Optimizing Hierarchical Storage Management For Database System

by

Xin Liu

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, 2014

© Xin Liu 2014

Author's Declaration

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

Caching is a classical but effective way to improve system performance. To improve system performance, servers, such as database servers and storage servers, contain significant amounts of memory that act as a fast cache. Meanwhile, as new storage devices such as flash-based solid state drives (SSDs) are added to storage systems over time, using the memory cache is not the only way to improve system performance. In this thesis, we address the problems of how to manage the cache of a storage server and how to utilize the SSD in a hybrid storage system.

Traditional caching policies are known to perform poorly for storage server caches. One promising approach to solving this problem is to use hints from the storage clients to manage the storage server cache. Previous hinting approaches are ad hoc, in that a predefined reaction to specific types of hints is hard-coded into the caching policy. With ad hoc approaches, it is difficult to ensure that the best hints are being used, and it is difficult to accommodate multiple types of hints and multiple client applications. In this thesis, we propose CLient-Informed Caching (CLIC), a generic hint-based technique for managing storage server caches. CLIC automatically interprets hints generated by storage clients and translates them into a server caching policy. It does this without explicit knowledge of the application-specific hint semantics. We demonstrate using trace-based simulation of database workloads that CLIC outperforms hint-oblivious and state-of-the-art hint-aware caching policies. We also demonstrate that the space required to track and interpret hints is small.

SSDs are becoming a part of the storage system. Adding SSD to a storage system not only raises the question of how to manage the SSD, but also raises the question of whether current buffer pool algorithms will still work effectively. We are interested in the use of hybrid storage systems, consisting of SSDs and hard disk drives (HDD), for database management. We present cost-aware replacement algorithms for both the DBMS buffer pool and the SSD. These algorithms are aware of the different I/O performance of HDD and SSD. In such a hybrid storage system, the physical access pattern to the SSD depends on the management of the DBMS buffer pool. We studied the impact of the buffer pool caching policies on the access patterns of the SSD. Based on these studies, we designed a caching policy to effectively manage the SSD. We implemented these algorithms in MySQL's InnoDB storage engine and used the TPC-C workload to demonstrate that these cost-aware algorithms outperform previous algorithms.

Acknowledgments

I would not have been able to complete my Ph.D. dissertation without the support of many people. First, I owe my deepest gratitude and respect to my supervisor Kenneth Salem. His scholarship, attention to detail, enthusiasm, and immense knowledge have helped me in all stages of my doctoral studies. His continues encouragement and positive attitude made it not only possible to earn my PhD degree but also a delightful experience.

My thanks are also due to my committee members for their numerous insightful suggestions for the final revision of this dissertation. I would like to thank my thesis committee at the University of Waterloo, Tamer Ozsu, Ashraf Aboulnaga, and Ladan Tahvildari, and my external examiner Bettina Kemme from University of McGill.

Last but not least, I would like to thank my family for their support, constant love, understanding, and care.

Table of Contents

List of Figures viii

1 Introduction 1

1.1 Cache Hierarchies . . . 2

1.2 Hybrid Storage . . . 3

1.3 Contributions . . . 4

1.4 Organization of the Thesis . . . 5

2 Related Work 7

2.1 Second-Tier Cache Management . . . 7

2.1.1 Hierarchy-aware Approaches . . . 8

2.1.2 Aggressively Collaborative Approaches . . . 8

2.2 Solid State Disks . . . 11

3 CLIC: Client-Informed Caching for Storage Servers 15

3.1 Generic Framework for Hints . . . 16

3.2 Hint Analysis . . . 18

3.2.1 Hint Benefit/Cost Analysis . . . 20

3.2.2 Tracking Hint Set Statistics . . . 23

3.2.3 Time-Varying Workloads . . . 24

3.2.4 Cache Management . . . 24

3.3 Handling Large Numbers of Hint Sets . . . 25

3.3.1 Frequently-Occurring Hint Sets . . . 27

3.4 Experimental Evaluation . . . 29

3.4.1 Comparison to Other Caching Policies 32
3.4.2 Limiting the Outqueue Size 34
3.4.3 Tracking Only Frequent Hint Sets 35
3.4.4 Increasing the Number of Hints 36
3.4.5 Multiple Storage Clients 42
3.5 Conclusion 43

4 Dynamic Priority CLIC 45

4.1 Re-reference Histogram of Hint Sets 47
4.2 Dynamic Benefit/Cost Model 47
4.3 Tracking Hint Statistics 49
4.4 Cache Management 50
4.5 DP-CLIC Priority vs. CLIC Priority 51
4.6 Experimental Evaluation 55
4.6.1 Evaluation with TPC-C Traces 57
4.6.2 Evaluation with TPC-H traces 59
4.6.3 Limiting the Histogram Size 60
4.6.4 Tracking Only Frequent Hint Sets for DP-CLIC 61
4.6.5 Increasing the Number of Hints for DP-CLIC 65
4.7 Conclusion 66

5 Classification of Hint Sets 69

5.1 Hybrid Algorithms 71
5.2 Impurity of Hint Types 72
5.3 Experimental Evaluation 74
5.4 Conclusion 79

6 Hybrid Storage Management for Database Systems 80

6.1 System Overview 83
6.2 Buffer Pool Management 85
6.2.1 Implementation of GD2L on MySQL 86
6.3 The Impact of Cost-aware Caching 89

6.4 SSD Management . . . 90
6.4.1 CAC: Cost-Adjusted Caching . . . 91
6.4.2 The Miss Rate Expansion Factor . . . 94
6.4.3 Sequential I/O . . . 96
6.4.4 Implementation of SSD Management on MySQL . . . 96
6.4.5 Failure Handling . . . 98
6.5 Evaluation . . . 98
6.5.1 Methodology . . . 100
6.5.2 Cost Parameter Calibration . . . 101
6.5.3 Analysis of GD2L and CAC . . . 101
6.5.4 Comparison with LRU2 and MV-FIFO . . . 106
6.5.5 Impact of the Eviction Zone . . . 108
6.6 Conclusion . . . 109

7 Conclusion and Future Work 111
7.1 Future Work . . . 112

References 114

List of Figures

1.1 Prices and Performance of SSDs and HDDs
Data are from Graefe [27] in 2007. . . . 2

1.2 Architecture of Multi-tier Caches . . . 3

2.1 Trend of HDD and SSD Price [59] . . . 10

2.2 FTL and NAND Flash Memory
FTL emulates sector read/write functionalities of a hard disk to use conventional disk file systems in NAND flash memory . . . 11

3.1 System Architecture . . . 17

3.2 Types of Hints in the DB2 I/O Request Traces . . . 18

3.3 Types of Hints in the MySQL I/O Request Traces . . . 19

3.4 Hint Set Priorities for the DB2_C60 Trace
Each point represents a distinct hint set. All hint sets are shown. . . . 22

3.5 Priority Calculation Timeline . . . 24

3.6 Structures Used by CLIC
Arrows show possible movements among queues in response to I/O requests . . . 25

3.7 Hint-Based Server Cache Replacement Policy
This pseudo-code shows how the server handles a request for page p with hint set H and request sequence number s. . . . 26

3.8 Cumulative Hint Set Frequency in the DB2_C300_400 Trace . . . 27

3.9 I/O Request Traces.
The page sizes for the DB2 and MySQL databases were 4KB and 16KB, respectively. For the TPC-C workloads, the table shows the initial database size. The TPC-C database grows as the workload runs. . . . 31

3.10 Read Hit Ratio of Caching Policies for the DB2 TPC-C Workloads . . . 33

3.11 Read Hit Ratio of Caching Policies for the DB2 TPC-H Workloads . . . 35

3.12 Read Hit Ratio of Caching Policies for the MySQL Workloads 36

3.13 Effect of Outqueue Size on Read Hit Ratio
Each bar represents a different outqueue size. 37

3.14 Effect of Top-K Hint Set Filtering on Read Hit Ratio 38

3.15 Effect of Top-K Hint Set Filtering on Read Hit Ratio with TPC-C Workload Traces 40

3.16 Effect of Top-K Hint Set Filtering on Read Hit Ratio with TPC-H Workload Traces 41

3.17 Read Hit Ratio with Three Clients
Read hit ratio is near zero for the DB2_C300 and DB2_C540 traces in the 180K page shared cache, so bars are not visible. 43

4.1 Temporal Locality of Upper-tier and Lower-tier Cache Accesses Using Reuse Distance Histograms. (a) Auspex Client Trace and (b) Auspex Server trace[76]. 46

4.2 Read Reference Distance Histograms of DB2 TPC-C Workload Traces . . . 47

4.3 Illustration of Read Reference Histogram 48

4.4 Hint Set Priority Produced by Dynamic Benefit/Cost Model of DB2 TPC-C Workload Traces 50

4.5 The DP-CLIC Cache Replacement Policy
This pseudo-code shows how the server handles a request for page p with hint set H and request sequence number s. 52

4.6 Hint Set Priorities for the DB2_C60_400 Trace
Each point represents a distinct hint set. All hint sets are shown. 54

4.7 Hint Set Statistics for the DB2_C60_400 Trace
Each bubble represents a distinct hint set - not all hint sets in the trace are shown. Each bubble's radius is proportional to the frequency of its hint set. 54

4.8 DP-CLIC Priority vs. CLIC Priority 55

4.9 Read Hit Ratio of Caching Policies for the DB2 TPC-C Workloads 56

4.10 DP-CLIC Priority vs. CLIC Priority 57

4.11 DP-CLIC Priority vs. CLIC Priority 58

4.12 Read Hit Ratio of Caching Policies for the DB2 TPC-H Workloads 59

4.13 CLIC priority vs. DP-CLIC priority of DB2_H80_400 trace 60

4.14 Effect of Width of Buckets on Read Hit Ratio (number of bucket = 3000) . 62

4.15 Effect of Numbers of Buckets on Read Hit Ratio (width of bucket = 50000) 62

4.16 Effect of Numbers of Buckets on Read Hit Ratio (width of bucket = 10000) 62
4.17 Effect of Width of Buckets on Read Hit Ratio (number of bucket = 3000) . 63
4.18 Effect of Number of Buckets on Read Hit Ratio (width of bucket = 100000) 63
4.19 Effect of Top-K Hint Set Filtering on Read Hit Ratio (DP-CLIC) 64
4.20 Effect of Top-K Hint Set Filtering on Read Hit Ratio with TPC-C Workload Traces 67
4.21 Effect of Top-K Hint Set Filtering on Read Hit Ratio with TPC-H Workload Traces 68

5.1 Architecture of Data Center 70
5.2 Parameters of the Hybrid Algorithm 72
5.3 CLIC Read Hit Ratio 75
5.4 CLIC Read Hit Ratio 76
5.5 DP-CLIC Read Hit Ratio 77
5.6 DP-CLIC Read Hit Ratio 78

6.1 System Architecture (the arrows represent read/write requests) 81
6.2 The Management of the Buffer Pool and the SSD 84
6.3 Storage Device Parameters 86
6.4 The GD2L Algorithm
This pseudo-code shows how GD2L handles a request for page p. L is initialized as 0 87
6.5 Buffer Pool Managed by GD2L on MySQL 88
6.6 Miss Rate/Write Rate While on HDD vs. Miss Rate/Write rate while on SSD.
Each point represents one page 90
6.7 The Measured and Estimated Statistics of a Page
Note that t_S and t_D represent the total time that the page is on the SSD and not on the SSD. 92
6.8 Summary of Notation 93
6.9 Miss Rate Expansion Factor for Pages from Three TPC-C tables. 95
6.10 The Data Structures Used by the SSD Manager 97
6.11 TPC-C Throughput 102

6.12 Performance Statistics (DB size=30GB)
I/O is reported as ms. per New Order transaction. 103

6.13 Performance Statistics (DB size=15GB)
I/O is reported as ms. per New Order transaction. 103

6.14 Performance Statistics (DB size=8GB)
I/O is reported as milliseconds of device time per New Order transaction. . . . 104

6.15 TPC-C Throughput 107

6.16 Performance Statistics (DB size=30GB)
I/O is reported as ms. per New Order transaction. 108

6.17 Performance Statistics (DB size=15GB)
I/O is reported as milliseconds of device time per New Order transaction. . . . 108

6.18 Performance Statistics (DB size=8GB)
I/O is reported as milliseconds of device time per New Order transaction. . . . 109

6.19 Throughput of TPC-C Runs in term of Eviction Zone Size 109

Chapter 1

Introduction

In computing systems, one of the most serious performance bottlenecks is accessing data on the hard disk drive (HDD). For example, in a database system, much of a transaction's lifetime is spent on waiting to access data on disks. The time required to access data on the HDDs is determined by the mechanical nature of the rotating disks and by the moving disk arms. Caching is a classical but effective way to improve system performance. The purpose of caching is to keep frequently accessed data in a faster media, and thus, to reduce disk I/O traffic. Consequently, conventional storage systems consist of two levels: the memory and HDDs. As the memory is significantly faster and significantly more expensive than the HDD, a larger memory means more performance gains but more cost to the system. In 1987, Gray, et al. [29], argued that data accessed every five minutes or more should be resident in memory. The "five-minute rule" is based on the trade-off between the dollar cost of accessing data in the memory versus using HDD.

Over the past decades, the access time gap and cost gap between the memory and HDDs have continued to increase. Meanwhile, new storage devices have been introduced. Figure 1.1 lists prices and performance of RAM, flash-based Solid State Drives (SSDs) and HDDs presented by Graefe [27] in 2007, from which we see that the performance of HDDs largely lags behind the memory. As the price of flash memory has fallen dramatically, it has made inroads into the laptop market, desktop storage market and the enterprise server market. Currently, SSDs are about $17-32\times$ more expensive per GB, but about $50-150\times$ less expensive per random I/O per second than are hard disks. Graefe [27] reviewed the five-minute rule for trading off memory and I/O capacity based on prices and performance of SSDs and HDDs current at that time. He pointed out that the five-minute rule still held if flash-based Solid State Drives (SSDs) were introduced to fill the gap between RAM and HDDs. Flash can accordingly augment the system to form a new tier in the storage hierarchy.

As new storage devices, e.g. SSDs, are added to storage systems over time, using the memory cache is not the only way to improve system performance. On the storage side, although the cost of flash per gigabyte is falling quickly, it is still expensive to replace

	RAM	Flash disk	SATA disk
Price and capacity	$3 for 8x64 Mbit	$999 for 32GB	$80 for 250GB
Access latency		0.1ms	12ms average
Transfer bandwidth		66MB/s API	300 MB/s API
Active power		1W	10W
Idle power		0.1W	8W
Sleep power		0.1W	1W

Figure 1.1: Prices and Performance of SSDs and HDDs
Data are from Graefe [27] in 2007.

all hard disks with SSDs. Hence, rather than as a replacement for HDDs, SSDs should be viewed as a means to enhance them. Hard disks are cost-effective and better suited for sequential accesses, while SSDs can be targeted for high throughput of small random I/O requests, as SSDs will have significantly better performance on random data accesses. Heterogeneous storage systems have been investigated inside and outside of industry. For example, OCZ launched a hybrid PCIe SSD (the RevoDrive Hybrid) [56] which integrates 100GB SSD capacity along with an onboard tegabyte HDD. Heterogeneous storage systems are designed to balance performance and storage capacity on a limited budget.

1.1 Cache Hierarchies

In this thesis, we address two challenges in the storage hierarchy: First, how to manage the lower-tier cache in a multi-tier cache system? Multi-tier block caches arise in many situations in modern computing systems. One scenario is illustrated in Figure 1.2. Most DBMS have their own buffer pool for various reasons. At the same time, the applications may store their data in a file system, which also manages a cache. Besides this, if the storage is provided by a dedicated storage server, the storage server also uses its memory for data caching. When applications issue I/O requests, the requested data may go through all tiers of these caches before it reaches its destination. Management of these caches has a significant impact on overall system performance.

In Chapter 3 and Chapter 4, we discuss how to manage the lower-tier cache. As illustrated in Figure 1.2, whether it is over a network or used for local access, running a database management system (DBMS) on top of a storage server results in at least two tier caches. The challenges of making effective use of caches below the first-tier are well known [54, 70, 76]. Zhou et al. [76], investigated the access patterns of second-tier caches and found that they are different from those at the first-tier caches. Filtered by the upper-tier cache, the request streams to the lower-tier cache have poor temporal locality. Thus, locality-based replacement policies, such as Least Recently Used (LRU), and CLOCK, do not work well for the second-tier cache. Another difficulty in managing multiple tiers of caches is the inclusiveness problem [70]. Multi-tier caches do not lead to performance in

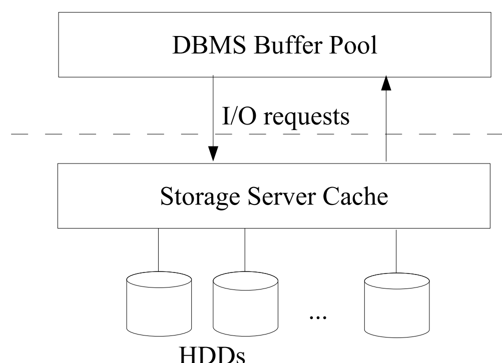


Figure 1.2: Architecture of Multi-tier Caches

proportion to their aggregate cache size, since caches fail to maintain exclusive content. Duplicated content in both the first-tier and second-tier caches result in wasting available cache space. To address these problems, many techniques have been proposed for improving the performance of second-tier caches. One promising class of techniques relies on *hinting*: the application that manages the first-tier cache generates hints and attaches them to the I/O requests that it directs to the second-tier. The cache at the second-tier then attempts to exploit these hints to improve its performance. Previous work has taken an *ad hoc* approach to hinting.

We provide two generic techniques for exploiting application hints to manage the second-tier cache. The first one, Client-informed Caching (CLIC), is discussed in Chapter 3. Unlike ad hoc techniques, CLIC does not hard-code responses to any particular type of hint. Instead, it is an adaptive approach that attempts to learn to exploit any type of hint supplied to it. Applications in the upper-tier cache are free to supply any hints that they believe to be of value to the lower-tier cache.

The second technique is dynamic priority CLIC (DP-CLIC), introduced in Chapter 4, which is an extension of CLIC. Both techniques analyze available hints and determine which can be exploited to improve performance. As any hints are allowed to pass to the lower-tier cache, tracking all hint sets may cause space and time overhead. Our techniques learn to ignore hints that do not help using a top-k algorithm (in Chapter 3) and a feature selection algorithm (in Chapter 5),

1.2 Hybrid Storage

The second question addressed in this thesis is how to place data in the memory and the SSD in a hybrid storage system. As SSDs become a new tier of the storage hierarchy, it is not uncommon for servers to consist of a memory cache and hybrid storage devices. For

example, a database server may use a hybrid storage system including a SSD and a HDD to improve performance while saving cost. Thus, besides managing its own buffer pool, the database server also needs to make data placement and replacement decisions for the SSD. The use of hybrid storage devices raises the question of how to make efficient use of both the memory and the SSD.

In Chapter 6, we are concerned with the use of hybrid (SSD and HDD) storage systems for database management. The promise of good performance for database workloads with SSDs [40] suggests that storage systems will likely continue to include SSDs and cost-effective HDDs. We consider hybrid storage systems in which the HDD and the SSD are both visible to the database management system (DBMS) buffer manager. As all data are stored in the HDD, the SSD forms a new tier between the buffer pool and the HDD. Thus, the DBMS buffer manager can use both the buffer pool and the SSD to improve system performance. However, managing both the buffer pool and SSD effectively is more difficult than managing one alone. The challenges arise from the impact of the buffer pool and the SSD on each other.

The hybrid storage management problem is different from the multi-tier cache problem. First, in the multi-tier cache problem, the upper-tier cache and the lower-tier cache belong to different servers, and are managed separately. Hints are passed for helping the lower-tier cache to understand the management of the upper-tier cache. However, the management of the lower-tier cache has no impact on the management of the upper-tier cache. In the hybrid storage management problem, the memory and the hybrid storage system belong to the same server. The buffer manager can make data placement and replacement decisions for both the memory and the SSD. Second, one key feature of flash memories is non-volatility, i.e. data stored on flash memory will not be lost even without power. Thus, the SSD can be also viewed as a part of the permanent storage. Unlike the memory cache in the storage server, dirty pages in the SSD do not need to be flushed to the HDD immediately.

We present cost-based caching algorithms for both the buffer pool and the SSD cache. To make replacement decisions for the buffer pool, the cost-aware algorithm considers not only page recency but also future retrieval costs. Correspondingly, the responsibility of the SSD caching algorithm is to identify pages with the largest access cost savings and place them in the SSD. As a characteristic of SSD is that it has fast random I/Os, caching hot pages in the SSD reduces the overall I/O access cost and improves the system performance.

1.3 Contributions

This thesis addresses the problems of two-tier cache management and hybrid storage management in database systems. In the area of two-tier cache management, it makes the following contributions:

- We define an on-line cost/benefit analysis of I/O request hints that can be used to

determine which hints provide potentially valuable information to the second-tier cache.

- We define an adaptive, priority-based cache replacement policy for the second-tier cache. This policy exploits the results of the hint analysis to improve the hit ratio of the second-tier cache.
- We extend CLIC's cost/benefit model to a dynamic cost/benefit model, on which dynamic priority CLIC (DP-CLIC) is based. The caching policy of DP-CLIC is priority-based, but DP-CLIC's priorities vary with reference distances.
- We propose to reduce the overhead of tracking hint set statistics using a frequency selection algorithm and a feature selection algorithm.
- We use trace-based simulation to provide a performance analysis of CLIC. Our results show that CLIC outperforms ad hoc hinting techniques and that its adaptivity can be achieved with low overhead. We also evaluate DP-CLIC by comparing it with CLIC. Our results show that DP-CLIC performs at least as well as CLIC in all cases, and outperforms CLIC for some traces.

In the area of hybrid storage management, it makes the following contributions:

- We present GD2L, a cost-aware algorithm for buffer pool management in DBMS with hybrid storage systems. GD2L takes the usual concerns of DBMS buffer management (exploiting locality, scan resistance) into account, but also considers the fact different devices in a hybrid storage system perform differently.
- We present CAC, an anticipatory cost-based technique for managing the SSD. Unlike previous techniques, CAC is intended to work together with a cost-aware buffer manager such as GD2L. The technique expects that moving a page into or out of the SSD will change the access pattern for that page, and it anticipates these changes when making SSD placement decisions.
- We present an empirical evaluation of GD2L and CAC. We have implemented both techniques in MySQL's InnoDB storage engine. We compare the performance of GD2L with of InnoDB's native buffer manager, which is oblivious to the location of pages in a hybrid storage system. We compare CAC to several alternatives, including a non-anticipatory cost-based technique and LRU2. Our evaluation uses transactional workloads.

1.4 Organization of the Thesis

The remainder of this thesis is structured as follows. Chapter 3 discusses Client-Informed Caching for Storage Servers. Chapter 4 introduces Dynamic Priority CLIC, an extension

of CLIC. Chapter 5 introduces how to classify hint sets with the feature selection algorithm to reduce the overhead for tracking hint set statistics. Chapter 6 presents the data management for hybrid database storage. Chapter 2 discusses the existing work related to the management of the lower-tier cache and the management of the SSD. Finally, Chapter 7 concludes the thesis.

Chapter 2

Related Work

Chapter 1 introduces two problems: the management of second-tier caches, and management of hybrid storage which includes SSD and HDD. In this chapter we survey other work related to these two problems.

2.1 Second-Tier Cache Management

Compared to other general-purpose replacement algorithms, e.g. Least Frequently Used (LFU), First In First Out (FIFO), and Most Recently Used (MRU), Least Recently Used (LRU) is the most widely used in practice. Based on these classical replacement algorithms, many general-purpose algorithms, such as 2Q [34], Adaptive Replacement Cache (ARC) [52], Clock with Adaptive Replacement (CAR) [4], and Multi-Queue (MQ) [76], have been proposed recently. Among them, ARC considers both recency and frequency in making replacement decisions, and adapts to changing workloads by automatically balancing recency and frequency. While any of the general-purpose algorithms can be used at any level of a cache hierarchy, researchers have recognized that cache management at lower-tiers of a hierarchy poses particular challenges. Many techniques have been proposed specifically for second (or lower) tier caches. For example, MQ was developed specifically for second-tier caches based on the access patterns of second-tier caches observed by Zhou et al. [76]. Since the requests to the second-tier cache have little temporal locality available, studies [52, 76] have shown that these new proposed algorithms (e.g. MQ, ARC) had better performance than LRU. ACME [1] is a mechanism that can be used to automatically and adaptively choose a good policy from among a pool of candidate policies, based on the recent performance of the candidates. Other approaches for second-tier cache management, presented in the following two subsections, have been classified by Chen et al. [13] as either hierarchy-aware or aggressively collaborative based on whether the approaches require changes to the first-tier.

2.1.1 Hierarchy-aware Approaches

Hierarchy-aware approaches, including eviction-based cache placement [14], X-RAY [3], and semantically-smart disks [2], exploit knowledge of the existence of the upper-tier cache, but they are transparent to it. An example of a hierarchy-aware technique is quick eviction of read pages from a second-tier cache, under the assumption that such pages are likely to remain in the first-tier cache. Eviction-based cache placement [14] was developed in the context of database systems. The storage server cache manager maintains a Client Content Track (CCT) table to estimate eviction information (which blocks have been evicted) for the storage client (DBMS) cache by monitoring the target memory location of each block request from the storage client. Instead of managing its cache in an on-demand fashion, which leads to cache misses. The storage server cache manager preloads the blocks evicted from the first-tier cache to the second-tier cache. Semantically-smart disks and X-RAY are related techniques which use gray-box methods [15], and they both assume the first tier is a file system. Semantically-smart disks [2] can exploit layout information (e.g. to categorize disk blocks as data blocks, inodes blocks, or superblocks) and extract semantic information for file systems that are similar to the Berkeley Fast File System. With this semantic information, it is able to infer higher-level file system behavior and improve caching performance by avoiding duplicating the contents of the file system cache. X-RAY [3] tracks the file system cache using a recency list. Like a semantically-smart disk, X-RAY can identify inode blocks (file meta-data) when they are flushed to the storage server cache, and can extract information (e.g. access and update timestamp) to update its recency list. Using the recency list, it can predict which blocks are likely to be in the first-tier cache, and which ones have been evicted. Like eviction-based cache replacement, X-RAY needs extra disk bandwidth to preload blocks evicted from the first-tier cache to the second-tier cache. X-RAY has also been used for DBMS clients [63].

2.1.2 Aggressively Collaborative Approaches

Aggressively collaborative approaches [13] require some modification to the first-tier. Wong et al. [70] observed the wasteful inclusiveness of second-tier caches. To maintain exclusivity among caches, they suggested adding a DEMOTE I/O operation to the SCSI command set in order to send blocks evicted from the first-tier to the second-tier cache. In contrast to the DEMOTE technique, which moves blocks down the cache hierarchy, PROMOTE [26] transfers blocks with high hit ratio from the lower-tier cache to the upper-tier cache gradually as they are read. Gill et al. [26] claimed that PROMOTE outperformed DEMOTE, especially in limited bandwidth scenarios as it avoided the bandwidth overheads caused by DEMOTE. PROMOTE must be used by all tiers, including the first-tier. This may be undesirable if the first-tier cache is managed by a database system or other application that prefers an application-specific policy for cache management.

Unified and Level-aware Caching (ULC) [33] uses a centrally-controlled cache placement and replacement protocol for multi-tier buffer caches. It gives complete responsibility for

management of the lower-tier caches to the first-tier cache manager, which moves blocks up or down the cache hierarchy using RETRIEVE or DEMOTE commands. Karma [72] relies on application hints to group blocks into "ranges", and then calculates the priority of each range based on its access pattern and replacement algorithm (e.g. LRU for random access pattern). Ranges with higher priorities will be maintained in the upper-tier cache. Each range is managed separately and exclusiveness is maintained by using DEMOTE and READ-SAVE commands. MC2 [71] applies Karma for managing multi-tier caches shared by multiple clients. Multiple clients raise another problem: the goal of the cache manager is not only to improve the overall system performance (e.g. latency, hit ratio), but also to consider the performance of each client. To achieve the goal of reducing I/O response time for all clients, MC2 tries to allocate cache space fairly among clients.

Several hint-based techniques have been proposed, including importance hints [13] and write hints [44]. The Type-Queue (TQ) algorithm, which exploits I/O type hints (read-/write) for cache management, has been proposed by Li et al. [44]. They distinguished different kinds of write requests (e.g. synchronous write, asynchronous replacement write) that provide clues about future data access by storage clients. Thus, these hints can be used by the storage server to maintain data exclusiveness. In their work on informed prefetching and caching, Patterson et al. [58] distinguished hints that disclose from hints that advise, and advocated the former. Most subsequent hint-based techniques, including CLIC [45], use hints that disclose. Informed prefetching and caching rely on hints that disclose sequential access to entire files or to portions of files. Unlike CLIC, all of these techniques are designed to exploit specific types of hints. As was discussed in Chapter 1, this makes them difficult to generalize and combine.

Content-aware caching [13], instead of piggybacking simple hints onto each I/O request, sends a summary of the storage client cache content periodically to the storage server. The storage server cache manager can then use the summary to avoid duplicating the contents of the upper-tier cache.

Although all aggressively collaborative techniques require changes to the first-tier, they vary considerably in the intrusiveness of the changes required. ULC [33] requires all tiers to understand the new protocol, and thus extensive code update in each tier is needed. Hint-based techniques are arguably the least costly. Hints are small and can be piggybacked onto regular I/O requests. More importantly, hint-based techniques do not require any changes to the policies used to manage the upper-tier cache.

A previous study [13] suggested that aggressively collaborative approaches provided little benefit beyond that of hierarchy-aware approaches and thus, the loss of transparency implied by collaborative approaches was not worthwhile. However, that study only considered one ad hoc hint-based technique. Li et al. [44] found that the hint-based TQ algorithm could provide substantial performance improvements in comparison to hint-oblivious approaches (LRU and MQ) as well as simple hint-aware extensions of those approaches.

There has also been work on the problem of sharing a cache among multiple competing client applications [8, 51, 65, 71]. Often, the goal of these techniques is to achieve specific

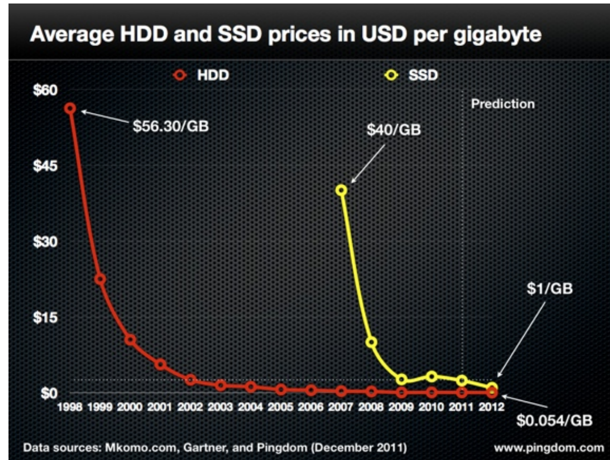


Figure 2.1: Trend of HDD and SSD Price [59]

quality-of-service objectives for the client applications, and the method used is to somehow partition the shared cache. This work is largely orthogonal to CLIC, in the sense that CLIC can be used, like any other replacement algorithm, to manage the cache contents in each partition. CLIC can also be used to directly control a shared cache, as in Section 3.4.5, but it does not include any mechanism for enforcing quality-of-service requirements or fairness requirements among the competing clients.

The problem of identifying frequently-occurring items in a data stream occurs in many situations [50, 53, 15]. Metwally et al. [53] classify solutions to the frequent-item problem as counter-based techniques or sketch-based techniques. The former maintain counters for certain individual items, while the latter collect information about aggregations of items. For CLIC, we have chosen to use the Space-Saving algorithm [53] as it is both effective and simple to implement. A recent study [15] found the Space-Saving algorithm to be one of the best overall performers among frequent-item algorithms.

In machine learning and statistics, feature selection has been widely studied [30, 60]. Guyon and Elisseeff [30] provide an excellent overview of feature selection for the particular problem, e.g., classification, clustering. In feature selection, the important features are determined as an useful preprocessing step before building the classifier (and this is usually done by selecting attributes that are highly correlated with the class attribute). Yang and Pederson [73] discusse feature selection for text classification. Das et al. [16] propose rank-based attribute selection which is related to feature selection to select attributes that best distinguish tuples of a query. In our work, we use feature selection to select important hint types and thus to reduce the number of hint sets CLIC has to track.

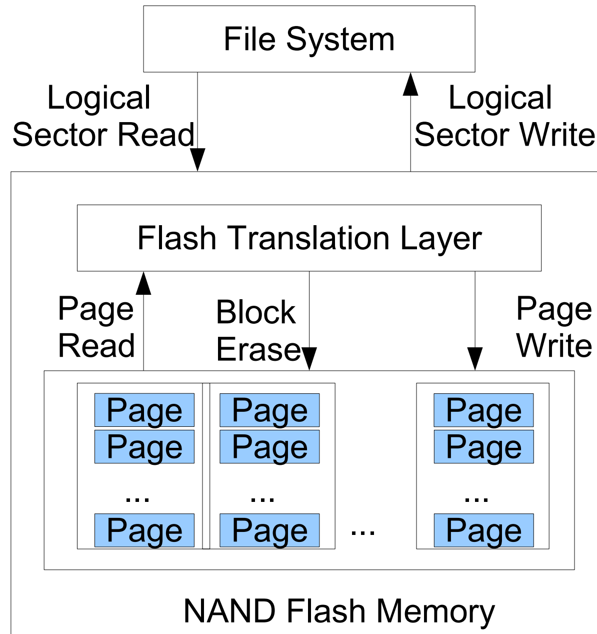


Figure 2.2: FTL and NAND Flash Memory
FTL emulates sector read/write functionalities of a hard disk to use conventional disk file systems in NAND flash memory

2.2 Solid State Disks

The flash memory disk has been widely used for portable computing devices, such as PDAs, MP3 plays, and mobile phones. As its price drops and capacity increases (Figure 2.1) [59], some recent studies have proposed that flash memory disk might be an attractive alternative for non-volatile data storage in desktop systems [36] and lower-level storage systems [38, 39, 40]. The Flash Translation Layer (FTL) [32], which is the driver to make linear flash memory appear to the OS like a disk drive, makes it easy to replace magnetic disks with flash disks in most applications (Figure 2.2).

Unlike DRAM, one key feature of flash memories is non-volatility, i.e. data stored on flash memory will not be lost even without power. This feature makes flash memories suitable for use as a consistent cache or for persistent storage. If flash memory is introduced to fill the gap between RAM and traditional rotating disks, a common question is whether it should work as a special part of main memory or a special part of persistent storage.

Placing hot data in fast storage (e.g. hard disks) and cold data in slow storage (e.g. tapes) is not a new idea. Hierarchical storage management (HSM) is a data storage technique which automatically moves data between high-cost and low-cost storage media. It

uses fast storage as a cache for slow storage [31]. The performance and price of SSDs suggests that "Tape is dead, disk is tape, flash is disk" [28] seems to have come true.

Some research has focused on how to partially replace hard disks with SSDs in database systems. Koltsidas, et al. [38] assumed that random reads from SSD are ten times faster than random reads from HDD, while random writes to SSD are ten times slower than random writes to HDD. They designed an algorithm to place read-intensive pages on the SSD and write-intensive pages on the HDD. Canim et al. [10] introduced an object placement advisor for DB2. Using run-time statistics about I/O behavior gathered by the buffer manager, the advisor helps the database administrator make decisions about SSD sizing, and about which database objects, such as tables or indices, should be placed on limited SSD storage. Ozmen et al. [57] presented a database layout optimizer which places database objects to balance workload and to avoid interference. It can generate layouts for heterogeneous storage configurations that include SSDs.

Flash memory has been used as cache for extending DRAM. Examples include ZFS [42], FlashCache [22], XtremSW Cache [21], and ioTurbine [25]. The flash memory cache sits between DRAM and the disks and is populated as entries are evicted from the DRAM cache. These flash-based caches are designed as write-through caches, which means dirty pages are written to both the flash-based cache and the HDD. Thus, all pages in the flash-based cache are consistent with these in the HDD. The main purpose of the flash-base cache is to offload the read workload from the underlying HDD. Recent studies [9, 11, 62] also propose to use SSD as a write through cache. For example, Canim et al. [11] investigated the use of SSD as a second-tier cache. The most-frequently read pages, identified by run-time I/O statistics gathering, are moved into the SSD. Reads are served from the SSD if the page is in the SSD, but writes need to go to the hard disks immediately.

Unlike DRAM, flash memory is non-volatile, i.e. data stored on flash memory will not be lost in case of a loss of power. As its persistence enables cache contents to survive crashes or power failures, recent studies have argued to use SSDs as write-back caches. Unlike a write-through cache, a write-back cache maintains dirty pages and writes are acknowledged immediately after the write to the cache. Koller et al. [37] has argued that write-back caching policy offers critical performance benefits as it reduces write latencies and write I/O traffic to the storage. Recent studies [41, 55] report a trend of increasing write/read ratios in production workloads as newer systems with larger DRAM caches filter more reads. Koller et al. [37] propose journaled write-back flash-based cache to offload the write workload from the storage devices. Do et al. [20] propose lazy cleaning, an eviction-based mechanism for managing an SSD as a second-tier write-back cache for database systems. Dirty pages are written to the SSD first, and are later copied to the HDD in batch mode. FlashStore [17] uses SSDs as a write-back cache between RAM and the HDD. It organizes data as key-value pairs, and writes the pairs in a log-structure on flash to improve the write performance. Kang et al. [35] proposed FaCE, an alternative write-back design. FaCE is based on the FIFO replacement algorithm. FaCE invalidates stale pages on the SSD and writes new versions to the end of the FIFO queue. Therefore,

FaCE always writes pages sequentially to the SSD, improving the performance by avoiding random writes. hStorage-DB [47] extracts semantic information from the query optimizer and query planner and passes it with I/O requests to the SSD manager. The semantic information includes hints about the request type, e.g, random or sequential. hStorage-DB associates a priority with each request type, and the SSD manager uses these priorities when making placement and replacement decisions. SSD can also be exploited to save energy for RAID systems consisting of hard disks. Snyder et al. [64] propose QMD which saves energy by using SSD as a write buffer.

As a write-back cache maintains dirty pages, it introduces data staleness and inconsistency at the storage devices. Thus, it needs to be recovered after a system crash. As the SSD is non-volatile, a naive way to restore the data in the SSD is to scan the whole SSD. However, scanning a whole SSD cache is time consuming if the SSD size is large. Do et al. [20] and Koller et al. [37] set checkpoints to flush dirty pages in the SSD to the storage devices. Debnath et al. [17] address this problem by checkpointing the hash map of the SSD and logging all writes to the SSD. As an extension to the work of Do et al. [20], DeWitt et al. [19] revisit the failure recovery problem of the SSD cache. They propose to flush the SSD buffer table during the DBMS checkpoint operation, and to log the updates made to the SSD buffer table in the regular database transactional log. Upon restart, the SSD buffer table can be reconstructed from the log. They also propose another method to restore the the SSD buffer table using the log. During a restart, the contents of the SSD buffer table are lazily verified on demand by checking the version in the SSD against that in the storage devices.

Most previous work focuses on the caching policy of the SSD and leaves DBMS buffer pool management unchanged. In addition to the placement policy for the SSD, Koltosidas, et al. [38] designed a cost-aware replacement algorithm for the buffer pool. The page I/O access cost is the I/O performance of the storage device that the page is on. For example, it considers the cost of writing a dirty page to SSD to be the highest and the cost of reading a page from SSD the lowest, and it always evicts the page with the lowest I/O access cost. One drawback of this algorithm is that it does not combine recency with the I/O access cost properly. Assume that a page in the SSD has been modified in the buffer pool, and then it is one of the pages with the highest cost. As long as there is a page with a lower cost, this dirty page will be kept in the buffer pool even if there are no further references to it. However, multiple reads may cost more than a single write. Another potential problem with this algorithm is that it delays the flushing of dirty pages.

Most replacement policies for the buffer cache, such as LRU and ARC, are cost-oblivious. Existing cost-aware algorithms for heterogeneous storage systems, e.g. balance algorithm [49] and GreedyDual, were proposed for file caching. Cao et al. [12] extended GreedyDual to handle cached objects of varying size, with application to web caching. Forney et al. [24] revisited caching policies for heterogeneous storage systems. They suggest partitioning the cache for different classes of storage according to the workload and performance of each class. Lv, et al. [48] designed another "cost-aware" replacement algo-

rithm for the buffer pool. However, it is designed for storage systems that only have SSD, not for heterogeneous storage systems. The algorithm is aware that SSD read costs and write costs are different and tries to reduce the number of writes to the SSD. Thus, it is "cost-aware" in a different sense than GD2L.

Our work, GD2L and CAC [46], builds upon previous studies [11, 12, 20, 49]. The GD2L algorithm for managing the buffer pool is a restricted version of GreedyDual [49], which we have adapted for use in database systems. Our CAC algorithm for managing the SSD is related to the previous cost-based algorithm of Canim et al. [11]. CAC is aware that the buffer pool is managed by a cost-aware algorithm and adjusts its cost analysis accordingly when making replacement decisions.

Chapter 3

CLIC: Client-Informed Caching for Storage Servers

The challenges of making effective use of caches below the first tier are well known [54, 70, 76]. Poor temporal locality in the request streams experienced by the second-tier caches reduces the effectiveness of recency-based replacement polices [76], and failure to maintain exclusivity among the contents of the caches in each tier leads to wasted cache space [70].

One promising approach is hint-based: let each individual sub-system manage its own cache, but with more information - hints. Specifically, hints are I/O related information carried from an upper-tier cache [70, 44, 72]. The system that manages the upper-tier cache generates hints and attaches them to individual I/O requests in order to provide some information about the request. The lower-tier cache then is able to exploit these hints to improve its performance. For example, hints can be used to pass data access and eviction predictions from an upper-tier to a lower-tier. A write hint [44] indicates whether the upper tier cache is writing a page to ensure recoverability of the page, or to facilitate replacement of the page. The lower-tier cache then may infer that replacement writes are better candidates for caching than the recovery writes, since they indicate pages that are eviction candidates in the upper-tier cache. Furthermore, hints can pass application semantics [13]. An importance hint indicates the priority of a particular page to the buffer cache manager in the upper-tier application. Given such hints, the lower-tier cache can infer that pages having higher priority in the upper-tier cache are likely to be retained there, and thus give these pages lower priority in the lower-tier cache.

Hinting is valuable because it is a way of making additional information available to the second (or lower) tier, which needs a good basis on which to make its caching decisions. However, previous work has taken an *ad hoc* approach to hinting. The general approach is to identify a specific type of hint that can be generated from the first-tier, e.g., a DBMS. A replacement policy that knows how to take advantage of this particular type of hint is then designed for the second-tier cache. For example, the TQ algorithm [44] is designed

specifically to exploit write hints. The desired response to each possible write hint is hard-coded into the TQ algorithm.

Ad hoc algorithms can significantly improve the performance of the second-tier cache when the necessary type of hint is available. However ad hoc algorithms also have some significant drawbacks. First, because the response to hints is hard-coded into an algorithm at the second-tier, any change to the hints requires changes to the cache management policy at the second-tier server. Second, even if change is possible at the server, it is difficult to generalize ad hoc algorithms to account for new situations. For example, suppose that applications can generate *both* write hints and importance hints. Clearly, a low-priority (to the first tier) replacement write is probably a good caching candidate for the second-tier, but what about a low-priority recovery write? In this case, the importance hint suggests that the page is a good candidate for caching in the second-tier, but the write hint suggests that it is a poor candidate. One response to this might be to hard code into the second-tier cache manager an appropriate behavior for all combinations of hints that might occur. However, each new type of hint will multiply the number of possible hint combinations, and it may be difficult for the policy designer to determine an appropriate response for each one. A related problem arises when multiple first-tier caches are served by a single cache in the second-tier. If the first-tier caches generate hints, how is the second-tier cache to compare them? Is a write hint from one first-tier cache more or less significant than an importance hint from another?

In this chapter, we propose CLient-Informed Caching (CLIC), a *generic technique* for exploiting application hints to manage a second-tier cache, such as a storage server cache. Unlike ad hoc techniques, CLIC does not hard-code responses to any particular type of hint. Instead, it is an adaptive approach that attempts to learn to exploit any type of hint that is supplied to it. Caches in the first-tier are free to supply any hints that they believe may be of value to the second-tier. CLIC analyzes the available hints and determines which can be exploited to improve second-tier cache performance. Conversely, it learns to ignore hints that do not help. Unlike ad hoc approaches, CLIC decouples the task of generating hints (done by the first-tier) from the task of interpreting and exploiting them. CLIC naturally accommodates multiple hint types, as well as scenarios in which multiple first-tier caches share a second-tier cache.

3.1 Generic Framework for Hints

We assume a system in which multiple storage server clients in the first-tier cache generate requests to a storage server, as shown in Figure 3.1. We are particularly interested in clients that cache data, since it is such clients that give rise to multi-tier caching.

There is a variety of realistic scenarios that fit the general architecture shown in Figure 3.1. For example, client application may be a database system accessing a shared

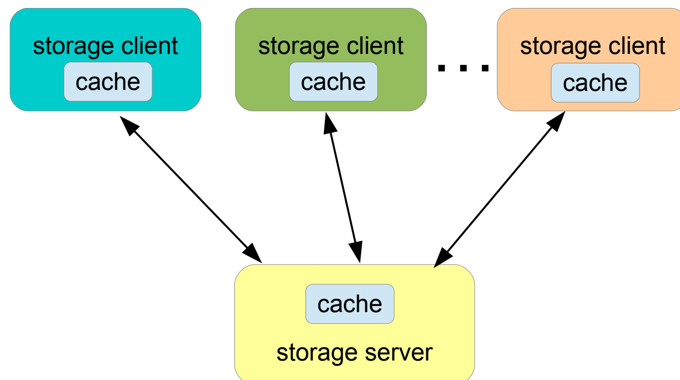


Figure 3.1: System Architecture

storage system through a storage area network. Alternatively, the storage server could represent a local file system serving a set of clients running on the same machine.

The storage server's workload is a sequence of block I/O requests from the various clients. When a client sends an I/O request (read or write) to the server, it may attach hints to the request. Specifically, each storage client may define one or more *hint types* and, for each such hint type, a *hint value domain*. When the client issues an I/O request, it attaches a *hint set* to the request. Each hint set consists of one hint value from the domain of each of the hint types defined by that client. For example, we used IBM DB2 Universal Database[1] as a storage client, and we instrumented DB2 so that it would generate five types of hints, as described in Figure 3.2. Thus, each I/O request issued by DB2 will have an attached hint set consisting of five hint values: a pool ID, an object ID, an object type ID, a request type, and a DB2 buffer priority. Together, a pool ID, object ID and object type ID uniquely identify a specific database object, such as a table or an index. The request type flag indicates whether a particular request was generated asynchronously (e.g., by a DB2 prefetch or page cleaning thread) and, for write requests, whether the request was generated for recoverability or buffer cache replacement purposes. (These are essentially *write hints* [44].) Finally, the DB2 buffer priority hint indicates the priority given to the requested page by DB2's buffer cache manager.

We also instrumented MySQL InnoDB the same way DB2 was instrumented, and Figure 3.3 shows the hint types in the MySQL InnoDB I/O request stream. From the two figures we see that the hint types of DB2 are different from that of MySQL. This is because the two database systems have different storage engines, and manage data differently.

CLIC does *not* require these specific hint types. We chose these particular types of hints because they could be generated easily from DB2 and MySQL InnoDB, and because

[1] DB2 Universal Database is a registered trademark of IBM.

DBMS	Hint Type	Value Domain Cardinality (TPC-C [67])	Value Domain Cardinality (TPC-H [68])	Description
DB2	pool ID	2	5	Identifies which DB2 buffer pool generated the I/O request.
DB2	object ID	21	23	Identifies a group of related database objects, such as a table and its associated indices.
DB2	object type ID	6	9	Identifies object type, such as table or index. Together, a pool ID, object ID and object type ID uniquely identify a database object.
DB2	request type	5	5	For read requests, distinguishes regular reads from prefetch reads. For writes, provides write hints ([44]), which distinguish between recovery writes, replacement writes, and synchronous writes. Synchronous writes are replacement writes that are not performed by an asynchronous page cleaning thread.
DB2	buffer priority	4	1	Identifies the priority of the page in its DB2 buffer cache.

Figure 3.2: Types of Hints in the DB2 I/O Request Traces

we believed that they might prove useful to the underlying storage system. Each client can generate its own types of hints. CLIC itself only assumes that the hint value domains are *categorical*. It neither assumes nor exploits any ordering on the values in a hint value domain. Each storage client may have its own hint types. In fact, even if two storage clients are instances of the same application (e.g., two instances of DB2) and use the same hint types, CLIC treats each client's hint types as distinct from the hint types of all other clients.

3.2 Hint Analysis

When a storage client issues a read request for a page that is not in the server's cache, the server obtains the page from persistent storage and returns it to the client. At this time, storage server also has an opportunity to place the page into its cache. Similarly, when a storage client issues a write request, it supplies a page to the server, which then has the opportunity to cache the page. For each such opportunity, the storage server must decide whether to take advantage of it by caching the page. If it chooses to cache, it must also choose which page to evict from its cache to make room for the newcomer. Our approach

DBMS	Hint Type	Value Domain Cardinality (TPC-H)	Description
MySQL	thread ID	5	ID of server thread that issued the request.
MySQL	request type	6	Read, read ahead, replacement write, or recovery write. For read ahead, provides read ahead linear, read ahead random, and read ahead merge. Read ahead linear is a read request for pages before or after a page that is a border in a linear read-ahead area. Read ahead random is a read request for all pages in a random read-ahead area when a number of pages in this area has been accessed recently. Read ahead merge is a read request for pages which the ibuf module wants to read in, in order to contract the insert buffer tree.
MySQL	file ID	9	MySQL is configured so that each table is stored in a separate file, together with any indexes defined on that table, so this hint distinguishes groups of database objects.
MySQL	fix count	9	indicates how many MySQL threads are have currently fixed (pinned) this page in the buffer pool

Figure 3.3: Types of Hints in the MySQL I/O Request Traces

is to base these caching decisions on the hint sets supplied by the client applications with each I/O request. CLIC associates each possible hint set H with a numeric priority, $\texttt{Pr}(H)$. When an I/O request (read or write) for page p with attached hint set H arrives at the server, the server uses $\texttt{Pr}(H)$ to decide whether to cache p. Cache management at the server will be described in more detail in Section 3.2.4, but the essential idea is simple: the server caches p if there is some page p' in the cache that was requested with a hint set H' for which $\texttt{Pr}(H') < \texttt{Pr}(H)$. In other words, the priority of a page in the server cache is equal to the priority of the most recent hint set with which that page was requested.

We expect that some hint sets may signal pages that are likely to be re-used quickly, and thus are good caching candidates. Other hint sets may signal the opposite. Intuitively, we want the priority of each hint set to reflect these signals. But how should priorities be chosen for each hint set? One possibility is to assign these priorities, in advance, based on knowledge of the client application that generates the hint sets. Most existing hint-based caching techniques use this approach. For example, the TQ algorithm [44], which exploits write hints, understands that replacement writes likely indicate evictions in the client application's cache, and so it gives them high priority, which are essentially the same as the request type hints shown in Figure 3.2. Based on knowledge of the storage client (the DB2 relational DBMS, in this case), the TQ algorithm gives higher priority to pages that are requested with the `replacement write` tag than those that are requested with the `recovery write` tag. TQ understands how the client works, and knows that pages

written as replacement writes are more likely to be needed again soon than those that are written with `recovery writes`. Similarly, *quick eviction* [13], gives low priority to any page in a `read` request, under the assumption that the storage client application is using a recency-based caching policy and is thus unlikely to need that page again soon.

This way of assigning priorities has been shown to be effective, when it can be applied. However, it has some significant disadvantages. Since the priority of each possible hint value is predefined and hard-coded into the storage server's cache management policy, incorporating new types of hints involves changing the server's cache management policy. New hint types can manifest themselves in two different ways. One is through new types of storage clients. It may be clear that DB2's replacement writes should have higher priority than DB2's recovery writes, but how does a DB2 replacement write compare to a write issued by another relational database system, or by a file system, that is sharing the same storage server? For that matter, how does a replacement write from one *instance* of DB2 compare to a replacement write from another instance of DB2 sharing the same storage server? A similar complication arises when a single application attaches more than one type of hint to each request. For example, our DB2 traces include both request type hints and DB2 buffer priority hints. Previous work has shown that recovery writes are poor server caching candidates and that pages with low DB2 buffer priority are good candidates. What should the server do with a recovery write that has low DB2 buffer priority?

3.2.1 Hint Benefit/Cost Analysis

CLIC takes a different approach to this problem. Instead of predefining hint priorities based on knowledge of the storage client applications, CLIC assigns a priority to each hint set by *monitoring and analyzing I/O requests that arrive with that hint set*. Next, we describe how CLIC performs its analysis. With CLIC, each storage client is responsible for *classifying* its I/O requests by attaching hint sets to them. Clients may generate any types of hints that are feasible to generate and that *might* prove valuable for cache management at the server. The storage server is responsible for assigning a priority to each hint set (class of requests) through its monitoring and analysis process. CLIC is adaptive, so adding new hint types and new storage client applications is easy. As new hints arrive, CLIC begins monitoring them and assigns priorities to them. To simplify the presentation, we will ignore, for now, the cost (in time and space) of performing the analysis.

We will assume that each request that arrives at the server is tagged (by the server) with a sequence number. Suppose that the server gets a request (p, H), meaning a request (read or a write) for a page p with an attached hint set H, and suppose that this request is assigned sequence number s_1. CLIC is interested in whether and when page p will be requested again after s_1. There are three possibilities to consider:

write re-reference: The first possibility is that the *next* request for p in the request stream is a write request occurring with sequence number s_2 ($s_2 > s_1$). In this case,

there would have been no benefit whatsoever to caching p at time s_1. A cached copy of p would not help the server handle the subsequent write request any more efficiently. A cached copy of p may be of benefit for requests for p that occur after s_2, but in that case the server would be better off caching p at s_2 rather than at s_1. Thus, the server's caching opportunity at s_1 is best ignored.

read re-reference: The second possibility is that the *next* request for p in the request stream is read request at time s_2. If the server caches p at time s_1 and keeps p in the cache until s_2, it will benefit by being able to serve the read request at s_2 from its cache. For the server to obtain this benefit, it must allow p to occupy one page "slot" in its cache during the interval $s_2 - s_1$.

no re-reference: The third possibility is that p is never requested again after s_1. In this case, there is clearly no benefit to caching p at s_1.

Of course, the server cannot determine which of these three possibilities will occur for any particular request, as that would require advance knowledge of the future request stream. Instead, we propose that the server base its caching decision for the request (p, H) on an analysis of previous requests with hint set H. Specifically, CLIC tracks three statistics for each hint set H:

$N(H)$: the total number of requests with hint set H.

$N_r(H)$: the total number requests with hint set H that result in a read re-reference (rather than a write re-reference or no re-reference).

$D(H)$: for those requests (p, H) that result in read re-references, the average number of requests that occur between the request and the read re-reference. These requests will be requests for pages other than p.

Using these three statistics, CLIC performs a simple benefit/cost analysis for each hint set H, and assigns higher priorities to hint sets with higher benefit/cost ratios. Suppose that the server receives a request (p, H) and that it elects to cache p. If a read re-reference subsequently occurs while p is cached, the server will have obtained a benefit from caching p. We arbitrarily assign a value of 1 to this benefit (the value we use does not affect the relative priorities of pages). Among all previous requests with hint set H, a fraction

$$f_{hit}(H) = N_r(H)/N(H) \tag{3.1}$$

eventually resulted in read re-references, and would have provided a benefit if cached. We call $f_{hit}(H)$ the *read hit rate* of hint set H. Since the value of a read re-reference is 1, $f_{hit}(H)$ can be interpreted as the expected benefit of caching and holding pages that are requested with hint set H. Conversely, $D(H)$ can be interpreted as the expected *cost* of

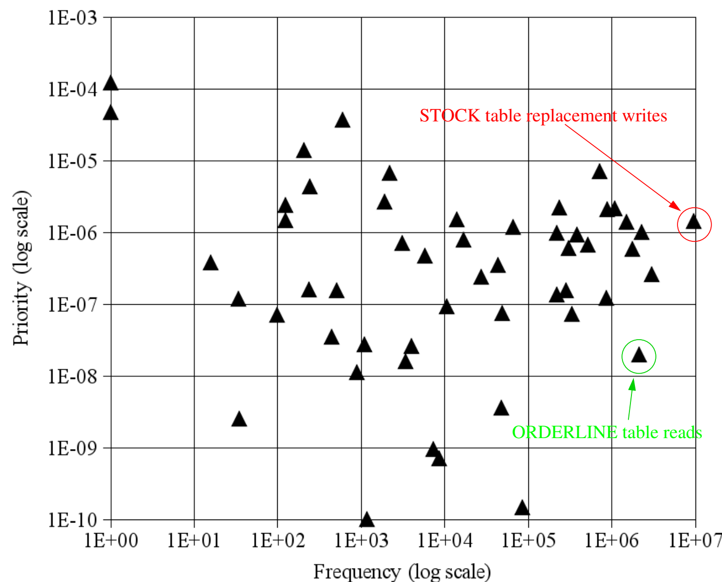


Figure 3.4: Hint Set Priorities for the DB2_C60 Trace
Each point represents a distinct hint set. All hint sets are shown.

caching such pages, as it measures how long such pages must occupy space in the cache before the benefit is obtained. We define the *caching priority* of hint set H as:

$$\texttt{Pr}(H) = \frac{f_{hit}(H)}{D(H)} \tag{3.2}$$

which is the ratio of the expected benefit to the expected cost.

Figure 3.4 illustrates the results of this analysis for a trace of I/O requests made by DB2 during a run of the TPC-C [67] benchmark. Our workload traces will be described in more detail in Section 3.4. Each point in Figure 3.4 represents a distinct hint set that is present in the trace, and describes the hint set's caching priority and frequency of occurrence. All hint sets with non-zero caching priority are shown. Clearly, some hint sets have much higher priorities, and thus much higher benefit/cost ratios, than others. For illustrative purposes, we have indicated partial interpretations of two of the hint sets in the figure. For example, the most frequently occurring hint set represents replacement writes to the STOCK table in the TPC-C database instance that was being managed by the DB2 client. We emphasize that CLIC does not need to understand that this hint represents the STOCK table, nor does it need to understand the difference between a replacement write and a recovery write. Its interpretation of hints is based entirely on the hint statistics that it tracks, and it can automatically determine that a request with the STOCK table hint set is a better caching opportunity than a request with the ORDERLINE table hint set.

3.2.2 Tracking Hint Set Statistics

To track hint set statistics, CLIC maintains a *hint table* with one entry for each distinct hint set H that has been observed by the storage server. The hint table entry for H records the current values of the statistics $N(H)$, $N_r(H)$ and $D(H)$. When the server receives a request (p, H), it increments $N(H)$. Tracking $N_r(H)$ and $D(H)$ is somewhat more involved, as CLIC must determine whether a read request for page p is a read re-reference. To determine this, CLIC records two pieces of information for every page p that is cached: $\texttt{seq}(p)$, which is the sequence number of the most recent request for p, and $\texttt{H}(p)$, which is the hint set that was attached to the most recent request for p. In addition, CLIC records $\texttt{seq}(p)$ and $\texttt{H}(p)$ for a fixed number (N_{outq}) of additional, uncached pages. This additional information is recorded in a data structure called the *outqueue*. N_{outq} is a CLIC parameter that can be used to bound the amount of space required for tracking read re-references. When the server receives a read request for page p with sequence number s, it checks both the cache and the outqueue for information about the most recent previous request, if any, for p. If it finds $\texttt{seq}(p)$ and $\texttt{H}(p)$ from a previous request, then it knows that the current request is a read re-reference of p. It increments $N_r(\texttt{H}(p))$ and it updates $D(\texttt{H}(p))$ using the re-reference distance $s - \texttt{seq}(p)$.

When a page p is evicted from the cache, an entry for p is inserted into the outqueue. The entry records the hint H with which p was last requested as well as $\texttt{seq}(p)$. An entry is also placed in the outqueue for any requested page that CLIC elects not to cache. (CLIC's caching policy is described in Section 3.2.4.) When an outqueue page is inserted into the cache, the page's entry is removed from the outqueue. If the outqueue is full when a new entry is to be inserted, the least-recently inserted entry is evicted from the outqueue to make room for the new entry.

Since CLIC only records $\texttt{seq}(p)$ and $\texttt{H}(p)$ for a limited number of pages, it may fail to recognize that a new read request (p, H) is actually a read re-reference for p. Some error is inevitable unless CLIC were to record information about all requested pages. However, CLICs approach to tracking page re-references has several advantages. First, since CLIC tracks the most recent reference to all pages that are in the cache, we expect to have accurate re-reference distance estimates for hint sets that are believed to have the highest priorities, since pages requested with those hint sets will be cached. If the priority of such hint sets drops, CLIC should be able to detect this. Second, by evicting the oldest entries from the outqueue when eviction is necessary, CLIC will tend to miss read re-references that have long re-reference distances. Conversely, read re-references that happen quickly are likely to be detected. These are exactly the type of re-references that lead to high caching priority. Thus, CLIC's statistics tracking is biased in favor of read re-references that are likely to lead to high caching priority.

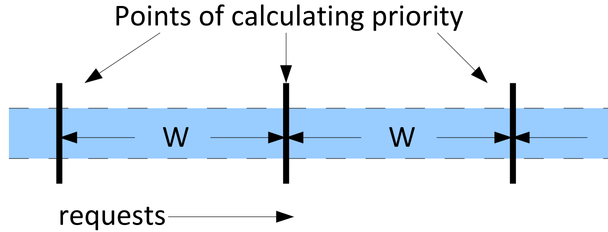


Figure 3.5: Priority Calculation Timeline

3.2.3 Time-Varying Workloads

To accommodate time-varying workloads, CLIC divides the request stream into non-overlapping windows, with each window consisting of W requests, as shown in Figure 3.5. At the end of each window, CLIC adjusts the priority for each hint set using the statistics collected during that window. The adjusted priority will be used to guide the caching policy during the next window. It then clears the statistics ($N(H)$, $N_r(H)$, $D(H)$) for all hint sets in the hint table so that it can collect new statistics during the next window.

Let $\texttt{Pr}(H)_i$ represent the priority of H that is calculated after the ith window, and that is used by CLIC's caching policy during window $i+1$. Priority $\texttt{Pr}(H)_i$ is calculated as follows

$$\texttt{Pr}(H)_i = r\widehat{\texttt{Pr}}(H)_i + (1-r)\texttt{Pr}(H)_{i-1} \tag{3.3}$$

where $\widehat{\texttt{Pr}}(H)_i$ represents the priorities that were calculated using the statistics collected during the ith window (and Equation 3.2), and r ($0 < r \leq 1$) is a CLIC parameter. The effect of Equation 3.3 is that the impact of statistics gathered during the ith window decays exponentially with each new window, at a rate that is controlled by r. Setting $r = 1$ causes CLIC to base its priorities entirely on the statistics collected during the most recently completed window. Lower values of r cause CLIC to give more weight to older statistics. For all of the experiments reported for CLIC and DP-CLIC, we have set $W = 10^6$ and $r = 1$.

3.2.4 Cache Management

In Section 3.2.1 , we described how CLIC assigns a caching priority to each hint set H. In this section, we describe how the server uses these priorities to manage the contents of its cache. Structures used by CLIC are summarized in Figure 3.6 and CLIC cache policy is described in Figure 3.7.

Figure 3.7 describes CLIC's priority-based replacement policy. This policy evicts a lowest priority page from the cache if the newly requested page has higher priority. The priority of a page is determined by the priority $\texttt{Pr}(H)$ of the hint set H with which that

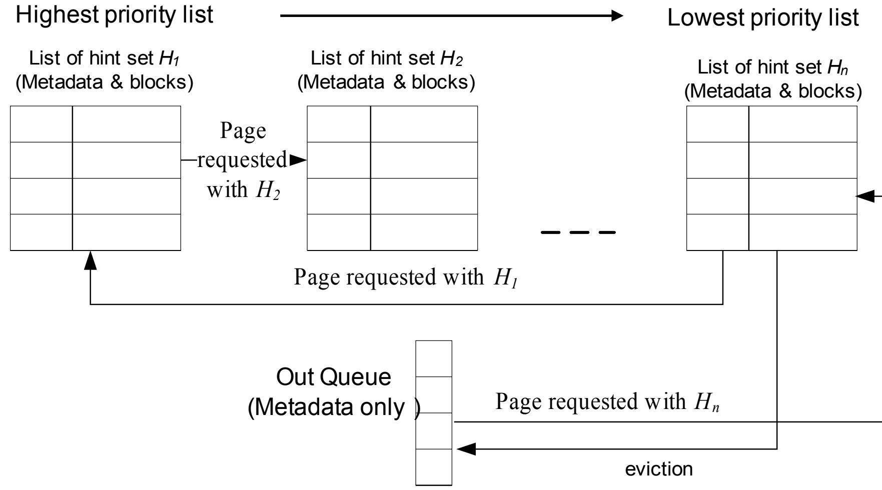


Figure 3.6: Structures Used by CLIC
Arrows show possible movements among queues in response to I/O requests

page was last requested. Note that if a page that is cached after being requested with hint set H is subsequently requested with hint set H', its priority changes from $\texttt{Pr}(H)$ to $\texttt{Pr}(H')$. The most recent request for each cached page always determines its caching priority.

The policy described in Figure 3.7 can be implemented to run in constant expected time. To do this, CLIC maintains a heap-based priority queue of the hint sets. For each hint set H in the heap, all pages with $\texttt{H}(p) = H$ are recorded in a doubly-linked list that is sorted by $\texttt{seq}(p)$. This allows the victim page to be identified (Figure 3.7, lines 7-11) in constant time. CLIC also maintains a hash table of all cached pages so that it can tell which pages are cached (line 1) and find a cached page in its hint set list in constant expected time. Finally, CLIC implements the hint table as a hash table so that it can look up $\texttt{Pr}(H)$ (line 12) in constant expected time.

As described in Section 3.2.3, CLIC adjusts hint set priorities after every window of W requests. When this occurs, CLIC rebuilds its hint set priority queue based on the newly adjusted priorities. Hint set priorities do not change except at window boundaries.

3.3 Handling Large Numbers of Hint Sets

As described in Section 3.2.2, CLIC's hint table records statistical information about every hint set that the server has observed. Although the amount of statistical information

```
1≻    if p is not cached then
2≻      if the cache is not full then
3≻        cache p
4≻        set seq(p) = s
5≻        set H(p) = H
6≻      else
7≻        let m be the minimum priority
8≻          of all pages in the cache
9≻        let v be the page with the
10≻         minimum sequence number seq(v)
11≻         among  all pages with priority m
12≻       if Pr(H)>m then
13≻         evict v from the cache
14≻         add entry for v (with seq(v)
15≻           and H(v)) to the outqueue
16≻         cache p
17≻         set seq(p) = s
18≻         set H(p) = H
19≻       else /* do not cache p */
20≻         add entry for p to the outqueue
21≻         set seq(p) = s
22≻         set H(p) = H
23≻   else /* p is already cached */
24≻     seq(p) = s
25≻     H(p) = H
```

Figure 3.7: Hint-Based Server Cache Replacement Policy
This pseudo-code shows how the server handles a request for page p with hint set H and request sequence number s.

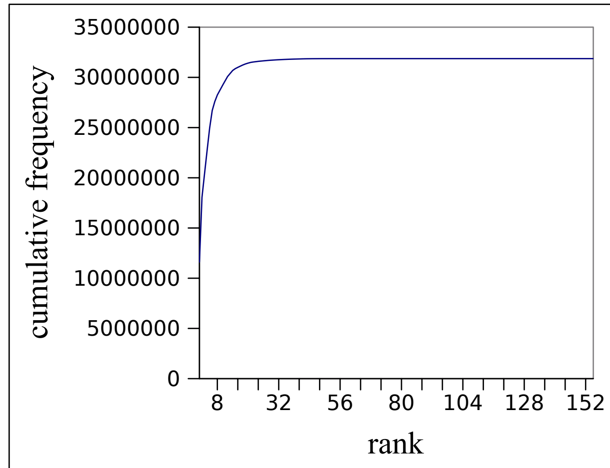


Figure 3.8: Cumulative Hint Set Frequency in the DB2_C300_400 Trace

tracked per hint set is small, the number of distinct hit sets from each client might be as large as the product of the cardinalities of that client's hint value domains. In our traces, the number of distinct hit sets is small. For other applications, however, the number of hint sets could potentially be much larger. In this section, we propose one technique for restricting the number of hint sets that CLIC must consider, so that CLIC can continue to operate efficiently as the number of hint sets grows. The technique is based on hint set frequency. We describe another technique, which is based on hint set generalization, in Chapter 5.

3.3.1 Frequently-Occurring Hint Sets

All of the hint types in our workload traces exhibit frequency skew. That is, some values in the hint domain occur much more frequently than others. As a result, some hint sets occur much more frequently than others. One way to cope with large numbers of hint sets is to reduce the number of hints that CLIC must consider. We propose to exploit this skew by tracking statistics for the hint sets that occur most frequently in the request stream and ignoring those that do not. Ignoring infrequent hint sets may lead to errors. In particular, we may miss a hint set that would have had high caching priority. However, since any such missed hint set would occur infrequently, the impact of the error on the server's caching performance is likely to be small.

For example, Figure 3.8 shows the cumulative frequency of hint sets in the DB2_300_400 trace, one of the traces we used in our evaluation of CLIC (see Figure 3.9). The figure shows that a few hint sets account for most of the I/O requests in the trace. While it is possible that this is merely an artifact of our traces, we expect that many application hints will exhibit skew.

The problem with this approach is that we must determine, on the fly, which hint sets

occur frequently, without actually maintaining a counter for every hint set. The simple way to do this is to count the number of requests with each hint set and then choose those hints sets with the most requests. However, this requires space proportional to the number of distinct hint sets, which is exactly what we are trying to avoid. What we need is a means of determining the frequently occurring hint sets without maintaining counts for every hint. Fortunately, this *frequent item problem* arises in a variety of settings, and numerous methods have been proposed to solve it. We have chosen one of these methods: the so-called *Space-Saving* algorithm [53], which has recently been shown to outperform other frequent item algorithms [15]. Given a parameter k, this algorithm tracks the frequency of k different hint sets, among which it attempts to include as many of the actual k most frequent hint sets as possible. It is an on-line algorithm which scans the sequence of hint sets attached to the requests arriving at the server. Although k different hint sets are tracked at once, the specific hint sets that are being tracked may vary over time, depending on the request sequence.

After each request has been processed, the algorithm can report the k hint sets that it is currently tracking, as well as an estimate of the frequency (total number of occurrences) of each hint set and an error indicator which bounds the error in the frequency estimate. By analyzing the frequency estimates and error indicators, it is possible to determine which of the k currently-tracked hint sets are guaranteed to be among the actual top-k most frequent hint sets and which are not. However, for our purposes this is not necessary.

We adapted the Space-Saving algorithm slightly so that it tracks the additional information we require for our analysis. Specifically:

$N(H)$: For each hint set H that is tracked by the Space-Saving algorithm, we use the frequency estimate produced by the algorithm, minus the estimation error bound reported by the algorithm, as $N(H)$.

$N_r(H)$: We modified the Space-Saving algorithm to include an additional counter for each hint set H that is currently being tracked. This counter is initialized to zero when the algorithm starts tracking H, and it is incremented for each read re-reference involving H that occurs while H is being tracked. We use the value of this counter as $N_r(H)$.

$D(H)$: We track the expected re-reference distance for all read re-references involving H that occur while H is being tracked, i.e., those read re-references that are included in $N_r(H)$.

For all hint sets H that are not currently tracked by the algorithm, we take $N_r(H)$ to be zero, and hence $\texttt{Pr}(H)$ to be zero as well.

In general, $N(H)$ will be be an underestimate of the true frequency of hint set H. Since $N_r(H)$ is only incremented while H is being tracked, it too will in general underestimate the true frequency of read re-references involving H. As a result of these underestimations,

$f_{hit}(H)$, which is calculated as the ratio of the $N_r(H)$ to $N(H)$, may be inaccurate. However, because we take the ratio of $N(H)$ to $N_r(H)$, the two underestimations may at least partially cancel one another, leading to a more accurate $f_{hit}(H)$. In addition, the higher the true frequency of H, the more time H will spend being tracked and the more accurate we expect our estimates to be.

To account for time-varying workloads, we restart the Space-Saving algorithm from scratch for every window of W requests. Specifically, at the end of each window we use the Space-Saving algorithm to estimate $N(H)$, $N_r(H)$, and $D(H)$ for each hint set H that is tracked by the algorithm, as described above. These statistics are used to calculate $\widehat{\texttt{Pr}}(H)$, which is then used in Equation 3.3 to calculate the hint set's caching priority ($\texttt{Pr}(H)$) to be used during the next request window. Once the $\widehat{\texttt{Pr}}(H)$ have been calculated, the Space-Saving algorithm's state is cleared in preparation for the next window.

The Space-Saving algorithm requires two counters for each tracked hint-set, and we added several additional counters for the sake of our analysis. Overall, the space required is proportional to k. Thus, this parameter can be used to limit the amount of space required to track hint set statistics. With each new request, the data structure used by the Space-Saving algorithm can be updated in constant time [53], and the statistics for the tracked hint sets can be reported, if necessary, in time proportional to k.

3.4 Experimental Evaluation

Objectives: We used trace-driven simulation to evaluate our proposed mechanisms. The goal of our experimental evaluation is to answer the following questions:

1. Can CLIC identify good caching opportunities for storage server caches, and thereby improve the cache hit ratio in compared to other caching policies? (Section 3.4.1)
2. How effective are CLIC's mechanisms for reducing the number of hint sets that it must track (Sections 3.4.3 and 3.4.4).
3. Can CLIC improve performance for multiple storage clients by prioritizing the caching opportunities of the different clients based on their observed reference behavior? (Section 3.4.5)

Simulator: To answer these questions, we implemented a simulation of the storage server cache. In addition to CLIC, the simulator implements the following caching policies for purpose of comparison:

OPT: This is an implementation of the well-known optimal off-line MIN algorithm [6]. It replaces the cached page that will not be read for the longest time. This algorithm requires knowledge of the future so it cannot be used for cache replacement in

practical systems, but its hit ratio is optimal so it serves as an upper bound on the performance of any caching algorithm.

LRU: This algorithm replaces the least-recently used page in the cache. Since temporal locality is often poor in second-tier caches, we expect CLIC to perform significantly better than LRU.

ARC: ARC [52] is a hint-oblivious caching policy that considers both recency and frequency of use in making replacement decisions.

TQ: TQ is a hint-aware algorithm that was proposed for use in second-tier caches [44]. Unlike the algorithms proposed here, it works only with one specific type of hint that can be associated with write requests from database systems. We expect our proposed algorithms, which can automatically exploit any type of hint, to do at least as well as TQ when the write hints needed by TQ are present in the request stream.

The TQ algorithm has previously been compared to a number of other second-tier caching policies that are not considered here. These include MQ [76], a hint-oblivious policy, and write-hint-aware variations of both MQ and LRU [44]. TQ was shown to be generally superior to those alternatives when the necessary write hints are present [44], so we use it as our representative of the state of the art in hint-aware second-tier caching policies.

The simulator accepts a stream of I/O requests with associated hint sets, as would be generated by one or more storage clients. It simulates the caching behavior of one of the five supported cache replacement policies (CLIC, OPT, LRU, ARC and TQ) and computes the *read hit ratio* for the storage server cache. The read hit ratio is the number of read hits divided by the number of read requests.

Workloads: We use DB2 Universal Database (version 8.2) and the MySQL[2] database system (Community Edition, version 5.0.33) as our storage system clients. DB2 is a widely-used commercial relational database system to which we had access to source code, and MySQL is a widely-used open source relational database system. We instrumented DB2 and MySQL so that they would generate I/O hints and dump them into an I/O trace file. The types of hints generated by these two systems are described in Figure 3.2.

To generate our traces, we ran TPC-C and TPC-H workloads on DB2 and a TPC-H workload on MySQL. TPC-C and TPC-H are well-known on-line transaction processing (TPC-C) and decision support (TPC-H) benchmarks. We ran TPC-C at scale factor 25. At this scale factor, the TPC-C database initially occupied approximately 600,000 4KB blocks, or about 2.3 GB, in the storage system. The TPC-C workload inserts new items into the database, so the database grows during the TPC-C run. For the TPC-H experiments, the database size was approximately 3.2 GB for the DB2 runs, and 5 GB for the MySQL runs. The DB2 TPC-H workload consisted of the 22 TPC-H queries and the two refresh

[2]MySQL is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Trace Name	DBMS	WkLoad	DB Size (pages)	DBMS Buffer Size (pages)	DBMS softmax	Requests	Distinct Hint Sets	Distinct Pages
DB2_C60_40	DB2	TPC-C	600K	60K	40	38101851	169	919215
DB2_C60_400	DB2	TPC-C	600K	60K	400	37699091	164	930688
DB2_C300_40	DB2	TPC-C	600K	300K	40	32102429	128	1130925
DB2_C300_400	DB2	TPC-C	600K	300K	400	31869377	154	1320882
DB2_C540_40	DB2	TPC-C	600K	540K	40	49279589	105	1684878
DB2_C540_400	DB2	TPC-C	600K	540K	400	21863719	140	1807431
DB2_H80_50	DB2	TPC-H	800K	80K	50	650713762	136	732803
DB2_H80_400	DB2	TPC-H	800K	80K	400	635375701	134	732905
DB2_H400_50	DB2	TPC-H	800K	400K	50	62842665	135	732931
DB2_H400_400	DB2	TPC-H	800K	400K	400	65675204	129	732723
DB2_H720_50	DB2	TPC-H	800K	720K	50	2679456	128	732764
DB2_H720_400	DB2	TPC-H	800K	720K	400	2973792	128	732690
MY_H65	MySQL	TPC-H	328K	65K	n/a	36266735	21	167502
MY_H98	MySQL	TPC-H	328K	98K	n/a	16561346	21	167501

Figure 3.9: I/O Request Traces.
The page sizes for the DB2 and MySQL databases were 4KB and 16KB, respectively. For the TPC-C workloads, the table shows the initial database size. The TPC-C database grows as the workload runs.

updates. The workload for MySQL was similar except that it did not include the refresh updated and we skipped one of the 22 queries (Q18) because of excessive run-time on our MySQL configuration.

On each run, we controlled the size of the first-tier cache, which in our case is the database system's internal buffer cache. We collected traces using a variety of different buffer cache sizes for each DBMS. We expect the buffer cache size to be a significant parameter because it affects the temporal locality in the I/O request stream that is seen by the underlying storage server. The larger DBMS buffer cache, the less temporal locality we expect to be available at the storage server. Since TPC-C is write-intensive, we also varied `softmax` when running TPC-C workloads on DB2. This parameter controls the urgency with which DB2 forces dirty pages from its buffer cache to disk for recoverability reasons. Smaller values of `softmax` result in more (and more frequent) write requests in the request stream. Figure 3.9 summarizes the I/O request traces that were used for the experiments reported here.

3.4.1 Comparison to Other Caching Policies

In our first experiment, we compare the cache read hit ratio of CLIC to that of other replacement policies that we consider (LRU, ARC, TQ, and OPT). We varied the size of the storage server buffer cache, and we present the read hit ratio as a function of the server's buffer cache size for each workload. For these experiments, we set $r = 1.0$ and the size of CLIC's outqueue (N_{outq}) to 5 entries per page in the storage server's cache. If the cache holds C pages, this means that CLIC tracks the most recent reference for $6C$ pages, since it tracks this information for all cached pages, plus those in the outqueue. For each tracked page, CLIC records a sequence number and a hint set. If each of these is stored as a 4-byte integer, this represents a space overhead of roughly 1%. To account for this, we reduced the server cache size by 1% for CLIC only, so that the total space used by CLIC would be the same as that used by other policies. ARC also employs a structure similar to CLIC's outqueue for tracking pages that are not in the cache. However, we did not reduce ARC's cache size. As a result, ARC has a small space advantage in these experiments.

Figure 3.10 shows the results of this experiment for the DB2 TPC-C traces. All of the algorithms have similar performance for the DB2_C60 traces. These two traces come from the DB2 configuration with the smallest buffer cache, and there is a significant amount of temporal locality in the traces that was not "absorbed" by DB2 buffer pool. This temporal locality can be exploited by the storage server cache. As a result, even LRU performs reasonably well. Both of the hint-based algorithms (TQ and CLIC) also do well.

The performance of LRU is significantly worse on the other TPC-C traces, as there is very little temporal locality. ARC performs better than LRU, as expected, though substantially worse than both of the hint-aware policies. CLIC, which learns how to exploit the available hints, does about as well as TQ, which implements a hard-coded response

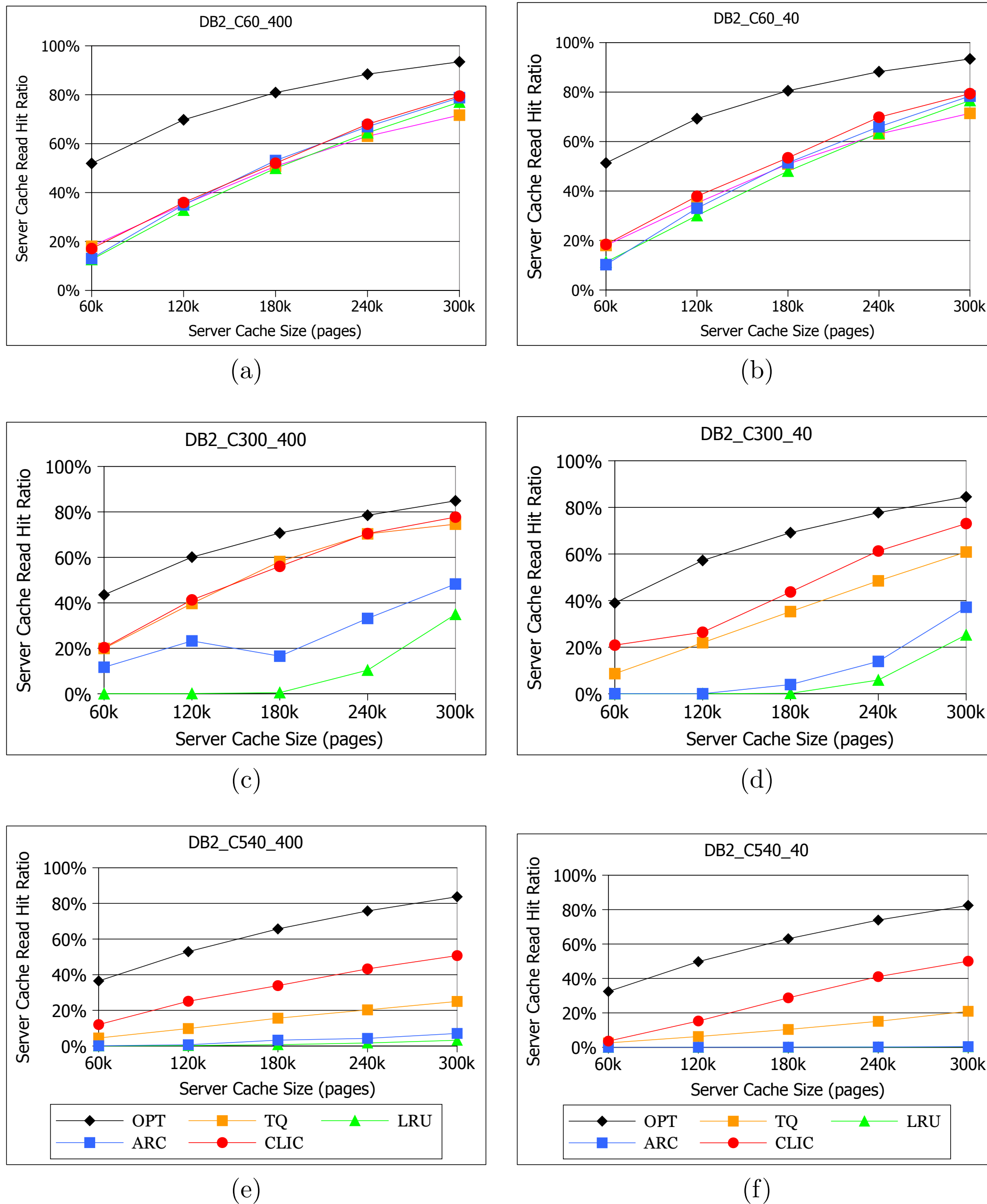


Figure 3.10: Read Hit Ratio of Caching Policies for the DB2 TPC-C Workloads

to one particular hint type on the DB2_C300_400 trace, and both policies' performance approaches that of OPT.

CLIC outperforms TQ on the DB2_C300_40 trace and CLIC's performance is closer to that of OPT. As the value of `softmax` is small, the DB2_C300_40 trace has more RECOV write requests and fewer REPLACE write requests. Under TQ, pages with REPLACE write hint are good candidates for caching at the storage server, but TQ ignores the RECOV

write hint. Thus, pages with read hints and pages with RECOV write hint are treated the same. CLIC exploits the available hints and determines that the RECOV write hint, in combination with other hint types, is also a good hint for caching. CLIC also outperforms TQ on the DB2_C540 trace, though it is also further from OPT. The DB2_C540 traces come from the DB2 configuration with the largest buffer cache, so they have the least temporal locality of all traces and therefore present the most difficult cache replacement problem. By analyzing these two traces, we note that DB2_C540_400 has much fewer REPLACE writes and DB2_C540_400 has no REPLACE writes. Thus, CLIC outperforms TQ because of its ability to learn that there is some value in the RECOV write hints in these traces.

Figures 3.11 and 3.12 show the results for the TPC-H traces from DB2 and MySQL, respectively. Again, CLIC generally performs at least as well as the other replacement policies that we considered. In some cases, e.g., for the DB2_H400_50 and DB2_H400_400 traces, CLIC's read hit ratio is more than twice the hit ratio of the best hint-oblivious alternative.

In one case, for the DB2_H80_50 and DB2_H80_400 traces with a server cache size of 300K pages, both LRU and ARC outperformed both TQ and CLIC. In this scenario, there is a relatively large amount of residual locality in the workload because the DB2 buffer cache is small. When the storage server cache is large enough (300K) to capture it, LRU and ARC have good performance.

3.4.2 Limiting the Outqueue Size

In this experiment, we study the effect of limiting the size of the outqueue that CLIC uses to track the sequence number of each page's most recent reference. If no limit were imposed, the outqueue could potentially have one entry for every page in the underlying storage system. In the previous section, we showed the performance of CLIC with the outqueue size equals to five times cache size, and in this section we show that the limited size of the outqueue does not have a large penalty on the performance.

We tested CLIC with the outqueue size equals to the server cache size, five times server size cache, and unlimited size. With unlimited outqueue, CLIC was given unlimited space for tracking page references and for recording hint statistics, and this space was not subtracted from the server cache size.

Figure 3.13 shows the results of this experiment for all six traces and a server cache size of 60K, 180K and 300K pages, respectively. In most cases, an outqueue limit of 5 entries per server cache page (the middle bar in each group in Figure 3.13) results in a server cache read hit ratio very close to what was obtained with CLIC with an unlimited outqueue. In most cases, an outqueue with only one entry per server cache page also does well (with some exceptions for the DB2_C540_40 trace). Similar experiments with server cache sizes of 60K pages and 300K pages resulted in identical conclusions. Thus, we can see that limiting the outqueue size saves space without degrading the performance of CLIC. For all of our remaining experiments, we use an outqueue limit of 5 entries per server cache page.

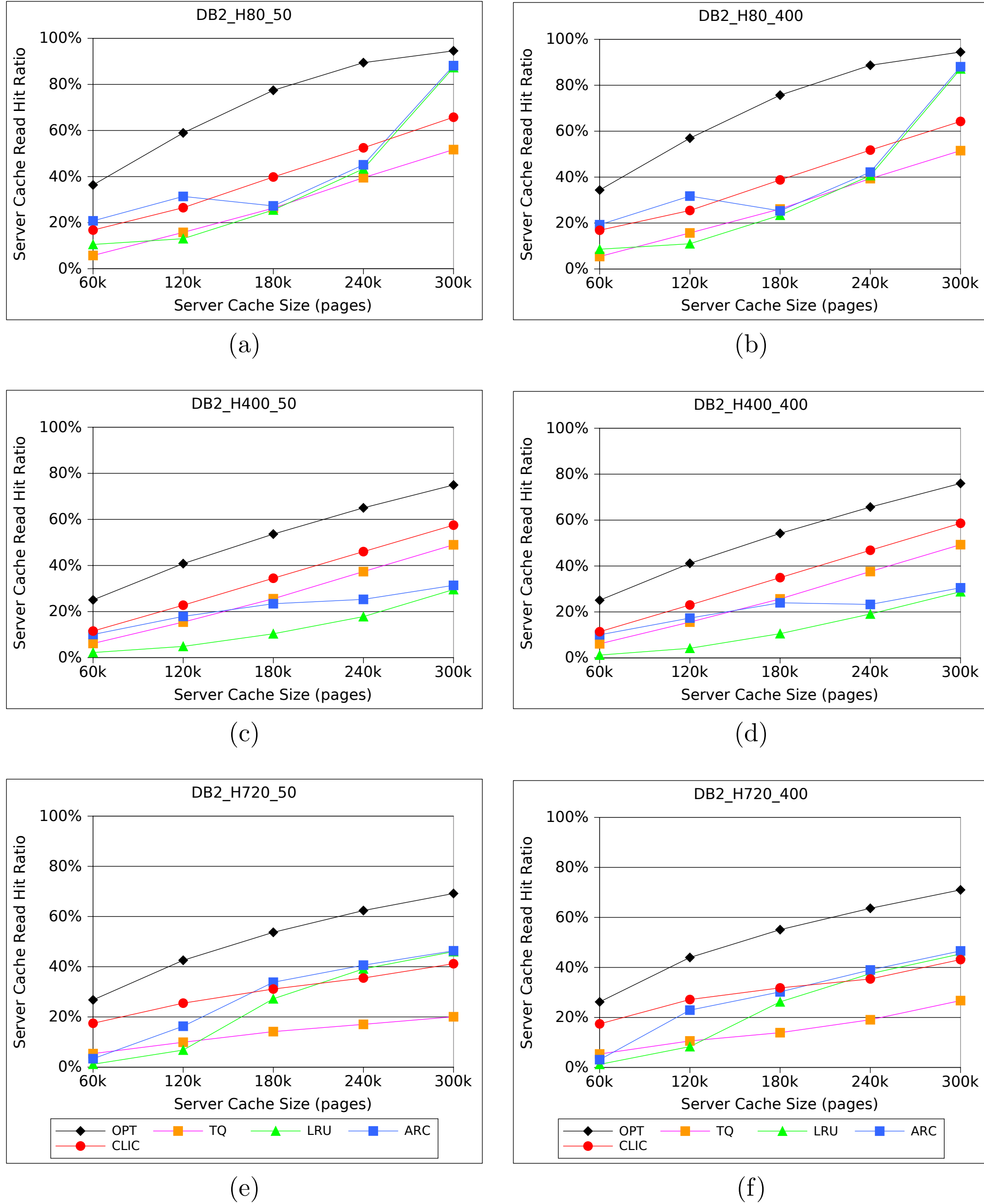


Figure 3.11: Read Hit Ratio of Caching Policies for the DB2 TPC-H Workloads

3.4.3 Tracking Only Frequent Hint Sets

In this experiment, we study the effect of tracking only the most frequently occurring hint sets using the top-k algorithm described in Section 3.3.1. In our experiment we vary k, the number of hint sets tracked by CLIC, and measure the server cache hit ratio.

Figure 3.14 shows some of the results of this experiment. The left side graphs (a) (c)

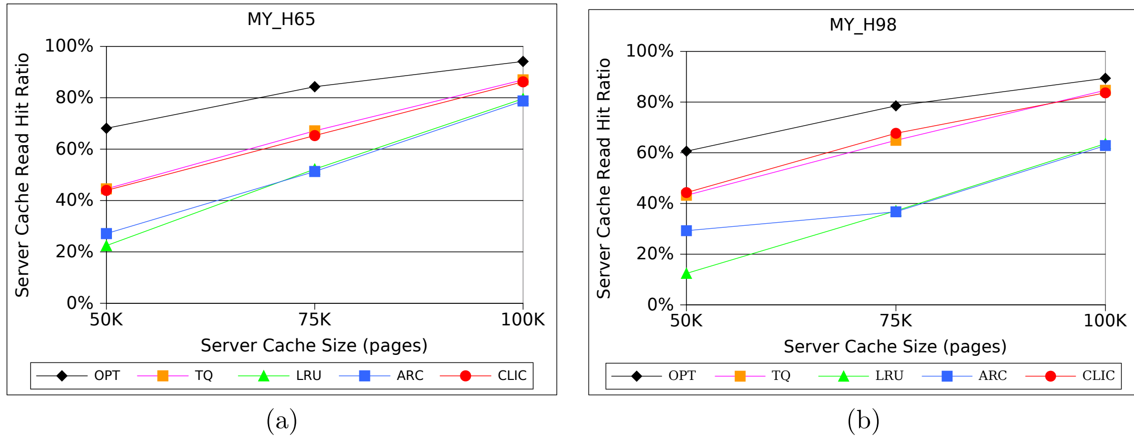


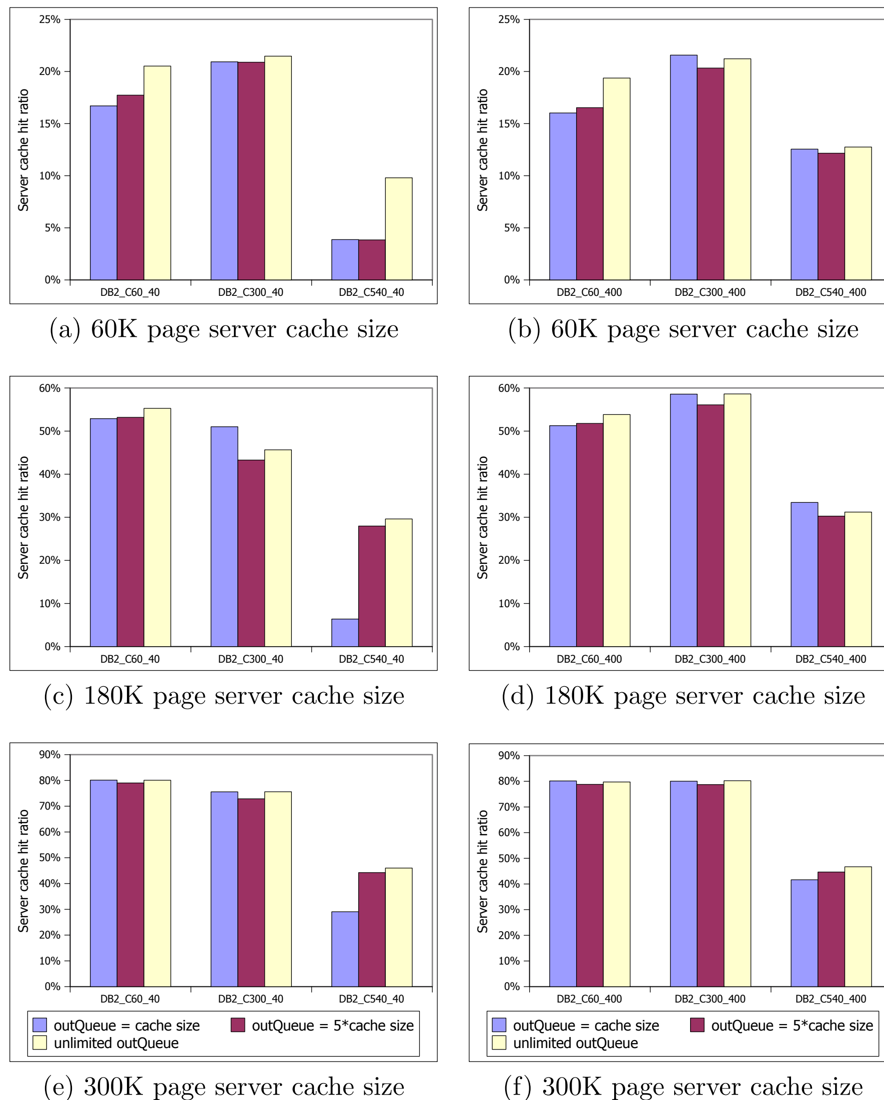
Figure 3.12: Read Hit Ratio of Caching Policies for the MySQL Workloads

(e) in Figure 3.14 show the results for the DB2 TPC-C traces, with a server cache size of 60K, 180K, and 300K pages respectively. We obtained similar results with the DB2 TPC-C traces for other server cache sizes. In all cases, tracking the 20 most frequent hints (i.e., setting $k = 20$) was sufficient to achieve a read hit ratio close to what we could obtain by tracking all of the hints in the trace. In many cases, tracking fewer than 10 hints sufficed. The curve for the DB2_C540_400 trace shows that the Space Saving algorithm that we use to track frequent hint sets can sometimes suffer from some instability, in the sense that larger values of k may result in worse performance than smaller k. This is because hint sets reported by the Space Saving algorithm when $k = k_1$ are not guaranteed to be reported by the space saving algorithm when $k > k_1$. We only observed this problem occasionally, and only for very small values of k.

The right side graphs (b) (d) (f) in Figure 3.14 show the results for the DB2 TPC-H traces, with a server cache size of 60K, 180K and 300K pages respectively. For all of the DB2 TPC-H traces and all of the cache sizes that we tested, $k = 10$ was sufficient to obtain performance close to that obtained by tracking all hint sets. Overall, we found the top-k approach to be very effective at cutting down the number of hints to be considered by CLIC.

3.4.4 Increasing the Number of Hints

In the previous experiment, we studied the effectiveness of the top-k approach at reducing the number of hints that must be tracked by CLIC. In this experiment, we consider a similar question, but from a different perspective. Specifically, we consider a scenario in which CLIC is subjected to useless "noise" hints, in addition to the useful hints that it has exploited in our previous experiments. We limit the number of hint sets that CLIC is able to track and increase the level "noise". Our objective is to determine whether the



(a) 60K page server cache size

(b) 60K page server cache size

(c) 180K page server cache size

(d) 180K page server cache size

(e) 300K page server cache size

(f) 300K page server cache size

Figure 3.13: Effect of Outqueue Size on Read Hit Ratio
Each bar represents a different outqueue size.

top-k approach is effective at ignoring the noise, and focusing the limited space available for hint-tracking on the most useful hints.

In practice, we hope that storage clients will not generate lots of useless hints. However, in general, clients will not be able to determine how useful their hints are to the server, and some hints generated by clients may be of little value. By deliberately introducing a controllable level of useless hints in this experiment, we hope to test CLIC's ability to tolerate them without losing track of those hints that are useful.

To study the effectiveness of these techniques, we added synthetic "noise" hints to our

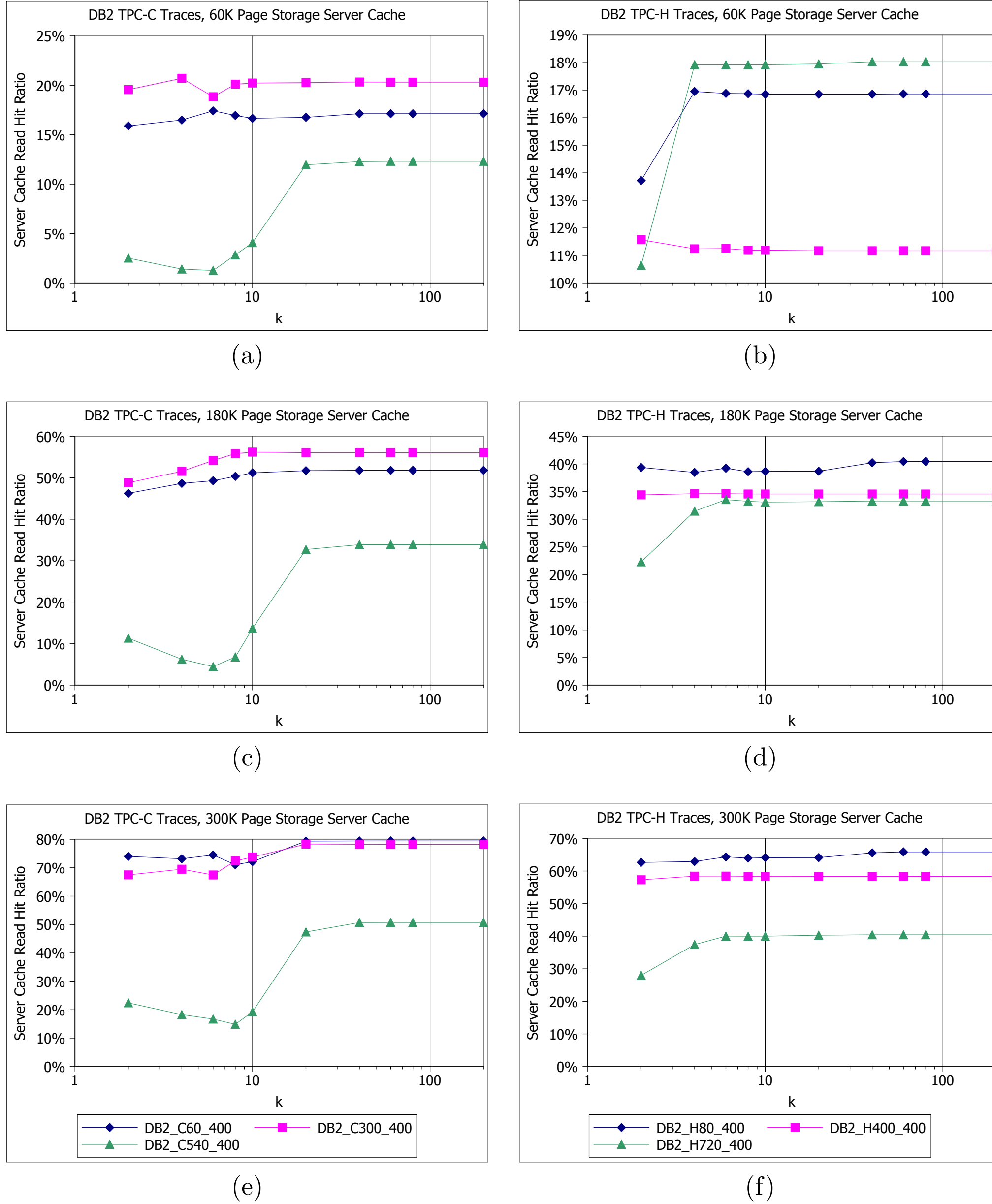


Figure 3.14: Effect of Top-K Hint Set Filtering on Read Hit Ratio

existing traces in addition to the original hints. In this experiment we consider how these techniques will perform as the number of distinct hint sets in the input trace is increased. Since each of our workload traces has a fixed number of distinct hint sets, we increased the number of hints by injecting additional synthetic hints into our traces.

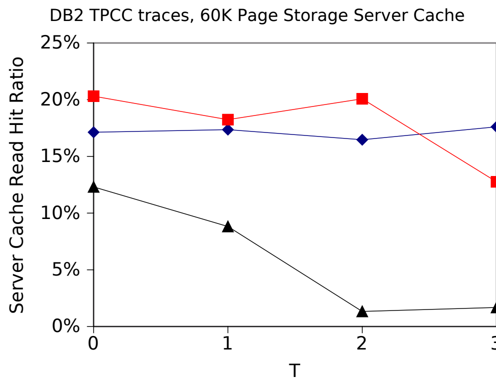
For this experiment we used our DB2 TPC-C and TPC-H traces, each of which contains 5 real hint types, and added T additional synthetic hint types. In other words, each request

will have $5+T$ hints associated with it, the five original hints plus T additional synthetic hints. Each injected synthetic hint is chosen randomly from a domain of D possible hint values. A particular value from the domain is selected using a Zipf distribution with skew parameter $z = 1$. When $T > 1$, each injected hint value is chosen independently of the other injected hints for the same record. Since the injected hints are chosen at random, we do not expect them to provide any information that is useful for server cache management. This injection procedure potentially increases the number of distinct hint sets in a trace by a factor D^T. For our experiments, we chose $D = 10$, and we varied T, which controls the amount of "noise".

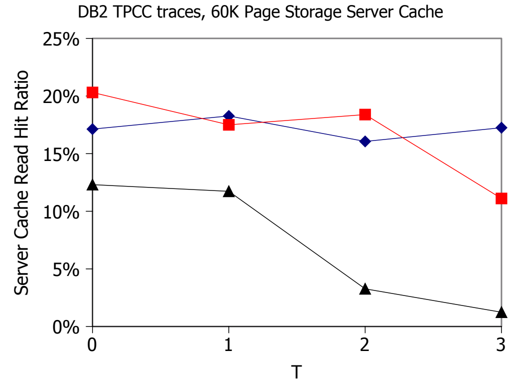
First, we evaluate the effectiveness of top-k approach when noise hint types are added to the DB2 TPC-C workload traces. Figure 3.15 shows the read hit ratios in a server cache of size 60K, 180K and 300K pages as a function of T. We fixed $k = 100$ and $k = 200$ for the top-k algorithm. With $k = 100$, the number of hints tracked by CLIC remains fixed at 100 as the number of useless hints increases. As T goes from 0 to 3, the total number of distinct hint sets in each trace increases from just over 100 (the number of distinct hint sets each TPC-C trace), to about 1000 when $T = 1$, and to more than 50000 when $T = 3$.

Ideally, the server cache read hit ratio would remain unchanged as the number of "noise" hints is increased. In practice, however, this is not the case. As shown in Figure 3.15, CLIC fares reasonably well for the DB2_C60_400 trace, suffering mild degradation in performance for $T \geq 2$. However, for the other two traces, CLIC experienced more substantial degradation, particularly for $T \geq 2$. The cause of the degradation is that high-priority hint sets from the original trace get "diluted" by the additional noise hint types. For example, with $D = 10$ and $T = 2$, each original hint set is split into as many as $D^T = 100$ distinct hint sets because of the additional noise hints that appear with each request. Since CLIC has limited space for tracking hint sets, the dilution eventually overwhelms its ability to track and identify the useful hints. Even when k is increased from 100 (the left side graphs of Figure 3.15) to 200 (the right side graphs of Figure 3.15), the read hit ratios have not been improved.

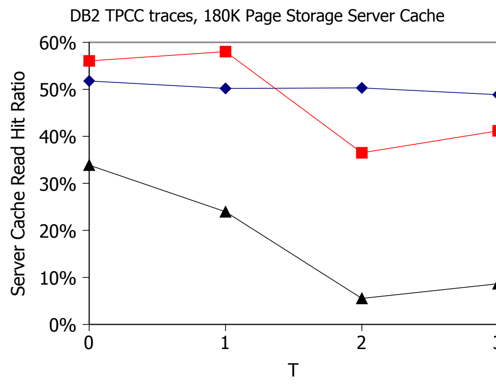
Secondly, we evaluate the effectiveness of top-k approach when noise hint types are added to the DB2 TPC-H workload traces. Figure 3.16 shows the read hit ratios in a server cache of size 60K, 180K and 300K pages as a function of T. Similarly, we fixed $k = 100$ and $k = 200$ for the top-k algorithm. For the DB2_H400_400 trace, CLIC fares reasonably well, suffering mild degradation in performance for $T \geq 2$. This is because for the original DB2_H400_400 trace, CLIC only needs to track $k = 2$ hint sets to obtain performance close to that obtained by tracking all hint sets (Shown in Figure 3.14). Even though the original hint sets have been diluted by the noise, the performance of CLIC does not degrade much. However, for the DB2_H720_400 trace, CLIC experienced more substantial degradation, particularly for $T \geq 2$. One interesting observation is made on DB2_H80_400. When T is increased, CLIC's performance actually increases in some cases. By analyzing the original hint sets of DB2_H80_400, we found that one hint set has the highest caching priority and frequency, and pages with this hint set occupy all of the server



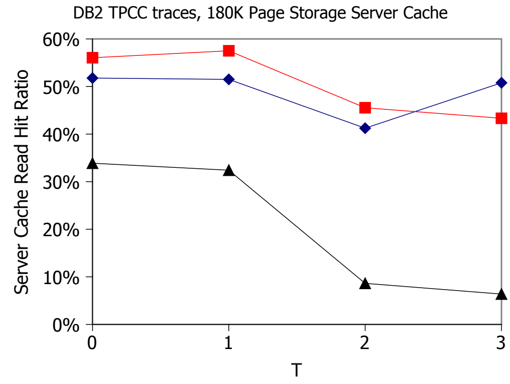
(a) k=100



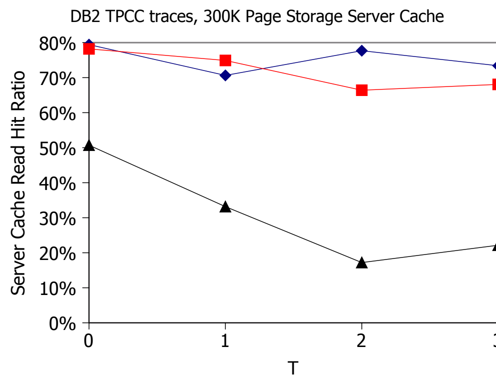
(b) k=200



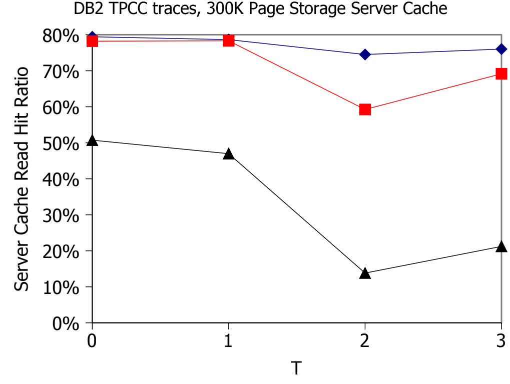
(c) k=100



(d) k=200



(e) k=100



(f) k=200

Figure 3.15: Effect of Top-K Hint Set Filtering on Read Hit Ratio with TPC-C Workload Traces

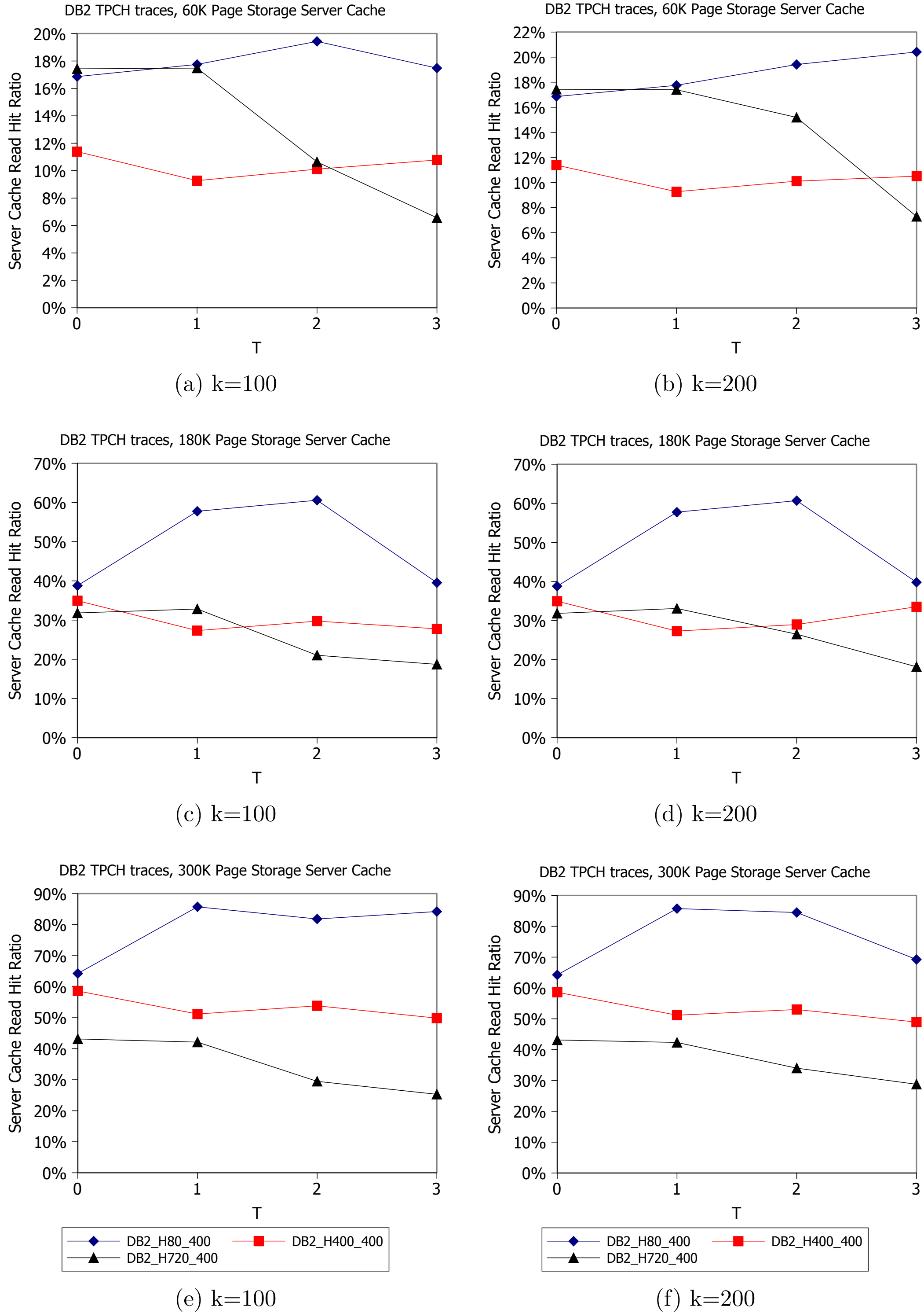


Figure 3.16: Effect of Top-K Hint Set Filtering on Read Hit Ratio with TPC-H Workload Traces

cache. The problem with this hint set is that some pages with this hint set have very long read re-reference distances, and CLIC is unable to evict any of these pages because they all have the highest caching priority. When noise hints are added, this hint set has been diluted into several hint sets with difference caching priorities. Thus, pages that once had the highest caching priority now have different caching priorities and some of them can be evicted from the cache. However, when more noise hints are added ($T = 3$), the performance starts to drop because k is not large to enough to collect all higher-priority hint sets.

This experiment suggests that it may be necessary to tune or modify CLIC to ensure that it operates well in situations in which the storage clients provide too many low-value hints. One way to address this problem is to increase k as the number of hints increases, so that CLIC is not overwhelmed by the additional hints. Controlling this trade-off of space versus accuracy is an interesting tuning problem for CLIC. (We address this problem in Chapter 5).

3.4.5 Multiple Storage Clients

One desirable feature of CLIC is that it should be capable of accommodating hints from multiple storage clients. The clients independently send their different hints to the storage server without any coordination among themselves, and CLIC should be able to effectively prioritize the hints to get the best overall cache hit ratio.

To test this, we simulated a scenario in which multiple instances of DB2 share a storage server. Each DB2 instance manages its own separate database, and represents a separate storage client. All of the databases are housed in the storage server, and the storage server's cache must be shared among the pages of the different databases. To create this scenario, we create a multi-client trace for our simulator by interleaving requests from several DB2 traces, each of which represents the requests from a single client. We interleave the requests in a round robin manner, one from each trace. We truncate all traces to the length of the shortest trace being interleaved to eliminate bias towards longer traces. We treat the hint types in each trace as distinct, so the total number of distinct hint sets in the combined trace is the sum of the number of distinct hint sets in each individual trace.

Figure 3.17 shows results for the trace generated by interleaving the DB2_C60, DB2_C400, and DB2_C540 traces. The server cache size is 180K pages, and CLIC uses top-k filtering with $k = 100$. The figure shows the read hit ratio for the requests from each individual trace that is part of the interleaved trace. The figure also shows the overall hit ratio for the entire interleaved trace. For comparison, the figure shows the hit ratios for the full-length (untruncated) traces when they use independent caches of size 60K pages each (i.e., the storage server cache is partitioned equally among the clients). The figure shows a dramatic improvement in hit ratio for the DB2_C60 trace and also an improvement in the overall hit ratio as compared to equally partitioning the server cache among the traces. CLIC is able

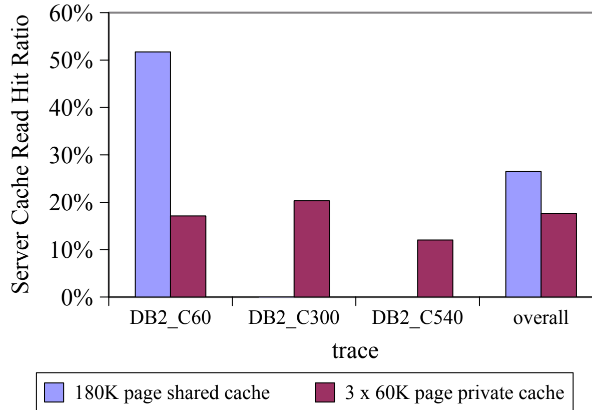


Figure 3.17: Read Hit Ratio with Three Clients
Read hit ratio is near zero for the DB2_C300 and DB2_C540 traces in the 180K page shared cache, so bars are not visible.

to identify that the DB2_C60 trace presents the best caching opportunities (since it has the most temporal locality), and to focus on caching pages from this trace. This illustrates that CLIC is able to accommodate hints from multiple storage clients and prioritize them so as to maximize the overall hit ratio.

Note that it is possible to consider other objectives when managing the shared server cache. For example, we may want to ensure fairness among clients or to achieve certain quality of service levels for some clients. This may be accomplished by statically or dynamically partitioning the cache space among the clients. In CLIC, the objective is simply to maximize the overall cache hit ratio without considering quality of service targets or fairness among clients. This objective results in the best utilization of the available cache space. Our experiment illustrates that CLIC is able to achieve this objective, although the benefits of the server cache may go disproportionately to some clients at the expense of others.

3.5 Conclusion

We have presented CLIC, a technique for managing a storage server cache based on hints from storage client applications. CLIC provides a general, adaptive mechanism for incorporating application-provided hints into cache management. We used trace-driven simulation to evaluate CLIC, and found that it was effective at learning to exploit hints. In our tests, CLIC learned to perform as well as or better than TQ, an ad hoc hint based technique. In many scenarios, CLIC also performed substantially better than hint-oblivious techniques such as LRU and ARC. Our results also show that CLIC, unlike TQ and other ad hoc techniques, can accommodate hints from multiple client applications.

A potential drawback of CLIC is the space overhead that is required learning which hints

are valuable. We considered a simple technique for limiting this overhead, which involves identifying frequently-occurring hints and tracking statistics only for those hints. In many cases, we found that it was possible to significantly reduce the number of hints that CLIC had to track with only minor degradation in performance. However, although tracking only frequent hints is a good way to reduce overhead, the overhead is not eliminated and the space required for good performance may increase with the number of hint types that CLIC encounters. In Chapter 5, we apply a feature selection technique to *generalize* hint sets by grouping related hint sets together into a common class. We expect that this approach, together with the frequency-based approach, can enable CLIC to accommodate a large number of hint types.

Chapter 4

Dynamic Priority CLIC

As discussed in Chapter 3, the lower-tier cache is hard to manage because the temporal locality in the request streams has been filtered by the upper-tier cache. Zhou et al. [76] use *reuse distance* (the same as re-reference distance defined in Chapter 3) histograms to observe the temporal locality of several traces of requests to the lower-tier cache. They point out that the reuse distance histograms of these traces exhibit two common patterns. First, all histograms are hill-shaped, which is not true of reuse histograms for the upper-tier cache. For example, Figure 4.1 shows the histograms obtained by grouping reuse distances by powers of two. Compared to the upper-tier cache (Figure 4.1(a)), in which 74 percent of references have a reuse distance less than or equal to 16, of the references to the lower-tier cache have 99 percent reuse distance is greater than 512 (Figure 4.1(b)). Thus, recency-based replacement algorithms, which work well for the upper-tier cache, do not work well for the lower-tier cache. Second, the beginning, peak and end of the "hill" region, while different, all depend on the size of the upper-tier cache and workload characteristics. Because most references occur in the hill portion, Zhou et al. [76] argue that a good algorithm for the lower-tier cache should retain blocks in the cache long enough to reach the hill. By examining block access frequency, they find that blocks accessed more frequently contribute more access in the hill region. Thus, they propose a cache replacement policy called MQ to combine the temporal locality and access frequency to make replacement decisions.

While it is clear that re-reference distance histograms for all requests to the lower-tier cache are hill-shaped, the re-reference distance histograms for requests with the same hint set have not been examined. Our hint-based replacement algorithm – CLIC (in Chapter 3) is based on the hint benefit/cost model. For each hint set passed by the storage clients, CLIC calculates the hint caching priority using the overall read hit rate and the mean re-reference distance of each hint set. All pages with the same hint set have the same caching priority, and are managed in one LRU list. However, if the re-reference distance histograms of each hint set are hill-shaped, LRU, which is designed to take advantage of temporal locality, may not be a good way for CLIC to manage pages with the same hint

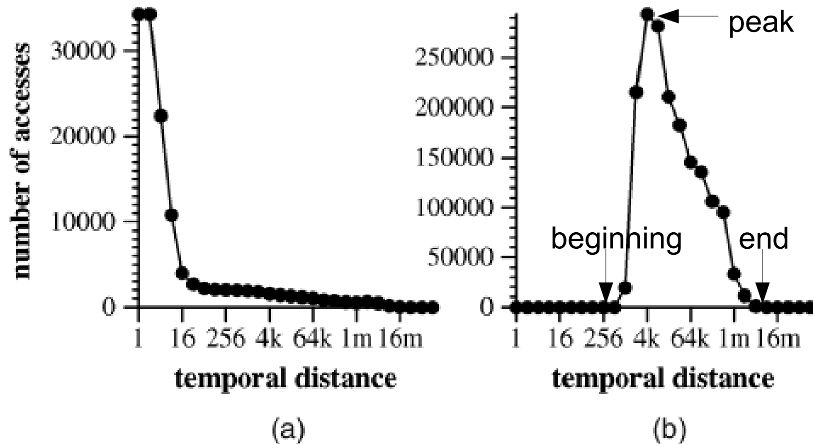


Figure 4.1: Temporal Locality of Upper-tier and Lower-tier Cache Accesses Using Reuse Distance Histograms. (a) Auspex Client Trace and (b) Auspex Server trace[76].

set.

Our hypothesis is that the re-reference distance histograms of requests with the same hint set exhibit an access pattern similar to that of all requests. In CLIC, the page priority is assigned as the priority ($\mathtt{Pr}(H)$) of the hint set H with which that page was last requested. The page priority will not change until the page is requested again. Intuitively, as the read hit rates do not distribute evenly and vary with the re-reference distance, the page caching priority should also depend on how long the page has stayed in the cache. Thus, we propose to collect re-reference histograms for each hint set so that read hit rates can be calculated based on re-reference distance. Then we propose to perform a dynamic benefit/cost analysis for each hint set. Since CLIC manages pages with fixed caching priorities, we address the following questions in this chapter:

- Are the re-reference distance histograms of requests with the same hint set also hill-shaped?
- Can the performance of CLIC be improved by considering how long a page has been in the cache?

We also extend CLIC to provide a new algorithm – dynamic priority CLIC (DP-CLIC), which is based on a dynamic benefit/cost model. Unlike the page caching priority in CLIC, the page caching priority of DP-CLIC not only depends on the hint sets attached to the most recent request for the page, but also on how long the page has been in the cache. DP-CLIC further improves the lower-tier cache performance by making replacement decision using dynamic caching priority.

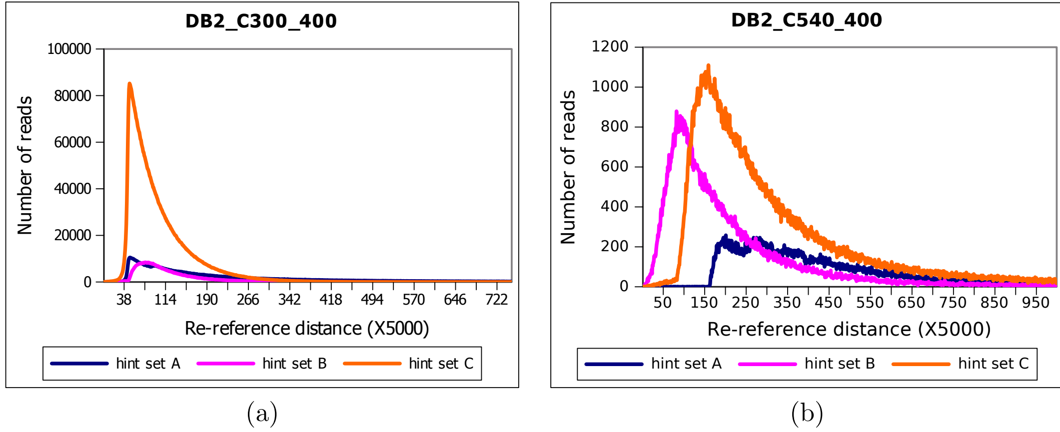


Figure 4.2: Read Reference Distance Histograms of DB2 TPC-C Workload Traces

4.1 Re-reference Histogram of Hint Sets

In order to understand the access pattern in terms of hint sets, we collected re-reference histograms for each hint set from the DB2 traces, using the approach described in Section 3.2.2. Figure 4.2 shows read reference histograms of several sample hint sets from the DB2 traces. We see that the read reference histograms are all hill-shaped, which correspond to that of all requests to lower-tier caches. Different hint sets' re-reference histograms have different peaks and hill shapes. After pages are placed in the cache, they are most likely to be read during the hill region. In each histogram, after the end of the hill region, there is a long "tail" which has lower read probability with longer distances. Intuitively, the caching priority should drop after the hill region because the benefit is lower and the cost is higher compared to that in the hill region. However, the benefit/cost model of CLIC cannot detect the hill region. Instead, it keeps the page caching priority unchanged until the page is requested again. In the worst case, a page which is requested with a high priority hint set may stay in the cache forever without any benefit.

4.2 Dynamic Benefit/Cost Model

To take into account the impact of hill-shaped access pattern on the caching priority of hint sets, in this section, we discuss a dynamic benefit/cost model for hint analysis. The purpose of this model is to calculate the dynamic caching priority for each hint set as a function of re-reference distance.

Besides the total count of requests with hint set H ($N(H)$ introduced in Section 3.2.1), DP-CLIC tracks read-reference and write-reference histograms for each hint set. Figure 4.3 illustrates the histogram and some notation used by the dynamic benefit/cost model with

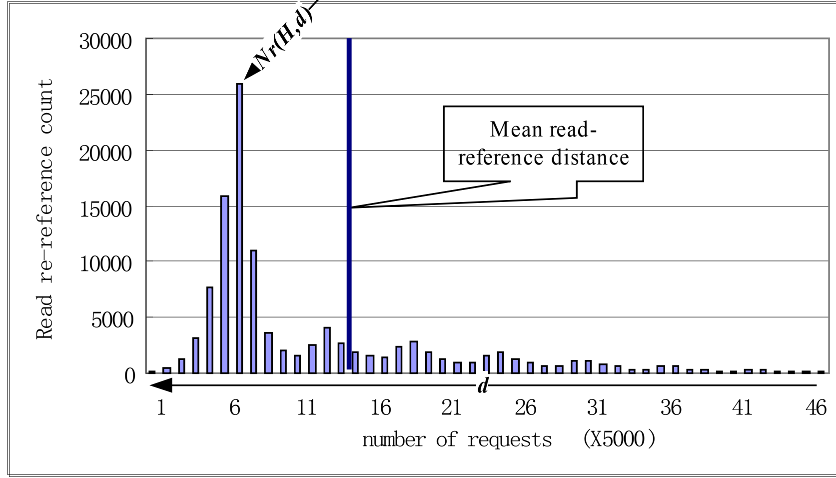


Figure 4.3: Illustration of Read Reference Histogram

a read-reference histogram of a sample hint set. The two histograms are collected based on the re-reference distance:

$N_r(H, d)$: the total number of requests in the input which have hint set H and for which the next request for the same page is a read that occurs at re-reference distance d. Note that the total number of read re-references equals to the sum of read references at each re-reference distance.

$N_w(H, d)$: the total number of requests in the input which have hint H and for which the next request for the same page is a write that occurs at re-reference distance d.

The sum of $N_r(H, d)$ and $N_w(H, d)$ may be less than $N(H)$ because some requests' pages may never be re-referenced in the input sequence:

$$\sum_{d=1}^{\infty} N_r(H, d) + \sum_{d=1}^{\infty} N_w(H, d) \leq N(H) \tag{4.1}$$

Suppose that page P was last referenced d requests ago with hint set H. The idea of DP-CLIC is to use re-reference histogram of H, plus d, to determine the cache's priority for page P. DP-CLIC estimates the conditional probability of a read request for P, given that d requests have occurred since P was last referenced. To do this, DP-CLIC defines two other statistics:

$N(H, d)$: the total number of re-references in the histogram for H with re-reference distance larger than d. As illustrated by Figure 4.3, this number does not include reads and

writes re-references at distance d or less:

$$N(H, d) = N(H) - \sum_{d'=0}^{d} (N_r(H, d') + N_w(H, d')) \tag{4.2}$$

$P_r(H, d', d)$: the probability that a page with hint set H will be read at re-reference distance d', given that there have been d requests since the page was last referenced ($d' \geq d$).

$$P_r(H, d', d) = N_r(H, d')/N(H, d) \tag{4.3}$$

At each re-reference distance d' ($d' \geq d$), $P_r(H, d', d)$ represents a probability that the server may obtain a benefit if it keeps caching p from d to d'. As defined in Section 3.2.1, the benefit of a read re-reference is 1. Hence, $P_r(H, d', d)$ can be interpreted as the conditional expected benefit of continuing to cache the page.

$$benefit(H, d', d) = P_r(H, d', d) \tag{4.4}$$

The cost of caching that page until d' $cost(H, d', d)$ is

$$cost(H, d', d) = d' - d \tag{4.5}$$

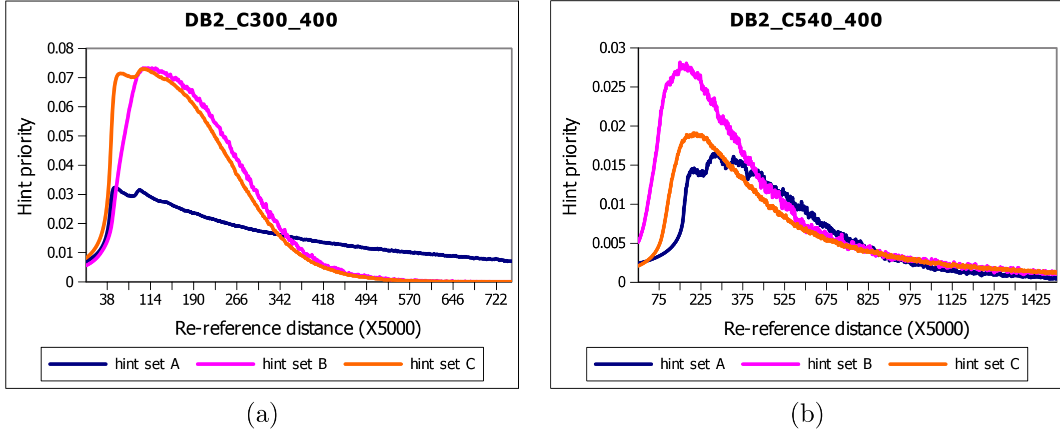
We define the caching priority pages with hint set H and distance d to their previous request as the sum of the expected benefit/cost at each re-reference distance d' ($d' \geq d$):

$$\mathtt{Pr}(H, d) = \sum_{d'=d}^{\infty} \frac{benefit(H, d', d)}{cost(H, d', d)} \tag{4.6}$$

Figure 4.4 shows the priority curves of the same example hint sets shown in Figure 4.2 (a priority curve is priority as a function of d). Like the curves of read-reference histogram in Figure 4.2, the priority curves are also hill-shaped. The priority curves start at a lower value. Following the increasing of the benefit (read hit ratio), the priority curves reach their peak points at the similar peak points of the read curves.

4.3 Tracking Hint Statistics

To track hint set statistics for the dynamic benefit/cost model, DP-CLIC maintains the same hint table as CLIC (Section 3.2.2). The dynamic benefit/cost model requires the read and write distributions based on re-reference distance for each hint set. Thus, besides the statistics $N(H)$, the hint table entry for H maintains two histograms: a read-reference histogram and a write-reference histogram. When the server receives a request for page p, with sequence number s, it checks both the cache and the outqueue for information about



(a) (b)

Figure 4.4: Hint Set Priority Produced by Dynamic Benefit/Cost Model of DB2 TPC-C Workload Traces

the most recent previous request, if any, for p. If it finds $\texttt{seq}(p)$ and $\texttt{H}(p)$ from a previous request, then it knows that the current request is a re-reference of p. If the request is a read request, DP-CLIC increments the read count in bucket $s - \texttt{seq}(p)$ (the re-reference distance) of $\texttt{H}(p)$. If the request is a write request, DP-CLIC increments the write count in bucket $s - \texttt{seq}(p)$ of $\texttt{H}(p)$.

The range of the histograms is determined by the number of buckets (N_{bucket}) per histogram and the width of the buckets (W_{bucket}), which are parameters of DP-CLIC. These two parameters control a trade off between the space consumption and the accuracy of the histograms. As shown in Figure 4.2, the most important portion of the histogram is the hill portion. Thus, the overall reference distance ($N_{bucket} \times W_{bucket}$) of the histogram should be able to cover the hill portions of all higher-priority hint sets. If W_{bucket} is small, the histograms are more accurate but a large N_{bucket} is needed to cover the hill portions. If W_{bucket} is large, the histograms have a coarse granularity but a small N_{bucket} is enough to cover the hill portions, and thus, space overhead can be reduced. However, the benefit/cost model may not be accurate because the histograms have a coarse granularity.

4.4 Cache Management

Section 3.2.4 describes how CLIC uses the hint set priorities to manage the contents of its cache. The cache management of DP-CLIC, described in Figure 4.5 is similar to that of CLIC. They both are priority-based replacement policies. However, the page priority in CLIC only depends on which hint set the page was requested with while the page priority in DP-CLIC also depends on how long the page has stayed in the cache. For each hint set, DP-CLIC maintains a dynamic caching priority in terms of re-reference distance. To

determine the caching priority for page p, $\texttt{Pr}(H, d)$, DP-CLIC needs the hint set H that p has been most recently requested with, and d, which indicates how long p has been cached.

The difference between CLIC and DP-CLIC is on the setting of page caching priority and on how they identify the page with the minimum priority in the cache. We explain the cache management of DP-CLIC using the algorithm in Figure 4.5. When the server receives a request (p, H), with sequence number s, the caching priority that DP-CLIC assigns to page p is set to the initial priority $\texttt{Pr}(H, 0)$ (in line 31). For a page v in the cache, DP-CLIC calculates its d (how long it has stayed in the cache) as $s - seq(v)$, and then sets the priority of page v to $\texttt{Pr}(\texttt{H}(v), s - seq(v))$. Theoretically, to find the page having the minimum priority, DP-CLIC needs to calculate and compare the priority of all pages in the cache.

In our implementation of the DP-CLIC cache management, we take advantage of the fact that the priority curves of hint sets are hill-shaped (from line 12 to line 30). When a page has just arrived in the cache, it has a lower priority, then its priority gradually grows until it reaches its peak point. After then, the priority decreases. DP-CLIC maintains a peak point for each hint set $H(peak)$ to record the re-reference distance at which the priority reaches the maximum. Like CLIC, DP-CLIC maintains a queue of the hint sets. For each hint set H in the queue, all pages with $\texttt{H}(p) = H$ are recorded in a doubly-linked list that is sorted by $\texttt{seq}(p)$. Based on the hill-shaped priority curves, the page with the minimum priority may be the MRU page or the LRU page in the list. DP-CLIC calculates the distance for both MRU and LRU pages, and there are three cases:

- Both MRU page and LRU page have not passed the peak point $H(peak)$. In this case, the MRU page has the minimum priority.
- Both MRU page and LRU page have passed $H(peak)$. In this case, the LRU page has the minimum priority.
- MRU page has not passed the peak point but LRU page has passed $H(peak)$. In this case, DP-CLIC compares the priority of the two pages and identifies the one with the minimum priority.

After identify the page with the minimum priority of each hint set, DP-CLIC identifies the minimum priority page in the cache among these pages. The run time is $O(n)$, in which n is the number of the hint sets.

4.5 DP-CLIC Priority vs. CLIC Priority

As discussed in Section 3.2.1, CLIC's benefit/cost model does not consider the variation of the benefit and cost, and hint set priorities of CLIC do not vary with the re-reference distance. Thus, the disadvantage of CLIC is that pages with high CLIC priority cannot

```
1≻  if p is not cached then
2≻    if the cache is not full then
3≻      cache p
4≻      set seq(p) = s
5≻      set H(p) = H
6≻    else
7≻      let m be the minimum priority
8≻        of all pages in the cache
9≻      let v be the page with the
10≻        minimum sequence number seq(v)
11≻        among  all pages with priority m
12≻      for H’ in all hint sets in the cache
13≻         let pm be the MRU page of H’
14≻         let pl be the LRU page of H’
15≻         if s-seq(pm) > H’(peak)
16≻            if m > Pr(H’, s-seq(pl))
17≻                m= Pr(H’, s-seq(pl))
18≻                v=pl
19≻           else if s-seq(pl) < H’(peak)
20≻               if m > Pr(H’, s-seq(pm))
21≻               m= Pr(H’, s-seq(pm))
22≻               v=pm
23≻          else
24≻              if Pr(H’, s-seq(pm)) < Pr(H’, s-seq(pl))
25≻                if m > Pr(H’, s-seq(pm))
26≻                  m= Pr(H’, s-seq(pm))
27≻                  v=pm
28≻              else if m > Pr(H’, s-seq(pl))
29≻                 m= Pr(H’, s-seq(pl))
30≻                 v=pl
31≻      if Pr(H,0)>m then
32≻        evict v from the cache
33≻        add entry for v (with seq(v)
34≻          and H(v)) to the outqueue
35≻        cache p
36≻        set seq(p) = s
37≻        set H(p) = H
38≻      else /* do not cache p */
39≻        add entry for p to the outqueue
40≻        set seq(p) = s
41≻        set H(p) = H
42≻  else /* p is already cached */
43≻    seq(p) = s
44≻    H(p) = H
```

Figure 4.5: The DP-CLIC Cache Replacement Policy
This pseudo-code shows how the server handles a request for page p with hint set H and request sequence number s.

be evicted from the cache even there is no further benefit to keep them in the cache. The purpose of DP-CLIC is to improve performance by detecting the variation (with time) of the caching priority and evicting pages from the cache when their priority is low. Intuitively, how much DP-CLIC can further improve performance depends on the read and write distributions of hint sets. In this section, we investigate CLIC priority and DP-CLIC priority based on hint sets that have different types of read and write distributions, with the goal of identifying types of reference distribution DP-CLIC may provide the greatest improvement relative to CLIC.

In CLIC's benefit/cost model, the benefit is the read hit rate and the cost is the average read re-reference distance of the hint set. Thus, the caching priority of a hint set (H) relies on its read hit rate ($f_{hit}(H)$) and the mean read-reference distance ($D(H)$). A higher priority of hint sets may be caused by higher $f_{hit}(H)$, or short $D(H)$, or both. Thus, we start to identify hint sets with different read and write distributions by looking at $f_{hit}(H)$ and $D(H)$ of hint sets. Figure 4.6 illustrates caching priorities of all distinct hint sets in the DB2_60_400 trace that have non-zero priorities. Each point in Figure 4.6 represents a distinct hint set that is present in the trace, and describes the hint set caching priority and frequency of occurrence. Figure 4.7 provides an illustration of the read hit rate (benefit) and the read re-reference distance (cost) of CLIC's hint analysis for the same trace. Only the most frequently occurring hints sets in the trace are represented. Each bubble in Figure 4.7 represents one hint set. The position of the center of the bubble for hint H is determined by $f_{hit}(H)$ and $D(H)$. The size of the bubble represents $N(H)$. In both figures, we identify three specific hint sets: hint "STOCK table replacement write", "STOCK index replacement write", and "ORDER table replacement write". From Figure 4.6 we see that these three hint sets are among hint sets that have high caching priority and high frequency of occurrence. Thus, pages with these three hint sets have better chances to be placed in the cache. These three hint sets are chosen also because they have significantly different benefit and cost values. As shown in Figure 4.7, hint "STOCK table replacement write" represents hint types that have a high read hit rate (about 95%), "STOCK index replacement write" represents hint types that have a low read hit rate but a short read re-reference distance, and "ORDER table replacement write" represents hint types that have medium read hit rate and read re-reference distance.

Figure 4.8 (a) and (b) (x axis is log scale) illustrates DP-CLIC priority curves and CLIC priorities of the three hint sets. As these three hint sets have higher caching priorities, CLIC may keep pages with these hint sets in the cache until they are referenced by new requests. Compared to CLIC, DP-CLIC is able to catch the variation of the benefit and cost by using the dynamic benefit/cost model.

STOCK table replacement write: its DP-CLIC priority curve has the lowest start priority among the three hint sets, and is flat and wide. Its DP-CLIC priority, which does not vary much with the re-reference distance, is similar to its CLIC priority.

STOCK index replacement write: With CLIC, this hint set has the highest priority,

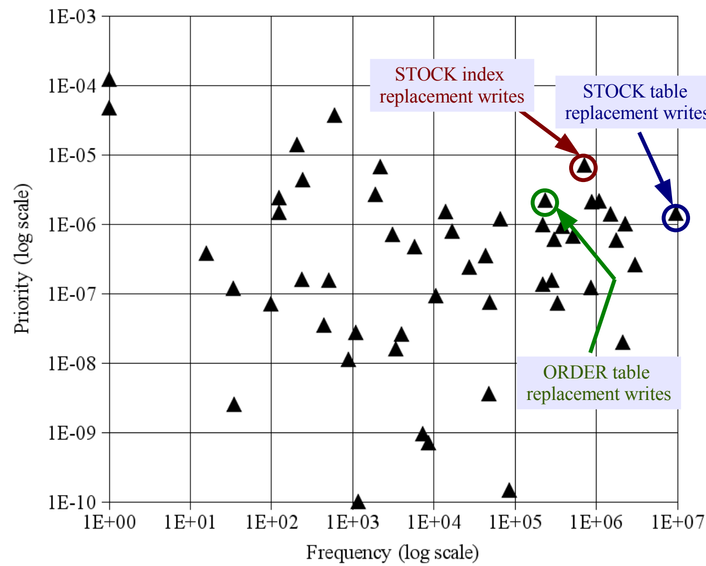


Figure 4.6: Hint Set Priorities for the DB2_C60_400 Trace
Each point represents a distinct hint set. All hint sets are shown.

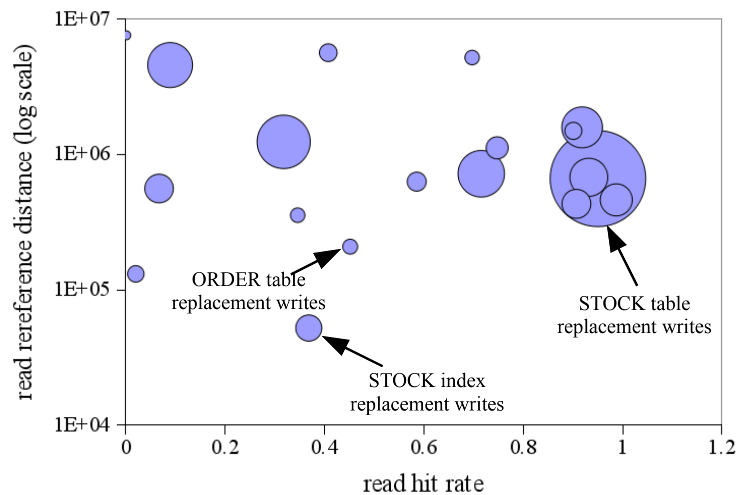
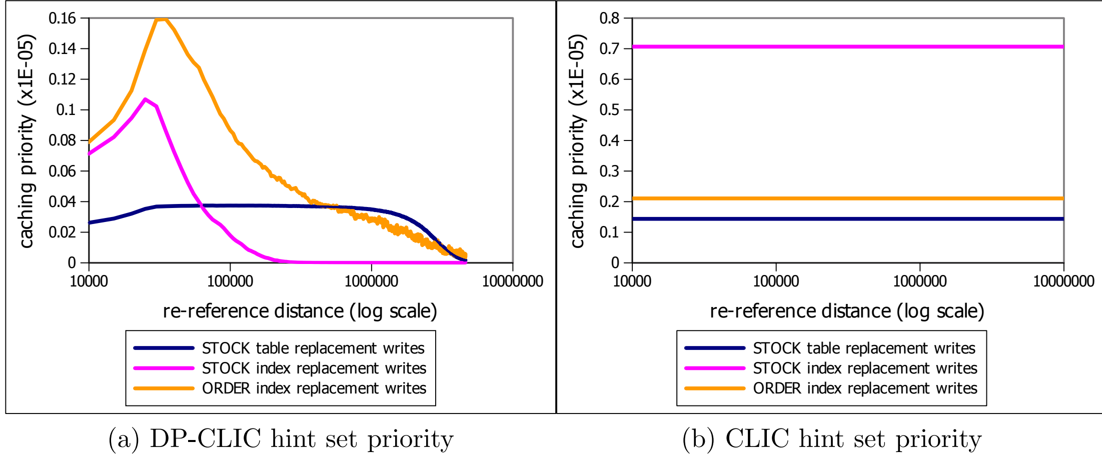


Figure 4.7: Hint Set Statistics for the DB2_C60_400 Trace
Each bubble represents a distinct hint set - not all hint sets in the trace are shown. Each bubble's radius is proportional to the frequency of its hint set.



(a) DP-CLIC hint set priority

(b) CLIC hint set priority

Figure 4.8: DP-CLIC Priority vs. CLIC Priority

and thus, pages with this hint set cannot be replaced from the cache. Its DP-CLIC priority curve drops quickly to zero after its peak point. With DP-CLIC, pages with this hint set are replaced quickly after storing in the cache.

ORDER index replacement write: its DP-CLIC priority curve drops after its peak point but slowly and has a long tail. Unlike CLIC, DP-CLIC can detect the drop of the priority and pages can be replaced by other higher priority pages.

4.6 Experimental Evaluation

We have evaluated CLIC by comparing its performance to several other algorithms' performance in Section 3.4, and CLIC outperforms other algorithms in most cases. In this section, we evaluate DP-CLIC by comparing it with CLIC. To evaluate DP-CLIC, we add the DP-CLIC caching policy to the simulator of the storage server cache used in Section 3.4. In the experiments with DP-CLIC, the width of each histogram bucket, W_{bucket} is set to 5000, and N_{bucket} is set to 3000. The purpose of the read and write histograms is to capture the benefit (read re-reference) as the function of the cost (re-reference distance). Thus, for each hint set, the most important part is the hill portion of the read histogram. With this setting, the histograms cover $15,000,000$ re-reference distances, which is much longer than the hill regions of all hint sets in our traces. As all traces have less than 150 hint sets, the overall space for tracking the histograms and storing the priority of all hint sets is less than $2K$ pages. We deducted $2K$ pages from the cache space when doing experiments with DP-CLIC. We use the DB2 TPC-C and TPC-H workloads listed in Table 3.9. We also collected information about which hint sets in these DB2 traces compete for the cache

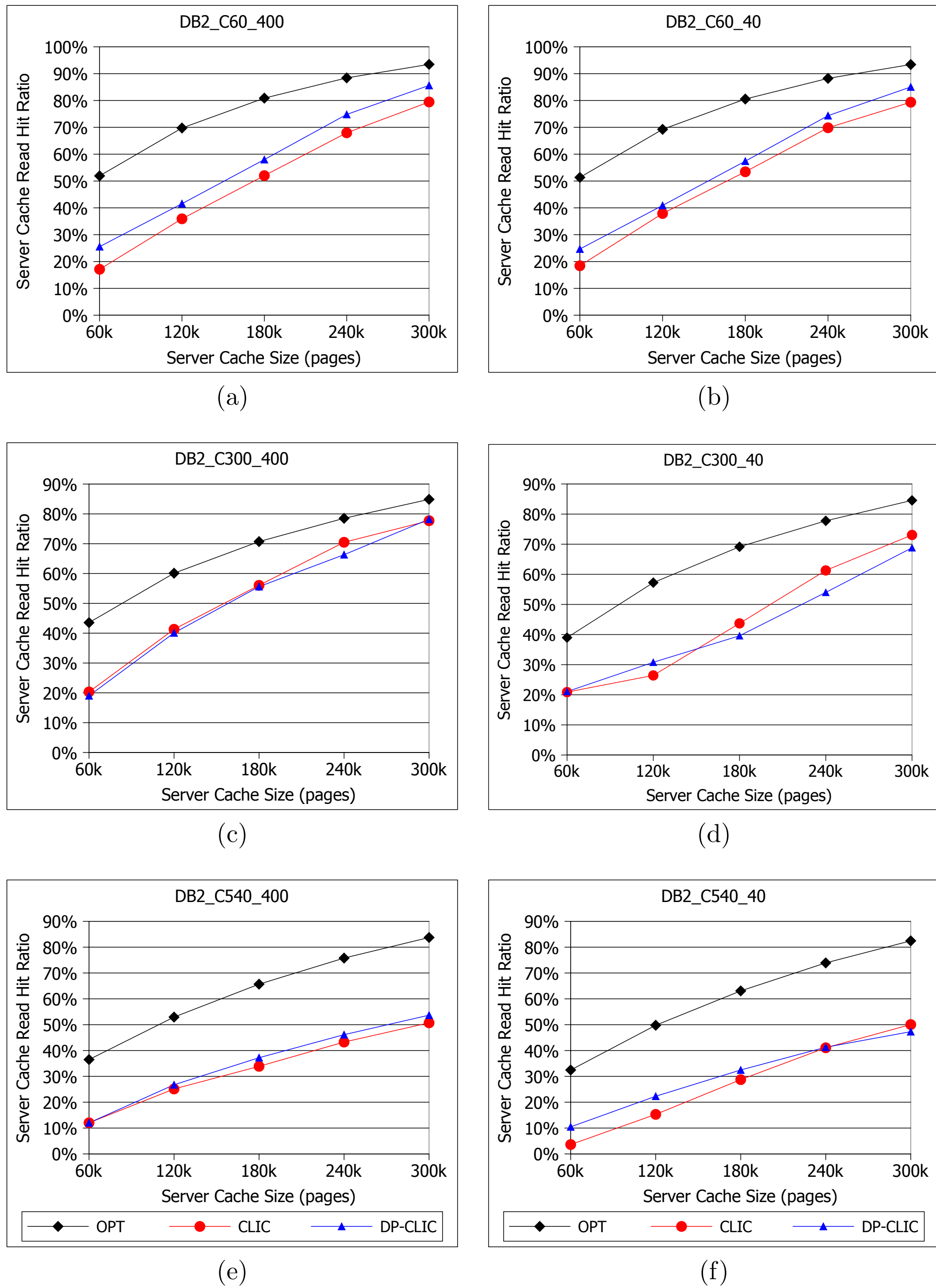
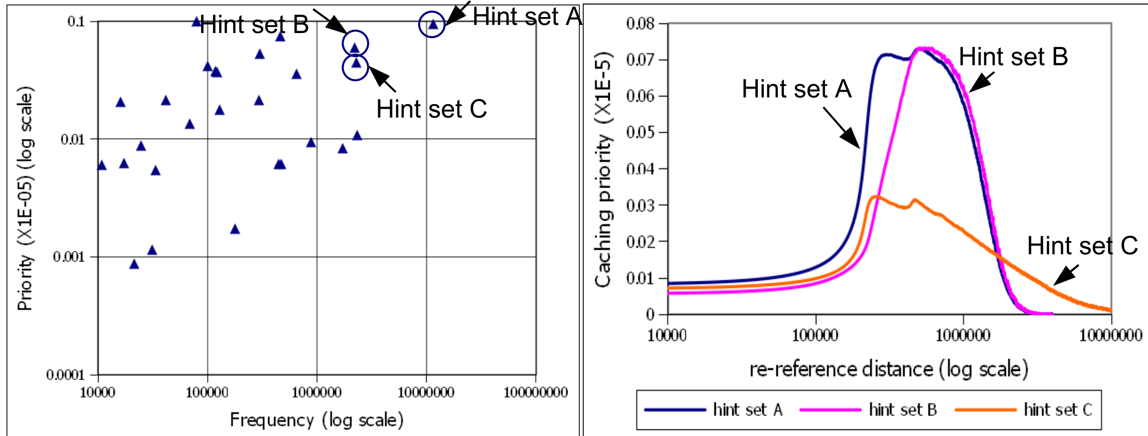


Figure 4.9: Read Hit Ratio of Caching Policies for the DB2 TPC-C Workloads

space in both CLIC and DP-CLIC. The goal of this experimental evaluation is to answer the following questions:

- Can DP-CLIC further identify good caching opportunities for storage server caches,



(a) CLIC hint set priorities

(b) DP-CLIC hint set priorities

Figure 4.10: DP-CLIC Priority vs. CLIC Priority

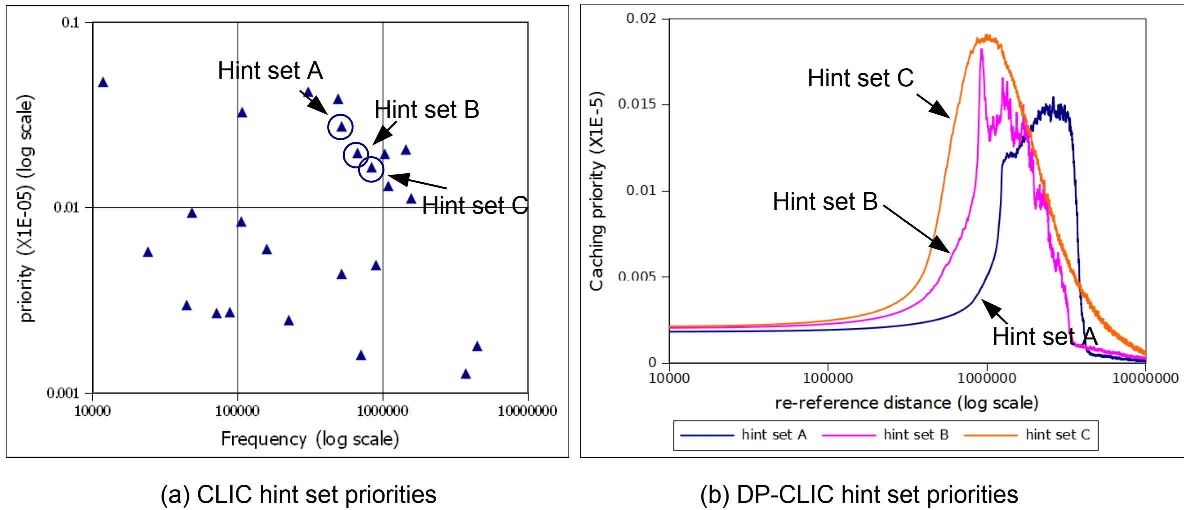
and thereby improve the cache hit ratio relative to CLIC by using dynamic priority model?

- What is the impact of the histogram size on DP-CLIC performance?
- Can top-k work effectively with DP-CLIC for reducing the number of hint sets that DP-CLIC must track?

4.6.1 Evaluation with TPC-C Traces

We use the same experimental setting as described in Section 3.4.1. Figure 4.9 shows the results of the experiments for the DB2 TPC-C workload traces. There are several observations to be made from Figure 4.9.

For DB2_C60_400 and DB2_C60_40 traces, DP-CLIC outperforms CLIC with 5% better read hit rates on all cache sizes. In our experiments, we have logged the information about how many pages of each hint set have been placed in the second tier cache. We observed that the cache space was occupied mostly by pages with the three types of hint sets in Figure 4.8, which we have analyzed in Section 4.5. With CLIC, hint "STOCK index replacement writes" and "ORDER index replacement writes" both have higher caching priorities than hint "STOCK table replacement writes" does. Thus, pages with these two hint sets cannot be replaced by pages with hint sets having lower priorities. DP-CLIC detects the drop of the priorities of the two hint sets and evicts pages with them when their priority is low. The improvement is not large because the number of pages requested with these two hint sets is small (about 4%).



(a) CLIC hint set priorities (b) DP-CLIC hint set priorities

Figure 4.11: DP-CLIC Priority vs. CLIC Priority

For DB2_C300_400 and DB2_C300_40 traces, DP-CLIC has almost the same performance as CLIC does. We identify three hint sets with which pages occupied the cache, because they have the highest caching priorities and are frequently occurring. Figures 4.10 (a) and (b) show their CLIC and DP-CLIC priorities. By analyzing these hint sets, we found that DP-CLIC cannot improve the read hit ratios because the characteristics of these hint sets do not provide space to improve. Among these three hint sets, only hint set C has a long tail in its priority curve that may have bad effect on CLIC performance. However, it has the lowest caching priority among the three hint sets with CLIC. CLIC can replace pages with hint set C with pages having higher priority. Thus, DP-CLIC cannot further improve the performance.

For DB2_C540_400 and DB2_C540_40 traces, DP-CLIC outperforms CLIC with about 5% better read hit rates on most cache sizes. We identify three hint sets with which pages occupied the cache. Figures 4.11 (a) and (b) show their CLIC and DP-CLIC priorities, respectively. Because the upper-tier cache has a large size, these two traces have the least temporary locality among all traces. Under DP-CLIC, the hill regions of hint sets' priority curves start later and are wider compared to hint sets of other traces. From Figure 4.11 we see that Hint set A has higher priority than both Hint set B and Hint set C under CLIC. Thus, with CLIC, pages with Hint set A can not be replaced by pages with Hint set B and C. However, under DP-CLIC, the priority of Hint set A starts lower and its hill region starts later than the other two hint sets, which means pages with Hint set A need to stay in the cache longer to achieve the benefit. With DP-CLIC, if pages with Hint set A are in the cache, they can be replaced by pages with Hint set B and C before their priorities reach the hill region. Thus, DP-CLIC improves the read hit ratios by evicting pages when their priorities are still low.

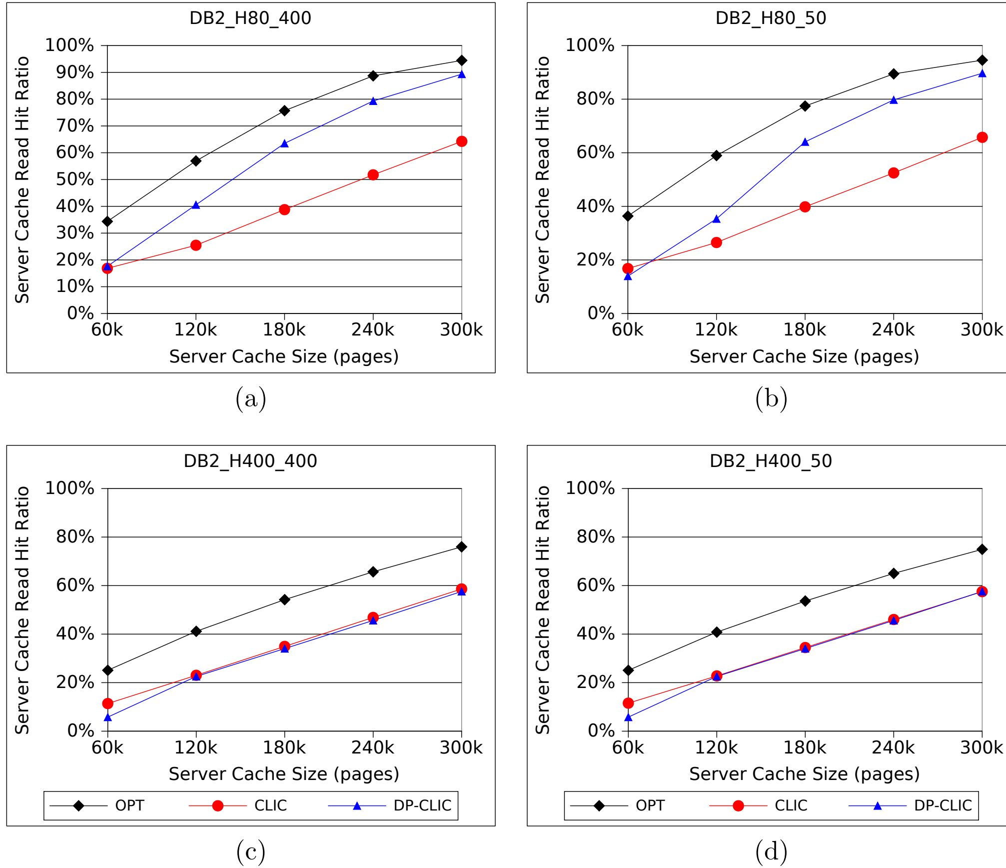
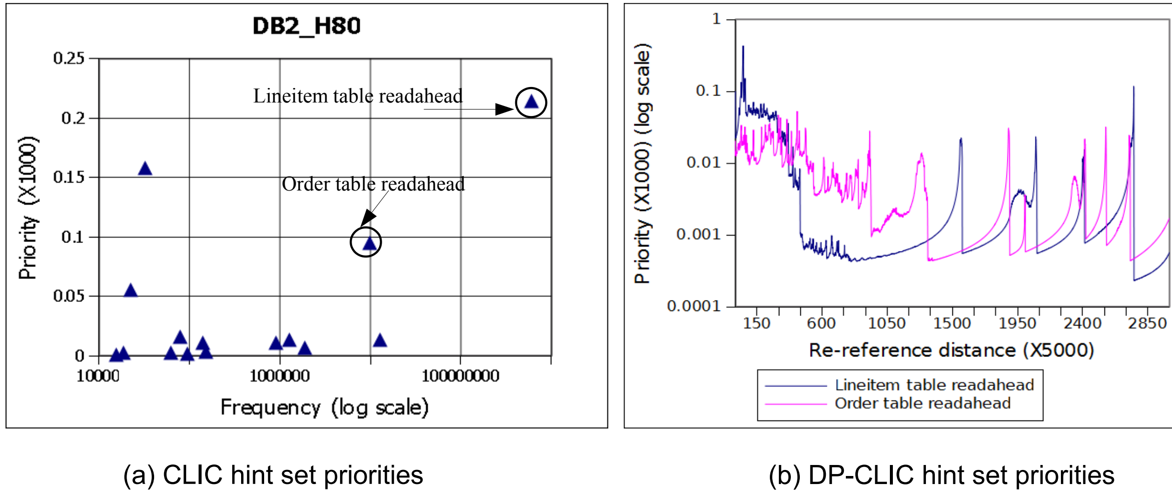


Figure 4.12: Read Hit Ratio of Caching Policies for the DB2 TPC-H Workloads

4.6.2 Evaluation with TPC-H traces

Figure 4.12 shows the results for DB2 TPC-H traces. For DB2_H400, DP-CLIC performs as well as CLIC on most cache sizes because the characteristics of their hint sets do not provide space to improve. For DB2_H80, DP-CLIC achieves the greatest advantage over CLIC. To further investigate the advantage of DP-CLIC in this case, Figure 4.13(a) shows hint set CLIC priority for the DB2_H80 trace, in which we indicate two hint sets. We see that among all hint sets, one hint set (hint "Lineitem table readahead") has the largest priority and its frequency is much larger than all other hint sets. The other hint set, hint "Order table readahead" is the second most frequently occurring hint set, but it has much lower frequency and lower caching priority. Thus, pages with other hint sets cannot compete cache space with pages with hint "Lineitem table readahead" in CLIC. The performance of CLIC mainly depends on the access pattern of this hint set. CLIC keeps pages with this high priority hint set in the cache until they are referenced, because they have the highest priority and cannot be replaced. Figure 4.13(b) shows the dynamic priority curve of hint "Lineitem table readahead", which reaches its peak quickly and drops quickly after



(a) CLIC hint set priorities

(b) DP-CLIC hint set priorities

Figure 4.13: CLIC priority vs. DP-CLIC priority of DB2_H80_400 trace

the peak. The priority curve has a long tail because about 30% reads occur after the end of the hill region. Figure 4.13(b) also shows the priority curve of hint "Order table readahead". We see that pages with hint "Order table readahead" can have higher caching priorities than pages with hint "Lineitem table readahead". Thus, with DP-CLIC, pages with other hint sets may be able to compete the cache with pages with hint "Lineitem table readahead". DP-CLIC outperforms CLIC by identifying the variation of caching priority and replacing pages when their priority is low.

4.6.3 Limiting the Histogram Size

In Section 4.3, we discussed the two parameters of DP-CLIC related to tracking hint histograms. In this section we show the effect of varying of the histogram parameters on the second-tier cache hit ratio. As in the other experiments in this chapter, space for tracking histograms has been deducted from the cache space.

The experiment results shown in the graphs of this section are normalized server cache hit ratios. Cache hit ratios in Figure 4.14 and Figure 4.17 are normalized with the cache hit ratio of the smallest width of histogram bucket in their figure. Cache hit ratios in Figure 4.15, Figure 4.16, and Figure 4.18 are normalized with the cache hit ratio of smallest number of buckets in their figure.

In the first group of experiments with TPC-C workloads, we test the effect of the width of the bucket W_{bucket} on the cache hit ratio. N_{bucket} is set to 3000, and we vary the width of bucket. Figure 4.14 shows the experiment results for the three TPC-C DB2 traces on different second tier cache sizes. The width of bucket varies from 5000 to 1000000. From

Figure 4.14 we see that: for different traces, DP-CLIC performance decreases at different values W_{bucket}. For trace 60-400, the read hit ratios drop when the width of the buckets is 50000. This is because the smallest hill portion of the higher-priority hint sets reach their peak point at about 50000 (Figure 4.8). When the width of each bucket is less than or equal to 50000, the histogram can still catch the rise and fall of the cache priority. When W_{bucket} is larger than 100000, the cache hit ratios start to decrease. Similarly, for trace 300-400 and trace 540-400, DP-CLIC performance decreases when W_{bucket} is 100000 and 1000000, respectively. From Figures 4.10 and 4.11 we see that the peak points of their higher-priority hint sets are close to 100000 and 1000000, respectively. Thus, DP-CLIC works well as long as W_{bucket} is smaller than the peak the hill portion and is able to capture rise and fall of important hint sets' priorities.

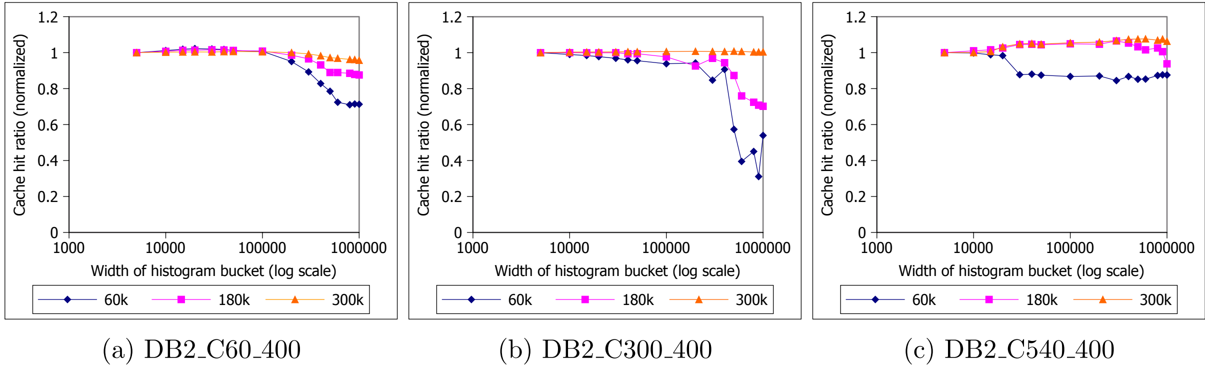
In the second group of experiments with TPC-C workloads, we test the effect of the number of buckets (N_{bucket}) on the cache hit ratio. First, we set W_{bucket} to 50000. Figure 4.15 shows the experiment results for this setting. For trace 60-400 and 300-400, the system reaches the highest hit ratio when N_{bucket} is 200. For trace 540-400, the system reaches the highest hit ratio when N_{bucket} is 500. When W_{bucket} is set to 10000, for all traces, the system needs a larger N_{bucket} to reach its highest hit ratio (Figure 4.16). This is because the overall reference distances need to cover all hill portions of important hint sets to achieve good performance.

We also did similar tests with TPC-H workloads. Figure 4.17 shows the normalized cache hit ratio as the function of W_{bucket}, and Figure 4.18 shows the normalized cache hit ratio as the function of N_{bucket}. From Figure 4.17 we see that the performance of DP-CLIC remains stable when W_{bucket} is less than or equal to 100000. According to the performance shown in Figure 4.17, we set W_{bucket} to 100000 to test the effect of N_{bucket} on the performance of DP-CLIC. From Figure 4.18 we see that we only need 10 buckets to achieve the best cache hit ratio for both traces. This is because W_{bucket} is large and 10 buckets are long enough to cover the hill portion of the important hint sets. Thus, for all traces, DP-CLIC only needs less than 1% of cache size to track dynamic priorities of hint sets.

4.6.4 Tracking Only Frequent Hint Sets for DP-CLIC

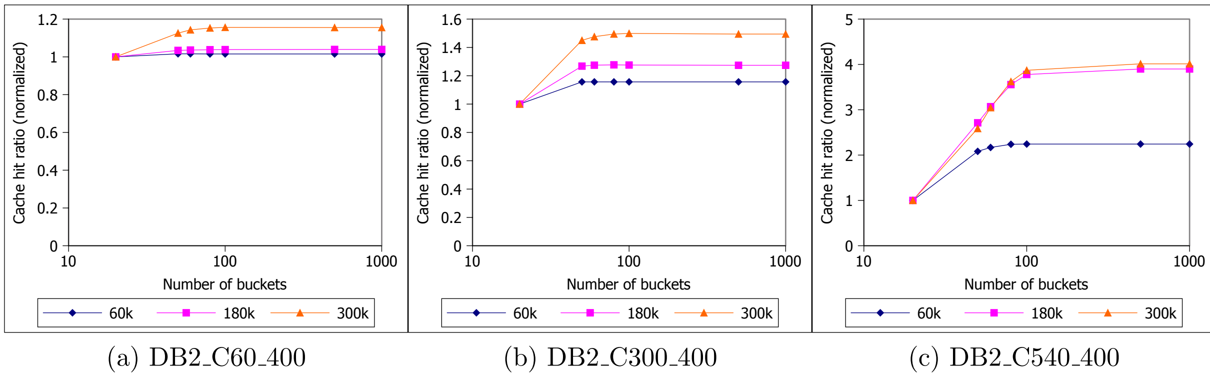
In Section 3.4.3, Figure 3.14 shows the experiment results with CLIC using top-k (described in Section 3.3.1) to reduce the number of hint sets to be tracked. For CLIC, tracking the 20 most frequent hint sets was sufficient. In this section, we study the effect of tracking only the most frequently occurring hint sets using the top-k algorithm with DP-CLIC. Similar to the experiments in Section 3.4.3, we vary k, the number of hint sets tracked by DP-CLIC, and measure the server cache hit ratio.

We did experiments with DB2 TPC-C and TPC-H workloads. Figure 4.19 shows the experiment results with DP-CLIC when using top-k to reduce the number of hint sets DP-CLIC needs to track. The left side graphs (a) (c) (e) show the results for TPC-C workloads.



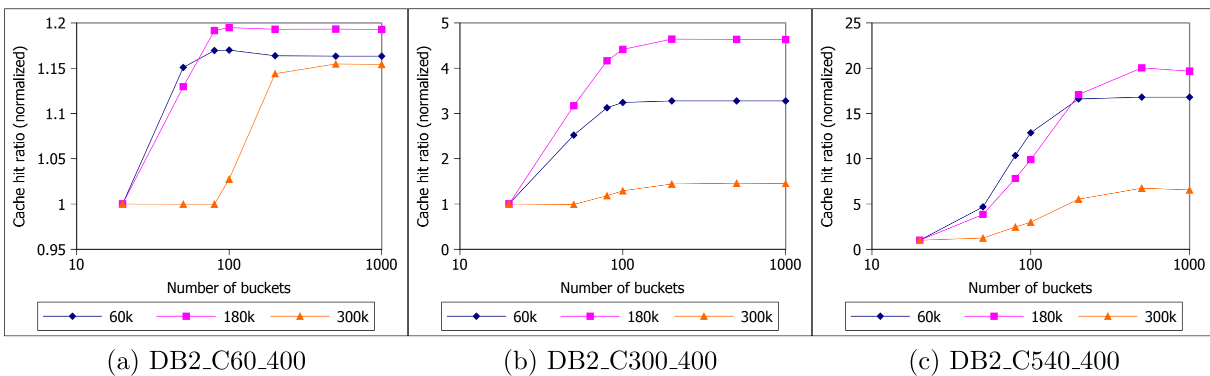
(a) DB2_C60_400 (b) DB2_C300_400 (c) DB2_C540_400

Figure 4.14: Effect of Width of Buckets on Read Hit Ratio (number of bucket = 3000)



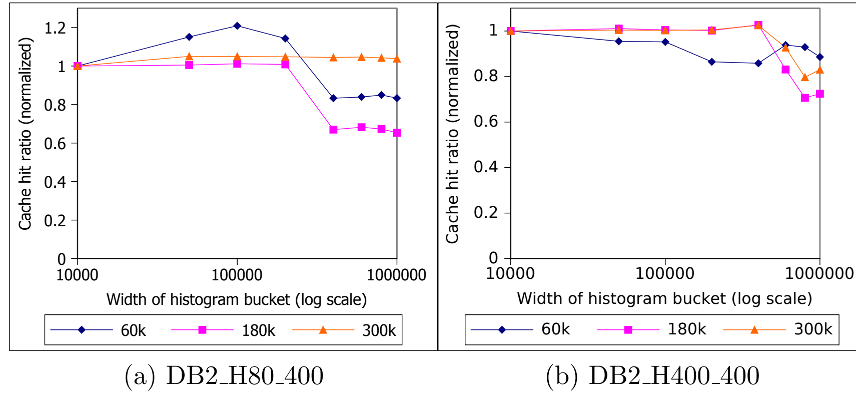
(a) DB2_C60_400 (b) DB2_C300_400 (c) DB2_C540_400

Figure 4.15: Effect of Numbers of Buckets on Read Hit Ratio (width of bucket = 50000)



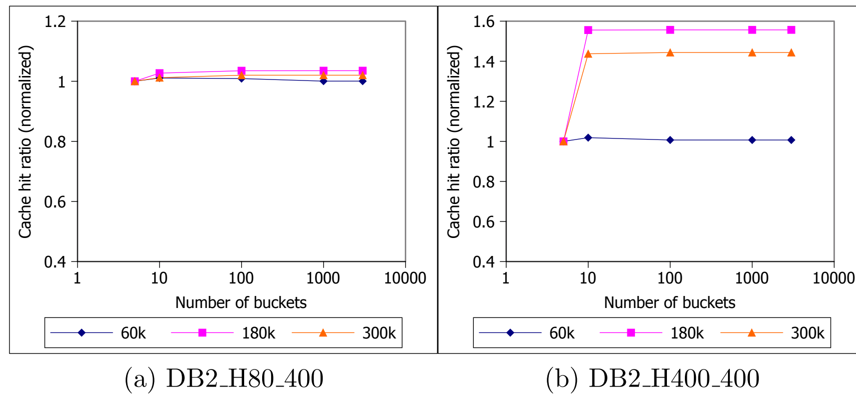
(a) DB2_C60_400 (b) DB2_C300_400 (c) DB2_C540_400

Figure 4.16: Effect of Numbers of Buckets on Read Hit Ratio (width of bucket = 10000)



(a) DB2_H80_400 (b) DB2_H400_400

Figure 4.17: Effect of Width of Buckets on Read Hit Ratio (number of bucket = 3000)



(a) DB2_H80_400 (b) DB2_H400_400

Figure 4.18: Effect of Number of Buckets on Read Hit Ratio (width of bucket = 100000)

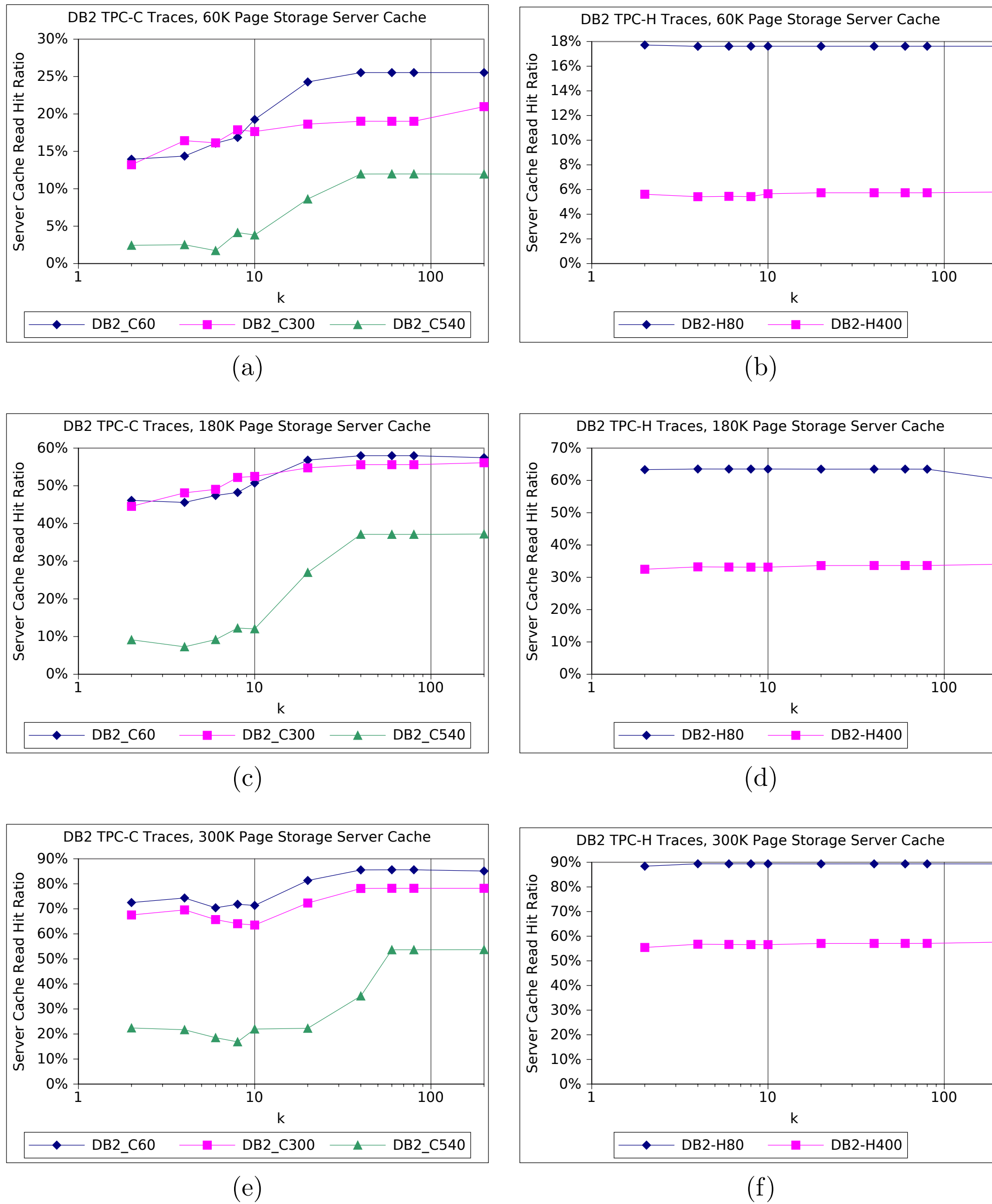


Figure 4.19: Effect of Top-K Hint Set Filtering on Read Hit Ratio (DP-CLIC)

We see that tracking the 50 most frequent hint sets (i.e., setting k = 50) was sufficient to achieve a read hit ratio close to what we could obtain by tracking all of the hints in the trace. The right side graphs (b) (d) (f) show the results for TPC-H workloads. We see that tracking k=2 most frequent hint sets is enough for TPC-H workload. Overall, we found that the top-k algorithm is also effective for DP-CLIC to achieve good performance while reducing the number of hint sets.

4.6.5 Increasing the Number of Hints for DP-CLIC

In Section 3.4.4, we considered a scenario in which CLIC was subjected to useless "noise" hints, in addition to the useful hints. By deliberately introducing a controllable level of useless hints in this experiment, we tested CLIC's ability to tolerate them without losing track of those hints that are useful. To test how DP-CLIC will perform with top-k as the number of distinct hint sets in the input trace is increased, we repeated the experiments similar to these in Section 3.4.4 with DP-CLIC. Similarly, we used our DB2 TPC-C traces and TPC-H traces, each of which contains 5 real hint types, and added T additional synthetic hint types. For the experiments in this section, we also chose $D = 10$, and we varied T, which controls the amount of "noise".

First, we evaluate the effectiveness of top-k approach when noise hint types are added to the DB2 TPC-C workload traces. Figure 4.20 shows the read hit ratios in a server cache of size 60K, 180K and 300K pages as a function of T. We fixed $k = 100$ and $k = 200$ for the top-k algorithm. As shown in Figure 3.15, similar to CLIC, DP-CLIC fares reasonably well for the DB2_C60_400 trace, suffering mild degradation in performance for $T \geq 2$. However, for the other two traces, DP-CLIC experienced more substantial degradation, particularly for $T \geq 2$. The cause of the degradation is that high-priority hint sets from the original trace get "diluted" by the additional noise hint types. Since CLIC has limited space for tracking hint sets, the dilution eventually overwhelms its ability to track and identify the useful hints. Even when k is increased from 100 (the left side graphs of Figure 4.20) to 200 (the right side graphs of Figure 4.20), the read hit ratios have not been improved.

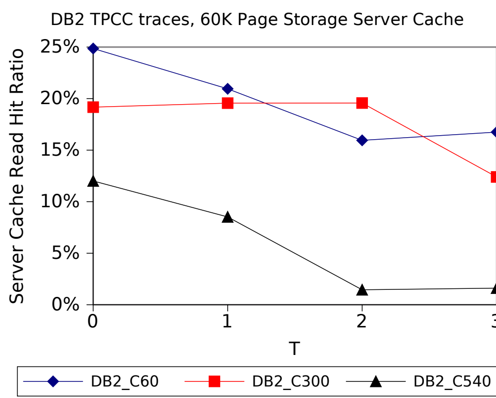
Secondly, we evaluate the effectiveness of top-k approach when noise hint types are added to the DB2 TPC-H workload traces. Figure 4.21 shows the read hit ratios in a server cache of size 60K, 180K and 300K pages as a function of T. Similarly, we fixed $k = 100$ and $k = 200$ for the top-k algorithm. For both DB2_H400_400 and DB2_H80_400 traces, DP-CLIC's performance is stable when $T \leq 2$. This is because for these original traces, DP-CLIC only needs to track $k = 2$ hint sets to obtain performance close to that obtained by tracking all hint sets (Shown in Figure 3.14). With $D = 10$ and $T = 2$, one useful original hint set is split into as many as $D^T = 100$ distinct hint sets. Even though useful hint sets have been diluted by noise hints, top-k can still identify and track them with $k = 100$ or $k = 200$. However, when more noise hints are added ($T = 3$), the performance starts to drop because k is not large to enough to collect all higher-priority hint sets. Thus, for DP-CLIC, top-k can not work efficiently when more noise hint types

are added. Controlling this trade-off of space versus accuracy is an interesting tuning problem for both CLIC and DP-CLIC.

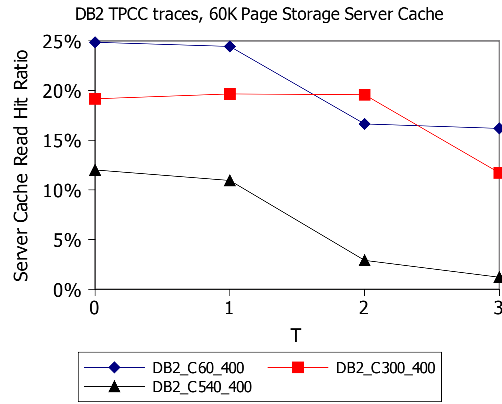
4.7 Conclusion

The main advantage of DP-CLIC is that it can identify the varying of page caching priority and evict pages when their caching priority is low. However, how much performance DP-CLIC can improve compared to CLIC depends on the read and write distributions of hint sets, which result in different priority curves. By analyzing the DB2 traces we found that the different read and write distributions are partially caused by the upper-tier cache and the workload. When the upper-tier cache is small (smaller than 50% of the database size), the temporal locality is not absorbed by the upper-tier cache and the majority of read re-references have short distances. When the upper-tier cache is large (larger than 50% of the database size), the upper-tier cache filters more temporal locality, and thus, the read re-reference distances in the lower-tier cache are much longer. DP-CLIC can detect the drop of priorities of each hint set by taking into account the variance of the read hit ratio (benefit) and the re-reference distances (cost), and can then evict pages when their priority is low. Thus, whether the upper-tier cache size is small or large, DP-CLIC can outperform CLIC.

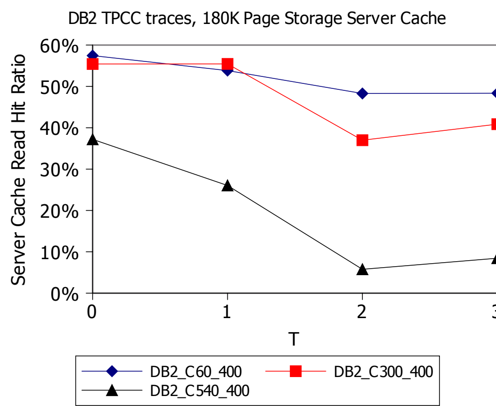
To capture the variance of the priority, DP-CLIC needs to collect the read and write histograms and calculate the dynamic priority for each hint set. The setting of the two parameters (N_{bucket} and W_{bucket}) of DP-CLIC decides the space and time overhead. The experimental results suggest that the histograms of hint sets do not need to have fine granularity. The experimental results also demonstrate that the space required for DP-CLIC to track and interpret hints is small.



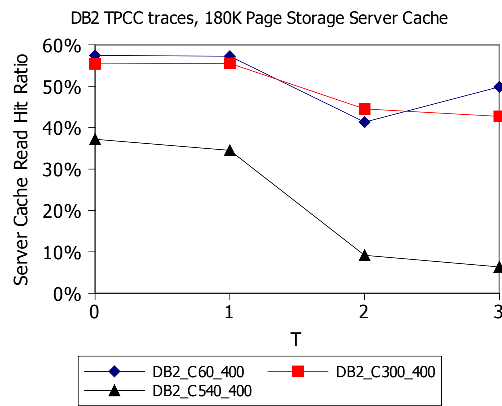
(a) k=100



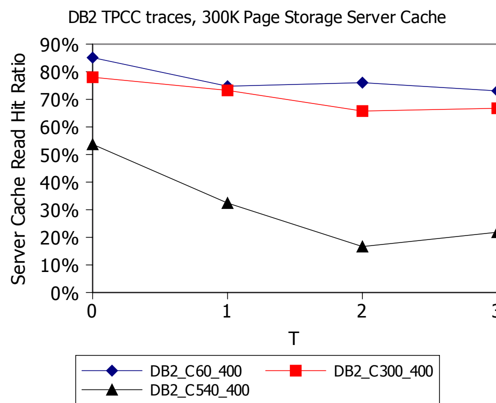
(b) k=200



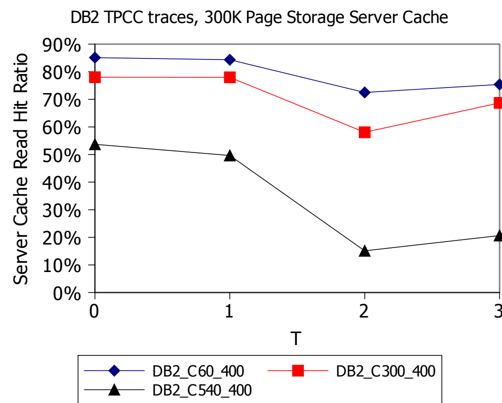
(c) k=100



(d) k=200

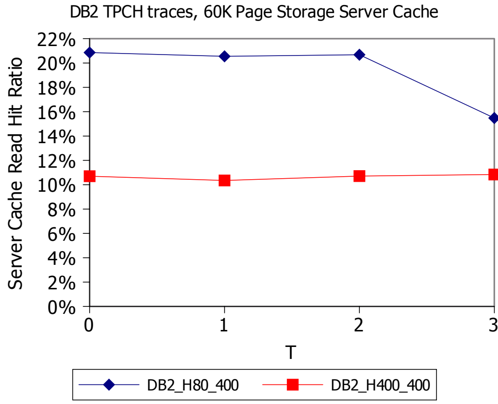


(e) k=100

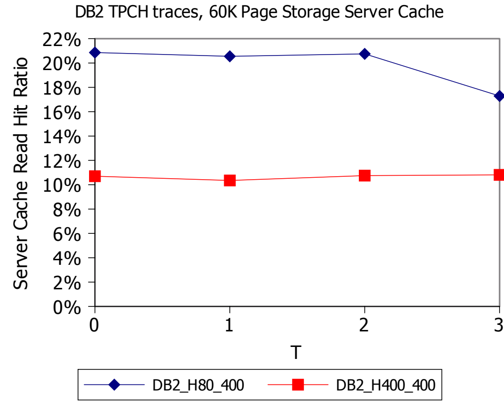


(f) k=200

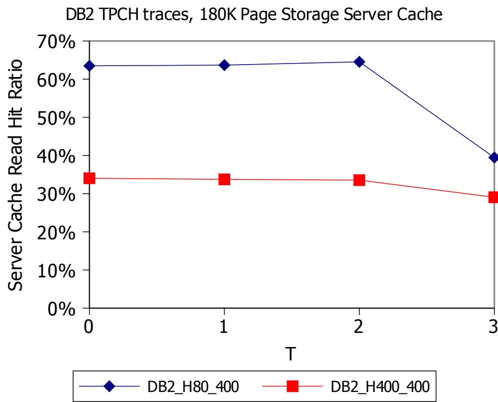
Figure 4.20: Effect of Top-K Hint Set Filtering on Read Hit Ratio with TPC-C Workload Traces



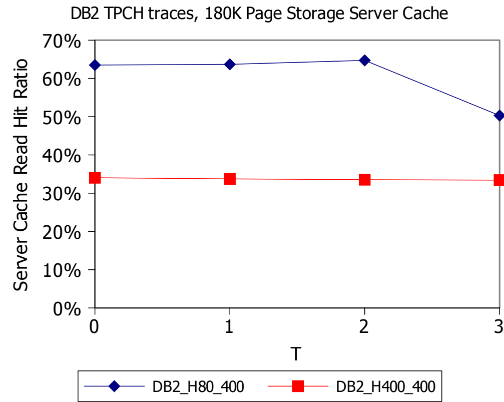
(a) k=100



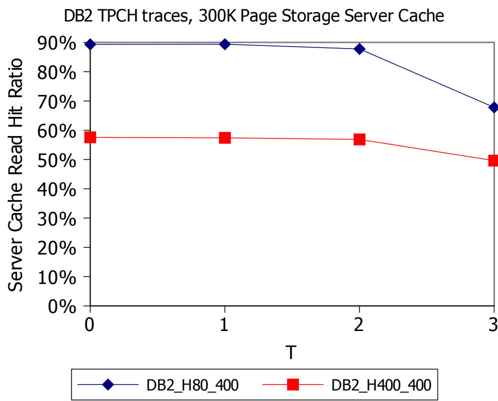
(b) k=200



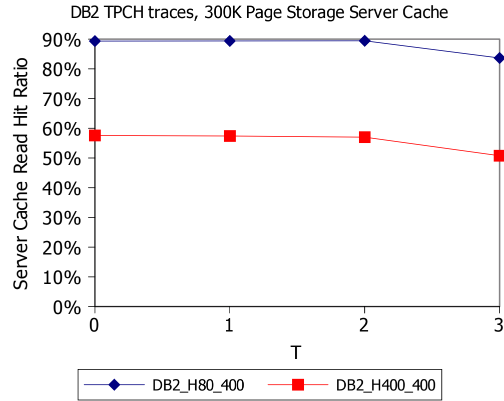
(c) k=100



(d) k=200



(e) k=100



(f) k=200

Figure 4.21: Effect of Top-K Hint Set Filtering on Read Hit Ratio with TPC-H Workload Traces

Chapter 5

Classification of Hint Sets

To reduce the number of hint sets that CLIC needs to consider, we introduced the top-k algorithm for tracking frequent hint sets in Section 3.3.1. This technique takes advantage of the observation that all of the hint sets in the traces exhibit frequency skew and tracks only the most frequently occurring hint sets. We studied the effect of the top-k algorithm on the performance of CLIC and DP-CLIC in Section 3.4.3 and Section 4.6.4. From the results of experiments with added noise hint types, we saw that the top-k algorithm may not work well in situations in which the storage clients provide too many low-value hints.

The top-k algorithm works well for our original traces because hint sets with high priorities have high frequency. One reason for this frequency skew is because of the hint types defined by the clients. Table 3.2 lists the five hint types in DB2 traces. Together, a pool ID, object ID and object type ID uniquely identify a database object. Pages in the same DB object and with the same I/O request have the same hint sets. Therefore, hint sets that identify a large DB object may occur more frequently than others. For example, hint set "STOCK table replacement writes" in DB2_60_400 occurs frequently because "STOCK table" is a large table. However, if other hint types are added to further identify pages with different priorities, large DB objects may be divided into smaller groups. Thus, the hint sets with high caching priorities may not have high frequency. Frequency-based technique may not be able to identify important hint sets.

CLIC is not limited to only accept hint sets provided by the DBMS buffer pool. As a generic approach, CLIC is designed to make replacement decisions for any lower-tier cache by analyzing hints from different caches of different upper-tiers. For example, in a data center as shown in Figure 5.1, a storage server may provide services to different DBMS servers or file servers through a storage area network (SAN). Through a network, file servers and database servers process requests to tables and files from application servers, such as mail servers, web servers. We may have hint types such as "server identifier" and "application server identifier". For each server, a hint value is assigned. These hint types can be useful or useless depending on what workloads are running on the servers. For example, if all application servers issue TPC-C workload transactions to a DBMS server,

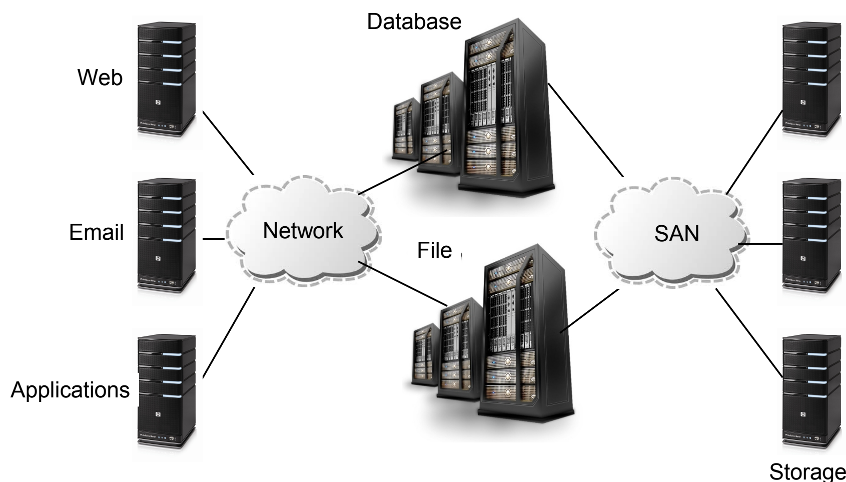


Figure 5.1: Architecture of Data Center

the "application server identifier" hint type maybe useless because the access patterns of the application servers may be very similar. If one useless hint type is added and it has ten hint values, each original hint set could be separated into ten distinct hint sets. If two such useless hint types are added, CLIC needs to track one hundred hint sets to track the real most important hint set. The number of hint sets may grow exponentially as the number of hint types grows. Moreover, the statistics collected for these hint sets will be diluted by useless hint types and may not be accurate enough for predicting the priority of hint sets. Thus, the frequency selection technique may not work efficiently in such scenarios.

To restrict the number of hint sets that CLIC tracks, an alternative to focusing on frequently-occurring hint sets is focusing on important hint types. For example, there are five hint types in DB2 TPC-C traces, and each hint type has several hint values. According to the value domain cardinality listed in Figure 3.2, theoretically, CLIC may need to trace 5040 hint sets in the worst case for a DB2 TPC-C workload trace. If we can identify that only "request type" and "object ID" are important among the five hint types, CLIC may need to trace only 105 hint sets in the worst case. The time and space required to track 100 hint sets will be much less than that required to track 5040 hint sets. For DP-CLIC, even though tracking read and write histograms required less than 1K pages for 100 hint sets, for 5040 hint sets, the whole cache could potentially be used for tracking statistics.

In this chapter, we introduce a new approach – classify hint sets using a feature selection technique. Unlike top-k, which filters infrequently occurring hint sets, the feature selection algorithm filters unimportant hint types. In machine learning and statistics, feature selection has been widely studied to provide more cost-effective predictors and to improve the

prediction performance of the predictor [30, 60] . The central assumption when using a feature selection technique is that the data contains redundant or irrelevant features. Redundant features are those which provide no more information than the currently selected features, and irrelevant features provide no useful information in any context. For example, the noise hint types in Section 3.4.4 are irrelevant features because they are randomly added and do not provide any useful information about the page requests. Thus, using the feature selection technique should be able to filter these noise hint types.

We propose hybrid algorithms that combine the feature selection technique with top-k. The hybrid algorithms utilize the limited space for tracking hint sets efficiently, and improve the read hit ratio compared what can be obtained by using frequency selection alone. We evaluate the hybrid algorithm with both CLIC and DP-CLIC.

5.1 Hybrid Algorithms

Since the noise hint types cause degradation of CLIC performance when the top-k algorithm is used, we propose a hybrid algorithm that combines the feature selection technique with the top-k algorithm. The hybrid algorithm has two steps: in the first step, it tracks k hint sets and identifies important hint types using the feature selection technique. In the second step, it only considers the hint types selected in the first step, and tracks reference statistics of k hint sets using the top-k algorithm. Therefore, we can limit the space for tracking reference statistics of hint sets.

As introduced in Section 3.3.1, the request stream is divided into non-overlapping windows, with each window consisting of W requests. CLIC restarts the top-k algorithm from scratch for every window of W requests. In the end of each window, CLIC calculates priorities for all tracked hint sets. As the hybrid algorithm has two steps, it needs $2W$ requests to track hint sets and build the benefit/cost model. Thus, CLIC restarts the hybrid algorithm from scratch for every $2W$ requests.

Step 1: The purpose of the first step is to filter irrelevant hint types by ranking the hint types according to their impurity (which will be introduced in Section 5.2). In the first W requests, we randomly choose k distinct hint sets about which to collect reference statistics as a learning sample. After W requests, we calculate the impurity for all hint types, and identify the most important hint types. The number of hint types t selected in the first step is a parameter of the hybrid algorithm. Table 5.2 shows the parameters of the hybrid algorithm.

Step 2: The purpose of the second step is to identify k frequently-occurring hint sets using the top-k algorithm. The hint sets tracked in this step are different from those in the first step. As important hint types have been identified in the first step, hint sets are grouped together by ignoring hint types which are not important. For example, if

Symbol	Description
k	The number of distinct hint sets tracked in step1 and step2
W	The number of requests for step1 and step2
t	The number of hint types selected in step1

Figure 5.2: Parameters of the Hybrid Algorithm

we identify that "noise1" as a useless hint type, we ignore the hint values of "noise1" when tracking hint sets. At the end of the second set of W requests, we calculate the caching priority for the tracked k hint sets. For DP-CLIC, read and write histograms are collected for the k frequently-occurring hint sets. At the end of the second set of W requests, DP-CLIC builds the DP priority model for each tracked hint sets.

5.2 Impurity of Hint Types

The purpose of using feature selection for CLIC is to reduce the number of hint types, and thus, to reduce the overall number of distinct hint sets CLIC needs to track. In our context, each hint set consists of a set of hint values, and each hint value is from the domain of one hint type. As we discussed in Chapter 3, CLIC associates a caching priority with each hint set. Assume that we have calculated the priorities for all hint sets, how can we decide which hint types are more important than others? In this section, we introduce Gini impurity [69] which is used for the classification and regression tree algorithm (CART) [7], for ranking the importance of hint types.

Gini impurity is a measure of homogeneity of each subset of a learning sample used by CART. CART is a classification method which uses historical data to construct *decision trees*. Decision trees split the learning sample into smaller and smaller subsets. CART algorithm will search for all possible variables and all possible values in order to find the best split that splits the data into subsets with maximum homogeneity. Gini impurity can be computed by summing the probability of each item being chosen times the probability of a mistake in categorizing that item. Gini impurity works well for ignoring noisy data [66].

With Gini impurity, we can measure the homogeneity of priority of a partition of hint sets when a learning sample is partitioned by hint types. For example, in the original DB2 traces, there are five hint types. Each hint set has five hint values which are from each of the five hint types' domains. If only one hint type is chosen, hint sets are to be grouped by the hint values of that hint type, and thus, hint sets are grouped into a larger partition. All hint sets in the large partition are assigned the same priority, which is the priority of the hint value associated with that partition. If all hint sets in the large partition have similar original priorities, the variance of their priorities is small. Thus, the original hint set priorities are close to the priority of whole partition.

To measure the impurity of hint types, we define *squared prioritization error* of the partition. The squared prioritization error of a partition is taken to be the frequency-weighted variance of the priorities of the training hint sets in that partition, i.e., of the hint sets that map to that partition. For example, if we consider partitioning hint sets using the "request type" hint type, each hint value in "request type" will result in one partition. Thus, all hint sets are to be categorized into four partitions, i.e. "sync read", "replacement write", "recovery write", and "sync write", depending on the value of "request type" in the hint sets. To calculate the variance of the priority of hint sets in one partition, we first need to define the priority of the whole partition. Suppose that a partition P groups together several hint sets H_i, and a caching priority $\texttt{Pr}(P)$ of the partition is calculated with $N_r(P)$, $N(P)$, and $D(P)$ of hint sets in this group.

$$\texttt{Pr}(P) = \frac{N_r(P)}{N(P)D(P)} \tag{5.1}$$

In Equation 5.1, $N(P)$, $N_r(P)$ and $D(P)$ are defined as the following:

$N(P)$: the total number of requests of partition P. As $N(H_i)$ represents the total number of requests with hint set H_i, $N(P)$ is calculated by summing up all $N(H_i)$ of hint sets belonging to this partition.

$$N(P) = \sum_{H_i \in P} N(H_i) \tag{5.2}$$

$N_r(P)$: the total number requests of partition P that result in a read re-reference (rather than a write re-reference or no re-reference). $N(P)$ is calculated by summing up all $N_r(H_i)$ of hint sets belonging to this partition.

$$N_r(P) = \sum_{H_i \in P} N_r(H_i) \tag{5.3}$$

$D(P)$: the average read re-reference distance for requests of partition P. The following equation first calculates the total read re-reference distance by summing up the overall read re-reference distances of all hint sets belonging to partition P, and then divides it by the total number of read re-reference of partition P.

$$D(P) = \frac{\sum_{H_i \in P} D(H_i) N_r(H_i)}{N(P)} \tag{5.4}$$

Note that the actual priorities $\texttt{Pr}(H_i)$ and frequencies $N(H_i)$ of the hint sets in the partition are observed from the I/O request stream as introduced in Section 3.2.2.

The squared prioritization error introduced by this partition is

$$\text{error}(P) = \frac{1}{N(P)} \sum_{H_i \in P} N(H_i) \left(\texttt{Pr}(P) - \texttt{Pr}(H_i)\right)^2 \tag{5.5}$$

Each hint value of the hint type may result in one partition P_i, and thus, each partition introduces a prioritization error. Lets error(P_i) represent the prioritization error of P_i. For example, if hint sets are partitioned by hint type "request type", it will result in four partitions, and thus, four squared prioritization errors. We define the impurity of the hint type as the total frequency-weighted prioritization error introduced by all partitions of this hint type(H_T):

$$N(H_T) = \sum_{P_i \in H_T} N(P_i) \tag{5.6}$$

$$I(H_T) = \frac{1}{N(H_T)} \sum_{P_i \in H_T} N(P_i)\mathrm{error}(P_i) \tag{5.7}$$

In our context, we use the impurity of hint type to identify the most important hint types for predicting the caching priority of hint sets. A small $I(H_T)$ of hint type H_T means that partitioning with this hint type leads to partitions that keep hint sets with similar priorities together. Thus, by ranking the impurity $I(H_T)$ of all hint types, we can identify good hint types for predicting the caching priority as hint types with the smallest $I(H_T)$.

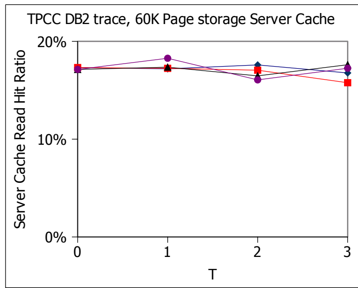
In the first step of the hybrid algorithm, we collect $N(H)$, $N_r(H)$ and $D(H)$ of k hint sets. At the end of the first step, we calculate $I(H_T)$ of each hint types with the statistics, and then choose t hint types with the smallest $I(H_T)$ as the useful hint types.

5.3 Experimental Evaluation

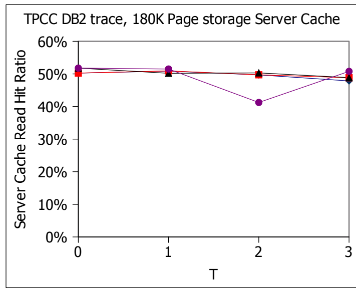
In Section 3.4.4 and Section 4.6.5, we tested CLIC's and DP-CLIC's ability to tolerate noise hint types without losing track of those hints that are useful. The results showed that top-k could not identify useful hint sets for both CLIC and DP-CLIC when more noise hint types were added. In this section, we study the effect of using the hybrid algorithm to identify useful hint sets by filtering noise hint types first. We implemented the hybrid algorithm in the storage server cache simulator for both CLIC and DP-CLIC. We repeated experiments similar to these in Section 3.4.4 and Section 4.6.5. In this experiment, we used our DB2 TPC-C traces and TPC-H traces, each of which contains 5 real hint types, and added T additional synthetic hint types. The goal of this experiment is to evaluate whether the hybrid algorithm can outperform top-k algorithm especially when more noise hint types are added, e.g. $T \geq 2$.

We set $W = 10^6$ requests, and a monitoring of $k = 100$ and $k = 200$ hint sets. In the first step of the hybrid algorithm, we set the number of the hint types selected t to be three, i.e., we only choose three most important hint types in the first step.

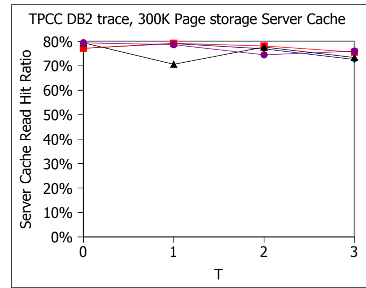
First, we evaluate the effectiveness of the hybrid algorithm when noise hint types are added to the DB2 TPC-C workload traces. Figure 5.3 shows the read hit ratios of both the hybrid algorithm and top-k as T increases for DB2 TPC-C workload traces. For each



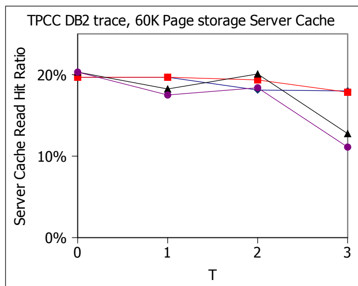
(a) DB2_C60_400



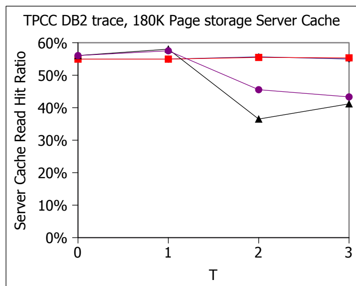
(b) DB2_C60_400



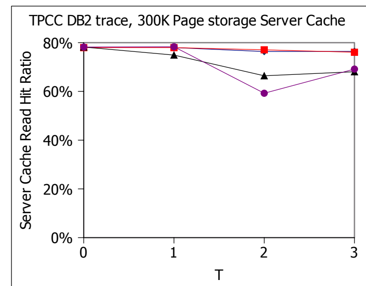
(c) DB2_C60_400



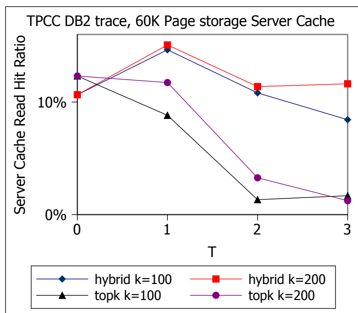
(d) DB2_300_400



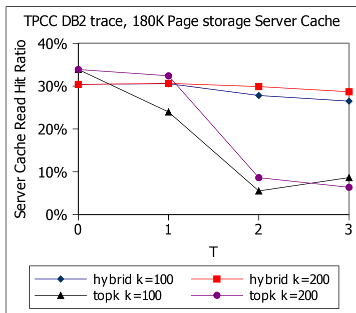
(e) DB2_300_400



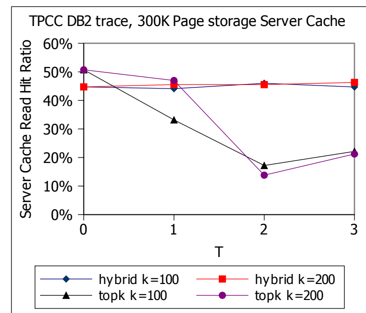
(f) DB2_300_400



(g) DB2_540_400

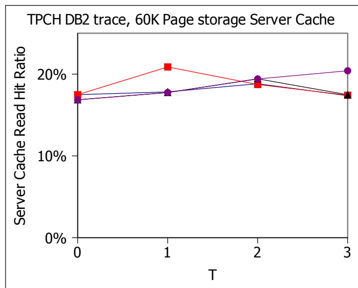


(h) DB2_540_400

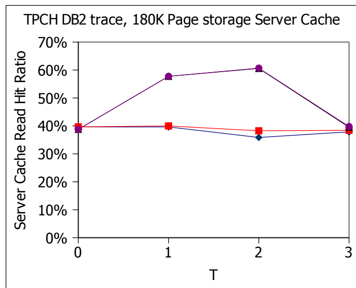


(i) DB2_540_400

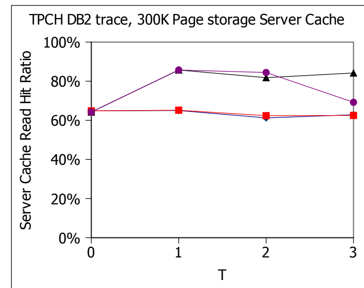
Figure 5.3: CLIC Read Hit Ratio



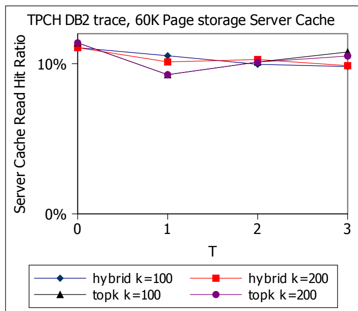
(a) DB2_H80_400



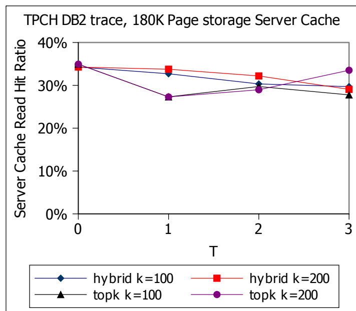
(b) DB2_H80_400



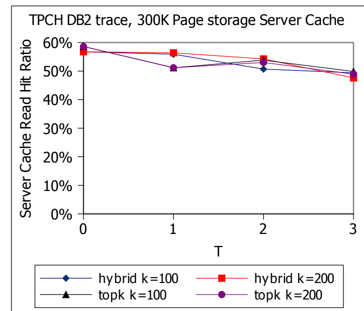
(c) DB2_H80_400



(d) DB2_H400_400

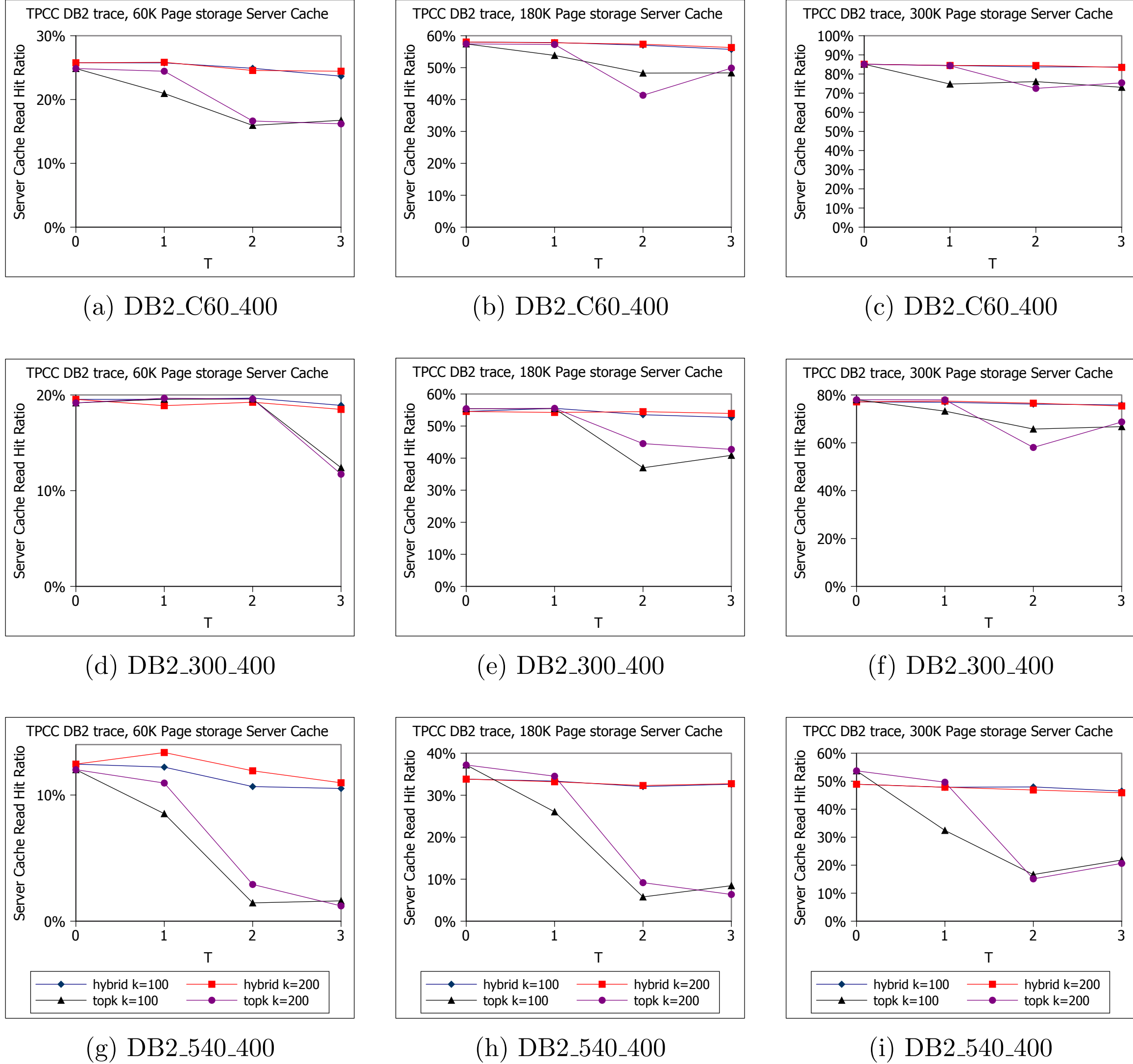


(e) DB2_H400_400



(f) DB2_H400_400

Figure 5.4: CLIC Read Hit Ratio



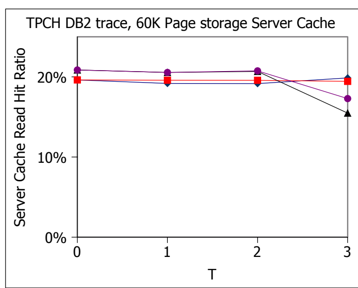
(a) DB2_C60_400 (b) DB2_C60_400 (c) DB2_C60_400

(d) DB2_300_400 (e) DB2_300_400 (f) DB2_300_400

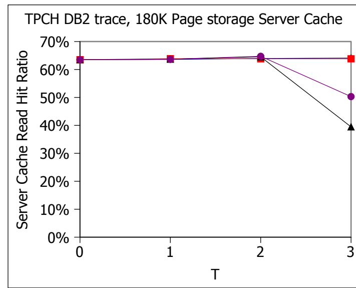
(g) DB2_540_400 (h) DB2_540_400 (i) DB2_540_400

Figure 5.5: DP-CLIC Read Hit Ratio

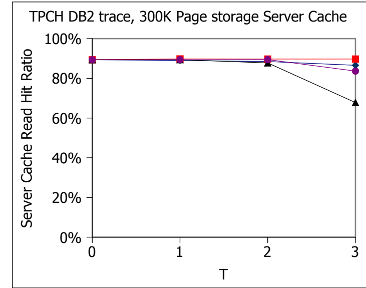
trace, we present three graphs to show the read hit ratios in a server cache of size 60K, 180K and 300K, respectively. As more "noise dimensions" are added to the hints, the read hit ratios of CLIC using the hybrid algorithm remained stable. Meanwhile, the read hit ratios of CLIC using top-k dropped. We analyzed the hint types selected by the first step in these experiment. In most cases, "request type", "object type ID" and "object ID" are identified as the three most important hint types. With the hybrid algorithm, CLIC had similar performance with $k = 100$ and $k = 200$. Thus, tracking $k = 100$ hint sets is sufficient for CLIC using the hybrid algorithm. We obtain the similar results for DB2 TPC-H workload traces as shown in Figure 5.4. Thus, CLIC's performance is more stable with the hybrid algorithm than with top-k when noise hints are added.



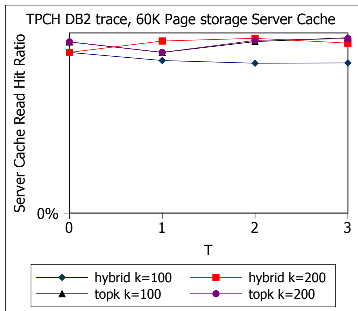
(a) DB2_H80_400



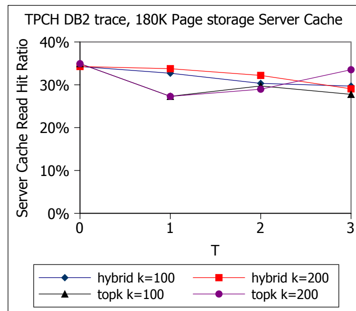
(b) DB2_H80_400



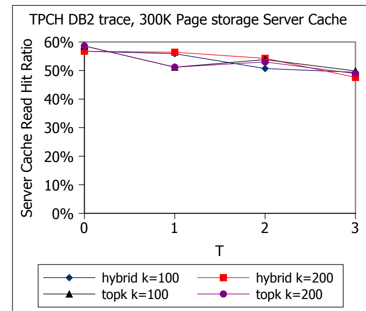
(c) DB2_H80_400



(d) DB2_H400_400



(e) DB2_H400_400



(f) DB2_H400_400

Figure 5.6: DP-CLIC Read Hit Ratio

The hybrid algorithm has the same effect for filtering noise hint types for DP-CLIC. Figure 5.5 to Figure 5.6 show the experiment results with DP-CLIC on DB2 TPC-C and TPC-H traces with noise hint types added. The setting of experiments is the same as the setting for evaluating CLIC with noise hints. We see the similar effect of the hybrid algorithm on DP-CLIC as that on CLIC. As T increases, the cache hit ratios of DP-CLIC did not degrade. Thus, from these experiment we conclude that the hybrid algorithm is more efficient than top-k for reducing space overhead when dealing with large numbers of hint types for both CLIC and DP-CLIC.

5.4 Conclusion

We have presented the top-k algorithm to reduce the number of hint sets that CLIC needs to track in Chapter 3. We found that for the original DB2 traces, it was possible to significantly reduce the number of hints that CLIC had to track with only minor degradation in performance. This is because the original traces do not have many irrelevant hint types. When more irrelevant hint types are added, CLIC and DP-CLIC need a large k to achieve good performance. In this chapter, we present an algorithm which filters irrelevant hint types first using a feature selection algorithm. The experimental results show that the hybrid algorithm can efficiently filter the irrelevant hint types, and thus, significantly reduce the number of hint sets CLIC had to track. With the hybrid algorithm, CLIC and DP-CLIC only need to track a small number of hint sets to achieve stable performance when more irrelevant hint types are in the traces.

Chapter 6

Hybrid Storage Management for Database Systems

Flash memories, which are semiconductor chips, have many attractive features, such as low power consumption, light weight, small size, shock resistance, and lack of moving parts. They have been used for many years in portable consumer devices (e.g, cameras, phones) because these features are particularly desirable for these devices. Recently, as the price of flash memories has dropped dramatically, flash-based solid state storage devices (SSDs) are now also becoming commonplace in server environments. Compared to HDDs, SSDs exhibit much better performance for random reads because they have no mechanical latency. Although SSDs are more expensive per bit than traditional hard disks (HDD), they are much cheaper in terms of cost per I/O operation. Thus, HDDs are cost effective for bulky, infrequently accessed data, while SSDs are well-suited to data that are relatively hot [27]. Servers in data centers may be configured with both types of persistent storage.

There are two approaches to use the SSD in the hybrid storage system. First, the SSD can be used as a part of the permanent storage sitting side by side with the HDD. When using the SSD as a permanent storage, it can partially replace the hard disk and save some cost on the storage device. The system should be able to identify hot data to place on the SSD before the workload starts. For example, Ozmen et al. [57] presented a database layout optimizer to generate layouts for heterogeneous storage configurations. After the data has been placed in the SSD, it will not be moved in and out from the SSD to avoid destroying the database layout. Secondly, the SSD can be used as a cache sitting between the main memory cache and the HDD. When using the SSD as a cache, it can always identify hot data dynamically as the workload is running. Thus, hot data can be placed in the SSD and also can be replaced from the SSD when they are cold. Comparing the two approach, the advantage of using SSD as a permanent storage is that it can save cost on the storage device. However, as the capacity of the SSD is small and the cost of the hard disk is much cheaper, the cost saving is limited. The advantage of using the SSD as a cache is that it can use the SSD more efficiently as placement and replacement decision

can be made dynamically. Thus, we consider using the SSD as a cache is a better approach than using the SSD as a part of permanent storage.

In this chapter, we are concerned with the use of such hybrid (SSD and HDD) storage systems for database management, in which the SSD is using as a cache. We consider hybrid storage systems in which the two types of devices are visible to the DBMS, so that it can use the information at its disposal to decide how to make use of the two types of devices. This is illustrated in Figure 6.1. When data is written to storage or evicted from the buffer pool, the DBMS chooses which type of device to write it to. The incentive for adding SSD to the storage hierarchy is to speed up I/O by storing frequently accessed data in SSD.

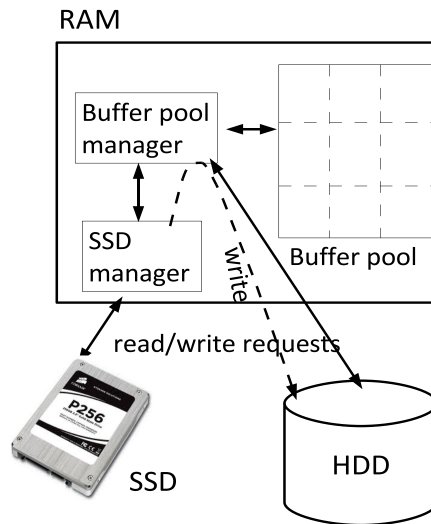


Figure 6.1: System Architecture (the arrows represent read/write requests)

Previous work has considered how to place data in a hybrid storage system for DBMS [38, 10, 11, 57, 20]. However, adding SSDs to the storage hierarchy not only raises the question about how to manage the SSDs, but also raises the question about whether the current DBMS buffer pool replacement algorithms still work effectively in a hybrid system. Thus, there are two problems for the design of algorithms for DBMS-managed hybrid storage systems:

- In a hybrid storage system, the SSD has faster random read and write than the HDD does. To make page replacement decisions for the buffer pool, the DBMS buffer pool replacement algorithm needs to be aware that replacing an HDD page may cause a higher access latency than would replacing an SSD page.

- If the DBMS makes buffer pool replacement decisions by considering where pages are located (SSD or HDD), page I/O access patterns may change when page locations change. When a page is cached in the SSD, it may be evicted from the buffer pool more quickly than when it is not cached in the SSD. As accesses to the SSD cache are misses from the buffer pool, a page may get more accesses when it is cached in the SSD.

To consider these two related questions, we take a broader view of the problem than has been taken in previous work. Our view includes the DBMS buffer pool as well as the two types of storage devices. First, we determine which data should be retained in the DBMS buffer pool. Currently, the buffer pool employs LRU or LRU-like algorithms to decide which data is to be replaced. One characteristic of these algorithms is that they are cost-oblivious: they do not consider the different retrieval costs of different storage devices and treat all data the same. However, blocks evicted from the buffer cache to an SSD are much faster to retrieve later than blocks evicted to the HDD, and thus buffer pool replacement decisions should be affected by the presence of hybrid storage. We consider *cost-aware buffer management*, which can take this distinction into account. Second, assuming that the SSD is not large enough to hold the entire database, we have the problems of deciding which data should be placed on the SSD. Because the SSD provides fast random I/Os, data that is accessed frequently (hot pages) from the storage device should be placed in the SSD. Thus, the responsibility of the SSD caching algorithm is to identify the hot pages. How to identify hot pages should depend on the physical access pattern for the data, which depends, in turn, on the DBMS workload and on the management of the DBMS buffer pool.

Because we consider both buffer pool management and management of the hybrid storage system, we have more scope for optimization than previous work in this area, at the expense of additional invasiveness in the design and implementation of the DBMS. In addition, we must account for the fact that the two problems we consider are mutually dependent. Replacement decisions in the buffer pool depend on the locations (SSD or HDD) of the pages being replaced, since placement affects both eviction cost and reloading costs. Conversely, page SSD placement decisions depend on how the page is used, e.g., how frequently it is read into or written from the buffer pool, which depends, in turn, on the buffer manager. For example, under a cost aware replacement policy, SSD pages are better eviction candidates than HDD pages are, because retrieving SSD pages has a much lower access cost than retrieving HDD pages. Consequently, moving a page from the HDD to the SSD may result in a significant increase in the physical read and write rates for that page.

In our work, we address these dependencies using an *anticipatory* approach to SSD management. When deciding whether to move a page into the SSD, our proposed admission and replacement policy (called CAC) predicts how such a move will affect the physical I/O load experienced by that page. The page is moved into the SSD only if it is determined

to be a good candidate under this predicted workload. The DBMS buffer manager then makes cost-aware replacement decisions based on the current placements of buffered pages.

The remainder of this chapter is organized as follows. Section 6.1 gives an overview of the system architecture that we assume. Section 6.2 presents a cost aware technique, called GD2L, for database buffer pool management, and Section 6.3 shows some empirical results that illustrate the effect of GD2L on the physical access patterns of database pages. Section 6.4 presents the CAC algorithm for managing the contents of the SSD device(s). The results of our evaluation GD2L and CAC are presented in Section 6.5.

6.1 System Overview

Figure 6.1 illustrates the system architecture we have assumed for this work. The DBMS manages two types of storage devices, SSDs and HDDs. All database pages are stored on the HDD, where they are laid out according to the DBMS's secondary storage layout policies. In addition, copies of some pages are located in the SSD and copies of some pages are located in the DBMS buffer pool. Any given page may have copies in the SSD, in the buffer pool, or both, although we expect the latter to be uncommon.

Figure 6.2 shows the pseudo code of the management of the buffer pool and the SSD. When the DBMS needs to read a page, the buffer pool is consulted first. If the page is cached in the buffer pool, the DBMS reads the cached copy. If the page is not in buffer pool but it is in the SSD, it is read into the buffer pool from the SSD. The SSD manager is responsible for tracking which pages are currently located in the SSD. If the page is in neither the buffer pool nor the SSD, it is read from the HDD.

If the buffer pool is full when a new page is read in, the buffer manager must evict a page according to its page replacement policy, which we present in Section 6.2. When the buffer manager evicts a page, the evicted page is considered for admission to the SSD if it is not already located there. SSD admission decisions are made by the SSD manager according to its *SSD admission policy*. If admitted, the evicted page is written to the SSD. If the SSD is full, the SSD manager must also choose a page to be evicted from the SSD to make room for the newly admitted page. SSD eviction decisions are made according to an *SSD replacement policy*. (The SSD admission and replacement policies are presented in Section 6.4.) If a page evicted from the SSD is more recent than the version of that page on the HDD, then the SSD manager must copy the page from the SSD to the HDD before evicting it, otherwise the most recent persistent version of the page will be lost. The SSD manager does this by reading the evicted page from the SSD into a staging buffer in memory, and then writing it to the HDD.

We assume that the DBMS buffer manager implements asynchronous page cleaning, which is widely used to hide write latencies from DBMS applications. When the buffer manager elects to clean a dirty page, that page is written to the SSD if the page is already

```
The current buffer pool request is for page p:
1     if p is not in the buffer pool
2        if p is on the SSD
3           read p from the SSD
4        else
5           read p from the HDD

On eviction of page p from the buffer pool:

6     if p is admitted to be placed in the SSD
7        write p to the SSD

On flushing dirty page p from the buffer pool:

8     if p is on the SSD
9        write p to the SSD
10    else if p is admitted to be placed in the SSD
11       write p to the SSD
12    else
13       write p to the HDD

On eviction of page p from the SSD:

14    if p is dirty (the version is newer than the one on the HDD)
15       write p to the HDD
```

Figure 6.2: The Management of the Buffer Pool and the SSD

located there. If the dirty page is not already located on the SSD, it is considered for admission to the SSD according to the SSD admission policy, in exactly the same way that a buffer pool eviction is considered. The dirty page will be flushed to the SSD if it is admitted there, otherwise it will be flushed to the HDD.

The buffer and SSD management techniques that we have described have two key properties. First, admission of pages into the SSD occurs only when pages are evicted or cleaned from the DBMS buffer pool. Most caching policies for managing multi-tier caches are access-based. They consider caching pages in all tiers when pages are read from storage devices. Unfortunately, access-based caching policies may cause duplication of pages in the buffer cache and the SSD, i.e., *cache inclusion* [70]. To minimize cache inclusion, pages are *not* admitted into the SSD when they are loaded into the buffer pool from the HDD. Second, each flush of a dirty page from the DBMS buffer pool goes either to the SSD or to the HDD, but not to both (at least not immediately). One advantage of this approach, compared to a write-through design, is that the SSD can potentially improve DBMS write performance, to the extent that writes are directed to the SSD. A disadvantage of this

approach is that the latest version of an unbuffered page might, in general, be found on either device. However, because the DBMS always writes a dirty buffer pool page to the SSD if that page is already on the SSD, it can be sure that the *SSD version (if any) of a page is always at least as recent as the HDD version.* Thus, to ensure that it can obtain the most recently written version of any page, it is sufficient for the DBMS to know which pages have copies on the SSD, and to read a page from the SSD if there is a copy of the page there. To support this, the SSD manager maintains an in-memory hash map that records which pages are on the SSD. To ensure that it can determine the contents of the SSD even after a failure, the SSD manager uses a checkpointing technique (described in Section 6.4.5) to efficiently retain its map so that it can be recovered quickly.

6.2 Buffer Pool Management

In this section, we describe our replacement algorithm for the buffer pool: a two-level GreedyDual (GD2L) algorithm. GD2L is a restricted version of the GreedyDual algorithm [74] that we have adapted for use in a DBMS.

Most existing cost-aware algorithms, e.g., the balance algorithm [49] and GreedyDual [74], were proposed for file caching. They take into account cached object size and access cost when making replacement decisions, and target different cases of cached objects: uniform object size with arbitrary retrieval cost, arbitrary object size with uniform retrieval cost, or arbitrary object size with arbitrary retrieval cost. In particular, the GreedyDual algorithm addresses the case in which the cached objects have uniform size, but incur different retrieval costs. Young [74] shows that GreedyDual has the same (optimal) competitive ratio as LRU and FIFO [49], $\frac{k}{k-h+1}$, where k denotes the size of the cache, and h denotes the size of the smallest cache for which OPT achieves the same cost it achieved by using a cache of size k ($k \geq h$).

GreedyDual is a range of algorithms which generalize well-known caching algorithms, such as LRU and FIFO. Initially, we present the GreedyDual generalization of LRU. In Section 6.2.1, we describe how a similar approach can be applied to the LRU variant used by InnoDB, and we also discuss how to extend it to handle writes.

GreedyDual associates a non-negative cost H with each cached page p. When a page is brought into the cache or referenced in the cache, H is set to the cost of retrieving the page into the cache. To make room for a new page, the page with the lowest H in the cache, H_{min}, is evicted and the H values of all remaining pages are reduced by H_{min}. By reducing the H values and resetting them upon access, GreedyDual ages pages that have not been accessed for a long time. The algorithm thus integrates locality and cost concerns in a seamless fashion.

GreedyDual is usually implemented using a priority queue of cached pages, prioritized based on their H value. With a priority queue, handling a hit and an eviction each

Symbol	Description
R_D	The read service time of HDD
W_D	The write service time of HDD
R_S	The read service time of SSD
W_S	The write service time of SSD

Figure 6.3: Storage Device Parameters

require $O(\log k)$ time, where k is the number of pages in the priority queue. Another computational cost of GreedyDual is the cost of reducing the H values of the remaining pages when evicting a page. To reduce the value H for all pages in the cache, GreedyDual requires k subtractions. Cao et al. [12] have proposed a technique to avoid the subtraction cost. Their idea is to keep an "inflation" value L and to offset all future settings of H by L.

These parameters of read/write service time are summarized in Figure 6.3. In our case, there are only two possible initial values for H: one corresponding to the cost of retrieving an SSD page and the other to the cost of retrieving an HDD page. We designed the GD2L algorithm for this special case. As shown in Figure 6.4, GD2L uses two queues to maintain pages in buffer pool: one queue (Q_S) is for pages with copies on the SSD, the other (Q_D) is for pages without copies on the SSD. Both queues are managed using LRU. With the technique proposed by Cao et al. [12], GD2L achieves $O(1)$ time for handling both hits and evictions.

Figure 6.4 describes the GD2L algorithm. When GD2L evicts the page with the smallest H from the buffer pool, L (the inflation value) is set to the H value. If the newly requested page is on the SSD, it is inserted to the MRU end of Q_S and its H value is set to $L + R_S$; otherwise, it is inserted to the MRU end of Q_D and its H value is set to $L + R_D$. Because the L value increases gradually as pages with higher H are evicted, pages in Q_D and Q_S are sorted by H value. The one having the smallest H value is in the LRU end. By comparing the H values of the two LRU pages of Q_D and Q_S, GD2L easily identifies the victim page that has the smallest H value in the buffer pool. The algorithm evicts the page with the lowest H value if the newly requested page is not in the buffer pool. In Figure 6.4, page q represents the page with the lowest H value ($H(q)$).

6.2.1 Implementation of GD2L on MySQL

We implemented GD2L for buffer pool management in InnoDB (the default storage engine of the MySQL database system). InnoDB maintains a buffer pool for caching database pages and has a control block for each cached page. InnoDB manages the buffer pool as several lists: One is the LRU list, which records all pages cached in the buffer pool; another is the free list, which keeps buffers ready to be allocated. InnoDB also maintains a hash table of all cached pages to facilitate fast lookup. InnoDB uses a variant of the

```
1≻    if p is not cached
2≻       let ps be the LRU page of Q_S
3≻       let pd be the LRU page of Q_D
4≻       if(H(ps) > H(pd))
5≻          q = pd
6≻       else
7≻          q = ps
8≻       set L = H(q)
9≻       evict q from the buffer pool
10≻      bring p into the buffer pool
11≻   if p is in the SSD
12≻      H(p) = L + R_S
13≻      put p to the MRU of Q_S
14≻   else if p is not in SSD
15≻      H(p) = L + R_D
16≻      put p to the MRU of Q_D
```

Figure 6.4: The GD2L Algorithm
This pseudo-code shows how GD2L handles a request for page p. L is initialized as 0

least recently used (LRU) algorithm. When room is needed to add a new page to buffer pool, InnoDB evicts the LRU page from the LRU list. Pages that are fetched on demand are placed at the MRU end of the LRU list. However, prefetched pages are placed near the midpoint (at the 3/8 point) of the LRU list, moving to the MRU position only if they are subsequently demanded. Since prefetching is used during table scans, this provides a measure of scan resistance.

To implement GD2L, we split InnoDB's LRU list into two LRU lists: Q_D and Q_S. As shown in Figure 6.5, the cached HDD pages are stored in Q_D and the cached SSD pages in Q_S. Newly loaded pages are placed either at the MRU end of the appropriate list or at the midpoint, depending on whether they were prefetched or loaded on demand. When a new page is inserted at the midpoint of Q_D or Q_S, its H value is set to the H value of the current midpoint page. When a page is moved to the MRU end of its list, its H value is set to $L + R_S$ if it is in Q_S, or $L + R_D$ if it is in Q_D, as shown in Figure 6.4.

When pages are modified in the buffer pool, they need to be copied back to the underlying storage device. In InnoDB, dirty pages are generally not written to the underlying storage device immediately after they are modified in the buffer pool. Instead, *page cleaner* threads are responsible for asynchronously writing back dirty pages. The page cleaners can issue two types of writes to the dirty pages: *replacement writes* and *recoverability writes*. As discussed in Chapter 3, the DBMS issues different types of writes for different purposes. Replacement writes are issued when dirty pages are identified as eviction candidates. To remove the latency associated with synchronous writes, the page cleaners try to ensure that pages that are likely to be replaced are clean at the time of the replacement. In contrast,

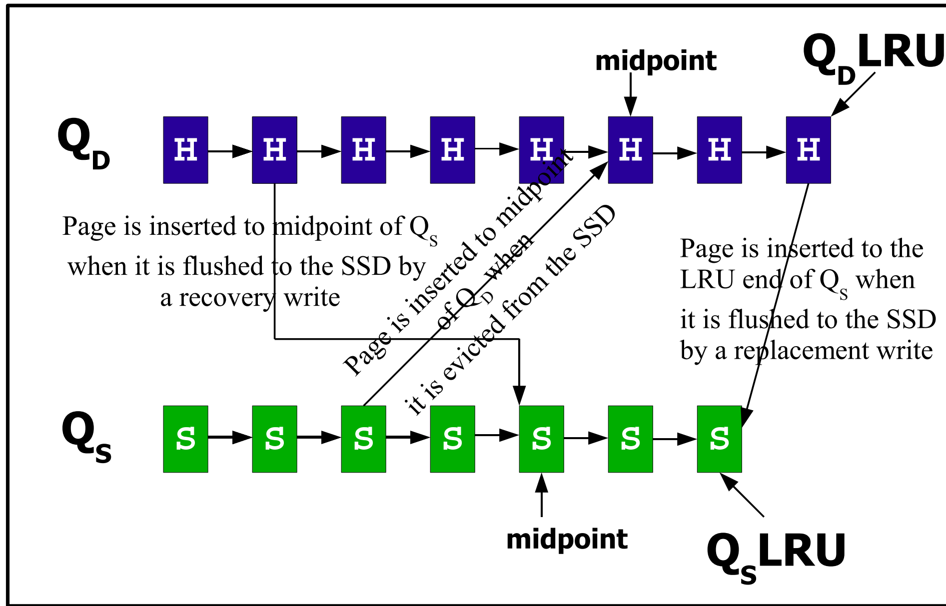


Figure 6.5: Buffer Pool Managed by GD2L on MySQL

recoverability writes are those that are used to limit failure recovery time. The DBMS uses write ahead logging to ensure that committed database updates survive failures. The failure recovery time depends on the age of the oldest changes in the buffer pool. The page cleaners issue recoverability writes for the least recently modified pages to ensure that a configurable recovery time threshold will not be exceeded.

Two problems occur in GD2L due to dirty page flushing:

- Which pages should the page cleaners identify as eviction candidates for issuing replacement writes?
- When pages are evicted or flushed from the buffer pool, they are considered for SSD admission. If HDD pages are written to the SSD, their page access cost changes. As GD2L manages HDD pages and SSD pages are in two different LRU lists that are sorted by H values, where should the page be inserted into Q_S?

In InnoDB, when the free space of the buffer pool is below a threshold, page cleaners start to check a range of pages from the tail of the LRU list. If there are dirty pages in the range, the page cleaners flush them to the storage devices. These are replacement writes. We changed the page cleaners to reflect the new cost-aware replacement policy. Since pages with lower H values are likely to be replaced sooner, the page cleaners consider H values when choosing which pages to flush. As GD2L maintains two LRU lists in the buffer pool (Q_D and Q_S), the page cleaners check pages from tails of both lists. If there are dirty pages in both lists, the page cleaners compare their H values and choose dirty pages with

lower H values to write back to the storage devices. We did not change the way the page cleaners issue recoverability writes, since those writes depend on page update time and not on page access cost.

The original GreedyDual algorithm assumed that a page's retrieval cost did not change. However, in our system a page's retrieval cost will change when it is moved into or out of the SSD. If a buffered page is moved into the SSD, then GD2L must take that page out of Q_D and place it into Q_S. This situation can occur when a dirty, buffered page than is not on the SSD is flushed, and the SSD manager elects to place the page into the SSD. If the page flush is a replacement write, it means that the page being flushed is a likely eviction candidate. In that case, GD2L removes the page from Q_D and inserts it at the LRU end of Q_S. To keep Q_S to be sorted by H value, the page's H value is set to be the H value of the previous LRU page of Q_S. If the page flush is a recoverability write, then the flushed page should not be inserted to the LRU end of Q_S because it is not an eviction candidate. As Q_S is sorted by page H value, we could find the correct position for the page in Q_S by looking through pages in Q_S and comparing H values. Instead, for simplicity, GD2L simply inserts the page at the midpoint of Q_S and assigns it the same H value as the previous Q_s midpoint page. Since recoverability writes are typically much less common than replacement writes, and since this situation only occurs when a page is moving into the SSD, this situation is relatively rare. Hence, we chose the simple approach for GD2L.

It is also possible that a buffered page that is in the SSD will be evicted from the SSD (while remaining in the buffer pool). This may occur to make room in the SSD for some other page. In this case, GD2L removes the page from Q_S and inserts it to Q_D. Since this situation is also uncommon, GD2L simply inserts the page at the midpoint of Q_D, as it does for recoverability writes.

6.3 The Impact of Cost-aware Caching

Cost-aware caching algorithms, e.g., GD2L, take into account page location when making replacement decisions. As a result, the page read rate and write rate might be different after the page location changes. In this section, we address the following questions: if the buffer pool uses the GD2L caching algorithm, how does the page read rate change when a page is placed in the SSD? GD2L also changes the mechanism for asynchronous cleaning of dirty pages. How does this impact on the page write rate?

To study the impact of the GD2L algorithm on the page access pattern, we drove the modified InnoDB with a TPC-C workload, using a scale factor of 10. We implemented GD2L in the InnoDB storage engine of MySQL database system, The initial size of the database was approximately 1GB. For managing the SSD, we used the policy that will be described in Section 6.4.

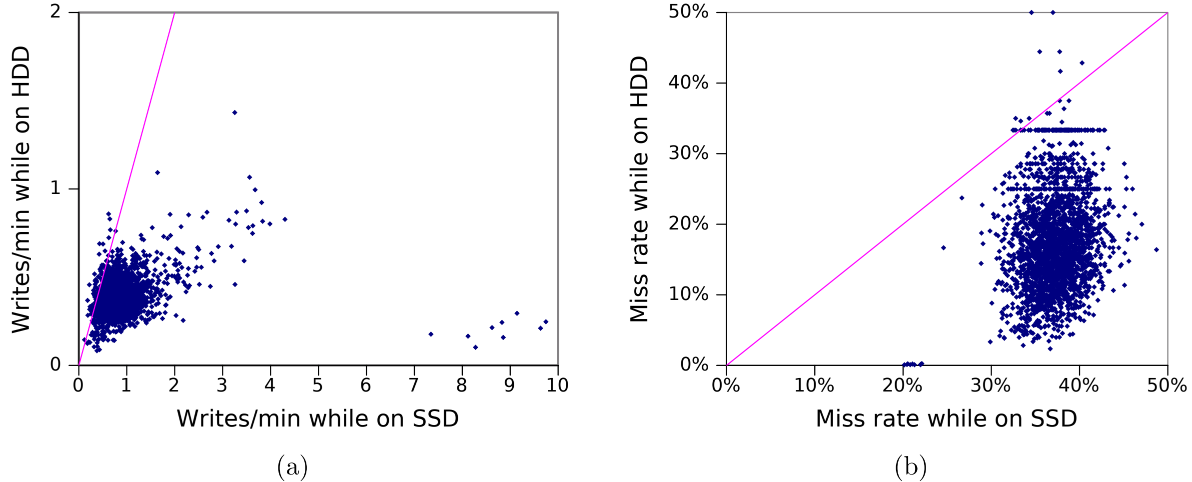


Figure 6.6: Miss Rate/Write Rate While on HDD vs. Miss Rate/Write rate while on SSD. Each point represents one page

In our experiments, we set the buffer pool size to 200M, the SSD size to 400M, and the running duration to sixty minutes. During the run, we monitored the amount of time each page spent in the SSD, and its read and write rates while on the SSD and while not on the SSD. We identified pages that had been cached in the SSD for at least twenty minutes and also had not been cached in the SSD for at least 20 minutes (about 2500 pages), and observed the buffer pool miss rate and write rate for these pages. A logical request on a page is realized as a physical request when the page is missed in the buffer pool. The page miss rate in the buffer pool is defined as the percentage of logical reads realized as physical reads of the page. Figure 6.6 shows the page miss rate in the buffer pool while the pages are on SSD vs. their miss rate while the pages are on HDD and the page write rate while the pages are on SSD vs. their write rate while the pages are on HDD. From the two graphs we see that most page miss rates and write rates are larger while the page is cached on SSD. This is as expected. Once pages are placed on SSD, they are more likely to be evicted from the buffer pool because they have lower retrieval costs. As SSD pages in the buffer pool are better eviction candidates, the *page cleaner* needs to flush dirty ones to the storage before they are evicted. As a result, page read and write rates go up while they are cached in SSD.

6.4 SSD Management

Section 6.1 provided a high-level overview of the management of the SSD device. In this section, we present more details about SSD management, including the page admission and replacement policies used for the SSD and the checkpoint-based mechanism for recovering SSD meta-data after a failure.

Pages are considered for SSD admission when they are cleaned or evicted from the DBMS buffer pool. Pages are always admitted to the SSD if there is free space available on the device. New free space is created on the SSD device as a result of *invalidations* of SSD pages. Pages in the SSD cache can be clean or dirty. A clean page in the SSD is a page whose version in the SSD is identical to the version on the HDD. A dirty page in the SSD is a page that has been updated in the SSD but not in the HDD. In another word, the version in the SSD is newer than the version in the HDD.

Consider a clean page p in the DBMS buffer pool. Suppose that there is also a copy of p in the SSD, and the SSD version of p is the same as the HDD version p (Since p is clean in the buffer pool, this also implies that all three copies of p are identical). If p is updated and hence made dirty in the buffer pool, the SSD manager invalidates the copy of p on the SSD if the SSD and HDD copies of p are identical. Invalidation frees the space that was occupied by p on the SSD. If the SSD version of p is newer than the HDD version, it cannot be invalidated without first copying the SSD version back to the HDD. Rather than pay this price, the SSD manager simply avoids invalidating p in this case.

If there is no free space on the SSD when a page cleaned or evicted from the DBMS buffer pool, the SSD manager must decide whether to place the page on the SSD and which SSD page to evict to make room for the newcomer. If a dirty page is chosen to be evicted from the SSD, it is written to the HDD first. The SSD manager makes these decisions by estimating the benefit, in terms of reduction in overall read and write cost, of placing a page on the SSD. It attempts to keep the SSD filled with the pages that it estimates will provide the highest benefit. Our specific approach is called Cost-Adjusted Caching (CAC). CAC is specifically designed to work together with a cost-aware DBMS buffer pool management, like the GD2L algorithm presented in Section 6.2. We present the specifics of CAC in Section 6.4.1.

6.4.1 CAC: Cost-Adjusted Caching

To decide whether to admit a page p to the SSD, CAC estimates the benefit $B(p)$, in terms of reduced access cost, that will be obtained if p is placed on the SSD. The essential idea is that CAC admits p to the SSD if there is some page p' already on the SSD cache for which $B(p') < B(p)$. To make room for p, it evicts the SSD page with the lowest estimated benefit.

Suppose that a p has experienced $r(p)$ physical read requests and $w(p)$ physical write requests over some measurement interval prior to the admission decision. If the physical I/O load on p in the past were a good predictor of the I/O load p would experience in the future, a reasonable way to estimate the benefit of admitting p to the SSD would be

$$B(p) = r(p)(R_D - R_S) + w(p)(W_D - W_S) \tag{6.1}$$

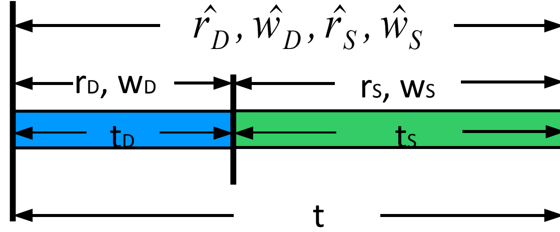


Figure 6.7: The Measured and Estimated Statistics of a Page
Note that t_S and t_D represent the total time that the page is on the SSD and not on the SSD.

where R_D,R_S,W_D, and W_S represent the costs of read and write operations on the HDD and the SSD (Figure 6.3).

Unfortunately, when the DBMS buffer manager is cost-aware, like GD2L, the read and write counts experienced by p in the past may be particularly poor predictors of its future physical I/O workload. This is because admitting p to the SSD, or evicting it from the SSD if it is already there, will change p's physical I/O workload. In particular, if p is admitted to the SSD then we expect that its post-admission physical read and write rates will be much higher than its pre-admission rates, as was illustrated by the experiments in Section 6.3. Conversely, if p is evicted from the SSD, we expect its physical I/O rates to drop. Thus, we do not expect Equation 6.1 to provide a good benefit estimate when the DBMS uses cost-aware buffer management.

To estimate the benefit of placing page p on the SSD, we would like to know what its physical read and write workload would be if it were on the SSD. Suppose that $\widehat{r_S}$ and $\widehat{w_S}$ are the physical read and write counts that p would experience if it were placed on the SSD, and $\widehat{r_D}$ and $\widehat{w_D}$ are the physical read and write counts p would experience if it were not placed on the SSD. (We will drop the references to specific pages in our notation when the page is clear from context.) Using these hypothetical physical read and write counts, we can write our desired estimate of the benefit of placing p on the SSD as follows

$$B = (\widehat{r_D} R_D - \widehat{r_S} R_S) + (\widehat{w_D} W_D - \widehat{w_S} W_S) \tag{6.2}$$

Thus, the problem of estimating benefit reduces to the problem of estimating values for $\widehat{r_D}$, $\widehat{r_S}$, $\widehat{w_D}$ and $\widehat{w_S}$.

To estimate $\widehat{r_S}$ for a page p, CAC uses *two* measured read counts: r_S and r_D. In general, p may spend some time in the SSD and some time not in the SSD. As the example shown in Figure 6.7, page p spent t_S time on the SSD, and t_D on the HDD only. r_S is count of the number of physical reads experienced by p during t_S (while p is on the SSD duration). r_D is the number of physical reads experienced by p during t_D (while it is not on the SSD). To estimate what p's physical read count would be if it were on the SSD full time ($\widehat{r_S}$), CAC uses

$$\widehat{r_S} = r_S + \alpha r_D \tag{6.3}$$

Symbol	Description
r_D, w_D	Measured physical read/write count while not on the SSD
r_S, w_S	Measured physical read/write count while on the SSD
$\widehat{r_D}$, $\widehat{w_D}$	Estimated physical read/write count if never on the SSD
$\widehat{r_S}$, $\widehat{w_S}$	Estimated physical read/write count if always on the SSD
m_S	Buffer cache miss rate for pages on the SSD
m_D	Buffer cache miss rate for pages not on the SSD
α	Miss rate expansion factor
t	The duration the workload has been running
t_D	The duration the page is not on the SSD
t_S	The duration the page is on the SSD

Figure 6.8: Summary of Notation

In this expression, the number of physical reads experienced by p while it was not on the SSD (r_D) is multiplied by a scaling factor α to account for the fact that it would have experienced more physical reads during that period if it had been on the SSD. Note that r_S is measured over a different time interval than $\widehat{r_S}$. We refer to the scaling factor α as the *miss rate expansion factor*, and we will discuss it further in Section 6.4.2. CAC estimates the values of $\widehat{r_D}$, $\widehat{w_D}$, and $\widehat{w_S}$ in a similar fashion:

$$\widehat{r_D} = r_D + \frac{r_S}{\alpha} \tag{6.4}$$

$$\widehat{w_S} = w_S + \alpha w_D \tag{6.5}$$

$$\widehat{w_D} = w_D + \frac{w_S}{\alpha} \tag{6.6}$$

The notation used in these calculations is summarized in Figure 6.8.

An alternative approach to estimating $\widehat{r_S}$ uses only the observed read count while the page is on the SSD (r_S), scaling it up to account for any time in which the page is not on the SSD. We assume that page read and write rates do not change if their location does not change, so that these counts have a linear relationship with the duration. $\widehat{r_S}$can be estimated as:

$$\widehat{r_S} = r_S \times \frac{t}{t_S} \tag{6.7}$$

$$\widehat{r_D} = r_D \times \frac{t}{t_D} \tag{6.8}$$

While this may be effective, it will work only if the page has actually spent time in the SSD, so that r_S can be observed. Nevertheless, a way to estimate r_S for pages that have not been observed in the SSD is required. In contrast, estimation using Equation 6.3 will work even if r_S or r_D are zero due to lack of observations.

6.4.2 The Miss Rate Expansion Factor

The purpose of the miss rate expansion factor (α) is to estimate how much a page's physical read and write rates will change if the page is admitted to the SSD. Admitting a page to the SSD does not affect that page's logical read and write rates. However, it will affect the page's physical read and write rates because a cost-aware DBMS buffer manager evicts SSD pages more aggressively than it evicts non-SSD pages. Thus, we want the miss rate expansion factor to capture how a page's miss rate in the DBMS buffer cache changes when the page is admitted to the SSD.

A simple way to estimate α is to compare the overall miss rates of pages on the SSD to that of pages that are not on the SSD. Suppose that m_S represents the overall miss rate of logical read requests for pages that are on the SSD, i.e., the total number of physical reads from the SSD divided by the total number of logical reads of pages on the SSD. Similarly, let m_D represent the overall miss rate of logical read requests for pages that are not located on the SSD. Both m_S and m_D are easily measured. Using m_S and m_D, we can define the miss rate expansion factor as:

$$\alpha = \frac{m_S}{m_D} \tag{6.9}$$

For example, $\alpha = 3$ means that the miss rate is three times higher for pages on the SSD than for pages that are not on the SSD.

While Equation 6.9 captures our intuition about increased miss rates for pages on the SSD, we have found that it is too coarse. In Equation 6.9, α is calculated using the buffer pool miss rates of all database pages. This relies on the assumption that all pages have the same expansion factor. However, since different tables may have different access patterns and the distribution of page requests is not uniform, this may not be true. As an example, Figure 6.9 illustrates miss rate expansion factors of pages grouped by table and logical read rates. The three lines represent pages holding the TPC-C STOCK, CUSTOMER, and ORDERLINE tables.

Since different pages may have substantially different miss rate expansion factors, we use different expansion factors for different groups of pages. Specifically, we group database pages based on the database object (e.g., table) for which they store data, and on their logical read rate, and we permit a different expansion factor for each group. We divide the range of possible logical read rates into subranges of equal size. We define a group as pages that store data for the same database object and whose logical read rates fall in the same subrange. For example, in our experiments, we defined the subrange width as one logical read per minute. If the maximum logical read rate of a table were 1000, this table might have 1000 groups. For each page group g, we define the miss rate expansion factor as in Equation 6.9:

$$\alpha(g) = \frac{m_S(g)}{m_D(g)} \tag{6.10}$$

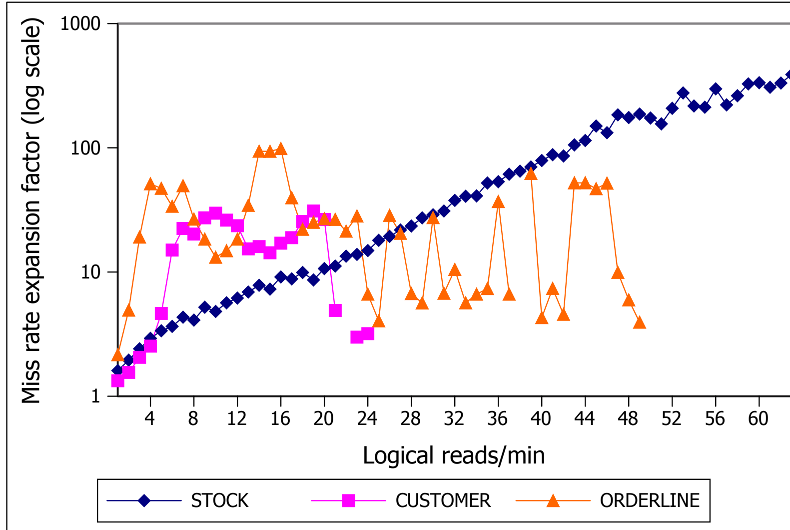


Figure 6.9: Miss Rate Expansion Factor for Pages from Three TPC-C tables.

where $m_S(g)$ is the overall miss rate for pages in g while they are in the SSD, and $m_D(g)$ is the overall miss rate for pages in g while they are not in the SSD.

Note that $m_D(g)$ and $m_S(g)$ may change over time for two reasons. First, the logical read count and the physical read count of pages in group g may changes. Second, pages may move from one group to another as their logical read rates fluctuate.

We track logical and physical read counts for each individual page, as well as miss rates for each group. Page read counts are updated with each logical or physical read request to the page. Group miss rates are updated lazily, when certain events occur, using the per-page statistics of the group's pages. Specifically, we update group miss rates when a page is evicted from the buffer pool, when a dirty page is flushed from the buffer pool to the SSD, and when a page is evicted from the SSD. Because pages are grouped based in part on their logical read rates, which can fluctuate, the group to which a page belongs may also change over time. If this occurs, we subtract the page's read counts from those of its old group and add them to the new group.

It is possible that $m_S(g)$ $m_D(g)$ will be undefined for some groups. For example, a group's $m_D(g)$ may be undefined because pages in the group have never been evicted from the buffer pool. We assume that $m_D(g) = 0$ for such groups. Similarly, $m_S(g)$ may be undefined because no pages in the group have been admitted to the SSD. We set $\alpha(g) = 1$ for such groups, which gives a better opportunity for them to be admitted to the SSD. Thus, we may have a chance to collect statistics for them.

A potential efficiency threat is the number of possible groups for which the system must maintain statistics. The number of possible groups depends on the size of the subrange. If we do not set the subrange size, there is only one group in each table. A smaller subrange size leads to more accurate $\alpha(g)$ at a cost of more space for collecting statistics. In our

evaluation (Section 6.5) we ignore this cost because the space requirement for tracking group statistics was less than 0.01% of the buffer pool size.

6.4.3 Sequential I/O

Hard disks have substantially better performance for sequential reads than for random reads. To account for this, CAC considers only random reads when estimating the benefit of placing a page in the SSD. In particular, the measured values r_S and r_D used in Equation 6.3 count only random reads. This requires that the SSD manager classify read requests as sequential or random. Two classification approaches have been proposed in recent work. Canim et al. [11] classify a page request as *sequential* if the page is within 64 pages of the preceding request. Do et al. [20] exploit the existing DBMS *read-ahead* (prefetch) mechanism: a page is marked as sequential if it is read from the disk via the read-ahead mechanism; otherwise, the page is marked as random. Do et al. [20] indicate that leveraging the read-ahead mechanism was much more effective. They observed that while the read-ahead mechanism was 82% accurate in identifying sequential reads, Canim's approach [11] was only 51% accurate. CAC adopts this approach for identifying sequential reads, using the read-ahead mechanism in the InnoDB buffer manager.

6.4.4 Implementation of SSD Management on MySQL

We present details of implementation of hybrid storage in InnoDB. The SSD cache is created as a file on the SSD when the DBMS is started. As is true of all files used by InnoDB, the file on the SSD is opened in unbuffered asynchronous I/O mode. Thus, data requested by InnoDB is not buffered in the file system cache. Blocks on the SSD have the same fixed size as blocks in the buffer pool. In MySQL, the block size in the buffer pool is 16K.

As shown in Figure 6.10, the SSD manager has the following main components:

Control block: For each block in the SSD, we maintain a control block in the memory The control block has attributes to record statistics, such as r_D, w_D, r_S, and w_S, which are necessary for calculating SSD page priority. To maintain group statistics for miss rate factors, the control block also tracks which group the page belongs to. To track t_s and t_d (the total time that the page is on the SSD and not on the SSD), we record the time when the page is placed in the SSD or evicted from the SSD. Thus, we can update t_d or t_s by deducting the recorded time from the current time when a page is moved to or evicted from the SSD.

Hash table: We maintain a hash table in the buffer pool for all pages on the SSD to facilitate fast lookup. When a new page is placed in the SSD, a hash key is created

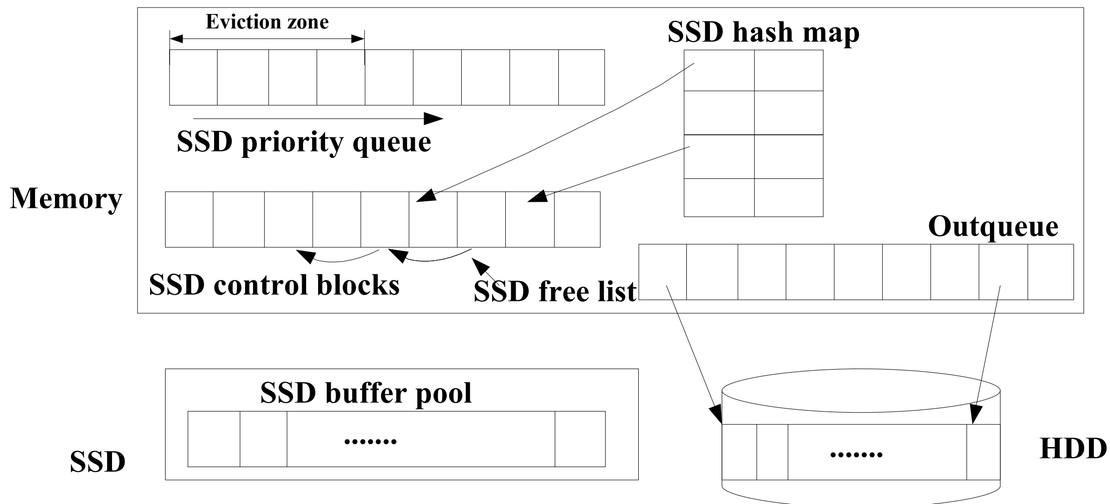


Figure 6.10: The Data Structures Used by the SSD Manager

using the page space ID and offset and inserted into the hash table. When a page is evicted from the SSD, it is also removed from the hash table.

Priority queue: A priority queue is used to organize pages placed on the SSD. The page with the smallest priority is on the top of the priority queue. The page priority is updated when the page is read from or written to the SSD. With the priority queue, the system is able to quickly identify an SSD replacement victim.

Free list: The SSD free list is for tracking SSD pages that are ready to be allocated. Initially, all SSD pages are in the free list. The SSD manager checks the free list first when a page is considered to be placed on the SSD. When SSD pages are invalidated, they are inserted in the free list.

Outqueue: We maintain an outqueue for recording statistics for a fixed number (N_{outq}) of uncached pages (neither in the buffer pool nor in the SSD). When a page is evicted from the SSD, an entry for the page is insert into the outqueue. An entry is also placed in the outqueue for pages that are evicted from the buffer pool and not placed in the SSD cache. Each entry in the outqueue records only the page statistics. When the outqueue is full, the least-recently inserted entry is evicted to make a room for a new entry.

6.4.5 Failure Handling

Since data present in the SSD may be more recent than that in the HDD, the system needs to ensure that it can identify the pages in the SSD after a system failure. TAC [11] does not have such a problem because it writes dirty pages to both the SSD and the HDD. Lazy cleaning, as proposed by Do et al. [20], handles this issue by flushing all dirty pages in the SSD to the HDD when taking a checkpoint. Neither of these approaches exploits the persistence of the SSD. In contrast, CAC assumes that the contents of the SSD will survive a failure, and it will read the latest version of a page from the SSD after a failure if the page was located there. The challenge with this approach is that the SSD manager's in-memory hash map indicating which pages are in the SSD is lost during a failure. Debnath et al. [17] address this problem by checkpointing the hash map and logging all writes to the SSD. During recovery, the hash map can be rebuilt based on the last written hash map and the log.

CAC's approach is also based on checkpointing, but it does not require logging of changes to the hash map. As each page header includes a page identifier, the hash map can be rebuilt without causing any runtime overhead by scanning all pages in the SSD during the failure recovery process. However, this may substantially increase recovery time. For example, based on the read service time of our SSD, to scan a 32G SSD requires about three minutes. Larger SSDs would introduce proportionally larger delays during recovery. To achieve faster recovery, CAC checkpoints the hash map periodically and also identifies a group of k low priority pages as an *eviction zone* on the SSD. Until the next checkpoint, CAC will evict only pages that fall into the eviction zone. After a failure, CAC initializes its hash map using the most recently checkpointed copy, and then checks the k SSD slots in the eviction zone to identify what is actually there. The eviction zone size (k) controls a trade-off between operational overhead and recovery time. CAC will checkpoint its hash map when all of its eviction candidates have been evicted from the SSD. Thus, smaller values of k result in more frequent hash map checkpoints, but faster recovery.

6.5 Evaluation

In this section, we present an experimental evaluation of GD2L and CAC. Our first objective is to provide some insight into the behavior of GD2L combined with CAC. Specifically, we wish to address two questions.

- First, how effective is GD2L relative to non-cost-aware buffer management?
- Second, when GD2L is used to manage the buffer pool, how important is it to use an anticipatory SSD manager, like CAC, that recognizes that page access patterns change when the page is moved between the SSD and the HDD?

Our second objective is to compare the performance of our proposed algorithms (GD2L with CAC) to that of other, recently proposed techniques for managing SSDs in database systems.

To answer these questions, we have implemented a variety of algorithms in MySQL's InnoDB storage manager. For the DBMS buffer pool, we have two alternatives to compare: the original buffer pool policies of InnoDB, which we refer to as LRU, and our implementation of GD2L. For SSD management we have implemented CAC as well as three alternatives, which we refer to as CC, MV-FIFO, and LRU2:

CC: CC is cost-based, like CAC, but it is not anticipatory. That is, unlike CAC it does not attempt to predict how a page's I/O pattern will change if that page is moved between the SSD and HDD. It uses Equation 6.1 to estimate the benefit of placing a page in the SSD, and evicts the page with the lowest benefit from the SSD when necessary. CC's approach for estimating the benefit of placing a page in the SSD is similar to the approach used by TAC [11], although TAC tracks statistics on a region basis, rather than a page basis. However, CC differs from TAC in that it considers pages for admission to the SSD when they are cleaned or evicted from the buffer pool, while TAC admits pages on read. Also, TAC manages the SSD as a write-through cache, while CC, like CAC, is write-back.

LRU2: LRU2 manages the SSD using the LRU2 replacement policy, as recently proposed by Do et al. [20] for their *lazy cleaning* (LC) technique. LRU2 is neither cost-based nor anticipatory. Our implementation of LRU2 is similar to LC. Both consider pages for admission when they are cleaned or evicted from the database buffer pool, and both treat the SSD as a write-back cache. Our LRU2 implementation cleans pages in the SSD only when they are evicted, which corresponds to the least aggressive (and best performing) version of LC implemented by Do et al. [20] in SQLServer. The invalidation procedure used for our implementation of LRU2 differs slightly from LC's in that our implementation invalidates an SSD page only if that page is identical to the version of the page on the HDD.

MV-FIFO: MV-FIFO manages the SSD as a FIFO queue of pages. Pages are admitted to the SSD when they are cleaned or evicted from the database buffer pool. If the page being cleaned or evicted already exists in the SSD and the existing version is older, the existing version is invalidated. MV-FIFO is neither cost-based nor anticipatory. It was proposed for SSD management by Kang et al. as the basis of their FaCE algorithm [35]. The FIFO organization of the SSD ensures that all writes to the SSD are sequential and hence fast - this is the chief advantage FaCE.

Either buffer pool technique can be combined with any of the SSD managers, and we will use the notation X+Y to refer to the combination of buffer pool manager X with the SSD manager Y. For example, LRU+MV-FIFO refers to the original InnoDB buffer manager combined with SSD management using MV-FIFO.

6.5.1 Methodology

We used the MySQL database management system, version 5.1.45, with the InnoDB storage engine modified to implement our buffer management and SSD management techniques. MySQL ran on test server with six 2.5GHz Intel Xeon cores and 4GB of main memory, running Ubuntu 10.10 Linux with kernel version 2.6.35-22-generic. The server has two 500GB RPM SCSI hard disks. One disk holds all system software, including MySQL, and the test database. The second disk holds the transaction logs. In addition, the server has a 32GB Intel X25-E SATA SSD. The database SSD cache is implemented as a single file on the SSD. All files in InnoDB use unbuffered I/O.

All of our experiments were performed using TPC-C workloads. Each of our experiments involves measuring performance under a TPC-C workload for a given system configuration, TPC-C scale factor, and combination of buffer pool and SSD algorithms. Our primary performance metric is TPC-C throughput, measured as the number of TPC-C New-Order transactions that are processed per minute (tpmC). Throughput is measured after the system has warmed up and reached its steady state performance. We also collected a wide variety of secondary metrics, including device utilizations and I/O counts measured at both the database and operating system levels. Experiment durations varied from from four to seven hours, largely because the amount of time required to achieve a steady state varies with the system configuration and the TPC-C scale factor. After each run, we restart the DBMS to clean up the buffer pool and we replace the database with a clean copy.

Like Do et al. [20], we have focused our experiments on three representative scenarios:

- database much larger than the size of the SSD cache,
- database somewhat larger than the SSD cache, and
- database smaller than the SSD cache.

To achieve this, we fixed the SSD size at 10GB and varied the TPC-C scale factor to control the database size. We used TPC-C scale factors of 80, 150, and 300 warehouses, corresponding to initial database sizes of approximately are 8GB, 15GB, and 30GB, respectively. The size of a TPC-C database grows as the workload runs. The number of TPC-C client terminals is set to twice the number of warehouses. For each of these scenarios, we tested database buffer pools sizes of 10%, 20%, and 40% of the SSD size (1GB, 2GB, and 4GB, respectively).

For experiments involving CAC or CC, the maximum number of entries in the outqueue is set to be the same as the number of database pages that fit into the SSD cache. We subtracted the space required for the outqueue from the available buffer space when using CAC and CC, so that all comparisons would be on an equal space basis. Unless otherwise stated, all experiments involving CAC use an eviction zone of 10% of the SSD cache size.

6.5.2 Cost Parameter Calibration

As introduced in Sections 6.2 and 6.4, both GD2L and CAC rely on device read and write cost parameters (listed in Figure 6.3) when making replacement decisions. One characteristic of an SSD is its I/O asymmetry: its reads are faster than its writes because a write operation may involve an erasing delay. We measure R_S and W_S separately.

To measure these access costs, we ran a TPC-C workload using MySQL and use *diskstats*, a Linux tool for recording disk statistics, to collect the total I/O service time. We also used InnoDB to track the total number of read and write requests it makes. As *diskstats* does not separate total service time of read requests and that of write requests, we measure the devices' read service time using a read-only workload. The read-only workload is created by converting all TPC-C updates to queries with the same search constraint and deleting all insertions and deletions. Thus, the modified workload has a disk block access pattern similar to that of the unmodified TPC-C workload. First, we stored the entire database on the SSD and run the read-only workload, for which we found that 99.97% of the physical I/O requests were reads. Dividing the total I/O service time (from *diskstats*) by the total number of read requests (from InnoDB), we calculate $R_S = 0.11$ms. Then, we ran an unmodified TPC-C workload, and measured the total I/O service time, the total number of reads, and the total number of writes. Using the total number of reads and the value of R_S obtained from the read-only experiment, we estimated the total I/O service time of the read requests. Deducting that from the total I/O service time, we have the total I/O service time spent on write requests. Dividing the total I/O service time on writes by the total number of write requests, we calculate $W_S = 0.27$ms. Similarly, we stored the database on the HDD and repeated this process to determine $R_D = 7.2$ms and $W_D = 4.96$ms. For the purpose of our experiments, we normalized these values: $R_S = 1$, $R_D = 70$, $W_S = 3$, and $W_D = 50$.

We also checked specifications for these devices. In the specification of SATA SSD, the random read latency is 0.075ms, based on page size 4KB. In InnoDB, the block size is 16KB. Thus, our measured read latency (0.11ms) is larger than the one in the specification. In the specification of the hard disk, the random read/write latency is specified as < 8.5ms. We can see that these cost values are close to what we measured. We suggest the cost parameters can be set based on the device specification.

6.5.3 Analysis of GD2L and CAC

To understand the performance of GD2L and CAC, we ran experiments using three algorithm combinations: LRU+CC, GD2L+CC, and GD2L+CAC. By comparing LRU+CC and GD2L+CC, we can focus on the impact of switching from a cost-oblivious buffer manager (LRU) to a cost-aware buffer manager (GD2L). By comparing the results of GD2L+CC and GD2L+CAC, we can focus on the effect of switching from a non-anticipatory

SSD manager to an anticipatory one. Figure 6.11 shows the TPC-C throughput of each of these algorithm combinations for each test database size and InnoDB buffer pool size.

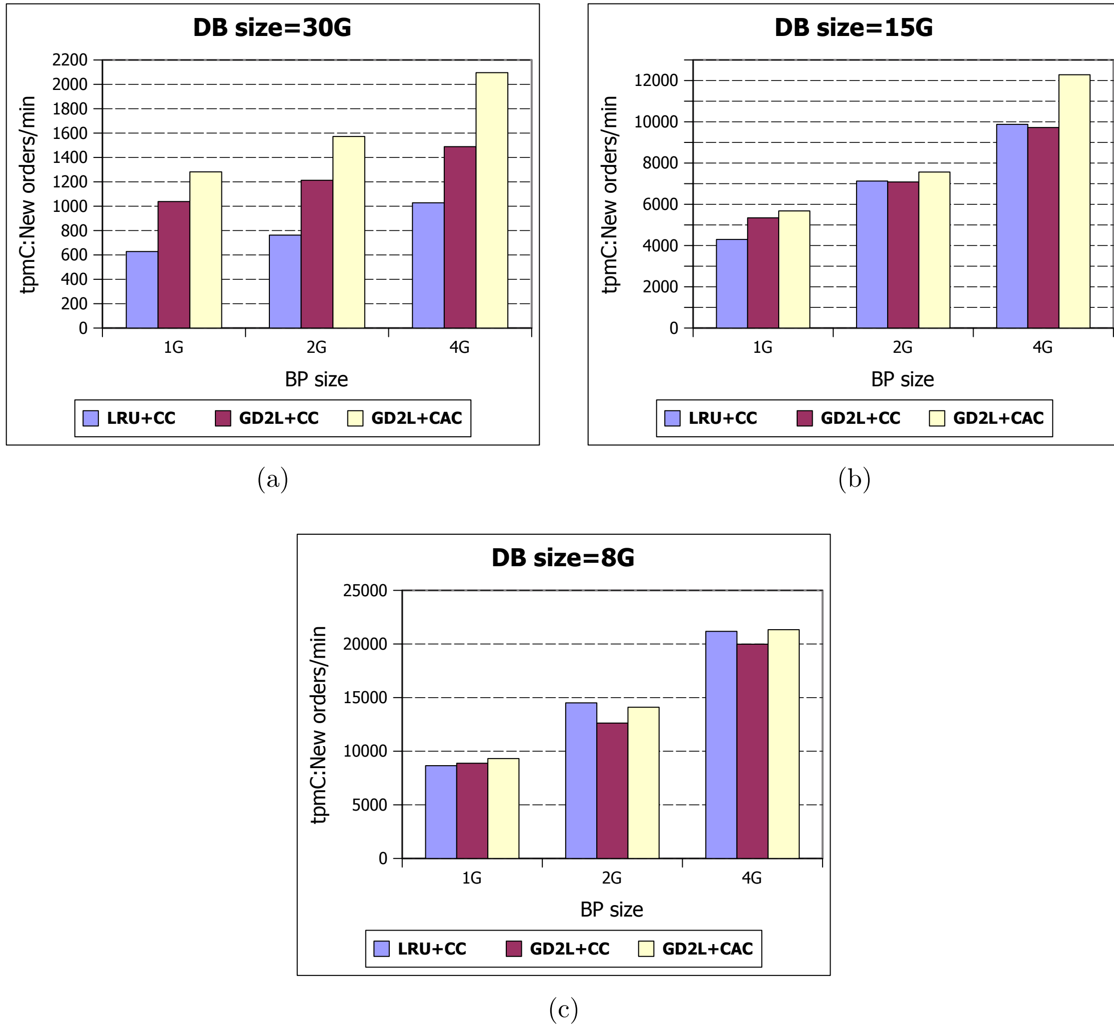


Figure 6.11: TPC-C Throughput

GD2L vs. LRU

By comparing LRU+CC with GD2L+CC in Figure 6.11, we can see that GD2L outperforms LRU when the database is much larger than the SSD. The two algorithms have similar performance for the two smaller database sizes. For the large database, GD2L provides TPC-C throughput improvements of about 40%-75% relative to LRU.

Figures 6.12 and 6.13 show the HDD and SDD device utilizations, buffer pool miss rates, and normalized total I/O cost on each device for the experiments with the 30GB,

Alg & BP size (GB)	HDD util (%)	HDD I/O (ms)	SSD util (%)	SSD I/O (ms)	total I/O (ms)	BP miss rate(%) (overall)	BP miss rate(%) (SSD)	BP miss rate(%) (HDD)
LRU+CC								
1	93	88.6	12	11.1	99.7	6.6	7.7	4.0
2	93	72.8	8	6.0	78.8	4.4	5.0	3.6
4	94	55.1	6	3.3	58.4	2.4	2.5	2.7
GD2L+CC								
1	92	53.1	21	12.1	65.2	8.8	9.3	6.1
2	90	44.3	20	9.7	54.0	7.4	7.8	5.5
4	90	36.3	14	5.8	42.1	4.7	4.9	4.2
GD2L+CAC								
1	85	39.6	19	8.8	48.4	7.4	46.2	0.7
2	83	31.8	20	7.8	39.6	6.3	45.5	0.2
4	82	23.5	20	5.8	29.3	4.8	37.2	0.4

Figure 6.12: Performance Statistics (DB size=30GB)
I/O is reported as ms. per New Order transaction.

15GB, and 8G databases. The normalized I/O cost for a device is the device utilization divided by the New Order transaction throughput. It can be interpreted as the number of milliseconds of device time consumed, on average, per completed New Order transaction. The normalized total I/O cost is the sum of the normalized costs on the HDD and SSD. Buffer pool miss rates include overall miss rate, SSD page miss rate, and HDD page miss rate.

Alg & BP size (GB)	HDD util (%)	HDD I/O (ms)	SSD util (%)	SSD I/O (ms)	total I/O (ms)	BP miss rate(%) (overall)	BP miss rate(%) (SSD)	BP miss rate(%) (HDD)
LRU+CC								
1	79	11.1	38	5.4	16.5	4.2	6.3	0.7
2	68	5.7	47	4.0	9.7	2.7	3.5	0.5
4	73	4.4	43	2.6	7.0	1.3	2.6	0.3
GD2L+CC								
1	21	2.5	68	8.0	10.4	6.1	6.9	0.4
2	18	1.5	62	5.3	6.8	3.7	4.3	0.3
4	14	0.9	61	3.8	4.7	2.3	3.0	0.07
GD2L+CAC								
1	30	3.2	73	7.8	11.0	5.7	21.1	0.06
2	21	1.6	78	6.2	7.8	4.0	18.7	0.04
4	48	2.3	60	2.9	5.3	2.0	9.8	0.04

Figure 6.13: Performance Statistics (DB size=15GB)
I/O is reported as ms. per New Order transaction.

For the 30GB experiments, in which GD2L-CC outperformed LRU+CC, Figure 6.12 shows that GD2L resulted in a much lower total I/O cost (per transaction) than LRU, despite the fact the GD2L had a higher miss rate in the InnoDB buffer pool. GD2L's higher

Alg & BP size (GB)	HDD util (%)	HDD I/O (ms)	SSD util (%)	SSD I/O (ms)	total I/O (ms)	BP miss rate (%)
LRU+CC						
1	5	0.4	72	5.0	5.4	2.9
2	6	0.2	65	2.6	2.9	1.5
4	37	1.1	40	1.1	2.2	0.4
GD2L+CC						
1	5	0.4	70	4.7	5.1	2.9
2	5	0.2	73	3.5	3.7	1.6
4	7	0.2	62	1.9	2.1	0.7
GD2L+CAC						
1	7	0.4	83	5.4	5.8	2.4
2	6	0.2	82	3.5	3.7	1.6
4	24	0.7	62	1.7	2.4	0.1

Figure 6.14: Performance Statistics (DB size=8GB)
I/O is reported as milliseconds of device time per New Order transaction.

miss rate is not surprising, since it considers replacement cost in addition to recency of use when making eviction decisions. Although the total number of I/O operations performed by GD2L+CC is higher than that of LRU+CC, GD2L+CC results in less I/O time per transaction because it does more of its I/O on the SSD and less on the HDD, compared to LRU+CC. From Figure 6.12 we see that the miss rate of SSD pages in the buffer pool is higher than that of HDD pages. This reflects GD2L's preference for evicting SSD pages, since they are cheaper to reload than HDD pages. In the case of the 30GB database, GD2L's shifting of I/O activity from the HDD to the SSD results in significantly higher throughput (relative to LRU+CC) since the HDD is the performance bottleneck in our test environment. This can be seen from the very high HDD utilizations shown in Figure 6.12

For the 15GB experiments, Figure 6.13 shows that GD2L+CC again has lower total I/O cost per transaction than LRU+CC, and shifts I/O activity from the HDD to the SSD. However, the effect is not as pronounced as it was for the larger database. Furthermore, as can be seen from Figure 6.11, this behavior does not lead to a significant TPC-C throughput advantage relative to LRU+CC, as it did for the 30GB database. This is because the SSD on our test server becomes more heavily utilized under the increased load induced by GD2L. (This situation did not happen in the 30GB case, because most of the database hot spot can fit in the SSD.) In a system with greater SSD bandwidth, we would expect to see a TPC-C throughput improvement similar to what we observed with the 30GB database.

For the experiments with the 8G database, both LRU-CC and GD2L-CC have very similar performance. In those experiments, the entire database can fit into the SSD. As more of the database becomes SSD-resident, the behaviour of GD2L degenerates to that of LRU, since one of its two queues (Q_D) will be nearly empty.

CAC vs. CC

Next, we consider the impact of switching from a non-anticipatory cost-based SSD manager (CC) to an anticipatory one (CAC). Figure 6.11 shows that GD2L+CAC provides additional performance gains above and beyond those achieved by GD2L+CC in the case of the large (30GB) database. Together, GD2L and CAC provide a TPC-C performance improvement of about a factor of two relative to the LRU+CC baseline in our 30GB tests. The performance gain was less significant in the 15GB database tests and non-existent in the 8GB database tests.

Figure 6.12 shows that GD2L+CAC results in lower total I/O costs on both the SSD and HDD devices, relative to GD2L+CC, in the 30GB experiments. Both policies result in similar buffer pool hit ratios, so the lower I/O cost achieved by GD2L+CAC is attributable to better decisions about which pages to retain in the SSD. To better understand the reasons for the lower total I/O cost achieved by CAC, we analyzed logs of system activity to try to identify specific situations in which GD2L+CC and GD2L+CAC made different placement decisions. One interesting situation we encountered was that in which a very hot page that is in the buffer pool is placed in the SSD. This may occur, for example, when the page is cleaned by the buffer manager and there is free space in the SSD, either during cold start or because of invalidations. When this occurs, I/O activity for the hot page will spike because GD2L will consider the page to be a good eviction candidate. Under the CC policy, such a page will tend to *remain* in the SSD because CC prefers to keep pages with high I/O activity in the SSD. In contrast, CAC is much more likely to evict such a page from the SSD, since it can (correctly) estimate that moving the page will result in a substantial drop in I/O activity. As shown in Figure 6.12, although the HDD I/Os of GD2L+CC and GD2L+CAC are about the same, GD2L+CAC has much lower buffer pool miss rate on HDD pages. The buffer pool miss rate of HDD pages is physical reads to the HDD divided by the logical read count of the HDD pages in the buffer pool. Thus, we found that GD2L+CAC tended to keep very hot pages in the buffer pool and *out* of the SSD, while with GD2L+CC such pages tend to remain in the SSD and bounce into and out of the buffer pool. Such dynamics illustrate why it is important to use an anticipatory SSD manager (like CAC) if the buffer pool manager is cost-aware.

For the experiments with smaller databases (15GB and 8GB), there was little difference in performance between GD2L+CC and GD2L+CAC. Both policies result in similar per-transaction I/O costs and similar TPC-C throughput. This is not surprising, since in these settings most or all of the hot part of the database can fit into the SSD, i.e., there is no need to be smart about SSD placement decisions. The SSD manager matters most when the database is large relative to the SSD.

6.5.4 Comparison with LRU2 and MV-FIFO

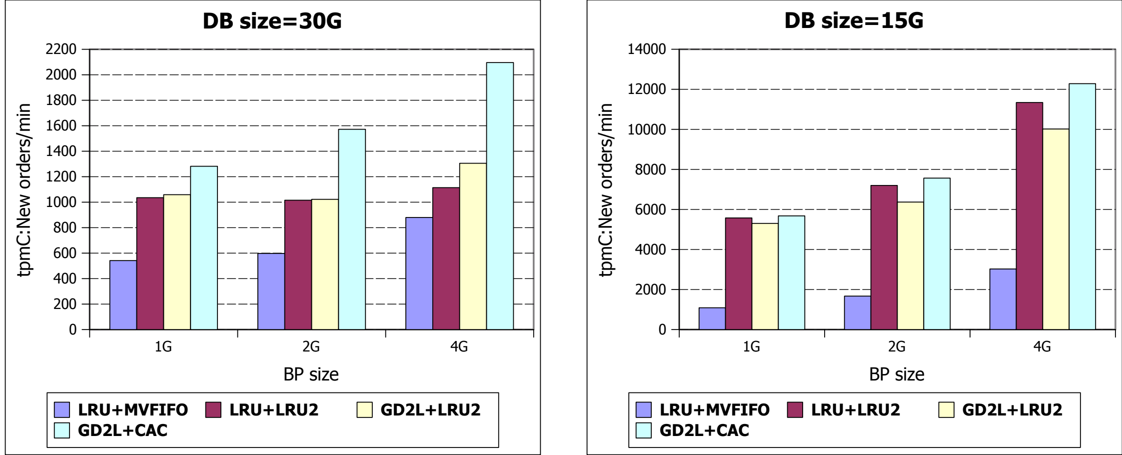
In this section we compare GD2L+CAC to two other recently proposed techniques for managing the SSD, namely lazy cleaning (LC) and FaCE. More precisely, we compare GD2L+CAC against LRU2 and MV-FIFO, which are similar to LC and FaCE but implemented in InnoDB to allow for side-by-side comparison. Since both LC and FaCE focus only on management of the SSD and not on management of the buffer pool, we combine them with InnoDB's default buffer manager for comparison to GD2L+CAC, resulting the combined algorithms LRU+LRU2 and LRU+MV-FIFO.

Figure 6.15 shows the TPC-C throughput achieved by all three of these algorithms on our test system for all three of the database sizes that we tested. In summary, we found that GD2L-CAC significantly outperformed the other two algorithms in the case of the 30GB database, achieving the greatest advantage over its closest competitor (LRU+LRU2) for larger buffer pool sizes. For the 15GB database, GD2L+CAC was only marginally faster than LC, and for the smallest database (8G) they essentially indistinguishable. LRU+MV-FIFO performed much worse than the other two algorithms in all of the scenarios we tested.

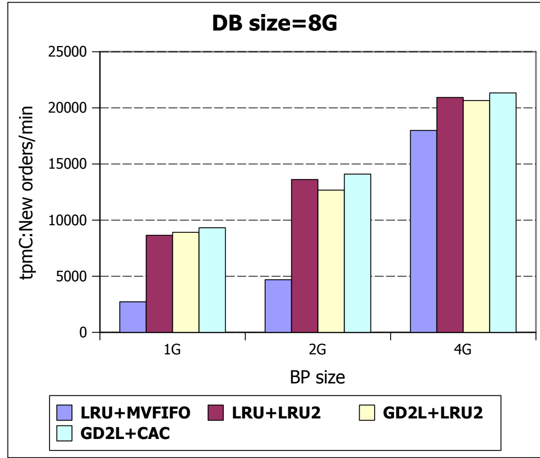
LRU+MVFIFO performed poorly in our environment because the performance bottleneck in our test system is the HDD. The goal of MV-FIFO is to increase the efficiency of the SSD by writing sequentially. Although it succeeds in doing this, the SSD is relatively lightly utilized in our test environment, so MV-FIFO's optimizations do not increase overall TPC-C performance. Interestingly, LRU+MV-FIFO performed poorly even in our tests with the 8GB database, and remained limited by the performance of the HDD. There are two reasons for this. The first is that MVFIFO makes poorer use of the available space on the SSD than LRU2 and CAC because of versioning. The second is disk writes due to evictions as SSD space is recycled by MV-FIFO.

Figure 6.16 shows the device utilizations, buffer hit rates and normalized I/O costs for the experiments with the 30GB database. LRU+LRU2 performs worse than GD2L+CAC in the 30GB database test because it has higher total I/O cost (per transaction) than GD2L+CAC. Furthermore, the additional cost falls primarily on the HDD, which is the performance bottleneck in our setting. Although it is not shown in Figure 6.16, GD2L+CAC does fewer reads per transaction on the HDD and more reads per transaction on the SSD than does LRU+LRU2. This may be due partly to CAC's SSD placement decisions and partly to GD2L's preference for evicting SSD pages.

In the 30GB tests, the performance of LRU+LRU2 remains relatively flat as the size of the database buffer pool increased, while GD2L+CAC's performance increases. One reason for this is that LRU+LRU2 generated more write I/O to the HDD because of SSD evictions than did GD2L+CAC, and the other reason is that LRU2 is a recency-based replacement policy. In the storage system, the SSD can be viewed as a second-tier cache below DBMS buffer pool. One challenge of making effective use of the lower-tier cache is that the upper-tier cache filters the temporal locality. Poor temporal locality in the request streams experienced by the lower-tier cache reduces the effectiveness of recency-based replacement



(a) (b)



(c)

Figure 6.15: TPC-C Throughput

policies, such as LRU and LRU2. As the size of the buffer pool becomes larger, more temporal locality is filtered by the buffer pool and the SSD is used less effectively.

When the database size is 15GB, GD2L+CAC's advantage disappears. In this setting, both algorithms have similar per-transaction I/O costs. GD2L+CAC directs slightly more of the I/O traffic to the SSD than does LRU+LRU2, but the difference is small. For the 8GB database there is no significant difference in performance between the two algorithms

Alg & BP size (GB)	HDD util (%)	HDD I/O (ms)	SSD util (%)	SSD I/O (ms)	total I/O (ms)	BP miss rate(%) (overall)	BP miss rate(%) (SSD)	BP miss rate(%) (HDD)
GD2L+CAC								
1	85	39.6	19	8.8	48.4	7.4	46.2	0.7
2	83	31.8	20	7.8	39.6	6.3	45.5	0.2
4	82	23.5	20	5.8	29.3	4.8	37.2	0.4
LRU+LRU2								
1	85	49.4	15	8.6	58.0	6.7	10.6	1.6
2	87	50.3	12	6.7	57.0	4.4	12.36	1.0
4	90	48.5	9	4.7	53.2	2.4	5.1	1.1
LRU+FIFO								
1	91	101.3	8	9.2	110.5	6.6	10.5	3.1
2	92	92.3	6	5.8	98.1	4.4	7.0	2.5
4	92	62.9	5	3.7	66.6	2.5	8.3	1.2

Figure 6.16: Performance Statistics (DB size=30GB)
I/O is reported as ms. per New Order transaction.

Alg & BP size (GB)	HDD util (%)	HDD I/O (ms)	SSD util (%)	SSD I/O (ms)	total I/O (ms)	BP miss rate(%) (overall)	BP miss rate(%) (SSD)	BP miss rate(%) (HDD)
GD2L+CAC								
1	30	3.2	73	7.8	11.0	5.7	21.1	0.06
2	21	1.6	78	6.2	7.8	4.0	18.7	0.04
4	48	2.3	60	2.9	5.3	2.0	9.8	0.04
LRU+LRU2								
1	34	3.8	65	7.2	11.0	4.9	10.5	0.12
2	45	3.7	56	4.6	8.4	2.6	11.6	0.08
4	42	2.2	56	3.0	5.2	1.3	9.3	0.04
LRU+FIFO								
1	92	50.4	10	5.6	56.0	4.9	7.9	1.9
2	92	33.1	9	3.4	36.5	2.7	4.7	0.32
4	97	20.7	11.3	2.0	22.8	1.4	5.3	0.4

Figure 6.17: Performance Statistics (DB size=15GB)
I/O is reported as milliseconds of device time per New Order transaction.

6.5.5 Impact of the Eviction Zone

To evaluation the impact of the eviction zone, we ran experiments with GD2L+CAC using different eviction zone sizes. In these experiments, the database size was 1GB, the buffer pool size is set to 200M and the SSD cache size is set to 400M. We tested k set to 1%, 2%, 5%, 10% and 100% of the SSD size, Our results showed that k values in this range had no impact on TPC-C throughput. In InnoDB, the page identifier is eight bytes and the size of each page is 16K. Thus, the hash map for a 400M SSD fits in ten pages. We measure the rate with which the SSD hash map was flushed, and find that even with $k = 1\%$, the highest rate of checkpointing the hash map experienced by any of the three

Alg & BP size (GB)	HDD util (%)	HDD I/O (ms)	SSD util (%)	SSD I/O (ms)	total I/O (ms)	BP miss rate (%)
GD2L+CAC						
1	7	0.4	83	5.4	5.8	2.4
2	6	0.2	82	3.5	3.7	1.6
4	24	0.7	62	1.7	2.4	0.1
LRU+LRU2						
1	6	0.4	72	5.0	5.4	4.0
2	6	0.2	70	3.1	3.3	1.5
4	34	1.0	74	2.1	3.1	0.4
LRU+FIFO						
1	92	28.6	12	3.7	32.3	2.9
2	95	12.2	15	1.9	14.1	1.6
4	82.2	6.0	22.2	0.6	6.6	0.4

Figure 6.18: Performance Statistics (DB size=8GB)
I/O is reported as milliseconds of device time per New Order transaction.

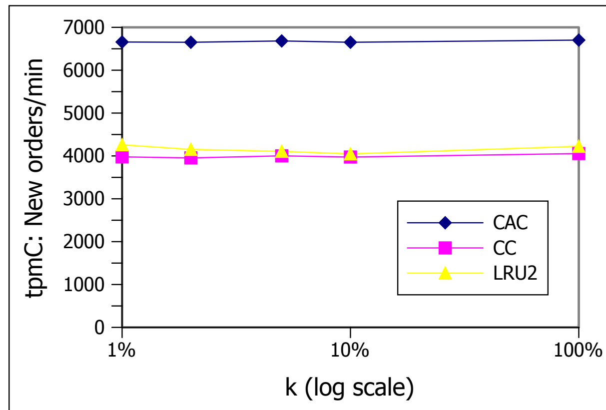


Figure 6.19: Throughput of TPC-C Runs in term of Eviction Zone Size

SSD management algorithms (CAC, CC, and LRU2) is less than three per second. Thus, the overhead imposed by checkpointing the hash map is negligible.

6.6 Conclusion

In this chapter we presented two new algorithms, GD2L and CAC, for managing the buffer pool and the SSD in a database management system. Both algorithms are cost-based and the goal is to minimize the overall access time cost of the workload. We implemented the two algorithms in the InnoDB storage engine and evaluated them using a TPC-C workload. We compared the performance of GD2L and CAC with other existing algorithms. For databases that are large relative to the size of the SSD, our algorithm provided sub-

stantial performance improvements over alternative approaches in our tests. Our results also suggest that the performance of GD2L and CAC and other algorithms for managing SSD caches in database systems will depend strongly on the system configuration, and in particular on the balance between available HDD and SSD bandwidth. In our test environment, performance was usually limited by HDD bandwidth. Other algorithms, like FaCE, are better suited to settings in which the SSD is the limiting factor.

Chapter 7

Conclusion and Future Work

The thesis proposes algorithms for managing caches and hybrid storage devices in modern storage systems. In the thesis, we address two independent problems: the management of the lower-tier cache in multi-tier caches is presented in Chapter 3 - 5, and the management of hybrid storage for database systems in presented in Chapter 6. More detailed conclusions and discussion of potential future research directions can be found at the ends of these chapters.

In the first part of the thesis, we have focused on the management of the lower-tier cache. CLIC, which is a generic cache replacement algorithm, is based on hints. The applications in the upper-tier server attach hints to their I/O requests that they think may be useful to the lower-tier cache. CLIC monitors the request streams of the different clients and collects statistics about the hints they contain. It uses these statistics to automatically learn the usefulness of different hints and attach caching priorities to these hints. We use TPC-C and TPC-H workload traces to evaluate CLIC by comparing it to other replacement algorithms. The experimental results show that CLIC outperforms other algorithms in most cases. In Chapter 4, we present DP-CLIC, which is an extension of CLIC. DP-CLIC can capture temporal variation of the page caching priority by considering how long the page has been in the cache. The experimental results show that DP-CLIC can further improve the performance of CLIC when the size of the upper-tier cache is small (about 10% of the database size) or are large (about 90% of the database size). To reduce the cost of cache management, CLIC uses top-K and feature selection techniques to limit the number of hint sets that CLIC needs to track. The experimental results show that these techniques make CLIC scale well and effectively limit the cost of cache management.

In the second part of the thesis, we have focused on how to make both buffer pool and SSD work efficiently together when the hybrid storage is used for DBMS systems. After the SSD has been added to the storage hierarchy to fill the performance and cost gaps between the memory and the HDD, the HDD is still the system performance bottleneck. Previous work focuses on how to make placement and replacement decisions for the SSD to reduce the I/O accesses to the HDD. Our work opens up one level to include the DBMS

buffer pool, and thus, we manage the buffer pool and SSD jointly to reduce overall I/O access cost. We propose a cost-aware algorithm, GD2L, to make replacement decisions for the DBMS buffer pool. GD2L is aware that there are two different storage devices which have significantly different I/O performance, and it takes into account that difference when make replacement decisions. To manage the SSD in the hybrid storage system, we propose an anticipatory algorithm CAC. CAC is aware that the page access pattern will be changed when the page's location changes. It adjusts page statistics when making placement and replacement decisions for the SSD. We implemented GD2L and CAC in InnoDB (MySQL default storage engine), and evaluated them using TPC-C workloads. By managing the buffer pool and the SSD jointly, we achieved better performance than managing them separately.

7.1 Future Work

There are many possible directions for future work:

- One possible direction is to apply CLIC to other multi-tier cache systems, such as a second-tier cache below a file system cache or a web cache. CLIC is a generic technique for exploiting application hints to manage a second-tier cache. As long as the first-tier cache can pass hints to the second-tier cache, CLIC is able to identify good hints for managing the second-tier cache efficiently. Thus, CLIC is not limited to database systems.

- Another possible direction is to deploy CLIC in the multi-tier caches in the cloud. Even though CLIC is designed for two-tier cache systems, it can be deployed in any lower-tier cache in multi-tier cache systems. Hints generated in the upper tier caches can be passed to the lower tier caches. One potential problem arising in this scenario is that the lower the cache is the more hint types are passed. When some hints have been passed through several tiers of caches, they might become useless. The advantage of CLIC is that it can filter these useless hints using the hybrid algorithm (Chapter 5). One technique we can add to CLIC is that it should not only identify good hints to manage the cache but also identify good hints to pass the lower tier cache. Another problem we need to think about is the page size in different tiers of caches. For CLIC, we assume that the upper-tier cache and lower-tier cache have the same page size. However, the assumption may not be hold for multi-tier caches in a cloud. Thus, techniques for merging and splitting pages should be considered when applying CLIC in the cloud.

- The hybrid storage management technique can be apply to other systems, for example, file systems. GD2L is based on GreedyDual and GreedyDual is designed for the file system originally. Thus, the algorithm is easy to apply to file systems. The hybrid storage management technique is not limited to the hybrid storage consisting SSD

and HDD. Whenever a system needs to manage data for a three storage hierarchy, GD2L and CAC can be applied.

References

[1] Ismail Ari, Ahmed Amer, Robert Gramacy, Ethan L. Miller, Scott Brandt, and Darrell D. E. Long. ACME: Adaptive caching using multiple experts. In *Workshop on Distributed Data and Structures 4 (WDAS)*, pages 143–158. Carleton Scientific, March 2002.

[2] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Lakshmi N. Bairavasundaram, Timothy E. Denehy, Florentina I. Popovici, Vijayan Prabhakaran, and Muthian Sivathanu. Semantically-smart disk systems: past, present, and future. *SIGMETRICS Perform. Eval. Rev.*, 33(4):29–35, March 2006.

[3] 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, pages 176–187, 2004.

[4] S. Bansal and D. Modha. CAR: Clock with adaptive replacement. In *Proc. of the 3nd USENIX Symposium on File and Storage Technologies*, FAST'04, pages 187–200, March 2004.

[5] Luiz Andre Barroso. Warehouse-scale computing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, 2010.

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

[7] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Statistics/Probability Series. Wadsworth Publishing Company, Belmont, California, U.S.A., 1984.

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

[9] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer. Mercury: Host-side flash caching for the data center. In

Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on, pages 1–12, 2012.

[10] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. An object placement advisor for db2 using solid state storage. *Proc. VLDB Endow.*, 2:1318–1329, August 2009.

[11] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. Ssd bufferpool extensions for database systems. *Proc. VLDB Endow.*, 3:1435–1446, September 2010.

[12] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *Proc. USENIX Symp. on Internet Technologies and Systems*, pages 18–29, 1997.

[13] 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.

[14] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. Eviction based cache placement for storage caches. In *USENIX Annual Technical Conference*. Usenix, 2003.

[15] Graham Cormode and Marios Hadjieleftheriou. Finding frequent items in data streams. *Proc. VLDB Endow.*, 1(2):1530–1541, August 2008.

[16] Gautam Das, Vagelis Hristidis, Nishant Kapoor, and S. Sudarshan. Ordering the attributes of query results. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 395–406, 2006.

[17] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: high throughput persistent key-value store. *Proc. VLDB Endow.*, 3:1414–1425, September 2010.

[18] Biplob Debnath, Cristian Ungureanu, Akshat Aranya, and Stephen Rago. Tbf: A memory-efficient replacement policy for flash-based caches. In *Proc. of the 2013 IEEE International Conference on Data Engineering*, ICDE '13, pages 1117–1128, 2013.

[19] David J. DeWitt, Jaeyoung Do, Jignesh M. Patel, and Donghui Zhang. Fast peak-to-peak behavior with ssd buffer pool. In *Proc. of the 2013 IEEE International Conference on Data Engineering*, ICDE '13, pages 1129–1140, 2013.

[20] Jaeyoung Do, Donghui Zhang, Jignesh M. Patel, David J. DeWitt, Jeffrey F. Naughton, and Alan Halverson. Turbocharging dbms buffer pool using ssds. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD'11, pages 1113–1124, 2011.

[21] EMC. EMC XtremCache, 2013. http://www.emc.com/collateral/hardware/data-sheet/h9581-xtremswcache-ds.pdf.

[22] Facebook. Facebook: FlashCache, 2012. http://assets.en.oreilly.com/1/event/45/Flashcache%20Presentation.pdf.

[23] Francesco Folino, Gianluigi Greco, Antonella Guzzo, and Luigi Pontieri. Editorial: Mining usage scenarios in business processes: Outlier-aware discovery and run-time prediction. *Data Knowl. Eng.*, 70(12):1005–1029, December 2011.

[24] Brian C. Forney, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Storage-aware caching: Revisiting caching for heterogeneous storage systems. In *Proc. of the USENIX Symposium on File and Storage Technologies*, FAST'02, pages 5–18, April 2002.

[25] Fusion-IO. Fusion-IO ioTurbine, 2013. http://www.fusionio.com/products/ioturbine/.

[26] Binny S. Gill. On multi-level exclusive caching: offline optimality and why promotions are better than demotions. In *Proc. of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 4–20, 2008.

[27] Goetz Graefe. The five-minute rule 20 years later: and how flash memory changes the rules. *Queue*, 6:40–52, July 2008.

[28] Jim Gray and Bob Fitzgerald. Flash disk opportunity for server applications. *Queue*, 6(4):18–23, July 2008.

[29] Jim Gray and Franco Putzolu. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for cpu time. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, SIGMOD '87, pages 395–398, 1987.

[30] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *J. Mach. Learn. Res.*, 3:1157–1182, March 2003.

[31] IBM. Tivoli Storage Manager HSM for Windows, 2012. http://www-01.ibm.com/software/tivoli/products/storage-mgr-hsm/.

[32] Intel. Understanding the Flash Translation Layer (FTL) Specification, 1998. http://www.jbosn.com/download_documents/FTL_INTEL.pdf/.

[33] Song Jiang and Xiaodong Zhang. ULC: A file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches. In *Proc. 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 168–177, 2004.

[34] 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.

[35] Woon-Hak Kang, Sang-Won Lee, and Bongki Moon. Flash-based extended cache for higher throughput and faster recovery. *Proc. VLDB Endow.*, 5(11):1615–1626, July 2012.

[36] Hyojun Kim and Seongjun Ahn. Bplru: a buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 16:1–16:14, February 2008.

[37] Ricardo Koller, Leonardo Marmol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao Zhao. Write policies for host-side flash caches. In *Proccedings of the 11th conference on File and storage technologies*, pages 45–58, 2013.

[38] Ioannis Koltsidas and Stratis D. Viglas. Flashing up the storage layer. *Proc. VLDB Endow.*, 1:514–525, August 2008.

[39] Sang-Won Lee and Bongki Moon. Design of flash-based dbms: an in-page logging approach. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 55–66, 2007.

[40] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory ssd in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1075–1086. ACM, 2008.

[41] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 213–226, 2008.

[42] Adam Leventhal. Flash storage memory. *Commun. ACM*, 51:47–51, July 2008.

[43] Xiaolei Li, Jiawei Han, Zhijun Yin, Jae-Gil Lee, and Yizhou Sun. Sampling cube: A framework for statistical olap over sampling data. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 779–790, 2008.

[44] Xuhui Li, Ashraf Aboulnaga, Kenneth Salem, Aamer Sachedina, and Shaobo Gao. Second-tier cache management using write hints. In *Proc. of the 4th conference on USENIX Conference on File and Storage Technologies*, FAST'05, pages 9–22, December 2005.

[45] 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*, FAST'09, pages 297–310, 2009.

[46] Xin Liu and Kenneth Salem. Hybrid storage management for database systems. *Proc. VLDB Endow.*, 6(8):541–552, June 2013.

[47] Tian Luo, Rubao Lee, Michael Mesnier, Feng Chen, and Xiaodong Zhang. hstorage-db: heterogeneity-aware data management to exploit the full capability of hybrid storage systems. *Proc. VLDB Endow.*, 5(10):1076–1087, June 2012.

[48] Yanfei Lv, Bin Cui, Bingsheng He, and Xuexuan Chen. Operation-aware buffer management in flash-based systems. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD'11, pages 13–24, 2011.

[49] Mark S. Manasse, Lyle A. McGeoch, and Daniel D. Sleator. Competitive algorithms for server problems. *J. Algorithms*, 11:208–230, May 1990.

[50] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *Proc. International Conference on Very Large Data Bases*, VLDB '02, pages 346–357. VLDB Endowment, 2002.

[51] Patrick Martin, Hoi-Ying Li, Min Zheng, Keri Romanufa, and Wendy Powley. Dynamic reconfiguration algorithm: Dynamically tuning multiple buffer pools. In *11th International Conference on Database and Expert Systems Applications (DEXA)*, pages 92–101, 2000.

[52] Nimrod Megiddo and Dharmendra S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proc. of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 115–130, 2003.

[53] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proc. International Conference on Database Theory (ICDT)*, January 2005.

[54] 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.

[55] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *Trans. Storage*, 4(3):10:1–10:23, November 2008.

[56] OCZ. OCZ RevoDrive Hybrid 1TB PCI-E SSD Review, 2013. http://www.hardwarecanucks.com/forum/hardware-canucks-reviews/47441-ocz-revodrive-hybrid-1tb-pci-e-ssd-review.html.

[57] Oguzhan Ozmen, Kenneth Salem, Jiri Schindler, and Steve Daniel. Workload-aware storage layout for database systems. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 939–950, 2010.

[58] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proc. ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 79–95, December 1995.

[59] Douglas Perry. SSD Prices Falling Faster Than HDD Prices, 2013. http://www.tomshardware.com/news/ssd-hdd-solid-state-drive-hard-disk-drive-prices,14336.html.

[60] L. Rokach and O. Maimon. Top-down induction of decision trees classifiers - a survey. *Trans. Sys. Man Cyber Part C*, 35(4):476–487, November 2005.

[61] Mohammad Sadoghi, Kenneth A. Ross, Mustafa Canim, and Bishwaranjan Bhattacharjee. Making updates disk-i/o friendly using ssds. *Proc. VLDB Endow.*, 6(11):997–1008, August 2013.

[62] Mohit Saxena, Michael M. Swift, and Yiying Zhang. Flashtier: a lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM european conference on Computer Systems*, EuroSys '12, pages 267–280. ACM, 2012.

[63] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Database-aware semantically-smart storage. In *Proc. of the USENIX Symposium on File and Storage Technologies*, FAST'05, pages 239–252, 2005.

[64] Sean M. Snyder, Shimin Chen, Panos K. Chrysanthis, and Alexandros Labrinidis. Qmd: Exploiting flash for energy efficient disk arrays. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, pages 41–49, New York, NY, USA, 2011. ACM.

[65] Gokul Soundararajan, Jin Chen, Mohamed Sharaf, and Cristiana Amza. Dynamic partitioning of the cache hierarchy in shared data centers. In *Proc. International Conference on Very Large Data Bases (VLDB'08)*, pages 635–646, Aug 2008.

[66] Roman Timofeev. Classification and regression trees (cart) theory and applications.

[67] The TPC-C Benchmark. [online] http://www.tpc.org/tpcc/.

[68] The TPC-H Benchmark. [online] http://www.tpc.org/tpch/.

[69] wiki. Decision Tree, 2013. http://en.wikipedia.org/wiki/Decision_tree_learning.

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

[71] Gala Yadgar, Michael Factor, Kai Li, and Assaf Schuster. Mc2: Multiple clients on a multilevel cache. In *Proc. Int'l Conference on Distributed Computing Systems (ICDCS'08)*, June 2008.

[72] Gala Yadgar, Michael Factor, and Assaf Schuster. Karma: know-it-all replacement for a multilevel cache. In *Proc. of the USENIX Symposium on File and Storage Technologies*, FAST '07, pages 25–38, 2007.

[73] Yiming Yang and Jan O. Pedersen. A comparative study on feature selection in text categorization. In *Proceedings of the Fourteenth International Conference on Machine Learning*, ICML '97, pages 412–420, 1997.

[74] Neal Young. The k-server dual and loose competitiveness for paging. *Algorithmica*, 11:525–541, 1994.

[75] Neal E. Young. On-line file caching. In *Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms*, SODA '98, pages 82–86, 1998.

[76] Yuanyuan Zhou, Zhifeng Chen, and Kai Li. Second-level buffer cache management. *IEEE Transactions on Parallel and Distributed Systems*, 15(7), July 2004.

[77] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 91–104, 2001.