on cache replacement algorithms that are *write-aware* and reduce the total number of I/Os generated by the cache. We begin with some observations about write-oblivious replacement algorithms:

- The MIN algorithm, proposed by Belady [9], is an optimal off-line algorithm which maximizes the number of read cache hits. The MIN algorithm always evicts the page with the lowest replacement cost, i.e., the page being referenced furthest in the future, during cache replacement.
- Other cache replacement algorithms, such as ARC, MQ, and TQ, follow a common approach, which is to imitate the replacement decisions of the MIN algorithm. Based on various techniques (e.g., by using recency, frequency, and I/O types), these algorithms attempt to identify the *ideal* victim page, i.e., the page being referenced furthest in the future, as accurately as possible, and to evict it in the next cache replacement.

In addition, we also observe that the existing write-oblivious algorithms, such as MIN, ARC, and MQ do not differentiate among the different types (e.g., read and write) of request received by the cache. They treat all the requests as if they are of the same type, and make replacement decisions accordingly. These algorithms also do not recognize the clean or dirty states of the cache pages. In their model of the page request workload, all requests are treated as *reads*, and the replacement algorithm's objective is maximizing the number of read cache hits. We refer to this workload model as a *read-only* workload model.

However, as we have discussed in Section 5.1, minimizing the number of reads is not always in the best interest of DBMS performance. Under some circumstances, the second-tier cache should be managed to minimize the total number of I/Os, including both reads and writes. To do this, it is necessary to distinguish read and write requests in the workload. Thus we define a read-write workload model as follows:

- $A = a, b, c, ...$ is a finite set of pages being requested by the first-tier
- There are three different types of request issued by the first-tier: read, write, and dsync
- Page request trace $X = x_1, x_2, ..., x_L$ is a finite sequence of L elements, where x_t denotes the request received by the cache at time t; and $x_t = (y_t, z_t)$ where $y_t \in A$ is the page being requested at time t (in terms of page requests), and $z_t \in \{read, write, dsync\}$ is the request type at time t.

5.2.2 Read-Write Cache Model

Under the read-only workload model, the cache does not distinguish different request types and different page states (e.g. dirty or clean). However, under the read-write workload model, the cache will behave differently. We define a read-write cache model as follows:

- Page states: at any given time each page in A must be in one of the following states:
  - Initial state: this is the state that the page has not been requested yet and therefore is not in the cache
  - Clean state: this is the state that the page is in the cache (brought into the cache by a read or write request) and its content is consistent with its copy on the lower level storage. The page either has not been updated (by a write request) since it was read into the cache, or the updates made to it have been synchronized to the lower level storage.

- Pre-clean state: this is the state that the page is in the cache and is dirty, which means that the page's content has been updated by a write request and that the update has not been synchronized to the lower level storage. However, there is no write request of the page in the trace before its next dsync request.
- Dirty but not pre-clean state: this is the state to describe the other dirty cached pages that are not in the pre-clean state
- Evicted state: this is the state to represent that the pages have been evicted from the cache and remain out of the cache

- State transitions: After the cache receives a request, the page being requested will transit from its current state to another state. If a cache replacement is required, the replaced page will also transit from its current state to the Evicted state. In summary, there are five different types of state transition:
  - transition r, indicates that the cache receives a read request of the page
  - transition w_w, indicates that the cache receives a write request of the page and there exists at least another write request of the page before the page's next dsync request (or there is no more dsync request of the page in the trace)
  - transition w_d, indicates that the cache receives a write request of the page and there exists no write request of the page, before the page's next dsync request
  - transition d, indicates that the cache receives a dsync request and the requested page will be synchronized to lower level storage if it is dirty
  - transition e, indicates that, based on the replacement algorithm used to manage the cache, the page is chosen as the victim and is evicted from the cache

- I/Os generated by the cache: During a state transition, the cache may generate an I/O request issued to the lower level storage. The I/Os include two types: read and write.

We illustrate the page states, states transitions, and the I/Os generated by the cache in Figure 5.1. The Initial, Clean, Pre-clean, Dirty but not pre-clean, and Evicted states are illustrated in ovals and labeled as Init, C, P, D, and E respectively. The state transitions are illustrated as directed arcs that connect the states. For example, if a page is in Pre-clean state and is evicted from the cache (the page is chosen as the victim page by the replacement algorithm) then the page will transit from P state to E state through a directed arc.

If an I/O is generated during a state transition then we associate it with the specific arc (as a number after the transition label, separated from the label by a slash). A read I/O is labeled as number *10* and a write I/O is labeled as number

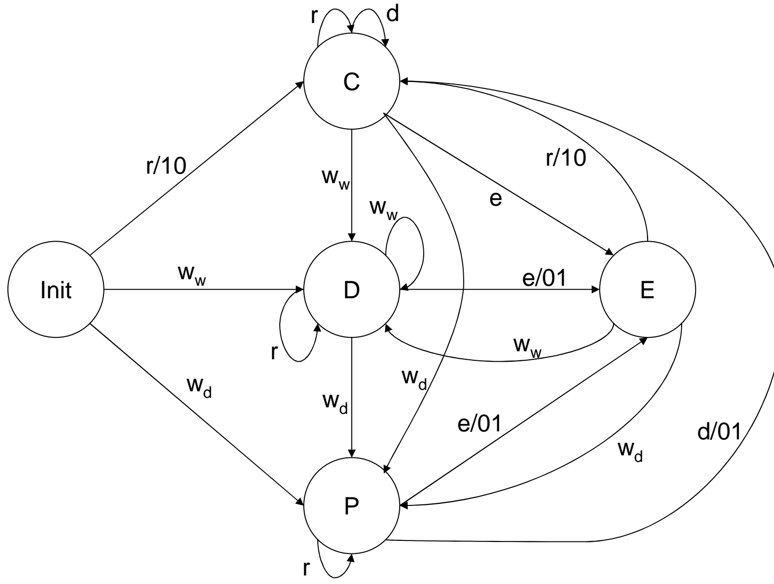


Figure 5.1: States, Transitions, and I/Os of Cache Pages

01. For example, the directed arc from P state to E state is associated with a number *01* because there must be a write I/O generated by synchronizing the dirty pre-clean page to disk.

5.3 Research Problems

Under the read-only workload model the MIN algorithm is optimal [62] in minimizing the reads issued by the cache. However, a similar approach does not minimize the total number of I/O operations under a read-write workload. In particular, we can extend the MIN algorithm to work under the read-write workload model, and we refer to the extended algorithm as MIN-RW. MIN-RW still uses the same approach used by the MIN to make replacement decisions, which is always to evict the page being read again furthest in the future. We argue that the MIN-RW algorithm does not minimize the total number of I/Os under the read-write workload model. It fails to recognize the following two factors while making replacement decisions:

- there exist different request types, e.g., read and write, in the workload received by second-tier caches. These different types of requests have different impacts on the total number of I/Os generated by the cache. For example, a cache miss of a read will *definitely* generate a read request to the lower level storage to fetch the missing page. However, a cache miss of a write request does not guarantee that a write will be issued to the lower lever storage.

- there exist different page states, clean and dirty, for the cache pages. These different states have different impacts on the total number of I/Os generated by the cache. For example, evicting a dirty page requires a write to be issued to the lower level storage, while evicting a clean page does not have such a requirement.

In Figure 5.2 and Figure 5.3 we give an example to show that the MIN-RW algorithm does not minimize the total number of I/Os under a read-write workload. In the figures, we illustrate a small cache of three pages and a workload trace including read and write requests. For example, Rx refers to a read request for page x. Similarly, Wy refers to a write request for page y. A request illustrated in red color represents a cache miss, indicating that the requested page is not in the cache and that a cache replacement is needed.

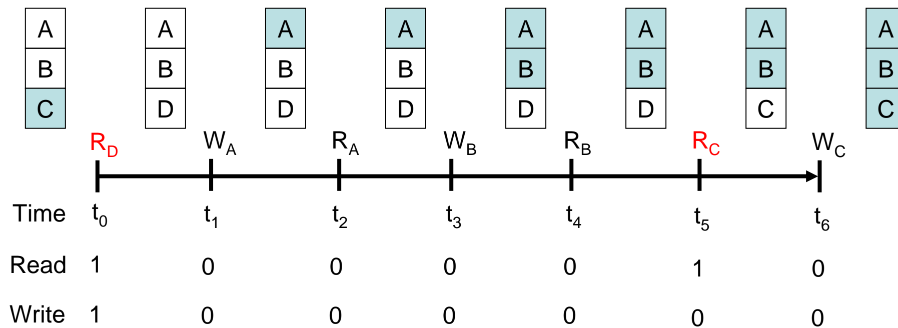


Figure 5.2: Using the MIN-RW Algorithm, Total Number of I/Os: 3

The cache states before and after a request are shown at the upper left and upper right of each request, respectively. After a clean page has been updated by a write request, we mark it as a dirty page by changing its color from white to blue. Beneath the request trace, we also show the I/O requests generated by the cache. To determine the total number of I/Os by the cache during the sequence of requests, we can simply sum the number of reads and writes listed in the two bottom rows.

In Figure 5.2 we illustrate a cache managed by using the MIN-RW algorithm. The total number of I/Os generated by the cache is three. In Figure 5.3, the cache is managed by using another replacement algorithm and the total number of I/Os generated is one. We notice that the MIN-RW algorithm is outperformed by the other algorithm in terms of the new metric, the total number of I/Os. The example shows that the MIN-RW algorithm is not optimal with respect to the new cache management objective under the read-write workload model. To achieve

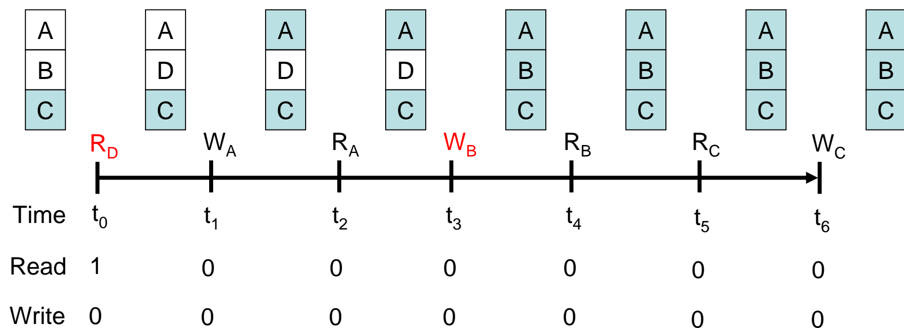


Figure 5.3: Using Another Replacement Algorithm, Total Number of I/Os: 1

the objective, the existing MIN and other write-oblivious algorithms should be re-studied and be extended to be write-aware.

The research problems we address can be summarized as follows:

- Problem 1: derive an off-line, optimal algorithm O, such that, for any given cache C and any page request trace X corresponding to C, using O to manage C generates no more I/Os, including both reads and writes, than any other algorithm for managing C
- Problem 2: given some on-line, write-oblivious algorithms, such as LRU and ARC, extend them to be write-aware, such that, for any given cache C and any page request trace X corresponding to C, using the extended write-aware algorithms to manage C generates fewer I/Os, including both reads and writes, than the original write-oblivious algorithms.

To solve the above research problems, we decide to adopt the same strategy used in developing the write-oblivious algorithms under the read-only workload model. First we characterize an off-line, optimal algorithm, which minimizes the total number of I/Os under the read-write workload model, then we extend some write-oblivious algorithms to be write-aware by imitating the replacement decisions of the optimal algorithm.

5.4 Partial Characterization of an Optimal Algorithm

In this section, we give a partial characterization of an optimal algorithm under the read-write workload model, with the objective of minimizing the total number of I/Os. First we give some preliminary definitions.

5.4.1 Preliminary Definitions

We give some preliminary definitions which are used in the characterization of an optimal cache replacement algorithm.

5.4.1.1 Cache Replacement Policy

We define a cache replacement policy as follows:

Definition 5.1 *Given a cache C and a request trace X received by C, a cache replacement policy $P = p_1, p_2, ..., p_L$ is a finite sequence of elements with the same length L as trace X, and with element p_t being the page replaced (evicted) by the cache at time t. p_t is determined by the cache size, trace X, and the replacement algorithm used to manage C. $p_t = \phi$ if the cache does not replace a page at time t, otherwise $p_t \in A$.*

We define a cache state as follows:

Definition 5.2 *We denote the state of cache C serving request trace X at time t as B_t, $B_t \subseteq A$, and we define it as the content of the cache at time t after the request x_t has been served and the cache replacement p_t has been made.*

We define a valid cache replacement policy as follows:

Definition 5.3 *A cache replacement policy P is valid if and only if $(p_t \neq \phi) \Rightarrow (p_t \in B_{t-1})$, for all $1 \leq t \leq L$.*

We define a demand cache replacement policy as follows:

Definition 5.4 *A cache replacement policy P is a demand policy if and only if the following requirements are satisfied:*

- $(z_t \neq dsync) \Rightarrow (y_t \in B_t)$
- $(a \in B_t) \Rightarrow (a \in B_{t-1})$ *or* $(a = y_t)$, *for any page* $a \in A$
- $(p_t = \phi) \Leftrightarrow (y_t \in B_{t-1})$ *or* $(z_t = dsync)$
- $(p_t \neq \phi) \Leftrightarrow (y_t \notin B_{t-1})$ *and* $(z_t \neq dsync)$ *and the cache is full at time* $t - 1$

for all $1 \leq t \leq L$.

Typically, a cache replacement policy is the realization of a replacement algorithm and a particular page request trace received by the cache. All the cache replacement policies discussed in this research are valid demand policies unless it is stated otherwise.

5.4.1.2 I/Os Generated by a Page Request

Given a cache C, which is managed by using replacement policy P, and a read-write workload X received by C, we denote the I/Os generated by request x_t as $n_{X,P}(x_t)$ and calculate it as follows:

- if the request type $z_t = dsync$ and the requested page y_t is dirty, then a write I/O is needed to synchronize the page to lower level storage and $n_{X,P}(x_t) = 1$. These are the writes labeled on the arc from state P to state C in Figure 5.1.
- if the request type $z_t = read$ or $z_t = write$ and the requested page y_t is not in cache, then $n_{X,P}(x_t)$ equals the sum of the following I/Os:
  - the I/O required to bring the requested page, y_t, to the cache. If $z_t = read$ then a read I/O is required to bring the page into the cache, else no I/O is required. These are the reads labeled on the arc leading to state C in Figure 5.1.
  - the I/O to synchronize the replaced page p_t to lower level storage. If p_t is dirty then a write I/O is required, else no I/O is required. These are the writes labeled on the arc from state D and P to state E in Figure 5.1.
- in all other cases $n_{X,P}(x_t) = 0$

In addition, we denote the total number of I/Os generated by the cache, which is managed by policy P, as $N_{X,P}$, and calculate it as follows:

$$N_{X,P} = \sum_{t=1}^{L} n_{X,P}(x_t) \tag{5.1}$$

5.4.1.3 Avoidable I/Os and Unavoidable I/Os

Among the total number of I/Os issued by a second-tier cache, we observe that some of the I/Os can be avoided by using replacement algorithms, but the other I/Os are not avoidable regardless of which algorithm is used to manage the cache. Under the read-write workload model, the I/Os generated by a second-tier cache can be categorized into two separate sets, avoidable I/Os and unavoidable I/Os, based on their avoidability and their definitions are given as follows:

Definition 5.5 *We define avoidable I/Os and unavoidable I/Os as:*

- *unavoidable I/Os: some of the I/O requests generated by a second-tier cache are unavoidable regardless of which replacement algorithm is used to manage the cache. These I/Os include:*

- *first time reads: when a page is accessed for the first time in the request trace, and the request type is a read, then the requested page needs to be fetched from the lower level storage by a read I/O.*
- *writes to synchronize pre-clean pages to lower level storage*

- *all other I/Os issued by a second-tier cache are categorized as avoidable I/Os*

In Lemma 5.1 we claim that the unavoidable I/Os categorized above are independent of the cache replacement algorithm and thus are unavoidable.

Lemma 5.1 *The unavoidable I/Os, which include first time reads and writes of pre-clean pages, are independent of the cache replacement algorithm. They will occur regardless of which algorithm is used to manage the cache.*

The proof of the correctness of Lemma 5.1 can be found in Appendix A.

5.4.1.4 Total I/O Cost of a Page Replacement

We observe that during a page replacement there are two factors determining the number of I/Os generated by the cache. These factors are:

- The clean or dirty state of the page being replaced: Replacing a dirty page generates one write I/O because the content of the dirty page needs to be synchronized to the lower level storage before its space can be used by a new page. In contrast, replacing a clean page does not need such a write.
- The type of the request being served by the cache: A cache replacement means there is a cache miss occurring. In such a case, a read request received by the cache causes a read I/O to be issued to the lower level storage to fetch the missing page. In contrast, a write request does not cause an I/O to fetch the page because the page content is carried from the first-tier cache.

Based on these observations we conclude that replacing a single page generates the following I/O costs at two different time points:

- The I/O cost to synchronize the evicted page to disk, at the time of the page's replacement. It can be either zero or one, depending on whether the page is clean or dirty, respectively.
- The I/O cost to bring the replaced page back into the cache, at the time when the page is next referenced (if applicable). It can be either zero or one, depending on whether the page's next request is a write or a read, respectively. If the page is not being referenced again, then it requires zero I/O to bring it back into the cache.

Now we give the definition of the *total I/O cost* of a page replacement:

Definition 5.6 *Given a cache C, page request trace X, and a cache replacement policy P, we define the total I/O cost to replace page p at time t to be the sum of the number of I/Os required to synchronize the page to disk and the number of I/Os to bring the page back into the cache when it is referenced next time by a read or write request (if applicable). The total I/O cost to replace page p at time t is denoted as $c_X(p, t)$ and is calculated as:*

- *if p is clean or is in pre-clean state*
  - *if the next non-dsync request to p is a read, then the total I/O cost to replace the page is one, which is caused by the next read request. (Clause 1)*
  - *if the next non-dsync request to p is a write or null (i.e., no more read or write requests to the page), then the total I/O cost to replace the page is zero (Clause 2)*
- *if p is in dirty but not pre-clean state*
  - *if the next non-dsync request to p is a read, then the total I/O cost to replace the page is two, which is caused by both synchronizing the dirty page to disk and reading it back into the cache. (Clause 3)*
  - *if the next non-dsync request to p is a write or null (i.e., no more read or write requests to the page), then the total I/O cost to replace the page is one, which is caused by synchronizing the dirty page to disk. (Clause 4)*

If there is no replacement at time t then $p_t = \phi$. We define $c_X(\phi, t) = 0$. (Clause 5)

We denote the sum of the total I/O cost generated at each page replacement (by using policy P to manage the cache) as O_P and calculate it as:

$$O_P = \sum_{t=1}^{L} c_X(p_t, t) \tag{5.2}$$

where L is the length of the request trace and p_t is the victim page (which is ϕ in case of a cache hit) being evicted at time t according to the replacement policy P. We denote O_P as the *total number of avoidable I/Os* generated by the cache and make the following claim:

Lemma 5.2 *The result of Equation 5.2, O_P, is equal to the total number of avoidable I/Os generated by a cache which is managed using policy P.*

The proof of the correctness of Lemma 5.2 can be found in Appendix B.

The result of Equation 5.2 (i.e., O_P) includes only a portion of the *total number of I/Os* generated by the cache (which is denoted as $N_{X,P}$ and introduced earlier). However, because O_P takes all the I/Os dependent on the cache replacement algorithm into account, it can be used as a metric to evaluate the performance of different algorithms.

5.4.2 Partial Characterization of an Optimal Algorithm

In this section we present several lemmas that describe the characteristics of an optimal algorithm and give their proofs in the appendices. First, we give some preliminary definitions:

Definition 5.7 *We denote the* re-reference distance *of page a at time t in trace X as $d_X(a,t)$, and define it as the distance, in terms of page requests, between time t and the time when the page a is next referenced by a read or write request. If the page will not be read or written again then the distance is defined to be infinite.*

We propose a lemma, Lemma 5.3, to characterize an optimal cache replacement algorithm and give the proof of the lemma in Appendix C.

Lemma 5.3 *Let X be a page request trace, B_0 and B'_0 be two different initial cache states where:*

$B'_0 = T_0 + \{a\}$

$B_0 = T_0 + \{b\}$

for $T_0 \subseteq A$, $a, b \notin T_0$, $c_X(a,0) > 0$, and $c_X(b,0) = 0$. We claim that for any demand policy P, corresponding to X and B_0, there must exist a demand policy P', corresponding to X and B'_0, such that $O_{P'} \leq O_P$.

Based on Lemma 5.3 we propose Theorem 5.1 as follows:

Theorem 5.1 *Let C be a cache, X be a page request trace received by C, B_0 be an initial state of C, P be a valid demand policy for C, X, and B_0. Suppose at time t the policy P makes a decision to replace page a with $c_X(a,t) > 0$, and at the same time there is another cached page b with $c_X(b,t) = 0$. Then there must exist another valid demand policy P' such that $p'_t = b$ and $O_{P'} \leq O_P$.*

The proof of Theorem 5.1 can be found in Appendix D. Based on the theorem we can conclude that it is always an optimal decision to replace a page p at time t with $c_X(p,t) = 0$, rather than a page with positive total I/O cost.

We propose another lemma, Lemma 5.4, to characterize an optimal cache replacement algorithm. The proof of Lemma 5.4 highly resembles the proof of Lemma 5.3 (which is presented in Appendix C) and thus is omitted in this thesis.

Lemma 5.4 *Let X be a page request trace, B_0 and B'_0 be two different initial cache states where:*

$B'_0 = T_0 + \{a\}$

$B_0 = T_0 + \{b\}$

for $T_0 \subseteq A$, $a, b \notin T_0$, $c_X(a, 0) = 0$, and $c_X(b, 0) = 0$. We claim that for any demand policy P, corresponding to X and B_0, there must exist a demand policy P', corresponding to$X$ and B'_0, such that $O_{P'} = O_P$.

Similar to Theorem 5.1, we propose another theorem, Theorem 5.2, based on Lemma 5.4. Also, because the proof of the theorem is trivial and highly resembles the proof of Theorem 5.1, it is omitted in this thesis. Based on Theorem 5.2 and Theorem 5.1, we can conclude that if there are several pages with zero total I/O cost, it does not matter which of these pages is replaced.

Theorem 5.2 *Let C be a cache, X be a page request trace received by C, B_0 be an initial state of C, P be a valid demand policy for $C, X, and B_0$. Suppose at time t the policy P makes a decision to replace page a with $c_X(a, t) = 0$, and at the same time there is another cached page b with $c_X(b, t) = 0$. Then there must exist another policy P' such that $p'_t = b$ and $O_{P'} = O_P$.*

The last lemma proposed by us to characterize an optimal algorithm is Lemma 5.5. Its proof can be found in Appendix E.

Lemma 5.5 *Let X be a page request trace, B_0 and B'_0 be two different initial cache states where:*

$B'_0 = T_0 + \{a\}$

$B_0 = T_0 + \{b\}$

for $T_0 \subseteq A$, $a, b \notin T_0$, $d_X(a, 0) < d_X(b, 0)$, and $c_X(a, 0) \geq c_X(b, 0) > 0$. For any demand policy P, corresponding to X and B_0, there must exist a demand policy P', corresponding to X and B'_0, such that $O_{P'} \leq O_P$.

We propose another theorem, Theorem 5.3, based on Lemma 5.5. Because the proof of the theorem is trivial and highly resembles the proof of Theorem 5.1, it is omitted in this thesis.

Theorem 5.3 *Let C be a cache, X be a page request trace received by C, B_0 be an initial state of C, P be a valid demand policy for $C, X, and B_0$. Suppose that at time t the policy P makes a decision to replace page a with $c_X(a, t) > 0$, and at the same time there is another cached page b with $d_X(a, 0) < d_X(b, 0)$ and $c_X(a, 0) \geq c_X(b, 0) > 0$. Then there must exist another policy P' such that $p'_t = b$ and $O_{P'} \leq O_P$.*

Based on Theorem 5.3, we can conclude that when a valid demand policy makes replacement decisions between two cached pages, it is always an optimal decision to replace the page with longer re-reference distance and equal or lower total I/O cost.

Based on the lemmas and theorems we can conclude how an optimal algorithm behaves in many cases. At any given time t, assuming that the cached page with the longest re-reference distance is page x (if there are multiple pages with infinite re-reference distance then choose either one of them as x), the cache must be in one of the following four states:

- Case A: There is a set of cached pages with the same total I/O cost of zero. In this case the optimal algorithm evicts one of the pages in the set during the next replacement. The replaced page can be any one of the pages in the set. (based on Lemma 5.3 and Lemma 5.4)
- Case B: All the cached pages have positive total I/O cost
  - Case B1: The total I/O cost of page x is $c_X(x,t) == 1$, then the optimal algorithm evicts x (based on Lemma 5.5)
  - Case B2: It must be $c_X(x,t) == 2$
    - Case B2a: All the pages in cache have the same I/O cost, which is two, then the optimal algorithm evicts page x (based on Lemma 5.5)
    - Case B2b: else this is the difficult case, which is not described by the lemmas. We do not have the knowledge about how an optimal algorithm behaves in this case.

Thus, the lemmas can determine the behavior of an optimal off-line algorithm in all of the cases, except Case *B2b*. Please note that we will continue to discuss the above cases in the following sections. In the next section we present the design of write-aware algorithms that imitate the behavior of an optimal off-line algorithm.

5.5 Write-aware Replacement Algorithms

In this section we present how to use an optimal replacement algorithm's characteristics presented in the previous section to design write-aware algorithms.

5.5.1 A Write-aware Off-line Algorithm

First, we present a write-aware off-line replacement algorithm, which is illustrated in Algorithm 5.1 and is referred to as *PRO-IO*. The PRO-IO algorithm makes

replacement decisions primarily based on the total I/O cost associated with cached pages. Since the behavior of an optimal algorithm is partially characterized in the previous section, we let the PRO-IO algorithm behave optimally whenever possible. In the difficult case, Case B2b, we let the PRO-IO algorithm imitate the original MIN algorithm, which evicts the page referenced furthest in the future by a read or write request. Although this particular replacement decision may not be optimal, it is used by the MIN to maximize the total number of cache hits under the read-only workload model, and is a reasonable choice in this difficult case.

Algorithm 5.1 The PRO-IO Cache Replacement Algorithm

1: /*assume the cache is full at time t and a cache replacement is required */
2: **if** there is a page y in the cache and $c_X(y, t) == 0$ **then**
3:    evict page y /* evict either of the pages with zero total I/O cost */
4: **else**
5:    evict the cached page with the longest re-reference distance
6: **end if**

Please note that for the entire Case B, the PRO-IO algorithm evicts the same victim page, which is the page being read or written furthest in the future. However, the decision is based on different rationales for different sub-cases. In Case B1 and B2a, the decision is based on Lemma 5.5. In Case B2b, which is the difficult case, the decision is based on our choice to let the PRO-IO algorithm imitate the existing MIN algorithm.

In the next section we will show empirically that the PRO-IO algorithm outperforms the MIN-RW algorithm in terms of minimizing the total number of I/O operations.

5.5.2 Write-aware On-line Algorithms

In reality, only on-line algorithms can be used to manage the caches. In this section we show how several existing write-oblivious on-line algorithms can be extended to be write-aware. We consider the following three write-oblivious algorithms:

- LRU, which is a purely recency-based algorithm
- ARC, which uses both recency and frequency to make replacement decisions
- TQ, which is proposed in Chapter 3 and uses I/O types to make replacement decisions

The current clean/dirty state of each cached page is known to an on-line algorithm. However, the type and the time of a page's next reference is unknown. To extend on-line replacement algorithms to be write-aware, we need to rely on some technique to predict the next reference type of the pages being accessed in

the workload. We have used the technique presented in Chapter 3 (Sections 3.2 and 3.3), which uses the type of a page's most recent request to predict the *type* of the page's next request. We also extend each on-line algorithm using its original technique, (based on recency, frequency, or I/O types) to predict the pages' next reference *time*.

In many cases (Case A and CaseB1), the total I/O cost to replace a page is a more important factor than the next reference time of the page in making replacement decisions. The next reference time is only used to break ties if multiple pages have the same positive total I/O cost (Case B2a) or in the difficult case (Case B2b).

Write-aware LRU

With the above observations in mind, we extend the existing write-oblivious LRU algorithm to include additional data structures, i.e., four sorted lists that contain the *clean* pages currently in cache. After the extension, the LRU list contains only the dirty pages. Each of the four lists being added corresponds to a particular request type, and the clean pages are placed into one of the lists according to the type of their most recent non-dsync requests. Within each type list, the pages are sorted based on the time of their most recent non-dsync requests and are organized in a tree-structure. The computational complexity is $O(logN)$ where N is the number of pages in the list. The reason to use a tree structure to sort the pages is that the pages may move between the lists when they are requested, and the sorting is based on each page's exact request time (not the relative order). The four request types are:

- READ, which predicts that the page's next request is *probably* a write
- RECOV write, which also predicts that the page's next request is *probably* a write, but with a probability lower than that indicated by a READ
- REPLACE write, which also *may* be followed by a write request, but with a probability lower than that indicated by a RECOV write
- SYNCH write, which also *may* be followed by a write request, but the probability is the lowest among those of the four request types

The rationales behind these predictions are based on our knowledge of DBMS bufferpool management, which have been introduced in Chapter 2 and Chapter 3. As we will see in the evaluation section, the accuracy of these predictions may vary according to database configurations and, of course, this is not the only approach to make these predictions. Other techniques can also be used to achieve the same functionality.

We show the detail of the write-aware LRU algorithm, which is referred to as LRU-IO, in Algorithm 5.2. The LRU-IO algorithm searches and evicts clean pages

in the four type lists in the following order: READ list, RECOV list, REPLACE list, and SYNCH list. Because the pages in these lists are clean pages, therefore, the total I/O cost of replacement for these pages is zero or one. The decision to evict the clean pages (in the four type-based clean lists) before dirty pages, and in the above order, is consistent with the behavior of the optimal algorithm described by Lemma 5.3 and Lemma 5.4 (to evict zero-cost pages whenever possible).

We can see that the LRU-IO algorithm favors clean pages over dirty pages in page replacement (by evicting clean pages first). As a result, it may evict a clean page with its next request as read (therefore the page's total I/O cost is one) *before* a dirty page with its next request as write (the page's total I/O cost is one, too). Making such a replacement may be inconsistent with the optimal algorithm's behavior described in Lemma 5.5, which is, if all the cached pages have the same positive total I/O cost, then the optimal algorithm must evict the one being referenced furthest in the future (which may not be a dirty page). However, we argue that for the following reasons the LRU-IO's decision to favor clean pages is reasonable:

- We observe that it is rare for a second-tier cache to enter the state described by Lemma 5.5, in which all the cached pages have the same positive total I/O cost. Typically, a second-tier cache contains both clean and dirty pages, and the cached page's next request type can be either read, write, or dsync. In such a common scenario the cached pages have different total I/O costs.
- LRU-IO is an on-line algorithm thus it cannot determine the total I/O cost for the cached pages. When there is a clean page in the cache, we decide to use a *greedy* approach, which evicts the clean pages (with zero or one total I/O cost) before the dirty pages (with one or two total I/O cost), for LRU-IO to make replacement decisions.

Based on the above observations we design the LRU-IO algorithm to emphasize the optimal algorithm's behavior described by Lemma 5.3 and to favor clean pages in replacement. If the LRU-IO algorithm cannot find a clean page in the cache, then it replaces the page at the LRU end of the LRU list (holding the dirty pages), which, according to the LRU algorithm, will be accessed furthest in the future. This decision is consistent with the optimal algorithm's behavior described by Lemma 5.5 (in Cases B1 and B2a, to evict the page with the longest re-reference distance).

Write-aware ARC

We extend the ARC algorithm [64] to be write-aware by using a similar methodology. The reason that we choose ARC rather than MQ (used in the evaluations of Chapter 3) is that the MQ algorithm, as we have introduced in Chapter 3, requires users to pre-configure a few important parameters. Furthermore, once those parameters are set, they do not automatically adapt to the changing workloads. On

Algorithm 5.2 LRU-IO Algorithm

```
LRU-IO(p : page access)
 1  /* assume the cache is full and it receives a request accessing page p */
 2  if this is a dsync request of p
 3     then if p is in the LRU list /* p is in cache and dirty */
 4             then move p to one of the four type-based clean lists based on p.lastRequestType
 5                  and insert it at the right position based on p.lastRequestTime
 6                  /* the pages are sorted based on their p.lastRequestTime in each clean list */
 7
 8           return /* nothing more to be done */
 9
10  /* this is a read or write request of p */
11  reset p.lastRequestTime = currentTime and p.lastRequestType = currentRequestType
12  if p is in cache /* cache hit */
13     then if p is in one of the four type-based clean lists /* p is clean */
14             then if this is a read request /* p remains clean */
15                     then move p to the right position in the READ clean list
16                          according to p.lastRequestTime
17                     else  /* this is a write and p becomes dirty */
18                          remove p from the clean list and add it to the MRU end of the LRU list
19
20             else  /* p is dirty and currently in the LRU list */
21                  move p to the MRU end of the LRU list
22
23     else  /* cache miss, need a cache replacement */
24           search a clean page in the four type-based clean lists in the following order:
25           READ list, RECOV list, REPLACE list, and SYNCH list
26           if a clean page is found in a clean list
27              then remove the page with the oldest lastRequestTime from the
28                   clean list and the cache
29              else  /* no clean page found in the cache, fall back to LRU */
30                   synchronize the page at the LRU end of the LRU list to disk and
31                   remove the page from the LRU list and the cache
32
33           /* now add p into the cache */
34           if this is a read request /* p is a clean page */
35              then insert p into the right position in the READ clean list
36                   according to p.lastRequestTime
37              else  /* this is a write request and p is dirty */
38                   add p to the MRU end of the LRU list
39
40
```

the contrary, ARC algorithm does not have parameters and it can adapt itself (as we will see in the following paragraph) based on the data access patterns in the changing workloads.

Originally, ARC uses two separate queues, a recency based queue and a frequency based queue, to manage the cached pages. The cache space allocation (i.e., the sizes of the two queues) are dynamically adjusted based on the workload. If the re-reference distances of the pages are relatively short, which means a strong temporal locality in the workload, the size of the recency based queue will be increased. In contrast, if the temporal locality is hard to exploit the size of the frequency based queue will be increased.

In ARC, the recency queue, as well as the frequency queue, are both LRU queues. For each of these two queues, we add four I/O type-based clean page lists as we did to the LRU algorithm, and move the clean pages into these lists according to the pages' most recent non-dsync requests. The two original LRU queues, i.e., the recency queue and frequency queue, contain only dirty pages. While making a replacement decision, the extended ARC algorithm, which we refer to as ARC-IO, will search for a clean page, first in the recency queue, then in the frequency queue. The searches are conducted in the same order among the type lists as that described in Algorithm 5.2. If no clean page is found in the cache, the ARC-IO algorithm will fall back to the original ARC and will make replacement decisions accordingly. The detail of ARC-IO is shown in Algorithm 5.3.

Write-aware TQ

The TQ algorithm, as proposed in Chapter 3, makes replacement decisions primarily based on the I/O types of the pages' most recent references. This fact makes our extension work easier because we do not need to create clean list for each of the request types. The extended write-aware TQ algorithm, which we refer to as TQ-IO, searches for a clean page starting from the high priority queue, and then from the low priority queue. We create a clean page list for each of the two priority queues. The pages in the clean list are sorted based on the same method used to sort the pages in the corresponding priority queue (i.e., using the LPR algorithm for the high priority queue and using the LRU algorithm for the low priority queue). If no clean page is found in the cache, the algorithm will fall back to the original TQ and will make replacement decisions accordingly.

In the following section we evaluate the performance of the extended write-aware algorithms and compare them with their original write-oblivious counterparts.

5.6 Evaluations

In the previous section we presented extensions to write-oblivious replacement algorithms to make them write-aware. The objective is to reduce the total number

Algorithm 5.3 ARC-IO Algorithm

```
1: /*assume the cache is full and requires a page replacement */
2: if there is a clean page in the recency based queue then
3:    if there is a page in the READ clean list of the recency based queue then
4:       evict page that has been read or written least recently
5:    else if there is a page in the RECOV clean list of the recency based queue
      then
6:       evict page that has been read or written least recently
7:    else if there is a page in the REPLACE clean list of the recency based queue
      then
8:       evict page that has been read or written least recently
9:    else
10:      /* there must be a page in the SYNCH clean list */
11:      evict page that has been read or written least recently
12:   end if
13: else if there is a clean page in the frequency based queue then
14:   if there is a page in the READ clean list of the frequency based queue then
15:      evict page that has been read or written least recently
16:   else if there is a page in the RECOV clean list of the frequency based queue
      then
17:      evict page that has been read or written least recently
18:   else if there is a page in the REPLACE clean list of the frequency based
      queue then
19:      evict page that has been read or written least recently
20:   else
21:      /* there must be a page in the SYNCH clean list */
22:      evict page that has been read or written least recently
23:   end if
24: else
25:   /* there is no clean page in the cache */
26:   evict a page according to the original ARC algorithm
27: end if
```

Parameter	Value	Default Value	Description
innodb_file_io_threads	4	4	number of threads working on bufferpool page cleaning
innodb_buffer_pool_size	480MB	384MB	bufferpool size
innodb_log_file_size	60MB	total log space be 25%-100% of bufferpool size	single log file size
innodb_log_files_in_group	2	-	number of log files, works together with innodb_log_file_size to determine the total log space size

Table 5.1: MySQL Database Configuration Parameters

of I/Os generated by the second-tier cache. In this section, we evaluate the effectiveness of the write-aware algorithms in achieving the objective. We expect to see that, compared with their original write-oblivious counterparts, the write-aware algorithms can significantly reduce the total number of I/Os.

We conduct our evaluations by cache simulations. The extended algorithms, as well as their original write-oblivious counterparts, are used to manage the simulated second-tier cache, which is driven by a page request trace issued from a database bufferpool. The total number of I/Os generated by the second-tier cache (including both avoidable and unavoidable I/Os) is monitored for performance comparison.

5.6.1 Evaluations by Using MySQL Bufferpool I/O Traces

As described in Chapter 4, we modified the MySQL InnoDB storage engine to use deferred synchronization of REPLACE bufferpool writes. With this modification, the REPLACE writes issued by MySQL become unsynchronized. We collect several bufferpool I/O traces issued to the lower level storage and use them to drive the cache simulations. The traces include different types of requests, such as reads, RECOV writes, REPLACE writes, and dsync requests.

5.6.1.1 Setup of Experiments

To collect the bufferpool I/O traces, we run a TPC-C workload kit against the modified MySQL. The scale factor of the workload is set to be 10 (SF=10) and the number of simulated users is set to be 100. The database size is about 1 GB and it increases (between 10% to 50% depending on the database configuration) while the workload is running. The bufferpool I/Os issued by MySQL are recorded in a trace file after the bufferpool is warmed up. Some important database configuration parameters are shown in Table 5.1, and a summary of the trace files is shown in Table 5.2.

5.6.1.2 The Baseline Case

In Figure 5.4 we show the results of the baseline case, in which we use the MySQL trace file *480mbuf_120mlog*. In this case, the bufferpool size is set to be about 50%

Trace Name	Bufferpool Size	Number of Requests	SYNCH Writes	REPLACE Writes	RECOV Writes	Reads	Dsyncs
120mbuf_120mlog	120 MB	47786295	0.00%	13.22%	0.00%	85.15%	1.63%
240mbuf_120mlog	240 MB	50176851	0.00%	26.77%	0.01%	68.23%	4.99%
480mbuf_120mlog	480 MB	17679140	0.00%	36.49%	8.64%	26.08%	28.79%
960mbuf_120mlog	960 MB	19498505	0.00%	0.82%	48.83%	0.71%	49.64%

Table 5.2: MySQL Bufferpool I/O Request Traces

of the initial database size.

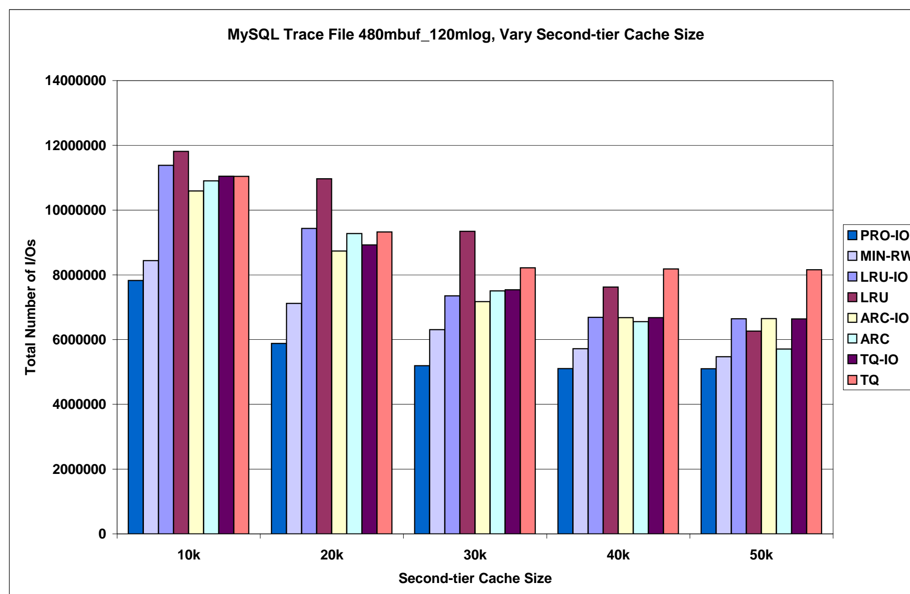


Figure 5.4: The Baseline Case, Vary Second-tier Cache Size

We can make the following observations from the results:

- As the second-tier cache size increases, the total number of I/Os generated by the cache decreases, which is a reasonable result
- The PRO-IO offline algorithm, although not necessarily optimal, outperforms other off-line and on-line algorithms. It shows the potential room for performance gains by the other replacement algorithms.
- By using the proposed approach, the extended write-aware on-line algorithms outperform their write-oblivious counterparts in most of the cases. However, the difference is not as significant as we expected. In some cases, e.g., when the simulated cache sizes are set to 40k and 50k, the extended ARC-IO algorithm underperforms the write-oblivious original ARC. We will discuss this further in the next part of this section.

5.6.1.3 Sensitivity Analysis, Varying MySQL Bufferpool Size

We conduct a sensitivity analysis by varying the size of the first-tier cache, i.e., the database bufferpool, to be about 12%, 24%, 48%, and 96% of the initial database size. We use several MySQL trace files, namely 120mbuf_120mlog, 240mbuf_120mlog, 480mbuf_120mlog, and 960mbuf_120mlog. The evaluation results are presented in Figure 5.5.

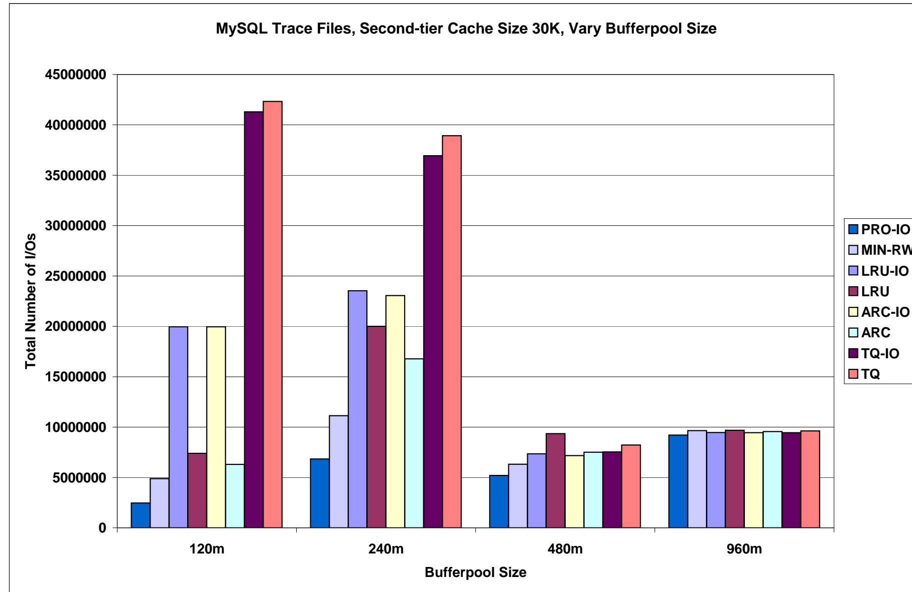


Figure 5.5: Sensitivity Analysis, Vary MySQL Bufferpool Size

We can make the following observations from the results:

- As the the bufferpool size increases, the total numbers of I/Os generated by the cache decreases. This is a reasonable result because the bufferpool can absorb more page accesses when its size enlarges. Consequently, fewer pages will need their requests to be served by the second-tier cache, and the cache will generate fewer I/Os.

- As in the baseline evaluation, the PRO-IO algorithm outperforms other algorithms, including the MIN-RW algorithm, in all the cases

- When the bufferpool size reaches an extreme large value, i.e., 960MB (about 96% of the initial database size), all the algorithms being studied have similar performance. This is because in this extreme case the database bufferpool captures most of the page accesses. Consequently, most of the I/Os received by the second-tier cache are unavoidable I/Os (i.e., first time reads and RECOV writes).

trace file	read	rd_rd	rd_rec	rd_rep	replace	rep_rd	rep_rec	rep_rep
120mbuf	40690598	39706684	5	956206	6318625	956977	34	5234219
240mbuf	34236419	31412376	150	2788925	13434720	2791728	884	10362467
480mbuf	4610302	2233690	59121	2286694	6451274	2313573	122498	3577523
960mbuf	137631	13202	120092	115	160794	3467	105801	45361

Table 5.3: MySQL Trace Analysis

- Using the proposed approach does not bring significant benefits to the write-aware on-line algorithms. In some cases, for example, when the bufferpool size is set to be comparably small (120MB or 240MB), the write-oblivious LRU and ARC algorithm even outperform their extended write-aware counterparts.

To explain why using the proposed approach (i.e., to imitate the optimal algorithm) does not improve cache performance as expected, or even hurts it, we conduct another analysis on the MySQL traces. The results are summarized in Table 5.3.

In Table 5.3, we show the numbers of the READs and REPLACE writes, in columns *read* and *replace*, respectively, for each trace. The number of times that a READ request is followed by another request type, namely READ, RECOV, and REPLACE, are shown in columns *rd_rd*, *rd_rec*, and *rd_rep*, respectively. Similar statistics for REPLACE writes can also be found in the table.

From these statistics we can see that the technique adopted by us to predict a page's next reference's type (by using the page's previous request type, which is proposed in Chapter 3) does not work well for the MySQL traces. For example, according to the technique, the next reference type following a READ request is *probably* a write. However, in the *120mbuf* case, that particular probability is only 2.35%. For another example, according to the technique, the next reference type following a REPLACE write is *probably* a read. However, in the *240mbuf* case, this particular probability is only 20.78%. To summarize, the adopted technique fails to predict the pages' next reference types for these MySQL traces. Consequently, without reasonably accurate predictions, the write-aware on-line algorithms cannot determine the total I/O cost for the cached pages, and their imitations of the optimal algorithm are based on the inaccurate information. In such a scenario, using the proposed approach cannot generate performance gains.

The above observations contradict those we made in the research presented in Chapter 3, in which the proposed technique is used to predict the next reference types for the cached pages and the evaluations show promising performance gains. Those experiments used traces from DB2, rather than MySQL. We conduct a similar analysis for a group of DB2 traces and show the results in Table 5.4 and Table 5.5. Based on these statistics, we can see that although the request type predictions are not completely accurate for these traces, the accuracy is much improved compared to that of the MySQL traces. For example, in all five DB2 trace files, a READ request predicts that there is *at least* a 63% possibility (in the 540k_4000 case) that the following request to the same page is a write (either a RECOV or a REPLACE).

Trace Name	Bufferpool Size in blocks	`softmax`	Number of Requests	SYNCH Writes	REPLACE Writes	RECOV Writes	Reads
60k_4000	300K (234 MB)	4000	42097520	0.05%	47.61%	0.00%	49.99%
300k_4000	300K (1.1 GB)	4000	43894440	0.00%	59.70%	0.57%	32.55%
540k_4000	300K (2.1 GB)	4000	72019329	0.03%	70.66%	0.55%	16.26%
300k_40	300K (1.1 GB)	40	56214748	0.00%	0.00%	42.89%	14.21%
300k_400	300K (1.1 GB)	400	43894440	0.00%	45.44%	2.21%	24.96%

Table 5.4: DB2 Bufferpool I/O Request Traces

trace file	read	rd_rd	rd_rec	rd_rep	replace	rep_rd	rep_rec	rep_rep
60k_4000	21045911	3097939	78	17534944	20042022	17409695	5	2035818
300k_4000	11155825	1437187	85592	9046763	20458269	9188692	52237	10607453
540k_4000	11711097	2543566	34961	7391702	50889685	8579872	60877	40546534
300k_40	7990047	898921	6652061	15	144	33	111	0
300k_400	10954211	1383230	188565	8808869	19943738	8974373	143084	10231643

Table 5.5: DB2 Trace Analysis

We reviewed the MySQL InnoDB storage engine code to find why the approach proposed in Chapter 3 does not accurately predict request types in above evaluations. We found that, compared with DB2, MySQL behaves differently in its bufferpool management:

- MySQL InnoDB storage engine uses the original LRU algorithm, which is purely based on recency, to manage page replacement. Compared with a generalized Clock algorithm, which is used by DB2 and is based on both recency and frequency, the LRU algorithm is disadvantaged and makes more *wrong* replacement decisions (i.e., evicts the pages being accessed sooner). Its disadvantage is especially true when the bufferpool size is small, such as in the *120mbuf_120mlog* and *240mbuf_120mlog* trace cases. This explains why in such traces a large percentage of REPLACE writes are followed by another REPLACE write. This is because the written pages are wrongly chosen as replacement candidates and are accessed again by the queries soon after they are written.

- MySQL InnoDB implements a mechanism to aggressively perform read-ahead operations. It monitors page access patterns and makes read-ahead decisions more frequently than necessary. In contrast, DB2 implements more sophisticated read-ahead policies and only initiates the operations when necessary. A TPC-C workload rarely needs the DBMS to initiate read-ahead operations. This explains why we observe such a high percentage of READ request followed by another READ for the same pages in the MySQL traces. This is because the pages being read-ahead and put into the bufferpool are rarely accessed by the queries, and therefore will be replaced as clean pages (without writes) before their next read request (accessed by queries or being read-ahead again). This problem is especially severe when the bufferpool size is small, because the read-ahead pages cannot stay for long in the bufferpool, and will be evicted soon after they are brought into the bufferpool.

- MySQL also issues bufferpool writes more *aggressively* than DB2 does. When a page is chosen to be synchronized for either RECOV or REPLACE reasons, its neighbors, i.e., pages with adjacent logical page ids, are also synchronized to disk, regardless of their current states. In DB2 code, only the pages having reached the *page cleaning threshold* are synchronized to disk.

Based on the above observations, we can conclude that although MySQL and DB2 are designed based on similar principles, (e.g., ARIES algorithm [67], WAL protocol [10, 30], asynchronous page cleaning mechanism), their different implementations make them behave differently in issuing bufferpool I/Os. Compared with DB2, MySQL's implementation is immature and it wastes I/O bandwidth by issuing unnecessary read and write requests. This fact makes it difficult to use the approach proposed in Chapter 3 to predict the future I/O types for the bufferpool pages. In the following section we will show the evaluations based on DB2 traces.

5.6.2 Evaluations by Using DB2 Bufferpool I/O Traces

In addition to the evaluations using MySQL traces, we also used DB2 traces to drive the simulated second-tier cache. The experimental setup and the results are as follows.

5.6.2.1 Setup of Experiments

While working on the technique presented in Chapter 3, we collected I/O request traces issued by the DB2 database bufferpool, which is driven by a TPC-C workload. We used some of those traces to conduct our current evaluations. The configurations of the DB2 database and the TPC-C workload are introduced in Section 3.4 and in Table 3.1. The statistics of the DB2 traces can be found in Table 5.4 and Table 5.5.

However, because we did not modify the DB2 code to use our deferred write synchronization approach (presented in Chapter 4), these traces need to be modified to include dsync operations. DB2 issues all bufferpool writes as synchronized writes. In the DB2 manual [42], users are recommended to use raw devices to bypass lower level cache tiers in order to preserve data durability. Therefore, all bufferpool writes issued by DB2 are synchronized writes.

Each entry of the DB2 I/O trace records the metadata of an I/O request issued by the database bufferpool. The information includes the I/O types, the requested page number, and a group of Log Sequence Numbers (LSN) associated with the I/O and the requested page:

- dbLSN, which describes the most recent update made to the database bufferpool at the time the I/O request is issued

- recLSN, which describes the oldest unsynchronized update carried by the requested page, if the request is a write
- pageLSN, which describes the newest unsynchronized update carried by the requested page, if the request is a write

During our modification of the DB2 traces, we assume that the bufferpool writes are not synchronized writes. We instrument the traces by adding a set of dsync requests immediately after each RECOV write to synchronize the following pages to disks:

- the page carried by the RECOV write itself, because the pages carried by the RECOV writes need to be synchronized to disks immediately after they are issued, to guarantee the recovery time or the log space limitations to the database users
- the pages carried by previously issued REPLACE writes, that have recLSNs older than the recLSN of the page carried by the current RECOV write, and have not yet been synchronized to disk

By adding the above dsync requests, we guarantee that, even though the REPLACE writes are treated as unsynchronized writes by the second-tier cache, data durability is still preserved in case of failure.

The above trace modification procedure is not perfect. To use the deferred write synchronization approach, DB2 needs to be modified to record and to monitor the previously issued REPLACE writes, in order to track the *LSN Gap* between the oldest and the newest *unsynchronized* updates made to the bufferpool pages. This requirement was discussed in Chapter 4. This is because a page cleaning session can be triggered by an old recLSN of a page carried by a previously issued REPLACE write (instead of the recLSNs carried by the bufferpool pages). In our modification of the DB2 traces, we ignored this requirement for simplicity reasons.

However, we argue that this omission will only have a trivial impact on the placement of dsync requests in the DB2 traces. The reason is that, for a database driven by a TPC-C workload (which makes frequent updates to the data objects), the updates made to the bufferpool are distributed across a large number of pages. Therefore, the recLSNs of the pages in the bufferpool, and the recLSNs of the pages carried by the previously issued REPLACE writes, are closely interleaved in sequence. The omission of the recLSNs of the pages carried by the REPLACE writes, while monitoring the LSN Gap, will not significantly delay the synchronizations of pages.

5.6.2.2 The Baseline Case

We use the trace file *300k_4000*, which is collected by setting the bufferpool size to be 50% of the initial database size and with a relatively large LSN Gap threshold.

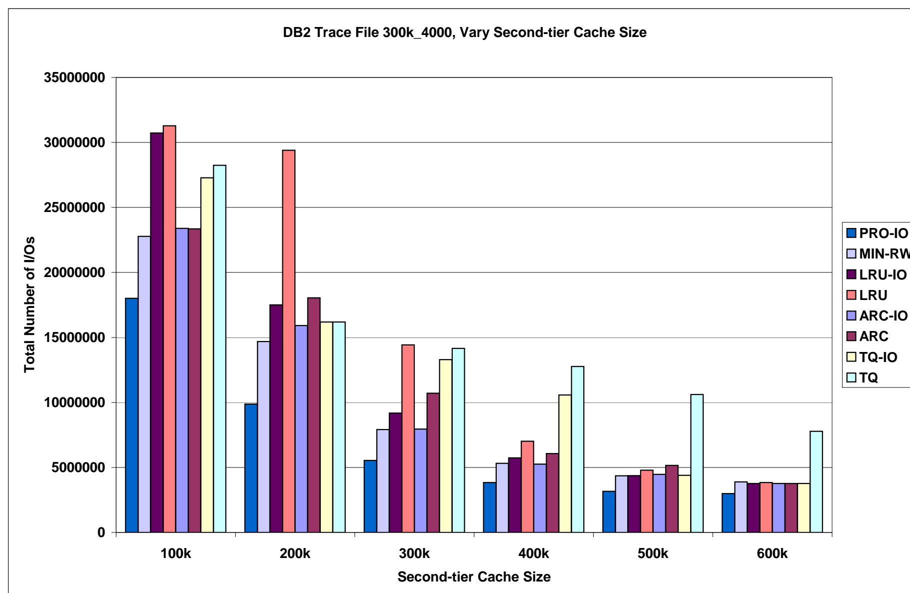


Figure 5.6: The Baseline Case, Vary Second-tier Cache Size

This means that there is a considerable percentage of unsynchronized REPLACE writes (59.7%) in the trace. The results are shown in Figure 5.6. Based on the results we can make the following observations:

- The extended write-aware on-line algorithms outperform their respective write-oblivious counterparts. The improvements (in terms of the total numbers of I/Os reduced) for the LRU, ARC, and TQ algorithms are as high as 41%, 26%, and 59%, respectively.

- The write-aware off-line algorithm, PRO-IO, outperforms all other on-line and off-line algorithms (as expected). It shows potential performance gains for other algorithms.

- As the second-tier cache size increases, the total number of I/Os generated by the cache decreases. This is reasonable because the cache can avoid more avoidable I/Os as its size gets larger.

- When the second-tier cache is extremely small, for example, 100K pages, the proposed approach is of little help. This is because most of the pages need to be evicted from the cache soon after they are cached, regardless of their replacement costs, due to the extremely small cache size. There are fewer opportunities for the proposed approach to help in this extreme scenario.

- When the cache size is extremely large, for example, 600K pages, all the algorithms perform almost the same, except for the original TQ. This is also a reasonable result. When the cache size is extremely large, most avoidable

I/Os can be saved by the algorithms, with or without using the proposed approach.

- In most of the cases (between the cases in which the second-tier cache size is extremely small or large), the write-aware algorithms outperform their write-oblivious counterparts.

- The TQ and TQ-IO algorithm perform badly in some cases (e.g. when the cache size is set to 400K pages). We analyzed the results and found that in these cases the TQ and TQ-IO algorithms generate many more eviction writes than the other algorithms do. The statistics are shown in Figure 5.7. Similar statistics that show TQ and TQ-IO generating large numbers of eviction writes can also be observed in the results of the MySQL evaluations. Because TQ algorithm was designed solely to maximize read hit ratio, it tends to evict the pages whose next request type is probably a write. This practice effectively decreases the ratio of write hits and results in more writes due to dirty page evictions. However, when the cache size is extremely large (500K and 600K pages), the TQ-IO algorithm can always find clean pages to evict and it generates small numbers of evictions writes in these cases.

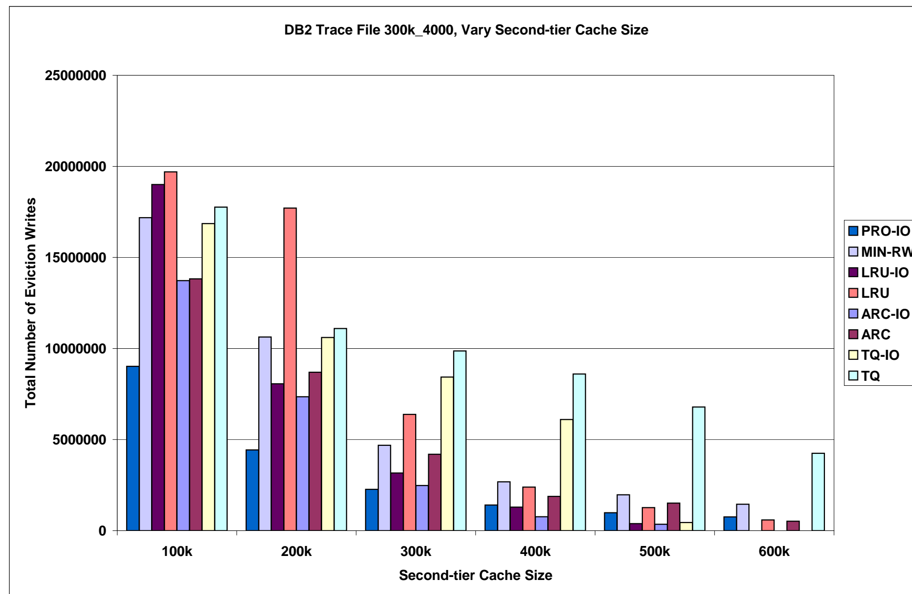


Figure 5.7: The Baseline Case, Vary Second-tier Cache Size

5.6.2.3 Sensitivity Analysis

We evaluated the sensitivity of the performance of the proposed approach to several configurable parameters. First, we vary the size of the first-tier cache, the database bufferpool, to be 10%, 50%, and 90% of the original database size, by choosing

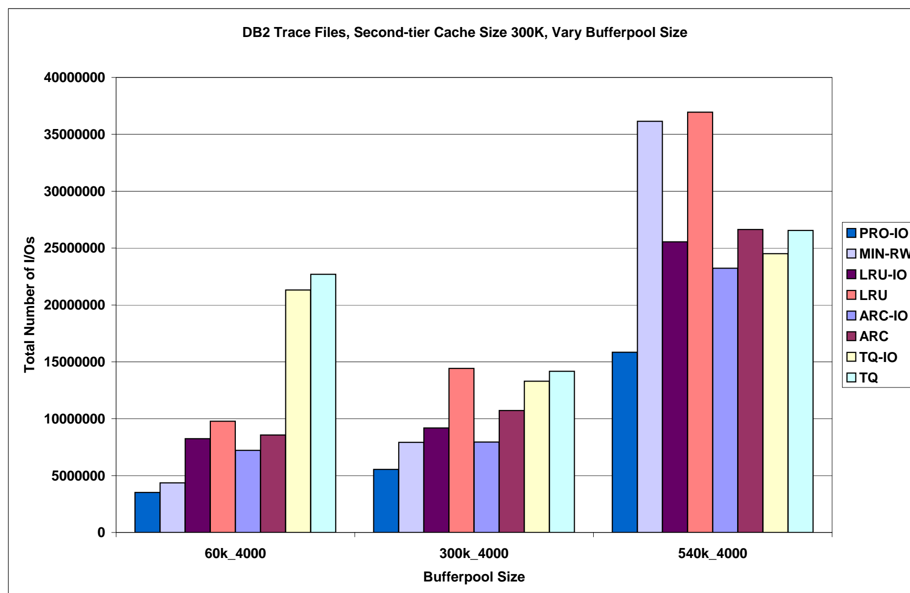


Figure 5.8: Sensitivity Analysis, Vary Bufferpool Size

specific traces corresponding to different bufferpool sizes. The simulated second-tier cache size remains at 300K pages. The results are shown in Figure 5.8. We can make the following observations based on the results:

- For on-line algorithms, the performance gaps between the write-aware and the write-oblivious algorithms (e.g., ARC/ARC-IO and TQ/TQ-IO)are not sensitive to the bufferpool size changes, except for the LRU and LRU-IO algorithms. When bufferpool size gets larger (300K and 540K), the performance gap between the write-aware LRU-IO and the write-oblivious LRU widens. This is because these two algorithms are recency-based and when the bufferpool size is extreme small (60K), the temporal locality in the request trace received by the second-tier cache can be used by both the algorithms to reduce the avoidable I/Os. However, when the bufferpool size gets larger and the temporal locality in the request trace becomes more difficult to exploit, using the proposed approach saves more I/Os for the write-aware LRU-IO.

- When the bufferpool size increases, the total number of I/Os generated by the algorithms (managing the second-tier cache) also increases in most cases. This is reasonable because as the bufferpool size increases, the DB2 database improves its performance and can process more transactions in the same amount of time. It issues more I/O requests to the trace file (the numbers are shown in Table 5.4).

- One unexpected result is that the MIN-RW algorithm performs badly in the 540k_4000 trace case. It generates more I/Os than some on-line algorithms.

	OPT-IO	MIN-RD	LRU-IO	LRU	ARC-IO	ARC	TQ-IO	TQ
TOTAL I/Os	15848056	36138625	25553429	36949588	23227747	26638983	24507090	26551668
READs	5412495	3046110	9286523	11087166	8268670	9104213	7745172	7419261
WRITEs	10435561	33092515	16266906	25862422	14959077	17534770	16761918	19132407

Table 5.6: DB2 540K_4000 Trace Case, Decomposed I/O Numbers

To find the explanation, we conduct an analysis on the results and show the statistics in Table 5.6. The table shows the total numbers of I/Os, and the the decomposed numbers of reads and writes, generated by different algorithms. We can see that, although the MIN-RW algorithm performs the best in reducing the reads, which is as expected, it generates a large number of writes which hurts its performance. The result shows that to achieve the objective of maximizing the read cache hits, the victim pages selected by the MIN-RW algorithm are mostly dirty pages. This is reasonable because the DB2 page cleaning threads are working actively to search for candidate victim pages in the bufferpool and to place them into the *flush_list*, even though the space pressure is minimal in this case (with extreme large bufferpool size). These pages are written out to the storage as REPLACE writes (composed of 71% of the total I/O requests). However, due to the extremely large bufferpool size, most of these *would-be* victim pages will not be evicted from the cache and thus do not need to be READ again. In the second-tier cache, the off-line MIN-RW algorithm knows the future and tends to evict these (not read again) dirty pages for replacement.

We vary another important database parameter, the LSN Gap threshold, by choosing different trace files (300k_40, 300k_400, and 300k_4000). The traces are collected as the DB2 LSN Gap threshold parameter, i.e., *softmax*, is set to different values. The larger the threshold, the less frequently DB2 will trigger a page cleaning session for recovery reasons. As a result, the pages with old updates can stay dirty in the bufferpool for a longer period of time.

The results are shown in Figure 5.9 and we can make the following observations:

- As that shown in the base-line case, the write-aware on-line algorithms outperform their respective write-oblivious counterparts in most of the cases (with the only exception in the 300k_40 case for the ARC algorithm). The improvements (in terms of the total numbers of I/Os reduced) for the LRU, ARC, and TQ algorithms can be up to 36%, 26%, and 12%, respectively.

- The write-aware off-line algorithm, PRO-IO, still outperforms all other algorithms.

- When the LSN Gap threshold value increases, the total number of I/Os generated by the algorithms decreases. This is because the bufferpool manager triggers page cleaning sessions less aggressively for recovery reasons. Therefore,

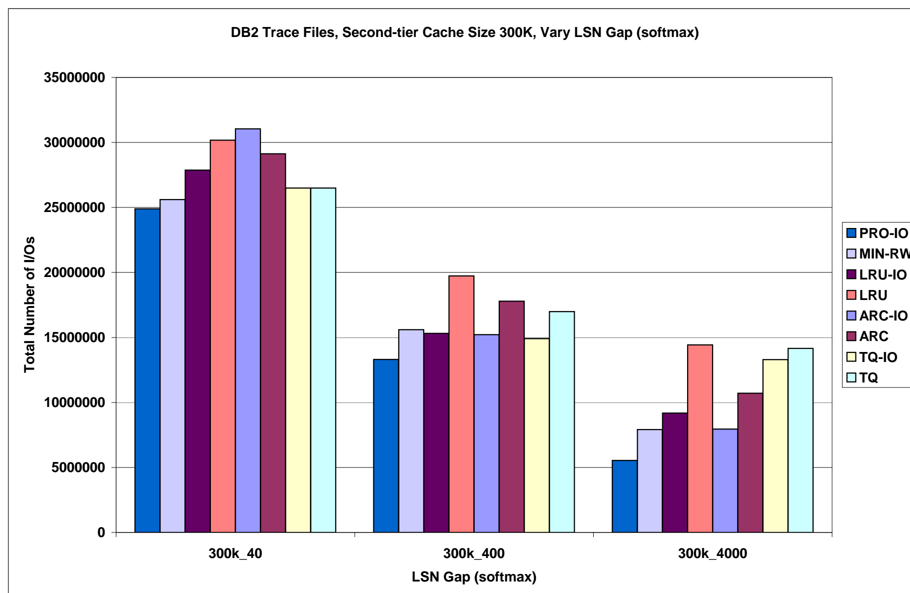


Figure 5.9: Sensitivity Analysis, Vary LSN Gap Threshold

it reduces the number of RECOV writes and the number of dsync requests. At the same time, the REPLACE write synchronizations are deferred for a longer period of time, and the dirty pages can stay longer in the second-tier cache. This gives the write-aware algorithms more opportunities to reduce the number of writes (because of more avoidable writes), and hence the total number of I/Os. For this reason, we can see that the performance gains by using the proposed approach become more significant.

- On the other hand, when the LSN Gap threshold becomes extremely small (the 300k_40 trace case), the performance gaps between the write-aware and the write-oblivious algorithms are not significant. The reason is that in such an extreme case the bufferpool manager aggressively triggers page cleaning sessions for recovery reasons, and generates a larger percentage of RECOV writes (43% in the 300k_40 trace case). Because the updated bufferpool pages are almost immediately written out (by RECOV writes) and become clean, REPLACE writes are rarely needed in this case(less than 1% in the trace). The write-aware algorithms cannot do much to reduce the total number of I/Os if most of the writes are synchronized RECOV writes.

5.7 Summary

Our contributions in this research are three fold. First, we present a partial characterization of an off-line, write-aware optimal replacement algorithm which minimizes the total number of I/Os, including both reads and writes, generated by

second-tier caches. In doing so, we propose three lemmas to describe the optimal algorithm's behavior under many possible cache states. Second, based on the above knowledge of the optimal algorithm, we propose to extend several write-oblivious on-line replacement algorithms to be write-aware. An off-line, primarily total I/O cost based replace algorithm is also proposed to show possible performance gains for other algorithms. Third, we evaluated the effects of the proposed write-aware algorithms by trace-driven cache simulations and discussed the experimental results. The results show that by using the write-aware algorithms a second-tier cache can reduce up to 50% of the total number of I/Os.

Chapter 6

Related Work

In this thesis we focus on second-tier cache management to support DBMS workloads. As pointed out by other researchers [69, 29, 20, 110, 22], second-tier cache management faces a different set of problems than those of the management of top-tier caches. For example, second-tier caches usually find weak temporal localities to exploit in their received requests. For another example, second-tier caches need to be managed to achieve data exclusivity, which is not an issue in the first-tier cache.

We observe two different trends in the proposed techniques for second-tier cache management. The first trend is towards unified, or aggressively collaborative management of different cache tiers [81, 38]. The advantage is obvious. The cache space, as a whole, can be allocated efficiently to avoid data redundancy; and the temporal localities can be revealed and be exploited by all cache tiers. However, there are expenses associated with the close collaboration among the cache tiers in that it consumes resources, such as CPU time and I/O bandwidth.

The other trend is to manage different cache tiers independently, or using a loosely collaborative approach [88, 91, 11]. Although the decentralized management reduces the requirements of system resources, it is usually less efficient than organizing multiple cache tiers as a whole. In this thesis we study second-tier cache management from several different angles; and we survey the related work in the following sections.

6.1 Increasing the Read Cache Hit Rate

As introduced earlier, the read requests issued by bufferpool managers are usually synchronous with respect to the query executions; and the write requests are usually issued asynchronously. This difference has been noted in the literature [69, 29, 20, 110, 22]. Consequently, various techniques have been proposed to increase the read cache hits in the first-tier cache (i.e., the database bufferpool), as well as in the

second-tier cache. The objective is to reduce the I/O latency endured by query executions.

Chou et al. [24] observe that the data access patterns of database operations are limited and are predictable. They argue that the bufferpool replacement algorithm can take advantage of this observation. Bonilla-Lucas et al. [12] find that under a TPC-C workload the references of the database pages are non-uniformly distributed. On average, index pages are accessed more frequently than are the table pages. Also, among all the pages, a small percentage (11.1%) of pages receive about half (45.2%) of the total references. These facts indicate that bufferpool managers have ample opportunities to take advantage of the temporal localities in the page references.

However, as discovered in [69, 29, 20, 110, 22], it is a different scenario in the second-tier cache. It is often difficult for the cache managers in the second-tier to identify temporal locality in their workload. The reason is that the re-reference distances of the data blocks cached in a second-tier cache are longer than those of the blocks cached by the first-tier cache. To overcome this obstacle, different techniques are proposed. One solution is to use frequency, as well as recency, to capture the locality information [75, 51, 111, 48, 64, 7, 108, 43].

Zhou et al. [111] propose the MQ algorithm, which is specifically designed for second-tier cache management. In MQ, the blocks are organized into multiple queues based on their reference frequencies; and recency is only used to break the tie within each individual frequency queue. Megiddo and Modha [64] present the ARC algorithm, which partitions the cache space into two separate parts (recency-based and frequency-based, respectively) and dynamically adjusts their sizes according to the received workload.

Another technique, using a *ghost cache*, is also commonly adopted [111, 64, 57] in second-tier cache management. Because the re-reference distances are usually longer than those observed in the first-tier cache, the second-tier cache needs to memorize a lengthy reference history, which usually contains a number of blocks larger than the cache size. The ghost cache, a facilitating data structure, is set up to catch the locality information for the pages that are out of cache.

The same problem also exists in processor research. To effectively identify the temporal locality information in the L2 cache of processors, some researchers propose to use data mining techniques. Wong and Baer [105] propose that, because the locality information is hard to identify in the L2 cache, extra data structures, such as a locality table, are needed in addition to the LRU list. Li et al. [59] propose to use data mining techniques to reveal the correlations among the requested data blocks and to use the derived information to guide caching, prefetching, and data layout.

Preserving data exclusiveness is another challenge for second-tier cache management. Some researchers suggest unified or aggressively collaborative management of the multiple cache tiers. Pai et al. [81] argue that operating systems can be

modified to unify all the caching and buffing operations within the system. Applications, inter-process communication, file systems, and networks can therefore share a single copy of data safely and concurrently. A similar approach is also proposed by He et al. [38].

Cao et al. [16, 15] suggest giving the applications direct control over the file system cache, for example, for deciding the replacement algorithms. The authors claim that the applications can improve the overall cache hit rates based on their knowledge of their own data access patterns. With the same consideration, Carrera and Bianchini [18] propose to give file systems direct control over the disk controller cache.

Jiang et al. [47] argue that in a client-server system the temporal locality of data references is often analyzed by different clients or by different cache tiers independently. This practice results in suboptimal placement and replacement of data blocks. The authors propose a generalized protocol to facilitate global and consistent locality quantification. Jiang and Zhang [49] propose to let the storage client indicate to the storage server the temporal locality of the data blocks such that the storage server cache can improve its hit rate. Sivathanu et al. [96] argue that DBMSs can be instrumented to explicitly gather data access statistics and then to pass these statistics to the storage systems. The authors claim that such DBMS amendment is natural and simple to implement, and that it can help storage systems in data placement and replacement.

While the above unified,or aggressively collaborative approaches can effectively achieve data exclusiveness and identify temporal locality, some researchers argue that the resource costs, in terms of CPU time and I/O bandwidth, are expensive [88, 90, 22]. They propose to use lightweight, loosely collaborative techniques to achieve the same performance goals. For example, Richardson et al. [88] propose a hint-based lower tier cache management strategy, in which the files' names, types, and other attributes can be used as hints by the lower tier storage systems. The authors observe that such attribute hints can provide useful information about data reference patterns. A similar approach is proposed by Phunchongharn et al. [87], who suggest to use the file metadata as hints to guide the caching policies and data allocations on storage systems.

Sarkar and Hartman [90, 91] propose that, in a distributed client-server system, the clients can use the approximate location information as hints to infer where to look for the missed data blocks. Using such an approach reduces the load on the network and the CPU usage on the server. The same hint-based cache coordination is also found in processor architecture research. Source and target hints are attached to the instructions to indicate the source and the destination cache levels where the data being accessed is likely to reside. Beyls and D'Hollander [11] argue that the re-reference distances of instructions can be used to determine the appropriate location hint values. Arpaci-Dusseau et al. [4] propose that because the operating systems are rigidly designed and implemented, the applications can take advantage of their knowledge of the OS and can infer the content of the file system cache. Similar

approaches are also proposed in other work [13].

Our work, presented in Chapter 3 and published as [57], belongs to this hint-based, decentralized cache management category. We argue that the types of the I/Os issued by DBMS bufferpool managers can be used as hints to guide second-tier cache replacement. As an extension, Liu et al. [61] propose a generic hint-based algorithm for managing storage server caches. It automatically interpret hints generated by storage clients and translates them into server caching policies.

In addition to using hints, other hierarchy-aware techniques have been proposed by researchers. For example, some researchers [104, 23, 107, 34] propose *demote* or *promote* based approaches to migrate blocks among the cache tiers and to achieve data exclusiveness. In other work [97, 6], the storage servers are implemented to infer the contents of upper level cache tiers to guide the servers' cache replacement. The authors claim that these lightweight and loosely collaborative techniques can achieve similar performance without much consumption of CPU time and I/O bandwidth.

6.2 Deferring Write Synchronization

To achieve concurrency and data consistency, most modern DBMS systems, such as DB2 [42, 103, 102], Oracle [77], and SQL Server [65], use the ARIES algorithm [67] to manage concurrency control, crash and recovery. To comply with the ARIES algorithm, Write Ahead Logging (WAL) protocol [10, 30] is widely adopted by DBMS implementations. However, in case of a crash and recovery, it is not feasible to replay all the log entries previously recorded. To limit recovery time, and log space consumption, most commercial DBMS, such as DB2, Oracle, and SQL Server, apply the fuzzy checkpointing technique [66] to synchronize the aged updates from bufferpools to permanent storage. By doing this, the numbers of log entries that need to be replayed during recoveries can be reduced.

As presented in Chapter 4, we find that the current implementations of bufferpool write synchronization are more conservative than necessary and therefore waste I/O bandwidth. We propose to extend the current file system interface to let DBMS defer synchronizations of some bufferpool writes. By using our approach the second-tier cache can reduce the number of writes issued to storage devices.

Write optimization has long been a topic in caching research, and various techniques have been proposed by researchers. Log-structured file systems [80, 89, 54, 44], are proposed to reduce the number of writes. The idea is that the updates made to a file system can be written to a log structure on disk sequentially, and then be synchronized to the data and metadata blocks at a later time. Therefore, some blocks can be written at once sequentially. It also reduces the number of seeks and rotations performed by the disk and allows for fast crash recovery.

Soft update [33, 63, 32], is another technique designed to reduce the number of writes issued by file systems. Ganger et al. [33, 32] observe that the writes of

metadata issued to storage devices need not be synchronized immediately. Their synchronization can be deferred to a later time when the corresponding data blocks are flushed to disks. By using soft update, the number of writes to update the on-disk metadata can be reduced significantly.

Carson and Setia [19] observe that bulk arrivals of the writes generated by periodic synchronizations of file system caches may cause I/O *traffic jams*. Mogul [66] proposes to use the *fuzzy checkpointing* in file systems to avoid the peak consumption of I/O bandwidth caused by periodic flushes of dirty blocks. In such an approach, each dirty block in the cache is associated with an age attribute to describe the time when the oldest unsynchronized update is made to the block. When the gap between the age and the current time reaches a certain threshold, the block will be flushed to disk. To address the same problem, i.e., bulk arrivals of cache synchronization writes, Batsakis et al. [8] suggest flushing dirty pages opportunistically when the I/O bandwidth is not saturated.

Wu et al. [107] propose that while using the DEMOTE approach [104], the writes of the blocks being demoted should be deferred and the blocks should be cached in a specific buffer in the storage client. This gives the client cache opportunities to achieve higher hit rates and to do better I/O scheduling. Olston and Widom [74] propose that, when the I/O bandwidth is a bottleneck, the synchronization of the cached data should be performed with a finer granularity. Only a selected few objects should be refreshed under such a situation to save I/O bandwidth. The selection of the objects to refresh should be based on their importance and the deviations between their in-cache and on-disk copies.

Flash drives are being used more widely as their price drops and their performance increases. Because their operations do not include seeks and rotations, the flash drives are usually exploited to reduce read and write latencies endured by the applications. Debnath et al. [26] suggest to defer the writes issued to spinning hard disks and detour them to dedicated flash drives where the writes can be performed sequentially. At a later time, multiple updates can be applied in bulk to the hard disks. A similar approach is presented by Li et al. [58], in which flash drives are used as buffers for DBMS writes. Based on the same observation, but in a quite opposite approach, Soundararajan et al. [98] suggest using a hard disk tier as a write cache to log the updates made to flash drives.

A similar idea, which uses *solid state storage* to cache writes, is proposed by Chen and Ng et al. [21, 71]. The authors propose to integrate non-volatile storage into a file system cache so that the writes can be buffered there to reduce the number of physical writes performed by disks. Data durability can be guaranteed in case of a system crash or a power failure. With the same consideration, Akyürek and Salem [2] suggest integration of non-volatile RAM into DBMS bufferpools, where it can be used to cache dirty blocks. The synchronizations of the dirty blocks can be deferred without jeopardizing data durability.

Some research discusses the desired time length for which writes should be deferred. Chen [21] determines an optimal value for the deferral intervals. Muntz et

al. [70] propose that the writes should be deferred until the I/O bandwidth becomes idle and should yield to other I/O requests with higher priorities. Nightingale et al. [72] suggest that write synchronizations can be deferred until the write effects need to be exposed to the users.

In Chapter 4, we argue that the bufferpool writes are synchronized more conservatively than necessary and we propose to defer synchronization of the REPLACE writes to a later time. Dhamankar et al. [28, 27] make the same observations during their study of commodity disk caches. They propose using the existing Windows system call, *FLUSH_CACHE* (whose Unix counterpart is *fsync*), to synchronize the entire disk caches only when it is necessary. Their approach is similar to the *rec* test case studied by us in Chapter 4.

We make another observation that the current POSIX-compliant file system interfaces are not flexible enough to perform fine-grained synchronizations for bufferpool writes. Other researchers also identify weaknesses of the POSIX standards for supporting DBMS workloads. One argument [106, 99] is that the POSIX standards do not offer transaction guarantees to applications and therefore cannot preserve consistency for database transactions. The proposed solution is to extend the POSIX-compliant file system interface to be transaction-aware and to provide applications with additional functions, such as concurrency control and write ordering. Similar ideas can also be found in other work [94, 25, 92, 31]. Hall and Bonnet [37] observe that read and write requests issued by bufferpool managers should be given different properties in caching and I/O scheduling in lower cache tiers. Read requests, because they are on the critical path of query executions, should be given priorities higher than those of the writes. The authors also suggest to modify operating systems to include prioritized I/O operations.

Guo [36] observes that the concurrency and consistency requirements of the applications may vary under different scenarios. The author proposes to implement a more flexible caching scheme in which the applications can explicitly express their concurrency and consistency requirements. The use of these requirements will reduce resource consumption at storage servers. Sears and Brewer [93], propose another approach regarding concurrency and consistency control. Instead of page-oriented management, they propose a finer-grained concurrency control mechanism, which is object-oriented. They claim that the new technique reduces communication costs among applications, log managers, and buffpool managers. Our approach differs from these in that it does not relax the requirements of transaction consistency, yet still saves I/O bandwidth.

6.3 Reducing Total Number of I/Os

In Chapter 5, we argue that maximizing read cache hits is not the sole objective of second-tier cache management. When I/O bandwidth becomes a performance bottleneck, the goal should be changed to minimizing the total number of I/Os,

including both reads and writes. In addition, under such a situation the cache replacement algorithm should take all the I/O related factors, such as clean or dirty page states and read or write request types, into account while making replacement decisions.

Similar observations have been made by other researchers. Jeong and Dubois [45] notice that cache misses can generate different types of costs, such as time latency, power consumption, bandwidth consumption, or any other properties attached to a cache miss. Also, the costs may be non-uniformly distributed across cached blocks. In practice, the replacement algorithms should take asymmetric cost distributions into account when doing replacements.

Lee et al. [55] make the same observations during their study on non-volatile RAM (NVRAM). They note that the read hit ratio, which has been a commonly adopted metric to measure cache performance, is no longer adequate for caching with NVRAM. Instead of using the read hit ratio, the total number of disk accesses should be counted to assess user perceived cache performance. Lee et al. also argue that because of the changed performance objective, the existing MIN algorithm [9] may no longer be optimal.

Flash drives, as they are being used more widely, are attracting more research effort due to their asymmetric read and write performance. To write a single block to a flash drive, a contiguous area of blocks needs to be erased in advance. This characteristic makes random writes more expensive than sequential writes or read operations. Much work has been proposed to reduce random writes issued to flash drives. Park et al. [85] suggest dividing the cache space into separate partitions to serve reads and writes respectively. The write cache size can be dynamically adjusted according to access patterns to effectively absorb logical writes. Similar ideas are proposed by other researchers. Jin et al. [50] suggest that the pages being updated most frequently should be kept in cache to reduce the number of physical writes performed by flash drives. Moshnyaga et al. [68] notice that mobile devices, such as cell phones, rely on a DRAM cache tier, between the processor and the flash drive, to buffer data blocks and instructions. The authors suggest to keep the updated blocks in the DRAM as long as possible to reduce the number of writes issued to the underlying flash drive.

Lee, Koh, and Bahn [56] propose detecting access patterns of data blocks and placing them accordingly to different types of storage media. For example, blocks with frequent random writes should be migrated from flash drives to hard disks, where they can be served more efficiently. He and Veeraraghavan [39] present a similar technique, in which the columns of database tables are organized into groups, according to their update frequencies, and are stored separately. Therefore, a single bufferpool page can contain more updates and the updates can be synchronized to flash drives simultaneously. In general, the total number of random writes can be reduced.

Kim, Whang, and Song [52] suggest writing only page-differentials as logs into flash drives. The writes can be organized into sequences and the page synchroniza-

tions can be performed later in bulk. Zhou and Meng [109] present a novel idea to convert random writes into sequential operations. The authors suggest inserting unmodified clean blocks into dirty blocks of a write sequence so that the writes can be performed sequentially. Hu et al. [40] argue that applications could be re-designed to minimize the intermediate data, such as temporary database tables, to reduce unnecessary writes issued to flash drives.

Park et al. [84, 83] propose exploring read and write access histories separately, so that future references of each I/O type can be predicted more precisely. Canim et al. [14] observe that high-end Solid State Disks (SSD) being marketed today do not show noticeable performance penalties for random versus sequential writes due to their internal write buffering. Therefore, the costs of read and write operations become symmetric. However, these high-end devices are more expensive and are not widely used.

Asymmetric costs of cache misses are also recognized in the studies of object-oriented database systems (OODB). Due to the variable sizes of data objects, the cost of replacing an object varies. Park et al. [82] propose extending the previous page-based replacement algorithms to include the asymmetric costs of objects. The authors propose a new metric, the I/O cost per unit time and unit space, to quantify the replacement costs of cached objects. Norvag and Bratbergsengen [73] propose using log-structured file systems to write only the delta-object, a piece of data which contains only the changes from the last version of the object, to disks. Thus, I/O bandwidth can be saved and the disk service time can also be improved due to the reduction of seeks and rotations. Similar approaches are also found in Web caching research [17, 5], where the Web documents also vary in size.

Due to the asymmetric costs of cache misses, cost-aware replacement algorithms have been proposed [35, 46, 53, 100, 78]. Gill and Modha [35] observe that there are two factors determining the cost of write caching. The first factor is the write hit rate and the second factor is the spatial locality of the written blocks. The higher the write hit rate, the fewer number of writes are issued by the cache. The better the spatial locality, the shorter the service time is for the writes. The authors propose a cache replacement algorithm that dynamically adapts the balance between the temporal and spatial localities of the cached blocks to improve write efficiency.

Some researchers [46, 86, 53, 60, 100, 78] observe that evicting clean pages is less expensive than evicting dirty pages, because the clean pages can be discarded immediately without writes. Tang and Meng [100] propose a replacement algorithm that dynamically selects clean or dirty pages to evict according to page access patterns. The key idea is that if the number of clean (dirty) pages in the cache exceeds a particular threshold, then the next victim should be a clean (dirty) page. A similar approach is taken by Ou and Härder [78], who propose making replacement trade-offs between clean and dirty pages in a controlled fashion, i.e., depending on the read to write cost ratio and on the data access patterns.

Beside I/O bandwidth utilization, power consumption is another type of cost generated by cache misses. Zhu and Zhou [112] propose a novel approach to min-

imize power consumption of storage systems. The authors argue that the overall disk idle time can be raised if the frequently accessed data blocks are organized and are placed on a dedicated set of disks (the cold blocks are placed on remaining disks). Ou et al. [79] observe that due to the expensiveness of the random writes, an ideal strategy to reduce the power consumption of flash drives is to reduce writes, especially random writes, issued from the cache. The authors argue that the existing write-oblivious replacement algorithms should be modified to defer the writes of dirty blocks.

Koltsidas and Viglas [53] claim that the replacement cost for a cached block should take into account both the clean or dirty block state and the I/O costs of different types of storage device (such as magnetic disk or flash drive, on which the block is stored). The authors propose a replacement algorithm which divides the cache into two separate parts, i.e., a recency based segment, which is managed by LRU; and a I/O cost based segment, which is managed according to the blocks' I/O costs on their respective storage medias. The victim page will be selected dynamically from the two segments based on the segments' sizes.

With the presence of different cost types and asymmetric cost distribution among cache misses, maximizing read cache hits no longer guarantees optimal cache performance. Some research has been done to pursue new optimal algorithms under such scenarios. Jeong and Dubois [46] propose an off-line, cost-sensitive replacement algorithm in the context of non-uniformly distributed miss costs. The main idea is to express all possible replacement sequences (for a particular request trace) in a huge tree structure. The size of the tree exponentially increases at each level (i.e., each request) and the optimal replacement algorithm is represented by the branch with the smallest accumulated cost. To deal with the huge search space, the authors provide several techniques to prune the unnecessary branches. The authors admit that this tree search approach is not guaranteed to find the optimal replacement policy, but the resulting off-line policy can be used to provide a lower bound on the achievable costs of realistic on-line algorithms. They also argue that the derived policy outperforms the existing MIN algorithm in terms of the new replacement metric.

Our research presented in Chapter 5 is based on similar observations, namely that cached pages have different replacement costs. However, our work is different from the related work in the following ways:

- In our research we proposed a read-write cache model which generalizes the cache behavior under a read-write workload. Other related work does not adopt the same (or similar) approach to give a complete cache model to describe the cache behavior under a read-write workload.

- We characterized the behavior of an optimal, off-line algorithm in many cases (except the difficult case B2b). To the best of our knowledge, this is the first work to describe the characteristics of an optimal cache replacement algorithm under the read-write workload/cache model.

Chapter 7

Conclusions and Future Work

In this thesis we present our work on second-tier cache management to support DBMS workloads. Our research focuses on increasing the read hit ratio in second-tier caches, which directly shortens I/O latency endured by query executions; and on reducing the number of writes issued by the cache, which relieves I/O bandwidth contentions between reads and writes.

We make novel observations about DBMS systems and second-tier cache management. Based on these observations, we propose several techniques to improve cache performance. Our evaluations show promising performance gains and some of the proposed techniques have been implemented in an open source operating system and in an open source DBMS.

7.1 Summary of Contributions

The ultimate goal of our research on second-tier cache management is to minimize the I/O latency endured by query executions, such that the DBMS performance can be improved in terms of both query response time and query throughput. We conduct our research from two particular angles: the rationale behind different types of bufferpool I/Os and the relationship between bufferpool reads and writes.

First of all, we find that the types of DBMS bufferpool I/Os indicate different bufferpool states, and states of the pages carried by the I/Os. Furthermore, the I/O type information can be used by second-tier caches as hints to infer the future access patterns of the pages being accessed. The cache then can adjust its caching policy based on these access patterns. Accordingly, we make the following contributions:

- We observe that the different bufferpool I/O types indicate different bufferpool states (e.g., whether the bufferpool is under space pressure or whether there is a page cleaning session being performed) and different states of the pages carried by the I/Os (e.g., whether the page is going to be evicted by the bufferpool). This type information can be used by second-tier caches as

hints to predict the future access patterns of the pages and to guide their cache replacements.

- We propose extend the current hint-oblivious replacement algorithms to be hint-aware. We also present a new, primarily hint-based algorithm, *Type Queue*. Our trace-driven (DB2 I/O traces collected by running a TPC-C workload) evaluations show that the hint-aware algorithms perform significantly better than the hint-oblivious algorithms.

In addition, our research also focuses on the relationship between bufferpool reads and writes. Although DBMS bufferpool writes are usually issued asynchronously with respect to query executions, they still compete with reads for I/O bandwidth. Therefore, the contention for I/O bandwidth still exist between reads and writes, and by reducing writes we can shorten the read latency endured by query executions. Our first effort to reduce writes can be summarized as follows:

- We observe that current bufferpool write synchronizations are performed more conservatively than necessary. REPLACE writes, unlike RECOVER writes, do not need immediate synchronizations and their synchronizations can be deferred to a later time. The deferral leads to more write cache hits, which can save I/O bandwidth and result in better I/O scheduling.

- Current POSIX-compliant file system interfaces implemented by operating systems, such as Unix and Windows, are not flexible enough to support dynamic requirements of bufferpool write synchronizations. We extend the interfaces to include a finer-grained, page-based synchronization operation to provide applications with more flexibilities in controlling their write synchronizations.

- The proposed new I/O operation is implemented in Linux and we also modify the MySQL InnoDB storage engine to use it. Our evaluations show promising performance gains in terms of increased transaction throughput, with increases up to 27% for a TPC-C workload.

While we study the relationship between bufferpool reads and writes, we make the following observation: maximizing read hit ratios should not be the sole objective of cache management. Under some circumstances, for example when I/O bandwidth becomes the performance bottleneck for a DBMS, reducing the total number of I/Os issued by the cache can lead to better database performance. Our contributions in this part of work are:

- We present a partial characterization of an off-line, write-aware optimal replacement algorithm which minimizes the total number of I/Os, including both reads and writes, issued by second-tier caches. In doing so, we propose several lemmas to describe the optimal algorithm's behavior under many possible cache states.

- Based on our knowledge of the optimal algorithm, we extend several write-oblivious on-line replacement algorithms to be write-aware. We also propose an off-line, primarily total I/O cost based replacement algorithm. These write-aware algorithms are designed to imitate the optimal algorithm's behavior whenever possible.

- We evaluated the effects of the proposed write-aware algorithms by trace-driven (DB2 and MySQL I/O traces collected by running TPC-C workloads) cache simulations. The results show that by using the write-aware algorithms a second-tier cache can reduce the total number of I/Os by as much as 50%.

7.2 Future Work

Our work can be extended in the following directions:

- In Chapter 5 we present a partial characterization of an off-line optimal algorithm which minimizes the total number of I/Os generated by second-tier caches. Possible future work in this direction is to extend the *partial* characterization to a fully described optimal algorithm. The key is to solve the difficult case (Case B2b) that we encounter. In doing so, the current replacement cost (i.e., the total I/O cost), which is associated with each cached page and which is used in making replacement decisions, need to be rethought.

- In Chapter 4 we implement our proposed technique (i.e., deferring synchronization of writes by using dsync operations) in Linux and in the MySQL InnoDB storage engine. As possible future work, the other approaches proposed by us, such as the hint-based replacement algorithms (proposed in Chapter 3) and the write-aware replacement algorithms (proposed in Chapter 5), can also be implemented in second-tier caches, such as a file system cache or a storage server cache, and be evaluated accordingly. This will give us insight on the effectiveness of these approaches in more realistic scenarios.

- Several approaches have been proposed in this thesis to improve second-tier cache performance. However, they are presented and evaluated individually. Possible future work should conduct a study on the interaction and collaboration of these approaches. For example, in Chapter 3, we propose that a page whose most recent request is a REPLACE write has a higher priority to be cached than a page whose most recent reference is a read does. However, quite likely, the page being read recently is a clean page, which, according to the approach proposed in Chapter 5, should be kept in cache at the expense of the dirty pages (such as the one recently accessed by a REPLACE write). As a possible future work, such contradicting factors should be studied and eventually a unified second-tier cache management scheme should be developed for practical use.

Appendix A

Proof of the Correctness of Lemma 5.1

We adopt the same assumptions and definitions made in Chapter 5 and restate the Lemma 5.1 as follows:

- The unavoidable I/Os, which include first time reads and writes of pre-clean pages, are independent of the cache replacement algorithm. They will occur regardless of which algorithm is used to manage the cache.

Also, in Figure A.1 we re-draw the diagram presented in Figure 5.1, which will be used in the proof.

Proof:

By definition the unavoidable I/Os include the following two types of I/Os. We prove that each of these two types of I/Os is generated independently of the cache replacement algorithm:

- first time reads: In Figure A.1, this type of I/O is associated with the transition arc, which is generated by a read request, from the Init (initial) State to the C (Clean) state. For the following reasons, these read I/Os are determined by the request trace and are unavoidable regardless of which algorithm is used to mange the cache:
  - all pages being requested in the trace start from the Init state in Figure A.1
  - by the definition of the Read-Write Cache Model (in Section 5.2), a page whose first request is a read will transit from Init state to Clean state, which results in one read I/O.

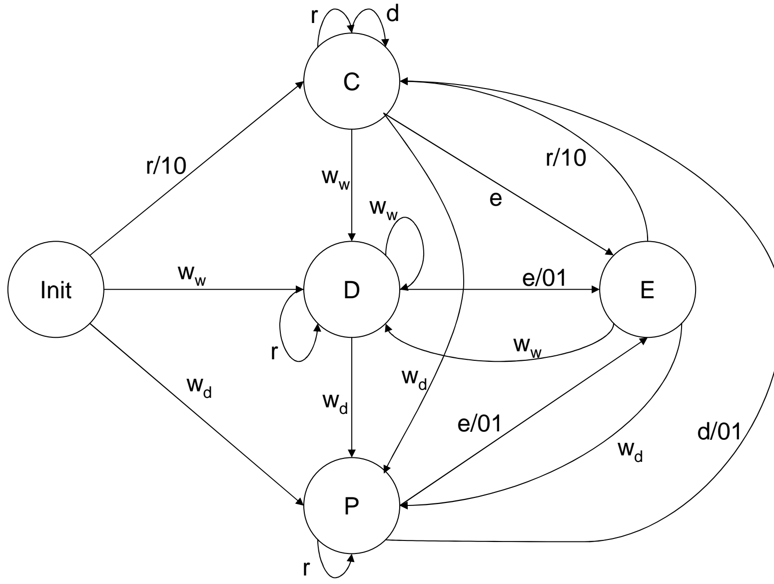


Figure A.1: States, Transitions, and I/Os of Cache Pages

- writes to synchronize pre-clean pages: In Figure A.1, this type of I/Os is associated with the following two transition arcs:
  - the transition arc from the P (Pre-clean) state to the E (Evicted) state, which is generated by a cache replacement. It indicates the page is chosen by the replacement algorithm as the victim page and is evicted from the cache. Because the page is dirty, a write I/O is required to synchronize its content to lower level storage.
  - the transition arc from the P (Pre-clean) state to the C (Clean) state, which is generated by a dsync request in the trace. It indicates the page is synchronized to the lower level storage by a dsync operation and becomes clean.

  Intuitively, it seems that the write associated with the former transition, from the P (Pre-clean) state to the E (Evicted) state, is dependent on the replacement algorithm. However, from Figure A.1 we can observe that once a page enters the pre-clean state, it will stay there until it is evicted (from P to E) or it is synchronized by a dsync request (from P to C). By the definition of the pre-clean state, a dsync request of the page must be in the trace. Assuming the time of the next dsync request is t then:

  - if the page is chosen as a victim page and be replaced before time t then a write must be issued to synchronize the page to lower level storage at the replacement time

  - otherwise the page will be synchronized to lower level storage at time t by the dsync request

We can conclude that a write I/O for a pre-clean page is unavoidable and that the write is independent of the replacement algorithm.

The proof has been completed.

Appendix B

Proof of the Correctness of Lemma 5.2

We adopt the same assumptions and definitions made in Chapter 5 and restate Equation 5.2 and Lemma 5.2 as follows:

- Equation 5.2: $O_P = \sum_{t=1}^{L} c_X(p_t, t)$
- Lemma 5.2: The result of Equation 5.2, O_P, is equal to the total number of avoidable I/Os generated by the cache, which is managed by using replacement policy P.

Also, in Figure B.1 we re-draw the diagram presented in Figure 5.1, which will be used in the proof.

Proof:

We discuss read I/Os and write I/Os separately in the following sections.

Read I/Os

In Figure B.1 it is seen that the read I/Os can only be generated by two state transitions:

- From state I to state C
- From state E to state C

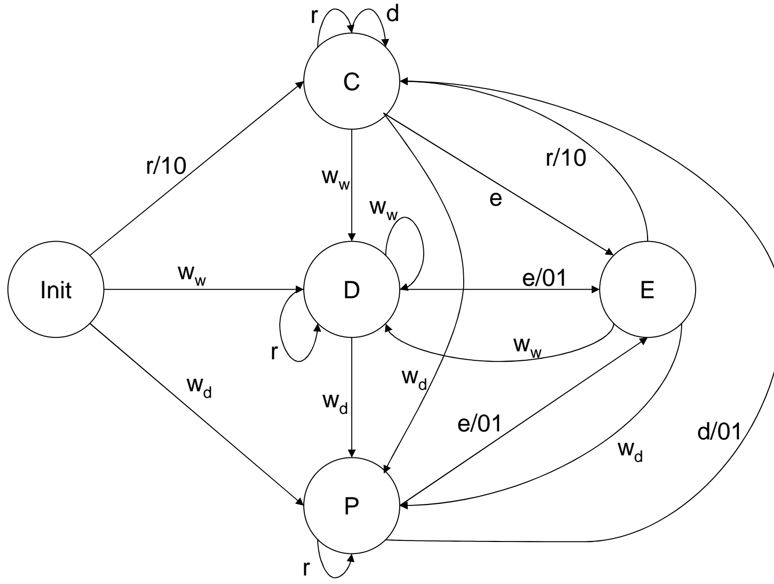


Figure B.1: States, Transitions, and I/Os of Cache Pages

The read I/O associated with the state transition from I to C is a first time read and by Definition 5.5 is unavoidable. In the definition of $c_X(p,t)$ (Definition 5.6) this unavoidable read I/O is not counted in any of the clauses.

The read I/O associated with state transition from E to C indicates that the page was evicted from the cache and now it is being referenced again by a read request. This read I/O is dependent on the caches replacement algorithm because the page was chosen as a victim by the algorithm and was evicted from the cache. The read is an avoidable I/O and is counted in Clause 1 and Clause 3 of Definition 5.6. The above two clauses cover all the three states (i.e., Clean, Pre-clean, and Dirty but not pre-clean) of cached pages.

Write I/Os

In Figure B.1 it is seen that the write I/Os can only happen when the page is dirty (i.e., in either D or P state) and it is to be evicted (during the transitions to E state) or to be synchronized (during the transitions to C state). Therefore, a write I/O can only be associated with the following arcs (state transitions):

- From state D to state E
- From state P to state E
- From state P to state C

The write I/O associated with the state transition from P to C or from P to E is the write to synchronize a pre-clean page. By Definition 5.5 such writes are unavoidable. In the definition of $c_X(p,t)$ (Definition 5.6) this unavoidable write I/O is not counted by any of the clauses.

The write I/O associated with state transition from D to E indicates that the page was evicted from the cache. The write is an avoidable I/O and is counted in Clause 4 of Definition 5.6.

Based on the definition of the Read-Write Cache Model (in Section 5.2), there is no other types of I/Os (other than avoidable read, unavoidable read, avoidable write, and unavoidable write) generated by the cache. We have shown that Equation 5.2:

- only counts avoidable I/Os
- does not count unavoidable I/Os

In addition, by definition the equation only counts each avoidable I/O generated by the cache *once*.

We have proved that the result of Equation 5.2, O_P, is equal to the total number of avoidable I/Os generated by the cache, which is managed by using replacement policy P.

Appendix C

Proof of the Correctness of Lemma 5.3

We adopt the same assumptions and definitions made in Chapter 5 and restate the Lemma 5.3 as follows:

Let X be a page request trace, B_0 and B'_0 be two different initial cache states where:

$B'_0 = T_0 + \{a\}$

$B_0 = T_0 + \{b\}$

for $T_0 \subseteq A$, $a, b \notin T_0$, $c_X(a, 0) > 0$, and $c_X(b, 0) = 0$. We claim that for any demand policy P, corresponding to X and B_0, there must exist a demand policy P', corresponding to X and B'_0, such that $O_{P'} \leq O_P$.

Before starting the formal proof of Lemma 5.3, we give some intuitions about the basic ideas used in the proof:

- The two different initial cache states, namely B_0 and B'_0, can be seen as two possible states of *one* cache, after the cache makes two different replacement decisions (evicting page a, which leads to cache state B_0, or evicting page b, which leads to cache state B'_0). Except for this one page difference (caching page a or page b), other cache contents (which are denoted as set T_0) in the two possible cache states are the same. The request trace received by the cache after it makes this particular replacement decision (evicting a or b) is denoted as X.

- The basic idea used in the proof is that, given an initial state B_0 and a request trace X, no matter which valid demand replacement policy one can devise (denoted as P), we can always construct another valid demand policy (denoted as P' and based on B'_0 and X), which performs at least as well as P. Therefore we can claim that evicting page b, which has a zero total I/O cost, is always an optimal replacement decision.

Our proof is based on a *case by case* analysis. To help readers better understand it, we also show a diagram for each case derived in the proof. The diagrams show that at some important time points, the total I/O costs of the replacement decisions made by the two different policies (P and P'). They help the readers to conclude that policy P' always generates less or equal total number of avoidable I/Os than policy P does.

Proof:

Given P, we construct P'. Suppose page a first occurs in the request trace X at time i_a, and b at i_b. If either page a or b does not occur in X, then set i_a or i_b equal to $L + 1$. We consider the following three cases:

Case A. $i_a < i_b$. In this case we consider the following three subcases:

Case A1. $p_j = b$ where p_j is the first occurrence of b in P, and $1 \leq j < i_a$. Here we set $p'_k = p_k$, for $1 \leq k \leq L$ and $k \neq j$, and $p'_j = a$. This results in $B_t = T_t + \{b\}$ and $B'_t = T_t + \{a\}$, for $0 \leq t \leq j - 1$ and $B_t = B'_t$, for $j \leq t \leq L$. Since pages a and b are both not referenced up to time j, it should be clear that P' is a valid demand policy.

Also, since in P we have costs $c_X(a, 0)$ and $c_X(b, j)$, and in P' we have costs $c_X(b, 0)$ and $c_X(a, j)$. We also know that $c_X(a, 0) = c_X(a, j)$ and $c_X(b, j) = c_X(b, 0)$, therefore $O_{P'} = O_P$. The above total I/O costs of page replacements made by policy P and P' are illustrated in Figure C.1.

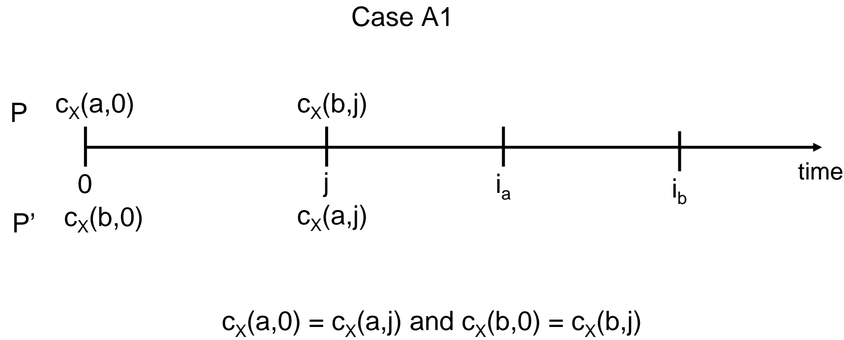


Figure C.1: Total I/O Costs of Page Replacements in Policy P and P'

Case A2. $p_{i_a} = b$ where p_{i_a} is the first occurrence of b in P. In this case we set $p'_k = p_k$, for $1 \leq k \leq L$ and $k \neq i_a$. We also set $p'_{i_a} = \phi$. As in Case A1, P' is a valid demand policy.

Also, since in P we have costs $c_X(a, 0)$ and $c_X(b, i_a)$, and in P' we have costs $c_X(b, 0)$ and $c(\phi, i_a)$, we also know that $c_X(a, 0) > c(\phi, i_a)$ and $c_X(b, i_a) = c_X(b, 0)$,

therefore $O_{P'} < O_P$. The above total I/O costs of page replacements in policy P and P' are illustrated in Figure C.2.

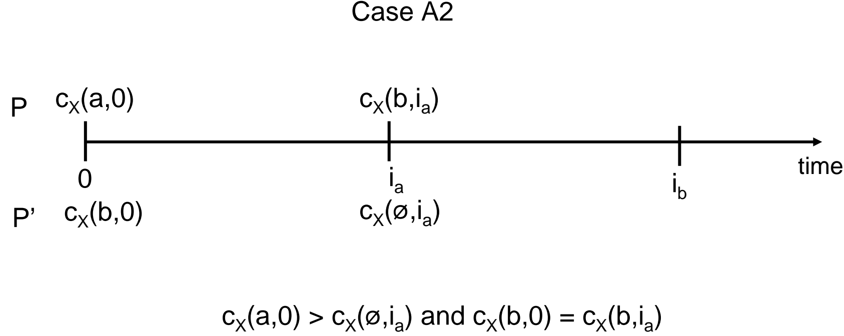


Figure C.2: Total I/O Costs of Page Replacements in Policy P and P'

Case A3. $p_k \neq b$, for $1 \leq k \leq i_a$. Then we must have $p_{i_a} = c$ where c is a page other than a or b. At time $t = i_a$ the states of the caches are given by:

$B'_{i_a} = T_{i_a} + \{a\}$

$B_{i_a} = T_{i_a} + \{b\} + \{a\} - \{c\}$

which can also be written as follows:

$B'_{i_a} = [T_{i_a} + \{a\} - \{c\}] + \{c\}$

$B_{i_a} = [T_{i_a} + \{a\} - \{c\}] + \{b\}$

We consider the following two sub-subcases:

Case A3A. If $d_X(c, i_a + 1) < d_X(b, i_a + 1)$. Let i_c be the time of the first occurrence of c in X, then $i_c < i_b$. In this sub-subcase we consider three more sub-sub-subcases.

Case A3A1. If $p_j = b$ where p_j is the first occurrence of b in P and $l < i_c$, then we set $p'_k = p_k$, for $i_a + 1 \leq k \leq L$ where $k \neq j$ and $p'_j = c$.

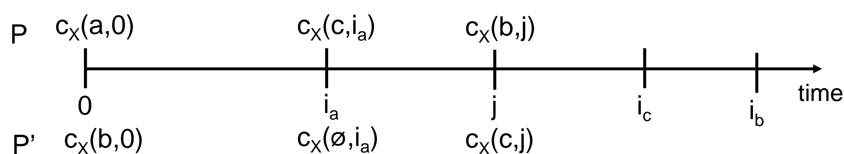
Since in P we have costs $c_X(a, 0)$, $c_X(c, i_a)$, and $c_X(b, j)$, and in P' we have costs $c_X(b, 0)$, $c(\phi, i_a)$, and $c_X(c, j)$, therefore $O_{P'} < O_P$. The above total I/O costs of page replacements in policy P and P' are illustrated in Figure C.3.

Case A3A2. If $p_j = b$ where p_j is the first occurrence of b in P and $j = i_c$, then we set $p'_k = p_k$, for $i_a + 1 \leq k \leq L$ where $k \neq j$ and $p'_j = \phi$.

Also, since in P we have costs $c_X(a, 0)$, $c_X(c, i_a)$, and $c_X(b, i_c)$, where in P' we have costs $c_X(b, 0)$, $c(\phi, i_a)$, and $c(\phi, i_c)$, therefore $O_{P'} < O_P$. The above total I/O costs of page replacements in policy P and P' are illustrated in Figure C.4.

Case A3A3. If $p_j = b$ where p_j is the first occurrence of b in P and $j > i_c$, we must have $p_{i_c} = d$, where d is a page other than b, or c (it could be a, because $i_a < i_c$). At time $t = i_c$ the states of the caches are given by:

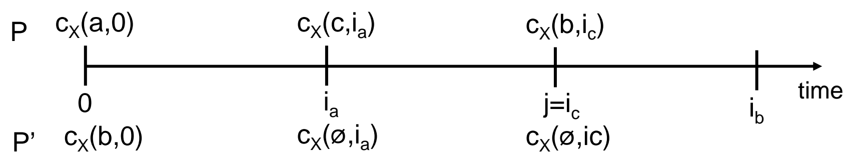
Case A3A1



$c_X(a,0) > c_X(\emptyset,i_a)$, $c_X(c,i_a) = c_X(c,j)$, and $c_X(b,j) = c_X(b,0)$

Figure C.3: Total I/O Costs of Page Replacements in Policy P and P'

Case A3A2



$c_X(a,0) > c_X(\emptyset,i_a)$, $c_X(c,i_a) > c_X(\emptyset,i_c)$, and $c_X(b,ic) = c_X(b,0)$

Figure C.4: Total I/O Costs of Page Replacements in Policy P and P'

$B'_{i_c} = T_{i_c} + \{c\}$

$B_{i_c} = T_{i_c} + \{b\} + \{c\} - \{d\}$

Which can also be written as follows:

$B'_{i_c} = [T_{i_c} + \{c\} - \{d\}] + \{d\}$

$B_{i_c} = [T_{i_c} + \{c\} - \{d\}] + \{b\}$

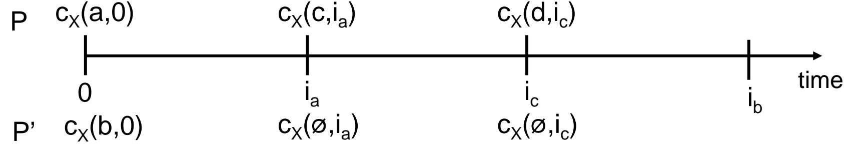
Please note that this is the same case as we met at Case A3. Since i_c is strictly larger than i_a, and the length of trace X is finite, thus this situation can only occur a finite number of times.

Also note that P' is valid as far as it is specified and that $p'_1, ..., p'_{i_c}$ generates fewer I/Os than $p_1, ..., p_{i_c}$. Some of the total I/O costs of page replacements in policy P and P' are illustrated in Figure C.5.

Case A3B. If $d_X(c, i_a+1) > d_X(b, i_a+1)$, we set $p'_k = p_k$ for $1 < k < i_a - 1$ and $p'_{i_a} = \phi$ and consider two more sub-sub-subcases:

Case A3B1. If $p_j = b$, where p_j is the first occurrence of b in P and $j < i_b$, we set $p'_k = p_k$, for $i_a + 1 \leq k \leq L$, and $k \neq j$ and $p'_j = c$. Here $B'_t = B_t$, for $j < t < L$,

Case 3A3

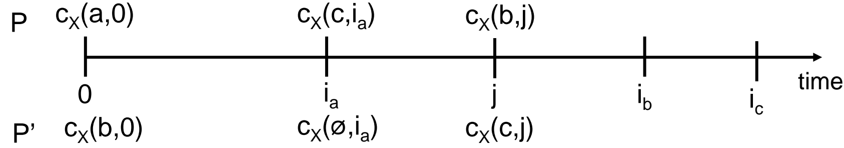


$c_X(a,0) > c_X(b,0)$, $c_X(c,i_a) > c_X(ø,i_a)$, and $c_X(d,i_c) > c_X(ø,i_c)$

Figure C.5: Total I/O Costs of Page Replacements in Policy P and P'

and as in Case A1, we see that $O_{P'} < O_P$ still holds. Some of the total I/O costs of page replacements in policy P and P' are illustrated in Figure C.6.

Case A3B1



$c_X(a,0) > c_X(ø,i_a)$, $c_X(c,i_a) = c_X(c,j)$, and $c_X(b,j) = c_X(b,0)$

Figure C.6: Total I/O Costs of Page Replacements in Policy P and P'

Case A3B2. If $p_j \neq b$, for $j < i_b$, we set $p'_k = p_k$, for $i_a + 1 \leq k \leq L$, and $k \neq i_b$ and $p'_{i_b} = c$. Again we have $B'_t = B_t$, for $i_b < t < L$, and we note that $p_{i_b} = \phi$, whereas $p'_{i_b} = c$. However, since $p_{i_a} = c$ and $p'_{i_a} = \phi$, therefore in P we have costs $c_X(a,0)$, $c_X(c,i_a)$, and $c(\phi,i_b)$, and in P' we have costs $c_X(b,0)$, $c(\phi,i_a)$, and $c_X(c,i_b)$. The relation $O_{P'} < O_P$ still holds. The above total I/O costs of page replacements in policy P and P' are illustrated in Figure C.7.

Case B: $i_b < i_a$. In this case we consider the following two subcases:

Case B1. $p_j = b$ where p_j is the first occurrence of b in P, and $1 \leq j < i_b$. Here we set $p'_k = p_k$, for $1 \leq k \leq L$ and $k \neq j$, and $p'_j = a$. This results in $B_t = T_t + \{b\}$ and $B'_t = T_t + \{a\}$, for $0 \leq t \leq j-1$ and $B_t = B'_t$, for $j \leq t \leq L$. Since pages a and b are both not referenced up to time j, it should be clear that P' is a valid demand policy.

Also, since in P we have costs $c_X(a,0)$ and $c_X(b,j)$, and in P' we have costs

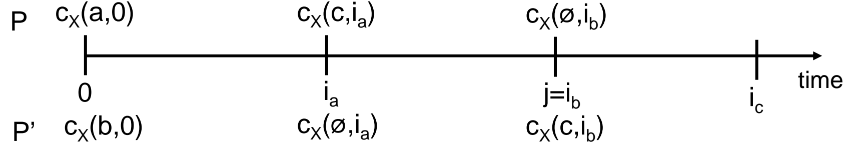


Figure C.7: Total I/O Costs of Page Replacements in Policy P and P'

$c_X(b,0)$ and $c_X(a,j)$, therefore $O_{P'} = O_P$. The above total I/O costs of page replacements in policy P and P' are illustrated in Figure C.8.

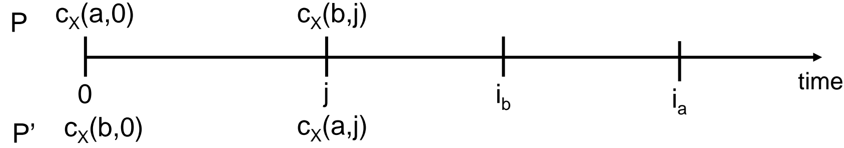


Figure C.8: Total I/O Costs of Page Replacements in Policy P and P'

Case B2. $p_j \neq b$, for $j < i_b$. In this case we set $p'_k = p_k$, for $1 \leq k \leq L$ and $k \neq i_b$. We also set $p'_{i_b} = a$. As in Case B1, P' is a valid demand policy.

Also, since in P we have costs $c_X(a,0)$ and $c(\phi, i_b)$, and in P' we have costs $c_X(b,0)$ and $c_X(a,i_b)$, therefore $O_{P'} = O_P$. The above total I/O costs of page replacements in policy P and P' are illustrated in Figure C.9.

Case C: $i_b = i_a = L+1$. Here we consider the following two sub-cases:

Case C1. $p_j = b$ where p_j is the first occurrence of b in P. We set $p'_k = p_k$, for $1 \leq k \leq L$ and $k \neq j$; and set $p'_j = a$. This results in $B_t = T_t + \{b\}$ and $B'_t = T_t + \{a\}$, for $0 \leq t \leq j-1$; and $B_t = B'_t$, for $j \leq t \leq L$.

Also, since in P we have costs $c_X(a,0)$ and $c_X(b,j)$, and in P' we have costs $c_X(b,0)$ and $c_X(a,j)$, therefore $O_{P'} = O_P$. The above total I/O costs of page replacements in policy P and P' are illustrated in Figure C.10.

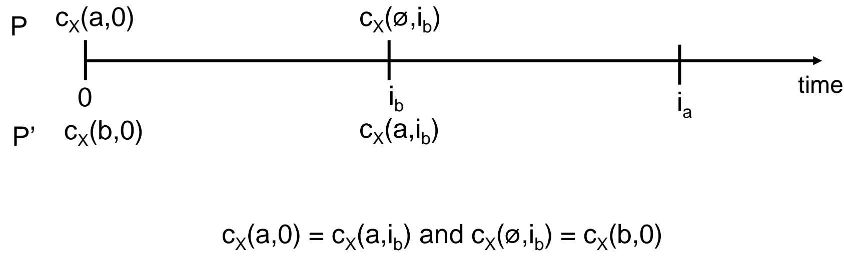


Figure C.9: Total I/O Costs of Page Replacements in Policy P and P'

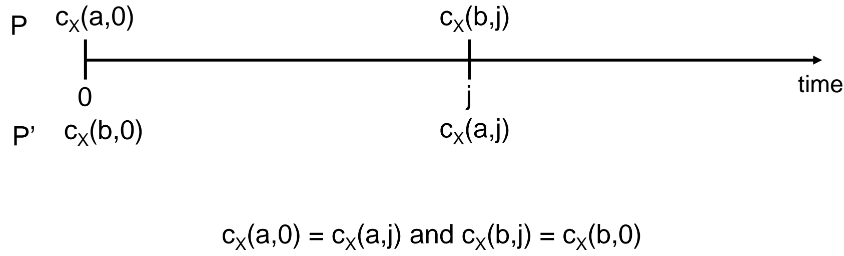


Figure C.10: Total I/O Costs of Page Replacements in Policy P and P'

Case C2. b does not occur in P. We set $p'_k = p_k$, for $1 \leq k \leq L$. This results in $B_t = T_t + \{b\}$ and $B'_t = T_t + \{a\}$, for $0 \leq t \leq L$.

Also, since in P we have cost $c_X(a, 0)$ and in P' we have cost $c_X(b, 0)$, therefore $O_{P'} < O_P$. The above total I/O costs of page replacements in policy P and P' are illustrated in Figure C.11.

To summarize, the above cases are complete to describe the states of the cache managed by policy P. In all the cases we can construct a demand policy P' such that $O_{P'} \leq O_P$.

Proof of Lemma 5.3 has been completed.

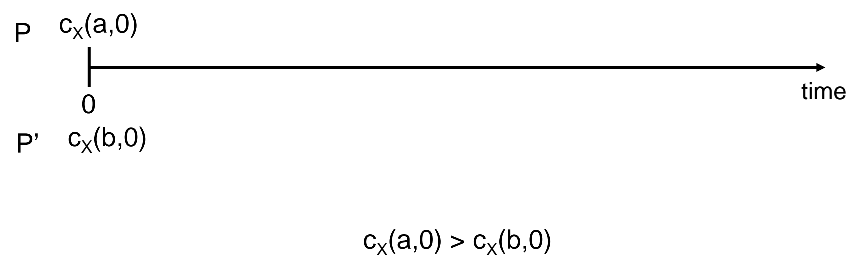


Figure C.11: Total I/O Costs of Page Replacements in Policy P and P'

Appendix D

Proof of the Correctness of Theorem 5.1

We adopt the same assumptions and definitions made in Chapter 5 and restate the Theorem 5.1 as follows:

- Let C be a cache, X be a page request trace received by C, B_0 be an initial state of C, P be a valid demand policy for $C, X, and B_0$, and at time t the policy P makes a decision to replace page a with $c_X(a,t) > 0$, and at the same time there is another cached page b with $c_X(b,t) = 0$, then there must exist another policy P' such that $O_{P'} \leq O_P$.

Proof:

Given P we construct P' corresponding to $C, X, and B_0$, such that $p'_i = p_i$ for $1 \leq i \leq t$. We know that the two policies, P and P' generate the same number of I/Os during the time period from 1 to t:

$\sum_{i=1}^{t} c_X(p_i, i) = \sum_{i=1}^{t} c_X(p'_i, i)$

Treat B_t and B'_t as the two initial cache states in Lemma 5.3, treat the remaining trace $x_{t+1}, x_{t+2}, ..., x_L$ as the request trace in Lemma 5.3, and treat the remaining policy $p_{t+1}, p_{t+2}, ..., p_L$ as the given policy in in Lemma 5.3, then based on Lemma 5.3, we can always find another policy P'' such that

$\sum_{i=t+1}^{L} c_X(p''_i, i) \leq \sum_{i=t+1}^{L} c_X(p_i, i)$

Let $p'_i = p''_i$ for $t+1 \leq i \leq L$, then the construction of policy P' has been completed and we have:

$\sum_{i=0}^{L} c_X(p'_i, i) \leq \sum_{i=0}^{t} c_X(p_i, i)$

The proof has been completed.

Appendix E

Proof of the Correctness of Lemma 5.5

We adopt the same assumptions and definitions made in Chapter 5 and restate the Lemma 5.5 as follows:

Let X be a page request trace, B_0 and B'_0 be two different initial cache states where:

$B'_0 = T_0 + \{a\}$

$B_0 = T_0 + \{b\}$

for $T_0 \subseteq A$, $a, b \notin T_0$, $d_X(a, 0) < d_X(b, 0)$, and $c_X(a, 0) \geq c_X(b, 0) > 0$. For any demand policy P, corresponding to X and B_0, there must exist a demand policy P', corresponding to X and B'_0, such that $O_{P'} \leq O_P$

Proof:

Given P, we construct P'. Suppose page a first occurs in the request trace X at time i_a, and b at i_b. Thus,$i_a < i_b \leq L$ is assumed. If either page a or b does not occur in X, then set i_a or i_b equal to $L + 1$. We consider following three cases:

Case 1. $p_j = b$ where p_j is the first occurrence of b in P, and $1 \leq j < i_a$. Here we set $p'_k = p_k$, for $1 \leq k \leq L$ and $k \neq j$, and $p'_j = a$. This results in $B_t = T_t + \{b\}$ and $B'_t = T_t + \{a\}$, for $0 \leq t \leq j - 1$ and $B_t = B'_t$, for $j \leq t \leq L$. Since pages a and b are both not referenced up to time j, it should be clear that P' is a valid demand policy.

Also, since in P we have costs $c_X(a, 0)$ and $c_X(b, j)$, and in P' we have costs $c_X(b, 0)$ and $c_X(a, j)$, therefore $O_{P'} = O_P$. The above total I/O costs of page replacements in policy P and P' are illustrated in Figure E.1.

Case 2. $p_{i_a} = b$ where p_{i_a} is the first occurrence of b in P. In this case we set $p'_k = p_k$, for $1 \leq k \leq L$ and $k \neq i_a$. We also set $p'_{i_a} = \phi$. As in Case 1, P' is a valid demand policy.

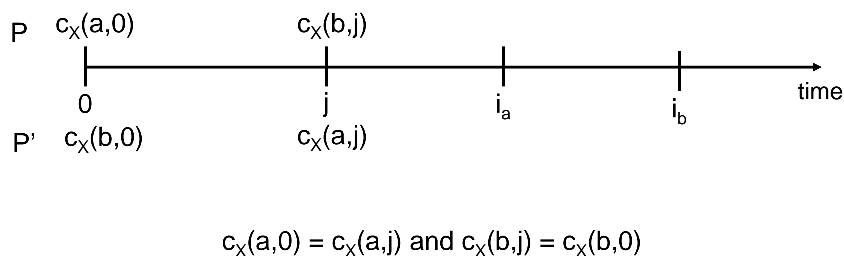


Figure E.1: Total I/O Costs of Page Replacements in Policy P and P'

Also, since in P we have costs $c_X(a,0)$ and $c_X(b,i_a)$, and in P' we have costs $c_X(b,0)$ and $c(\phi,i_a)$, therefore $O_{P'} < O_P$. The above total I/O costs of page replacements in policy P and P' are illustrated in Figure E.2.

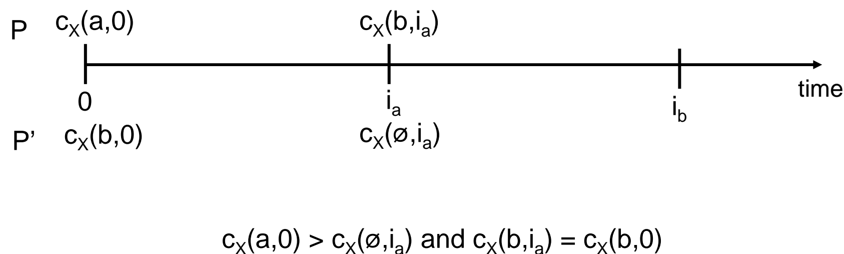


Figure E.2: Total I/O Costs of Page Replacements in Policy P and P'

Case 3. $p_j \neq b$, for $1 \leq j \leq i_a$. Then we must have $p_{i_a} = c$ where c is a page other than a or b. At time $t = i_a$ the states of the caches are given by:

$B'_{i_a} = T_{i_a} + \{a\}$

$B_{i_a} = T_{i_a} + \{b\} + \{a\} - \{c\}$

which can also be written as follows:

$B'_{i_a} = [T_{i_a} + \{a\} - \{c\}] + \{c\}$

$B_{i_a} = [T_{i_a} + \{a\} - \{c\}] + \{b\}$

We consider the following two subcases:

Case 3A. If $d_X(c, i_a+1) < d_X(b, i_a+1)$. Let i_c be the time of the first occurrence of c in X, then $i_c < i_b$. In this subcase we consider three more sub-subcases.

Case 3A1. If $p_j = b$ where p_j is the first occurrence of b in P and $j < i_c$, then we set $p'_k = p_k$, for $i_a + 1 \leq k \leq L$ where $k \neq j$ and $p'_j = c$.

Since in P we have costs $c_X(a, 0)$, $c_X(c, i_a)$, and $c_X(b, j)$, and in P' we have costs $c_X(b, 0)$, $c(\phi, i_a)$, and $c_X(c, j)$, therefore $O_{P'} < O_P$. The above total I/O costs of page replacements in policy P and P' are illustrated in Figure E.3.

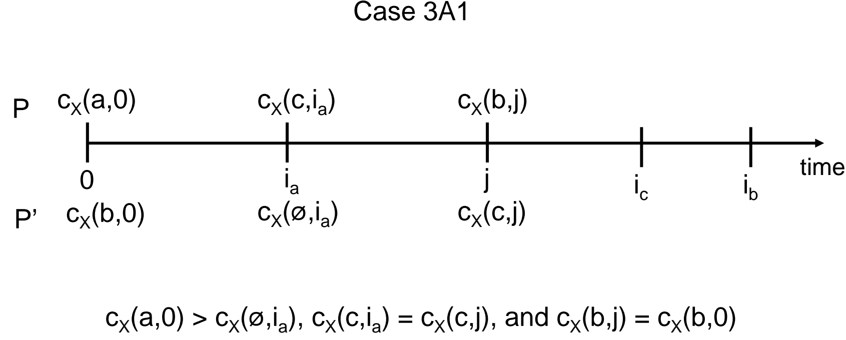


Figure E.3: Total I/O Costs of Page Replacements in Policy P and P'

Case 3A2. If $p_j = b$ where p_j is the first occurrence of b in P and $j = i_c$, then we set $p'_k = p_k$, for $i_a + 1 \leq k \leq L$ where $k \neq j$ and $p'_j = c$.

Also, since in P we have costs $c_X(a, 0)$, $c_X(c, i_a)$, and $c_X(b, i_c)$, where in P' we have costs $c_X(b, 0)$, $c(\phi, i_a)$, and $c(\phi, i_c)$, therefore $O_{P'} < O_P$. The above total I/O costs of page replacements in policy P and P' are illustrated in Figure E.4.

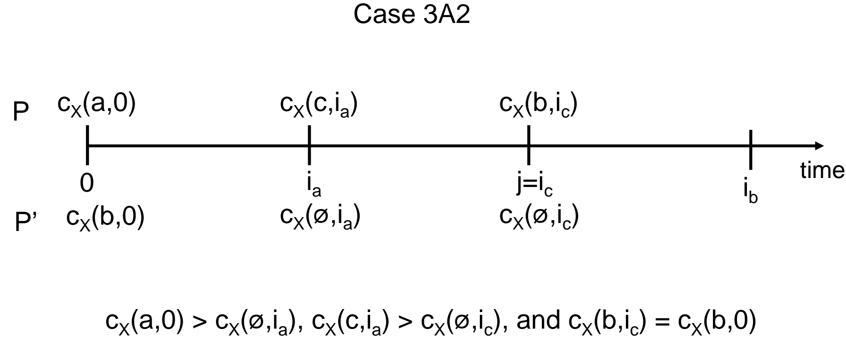


Figure E.4: Total I/O Costs of Page Replacements in Policy P and P'

Case 3A3. If $p_j = b$ where p_j is the first occurrence of b in P and $j > i_c$, we must have $p_{i_c} = d$, where d is a page other than b, or c (it could be a, because $i_a < i_c$). At time $t = i_c$ the states of the caches are given by:

$B'_{i_c} = T_{i_c} + \{c\}$

$B_{i_c} = T_{i_c} + \{b\} + \{c\} - \{d\}$

Which can also be written as follows:

$B'_{i_c} = [T_{i_c} + \{c\} - \{d\}] + \{d\}$

$B_{i_c} = [T_{i_c} + \{c\} - \{d\}] + \{b\}$

Please note that this is the same case as we met at Case 3. Since i_c is strictly larger than i_a, and the length of trace X is finite, thus this situation can only occur a finite number of times. Also note that P' is valid as far as it is specified and that $p'_1, ..., p'_{i_c}$ generates less I/O cost than $p_1, ..., p_{i_c}$. Some of the total I/O costs of page replacements in policy P and P' are illustrated in Figure E.5.

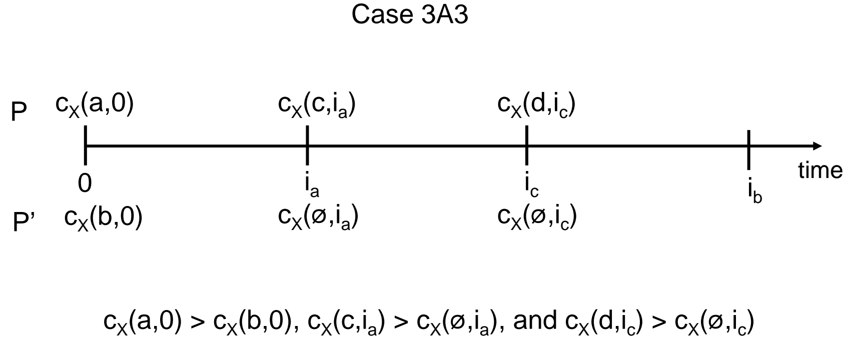


Figure E.5: Total I/O Costs of Page Replacements in Policy P and P'

Case 3B. If $d_X(c, i_a + 1) > d_X(b, i_a + 1)$, we set $p'_k = p_k$ for $1 < k < i_a - 1$ and $p'_{i_a} = \phi$ and consider two more sub-subcases:

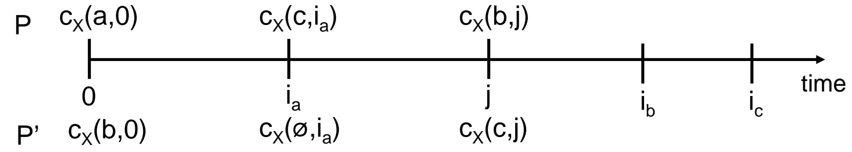
Case 3B1. If $p_j = b$, where p_j is the first occurrence of b in P and $j < i_b$, we set $p'_k = p_k$, for $i_a + 1 \leq k \leq L$, and $k \neq j$ and $p'_j = c$. Here $B'_t = B_t$, for $j \leq t \leq L$, and as in Case 1, we see that $O_{P'} < O_P$ still holds. Some of the total I/O costs of page replacements in policy P and P' are illustrated in Figure E.6.

Case 3B2. If $p_j \neq b$, for $j < i_b$, we set $p'_k = p_k$, for $i_a + 1 \leq k \leq L$, and $k \neq i_b$ and $p'_{i_b} = c$. Again we have $B'_t = B_t$, for $i_b < t < L$, and we note that $p_{i_b} = \phi$, whereas $p'_{i_b} = c$. However, since $p_{i_a} = c$ and $p'_{i_a} = \phi$, therefore in P we have costs $c_X(a, 0)$, $c_X(c, i_a)$, and $c(\phi, i_b)$, and in P' we have costs $c_X(b, 0)$, $c(\phi, i_a)$, and $c_X(c, i_b)$. The relation $O_{P'} \leq O_P$ still holds. The above total I/O costs of page replacements in policy P and P' are illustrated in Figure E.7.

To summarize, the above cases are complete to describe the states of the cache managed by policy P. In all the cases we can construct a demand policy P' such that $O_{P'} \leq O_P$.

Proof of Lemma 5.5 has been completed.

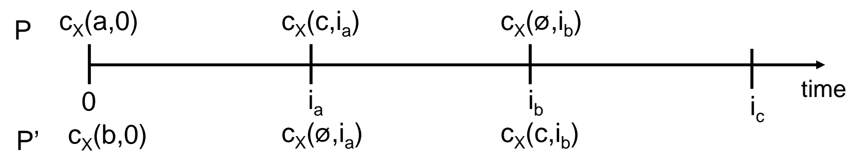
Case 3B1



$c_X(a,0) > c_X(\varnothing,i_a)$, $c_X(c,i_a) = c_X(c,j)$, and $c_X(b,j) = c_X(b,0)$

Figure E.6: Total I/O Costs of Page Replacements in Policy P and P'

Case 3B2



$c_X(a,0) > c_X(b,0)$, $c_X(c,i_a) = c_X(c,i_b)$, and $c_X(\varnothing,i_b) = c_X(\varnothing,i_a)$

Figure E.7: Total I/O Costs of Page Replacements in Policy P and P'

References

[1] Linux kernel. www.kernel.org. 1, 44

[2] Sedat Akyürek and Kenneth Salem. Management of partially safe buffers. *IEEE Trans. Comput.*, 44(3):394–407, March 1995. 91

[3] American National Standards Institute (ANSI). *ANSI SCSI Standard*, x3.131 - 1994 (r1999) edition, 1999. 33, 34

[4] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, October 2001. 89

[5] O. Bahat and A.M. Makowski. Optimal replacement policies for nonuniform cache objects with optional eviction. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, pages 427 – 437 vol.1, 2003. 94

[6] Lakshmi N. Bairavasundaram, Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*, June 2004. 90

[7] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with adaptive replacement. In *Proc. of the 3nd USENIX Symposium on File and Storage Technologies (FAST'04)*, March 2004. 88

[8] Alexandros Batsakis, Randal Burns, Arkady Kanevsky, James Lentini, and Thomas Talpey. AWOL: an adaptive write optimizations layer. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 67 – 80, 2008. 14, 54, 91

[9] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966. 55, 93

[10] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987. 8, 79, 90

[11] Kristof Beyls and Erik H. D'Hollander. Reuse distance-based cache hint selection. In *Proceedings of the 8th International Euro-Par Conference*, pages 265–274, 2002. 4, 87, 89

[12] R. Bonilla-Lucas, P. Plachta, A. Sachedina, D. Jimenez-Gonzalez, C. Zuzarte, and J.-L. Larriba-Pey. Characterization of the data access behavior for TPC-C traces. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 115–122, 2004. 88

[13] Nathan C. Burnett, John Bent, Andrea C. Arpaci-dusseau, and Remzi H. Arpaci-dusseau. Exploiting gray-box knowledge of buffer-cache management. In *Proceedings of the USENIX Annual Technical Conference (USENIX 02)*, pages 29–44, 2002. 90

[14] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. SSD bufferpool extensions for database systems. *Proc. VLDB Endow.*, 3(1-2):1435–1446, September 2010. 94

[15] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Trans. Comput. Syst.*, 14(4):311–343, November 1996. 89

[16] Pei Cao, Edward W. Felten, and Kai Li. Implementation and performance of application-controlled file caching. In *PROCEEDINGS OF THE FIRST SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION*, pages 165–178, 1994. 4, 89

[17] Pei Cao and Sandy Irani. Cost-Aware WWW Proxy Caching Algorithms. In *PROCEEDINGS OF THE 1997 USENIX SYMPOSIUM ON INTERNET TECHNOLOGY AND SYSTEMS*, pages 193–206, 1997. 94

[18] Enrique V. Carrera and Ricardo Bianchini. Improving Disk Throughput in Data-Intensive Servers. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, HPCA '04, pages 130–141, 2004. 4, 89

[19] Scott D. Carson and Sanjeev Setia. Analysis of the periodic update write policy for disk cache. *IEEE Transactions on Software Engineering*, 18(1):44–54, January 1992. 91

[20] Kerhong Chen, Richard B. Bunt, and Derek L. Eager. Write caching in distributed file systems. In *Proccedings of the 15th IEEE International Conference on Distributed Computing Systems*, pages 457–466, 1995. 1. 3, 15, 87, 88

[21] Peter M. Chen, Wee Teck Ng, Gurushankar Rajamani, and Christopher M. Aycock. The Rio File Cache: Surviving Operating System Crashes. In *Proc.*

7th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 74–83, 1996. 91

[22] Zhifeng Chen, Yan Zhang, Yuanyuan Zhou, Heidi Scott, and Berni Schiefer. Empirical evaluation of multi-level buffer cache collaboration for storage systems. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'05)*, pages 145–156, 2005. 3, 4, 15, 25, 87, 88, 89

[23] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. Eviction based cache placement for storage caches. In *Proceedings of the USENIX 2003 Annual Technical Conference*, pages 269–282, June 2003. 15, 90

[24] Hong-Tai Chou and David J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the 11th International Conference on Very Large Data Bases (VLDB)*, pages 127–141, August 1985. 88

[25] Andrew de los Reyes, Christopher Frost, Eddie Kohler, Mike Mammarella, and Lei Zhang. The KudOS architecture for file systems. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 1–14, 2005. 92

[26] Biplob Debnath, Mohamed F. Mokbel, David J. Lilja, and David Du. Deferred updates for flash-based storage. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–6, 2010. 91

[27] Robin Dhamankar and Hanuma Kodavalla. InProcDiskSim: testing database recovery on commodity disk drives. In *Proceedings of the Second International Workshop on Testing Database Systems*, DBTest '09, pages 1:1–1:6, 2009. 92

[28] Robin Dhamankar, Hanuma Kodavalla, and Vishal Kathuria. Enforcing Database Recoverability on Disks that Lack Write-Through. Technical Report MSR-TR-2008-36, Microsoft, March 2008. 92

[29] Michael J. Franklin, Michael J. Carey, and Miron Livny. Global memory management in client-server database architectures. In *Proc. International Conference on Very Large Data Bases (VLDB'92)*, pages 596–609, 1992. 3, 15, 25, 87, 88

[30] Michael J. Franklin, Michael J. Zwilling, C. K. Tan, Michael J. Carey, and David J. DeWitt. Crash recovery in client-server exodus. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, SIGMOD '92, pages 165–174, 1992. 8, 79, 90

[31] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized file system dependencies. In *Proc. of ACM Symposium on Operating System Principles*, pages 307–320, 2007. 92

[32] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Trans. Comput. Syst.*, 18(2):127–153, May 2000. 90

[33] Gregory R. Ganger and Yale N. Patt. Metadata update performance in file systems. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, OSDI '94, pages 49–60, 1994. 90

[34] Binny S. Gill. On multi-level exclusive caching: offline optimality and why promotions are better than demotions. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 49–65, 2008. 15, 90

[35] Binny S. Gill and Dharmendra S. Modha. Wow: wise ordering for writes - combining spatial and temporal locality in non-volatile caches. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, pages 129–142, 2005. 94

[36] Hongfei Guo. *"Good Enough" Database Caching*. PhD thesis, University of Wisconsin at Madison, 2005. 92

[37] Christoffer Hall and Philippe Bonnet. Getting priorities straight: improving Linux support for database I/O. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 1116–1127, 2005. 92

[38] Xubin He, Li Ou, Martha J. Kosa, Stephen L. Scott, and Christian Engelmann. A unified multiple-level cache for high performance storage systems. *International Journal of High Performance Computing and Networking*, 5(1/2):97–109, November 2007. 4, 87, 89

[39] Zhen He and Prakash Veeraraghavan. Fine-grained updates in database management systems for flash memory. *Information Sciences*, 179(18):3162–3181, August 2009. 93

[40] Jingtong Hu, Chun Jason Xue, Wei-Che Tseng, Qingfeng Zhuge, and Edwin H. M. Sha. Minimizing write activities to non-volatile memory via scheduling and recomputation. In *Proceedings of the 2010 IEEE 8th Symposium on Application Specific Processors (SASP)*, SASP '10, pages 101–106, 2010. 94

[41] IEEE Computer Society and The Open Group. *Portable Operating System Interface (POSIX) System Interfaces*, ieee std 1003.1, 2004 edition edition, 2004. 31

[42] International Business Machines Corporation. *IBM DB2 Universal Database Administration Guide: Performance*, version 8.2 edition. 6, 14, 38, 79, 90

[43] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 60–71, 2010. 88

[44] Martin Jambor, Tomas Hruby, Jan Taus, Kuba Krchak, and Viliam Holub. Implementation of a Linux log-structured file system with a garbage collector. *ACM SIGOPS Operating Systems Review*, 41(1):24–32, January 2007. 90

[45] Jaeheon Jeong and Michel Dubois. Cost-sensitive cache replacement algorithms. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, pages 327–337, 2003. 93

[46] Jaeheon Jeong and Michel Dubois. Cache replacement algorithms with nonuniform miss costs. *IEEE Transactions on Computers*, 55(4):353–365, April 2006. 94, 95

[47] Song Jiang, Kei Davis, and Xiaodong Zhang. Coordinated multilevel buffer cache management with consistent access locality quantification. *IEEE Transactions on Computers*, 56(1):95–108, January 2007. 4, 89

[48] Song Jiang and Xiaodong Zhang. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS'02)*, pages 31–42, June 2002. 88

[49] Song Jiang and Xiaodong Zhang. ULC: A file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS 2004)*, pages 168–177, March 2004. 89

[50] Xin Jin, Sanghyuk Jung, and Yong Ho Song. Write-aware buffer management policy for performance and durability enhancement in NAND flash memory. *Consumer Electronics, IEEE Transactions on*, 56(4):2393–2399, 2010. 93

[51] Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proc. International Conference on Very Large Data Bases (VLDB'94)*, pages 439–450, 1994. 88

[52] Yi-Reun Kim, Kyu-Young Whang, and Il-Yeol Song. Page-differential logging: an efficient and dbms-independent approach for storing data into flash memory. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 363–374, 2010. 93

[53] Ioannis Koltsidas and Stratis D. Viglas. Flashing up the storage layer. *Proceedings of the International Conference on Very Large Databases 2008(PVLDB'08)*, 1(1):514–525, August 2008. 94, 95

[54] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The linux implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev.*, 40(3):102–107, July 2006. 90

[55] Kyu Hyung Lee, In Hwan Doh, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Write-aware buffer cache management scheme for nonvolatile ram. In *Proceedings of the third conference on IASTED International Conference: Advances in Computer Science and Technology*, ACST'07, pages 29–35, 2007. 93

[56] Sehwan Lee, Kern Koh, and Hyokyung Bahn. Unifying buffer replacement and prefetching with data migration for heterogeneous storage devices. In *Proceedings of the 2010 IEEE 16th International Conference on Parallel and Distributed Systems*, ICPADS '10, pages 330–337, 2010. 93

[57] Xuhui Li, Ashraf Aboulnaga, Kenneth Salem, Aamer Sachedina, and Shaobo Gao. Second-tier cache management using write hints. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, pages 115–128, 2005. 88, 90

[58] Yu Li, Jianliang Xu, Byron Choi, and Haibo Hu. StableBuffer: optimizing write performance for DBMS applications on flash devices. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, CIKM '10, pages 339–348, 2010. 91

[59] Zhenmin Li, Zhifeng Chen, and Yuanyuan Zhou. Mining block correlations to improve storage performance. *ACM Transactions on Storage (TOS)*, 1(2):213–245, May 2005. 88

[60] Zhi Li, Peiquan Jin, Xuan Su, Kai Cui, and Lihua Yue. CCF-LRU: a new buffer replacement algorithm for flash memory. *Consumer Electronics, IEEE Transactions on*, 55(3):1351 –1359, august 2009. 94

[61] Xin Liu, Ashraf Aboulnaga, Kenneth Salem, and Xuhui Li. CLIC: client-informed caching for storage servers. In *Proccedings of the 7th conference on File and storage technologies*, pages 297–310, 2009. 90

[62] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, June 1970. 58

[63] Marshall Kirk Mckusick and Gregory R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of 1999 USENIX Annual Technical Conference*, pages 1–17, 1999. 90

[64] Nimrod Megiddo and Dharmendra S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the Second USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 115–130, March 2003. 53, 70, 88

[65] Microsoft Corporation. *Microsoft SQL Server 2000 Books Online*. 6, 14, 90

[66] Jeffrey C. Mogul. A better update policy. In *Proceedings of the USENIX Summer 1994 Technical Conference*, USTC'94, pages 99–111, 1994. 14, 54, 90, 91

[67] C. Mohan and Inderpal Narang. ARIES/CSA: a method for database recovery in client-server architectures. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, SIGMOD '94, pages 55–66, 1994. 8, 79, 90

[68] V.G. Moshnyaga, Hua Vo, G. Reinman, and M. Potkonjak. Reducing Energy of DRAM/Flash Memory System by OS-controlled Data Refresh. In *Proceedings IEEE International Symposium on Circuits and Systems - ISCAS*, pages 2108 –2111, May 2007. 93

[69] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems - or - your cache ain't nuthin' but trash. In *Proceedings of the USENIX Winter Conference*, pages 305–313, January 1992. 3, 15, 25, 87, 88

[70] D.A. Muntz, P. Honeyman, and C.J. Antonelli. In *In Proc. of IFIP/IEEE Intl. Conf. on Distributed Platforms*, pages 415 –429, 1996. 92

[71] Wee Teck Ng and Peter M. Chen. Integrating reliable memory in databases. *The VLDB Journal*, 7(3):194–204, August 1998. 91

[72] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 1–14, 2006. 92

[73] K. Norvag and K. Bratbergsengen. Write optimized object-oriented database systems. In *Proceedings of the 17th International Conference of the Chilean Computer Science Society*, SCCC '97, pages 164–173, 1997. 94

[74] Chris Olston and Jennifer Widom. Best-effort cache synchronization with source cooperation. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, SIGMOD '02, pages 73–84, 2002. 91

[75] 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 ACM SIGMOD International Conference on Management of Data (SIGMOD'93)*, pages 297–306, 1993. 88

[76] Oracle Corporation. *MySQL 5.1 Reference Manual*. 6, 14, 38

[77] Oracle Corporation. *Oracle Database Concepts*, version 10g release 2 (10.2) edition, June 2005. 6, 14, 90

[78] Yi Ou and Theo Härder. Clean first or dirty first?: a cost-aware self-adaptive buffer replacement policy. In *Proceedings of the Fourteenth International Database Engineering, Applications Symposium*, IDEAS '10, pages 7–14, 2010. 94

[79] Yi Ou, Theo Härder, and Daniel Schall. Performance and power evaluation of flash-aware buffer algorithms. In *Proceedings of the 21st international conference on Database and expert systems applications: Part I*, DEXA'10, pages 183–197, 2010. 95

[80] John Ousterhout and Fred Douglis. Beating the I/O bottleneck: a case for log-structured file systems. *ACM SIGOPS Operating Systems Review*, 23(1):11–28, January 1989. 90

[81] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Io-lite: a unified i/o buffering and caching system. *ACM Transactions on Computer Systems (TOCS)*, 18(1):37–66, February 2000. 4, 87, 88

[82] Chong-Mok Park, Kyu-Young Whang, Jeong-Joon Lee, and Il-Yeol Song. A cost-based buffer replacement algorithm for object-oriented database systems. *Information Sciences*, 138(1-4):99–118, August 2001. 94

[83] Junseok Park, Hyunkyong Choi, Hyokyung Bahn, and Kern Koh. Buffer Caching Algorithms for Storage Class RAMs. *International Journal of Computers*, 3(1):41–52, 2009. 94

[84] Junseok Park, Kern Koh, Hyunkyoung Choi, and Hyokyung Bahn. NBM: an efficient cache replacement algorithm for nonvolatile buffer caches. In *Proceedings of the 8th conference on Applied computer scince*, pages 320–325, 2008. 94

[85] Junseok Park, Hyejeong Lee, Seunghwan Hyun, Kern Koh, and Hyokyung Bahn. A cost-aware page replacement algorithm for NAND flash based mobile embedded systems. In *Proceedings of the seventh ACM international conference on Embedded software*, EMSOFT '09, pages 315–324, 2009. 93

[86] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. CFLRU: a replacement algorithm for flash memory. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 234–241, 2006. 94

[87] P. Phunchongharn, S. Pornnapa, and T. Achalakul. File type classification for adaptive object file system. In *TENCON 2006. 2006 IEEE Region 10 Conference*, pages 1–4, 2006. 4, 89

[88] K.J. Richardson and M.J. Flynn. Strategies to improve i/o cache performance. In *Proceeding of the Twenty-Sixth Hawaii International Conference on System Sciences*, pages 31 – 39, January 1993. 4, 87, 89

[89] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992. 90

[90] Prasenjit Sarkar and John Hartman. Efficient cooperative caching using hints. In *Proceedings of the second USENIX symposium on Operating systems design and implementation*, OSDI '96, pages 35–46, 1996. 4, 89

[91] Prasenjit Sarkar and John H. Hartman. Hint-based cooperative caching. *ACM Transactions on Computer Systems*, 18(4):387–419, 2000. 4, 87, 89

[92] Russell Sears and Eric Brewer. Stasis: flexible transactional storage. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 29–44, 2006. 92

[93] Russell Sears and Eric Brewer. Segment-based recovery: write-ahead logging revisited. *Proc. VLDB Endow.*, 2(1):490–501, August 2009. 92

[94] Margo Seltzer and Michael Olson. Libtp: Portable, modular transactions for unix. In *Proceedings of the 1992 Winter Usenix*, pages 9–25, 1992. 92

[95] Serial ATA International Organization. *Serial ATA Revision 3.0 Specification*, revision 3.0 edition, 2009. 33, 34

[96] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Database-aware semantically-smart storage. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 239–252, 2005. 4, 89

[97] Muthian Sivathanu, Vijayan Prabhakaran, Florentina Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proceedings of the Second USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 73–88, March 2003. 90

[98] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending SSD lifetimes with disk-based write caches. In *Proceedings of the 8th USENIX conference on File and storage technologies*, FAST'10, pages 101–114, 2010. 91

[99] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling transactional file access via lightweight kernel extensions. In *Proccedings of the 7th conference on File and storage technologies*, pages 29–42, 2009. 92

[100] Xian Tang and Xiaofeng Meng. ACR: An Adaptive Cost-Aware Buffer Replacement Algorithm for Flash Storage Devices. In *Proceedings of the 2010 Eleventh International Conference on Mobile Data Management*, MDM '10, pages 33–42, 2010. 94

[101] Transaction Processing Performance Council. *TPC Benchmark C*, revision 5.4 edition, 2005. 22, 49

[102] W. Wang and R. Bunt. A self-tuning page cleaner for db2. In *Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, MASCOTS '02, pages 81–89, 2002. 90

[103] Wenguang Wang and Richard B. Bunt. Simulating db2 buffer pool management. In *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '00, pages 13–22, 2000. 90

[104] Theodore M. Wong and John Wilkes. My cache or yours? making storage more exclusive. In *USENIX Annual Technical Conference (USENIX 2002)*, pages 161–175, June 2002. 15, 90, 91

[105] W. A. Wong and J.-L. Baer. Modified LRU policies for improving second-level cache behavior. In *Proceedings of 6th Intl. Symp. on High-Performance Comp. Arch. (HPCA)*, pages 49–60, 2000. 88

[106] Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Extending ACID semantics to the file system. *ACM Transactions on Storage (TOS)*, 3(2):1–42, June 2007. 92

[107] Jiesheng Wu, Pete Wyckoff, and Dhabaleswar K. Panda. Demotion-based exclusive caching through demote buffering: design and evaluations over different networks. In *Proceedings of the international workshop on Storage network architecture and parallel I/Os*, SNAPI '03, pages 73–80, 2003. 15, 90, 91

[108] Yingjie Zhao and Nong Xiao. 2c: A new approach to design the second level buffer cache. In *PROCEEDINGS Of the Seventh International Conference on Grid and Cooperative Computing, GCC '08.*, pages 56–61, 2008. 88

[109] Da Zhou and Xiaofeng Meng. A flash-aware random write optimized database. In *Proceedings of the 2010 Eleventh International Conference on Mobile Data Management*, MDM '10, pages 276–278, 2010. 94

[110] Yuanyuan Zhou, Zhifeng Chen, and Kai Li. Second-level buffer cache management. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):505–519, July 2004. 3, 15, 25, 87, 88

[111] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the USENIX 2001 Annual Technical Conference*, pages 91–104, June 2001. 13, 17, 53, 88

[112] Qingbo Zhu and Yuanyuan Zhou. Power-aware storage cache management. *IEEE Transactions on Computers*, 54(5):587–602, 2005. 94