Caching for flash-based databases and flash-based caching for databases

Vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von

Diplom-Informatiker Yi Ou

Dekan des Fachbereichs Informatik:
Prof. Dr. Arnd Poetzsch-Heffter

Promotionskommission:

Vorsitzender:	Prof. Dr. Christoph Grimm
Berichterstatter:	Prof. Dr. Dr. Theo Härder
	Assoc. Prof. Dr. Jianliang Xu

Datum der wissenschaftlichen Aussprache:
14. August 2012

D 386

To my family

Acknowledgements

This dissertation documents my work as a member of the research group Databases and Information Systems (DBIS) at the University of Kaiserslautern in the past four years. Nine years ago, I could not have believed that accomplishing such a work would be possible, when I made a career change and started as university student again. Until then, my school-time dream of a Computer Science study could not come true due to various constraints. Now this long-running *transaction* reached the *commit phase*. Fortunately, it is not *atomic* (everything or nothing), because I have gained knowledge and experience in every single step of its progressing. I am also fortunate that, in contrast to real *database transactions*, it did not run in *isolation*, because many people have supported my work—directly or indirectly.

First and foremost, I would like to thank my advisor, Prof. Dr. Dr. Theo Härder, for his encouragement and support, for keeping me highly motivated, for allowing me to manage my time flexibly, and, above all, for offering me the opportunity of doing database research under his guidance. In addition to the guidance, his office door is *consistently* open for providing advice and help. Without his help, this work simply could not be done. It is a great honor and great pleasure to be a member of his group.

I am grateful to Assoc. Prof. Dr. Jianliang Xu, for accepting to be the second examiner, for hosting my short visit to his research group at Hong Kong Baptist University, and for providing valuable advices on my recent research, which greatly contributed to this dissertation. I am grateful to Prof. Dr. Christoph Grimm for chairing the doctoral committee and providing a nice atmosphere during the scientific debate.

I am also grateful to Assoc. Prof. Dr. Peiquan Jin, for numerous inspiring discussions during his stay in our group as a visiting scientist; and to Dr. Ilia Petrov, another expert in my area of research, who has generously shared his know-hows and hands-on experiences with me. Prof. Dr. Gerhard Weikum has provided me the trace data used in their famous LRU-k paper. Maryela Weihrauch, Holger Karn, and their colleagues at IBM have also provided trace data for my experiments. Two students have supported my work: Jonas Jeske has supported me in collecting trace data and worked on the device-access code of my research prototype; Max Bechtold has been assisting me in preparing and tutoring of the practical course of this summer, so that I gained more time for this dissertation.

My research has benefited from my previous work experience on the XTC project. I am grateful to Dr. Michael P. Haustein, the project founder who offered me my first job as programmer and showed me the internals of a real database engine; and to Dr. Christian Mathis, who taught me a lot as a nice person, technical guru, and great team leader. I also learned a lot from the entire project team and I am fortunate to have been a member of them.

I would like to thank all my current and former colleagues for the nice atmosphere in the group. Special thanks go to Dr. Andreas M. Weiner, Joachim Klein, Dr. Boris Stumm, Sebastian Bächle, and Andreas Bühmann for kind help on various occasions; to Caetano Sauer, Daniel Schall, and Volker Höfner for excellent teamwork; and to Heike Neu, Manuela Burkart, and Steffen Reithermann for administrative support. The regular (and irregular) tea (or coffee, or whatever) breaks together with Dr. Leonardo A. Ribeiro, Dr. José de Aguiar Moraes Filho, Dr. Philipp Dopichaj, Dr. Karsten Schmidt, Yong Hu, Weiping Qu, Thomas Jörg, Dr. Jürgen Göres, Xiaofeng Xia, Huiying Duan, Dr. Nikolas Nehmer, Martin Größl, and Hongzhe Jia have been relaxing, interesting, and informative.

I would also like to thank all my friends for their affection, help, and encouragement. I am fortunate to have a *durable* friendship with a circle of old-time friends. Some of them are still visiting my parents regularly, although I have been away from hometown for many years due to work and study. I am greatly indebted to them, especially to Yang Huang and Shaofei Hu. Special thanks also go to my old colleagues and friends, Joachim Götze and Tino Fleuren, for sharing their knowledge with me from the first day we met on our student job, and for encouraging me in the final phase of my research work. Lunch meetings on campus with various friends in Kaiserslautern have not only made the meal more enjoyable but also broadened my perspective to technology, society, and economy.

Last but not least, I would like to thank my family, especially my wife, Jie, for love, support, and understanding; for having confidence in me and for sharing my frustrations and excitements along the journey, especially in the last four challenging years.

Mainz, August 2012

Yi Ou

Abstract

Database storage systems today are primarily based on two technologies: HDD (hard disk drive) and DRAM (dynamic random-access memory). It is increasingly difficult for these systems to deliver acceptable performance, due to fast expanding data volume, growing energy concern, and cost constraints. The emergence of *flash memory* has made cost-effective solutions possible. However, conventional storage systems are designed without the knowledge of flash memory limitations and flash device characteristics. Therefore, they can not fully exploit the potential of flash memory.

This dissertation investigates two major aspects of flash-incorporated database storage systems. The first aspect is related to the buffer management issues of two-tier storage systems where flash devices are used as the primary storage, i. e., *caching for flash-based databases*. The second aspect is related to the mid-tier cache management issues for three-tier storage systems where flash memory is used as a page cache to speed up accesses to the slower primary storage, i. e., *flash-based caching for databases*. The major contributions can be summarized as follows:

- It identifies the weaknesses of previously proposed buffer algorithms for flash-based storage systems and, as improvement, proposes the CFDC (clean-first dirty-clustered) algorithm, which is one of the earliest proposals addressing the *flash random write problem*.
- It examines the *parameter tuning problem*, which discourages the practical use of various previously proposed buffer algorithms, and proposes the CASA (cost-aware self-adaptive) algorithm, which automatically adapts itself to the *extent* of device *R/W asymmetry* and to changing workloads at runtime.
- From an *architectural perspective*, it empirically compares conventional storage systems and three-tier storage systems with flash as the mid-tier cache and delivers indicative implications to system designers.
- It identifies the *cold-page migration problem* in flash-based mid-tier caching and proposes two effective solutions. The results suggest an important architectural consideration that was ignored so far—native management of flash memory by the mid-tier cache manager.

Contents

Nomenclature

A_F	The set of FLAs, page 93
A_b	The set of BTAs, page 98
A_f	The set of FPAs, page 93
C_H	Average cost of a disk access, page 24
C_H	Average cost of a disk access, page 104
C_{FR}	Average cost of a flash-device read, page 88
C_{FW}	Average cost of a flash-device write, page 88
C_H	Average cost of a disk access, page 88
C_{fe}	Cost of a flash-memory erase operation, page 29
C_{fp}	Cost of a flash-memory program operation, page 29
C_{fr}	Cost of a flash-memory read operation, page 29
T_b	Bottom tier, page 83
T_m	Middle tier, page 83
T_t	Top tier, page 83
$\tilde{C}_R$	CASA normalized device read cost, page 69
$\tilde{C}_W$	CASA normalized device write cost, page 69
$algo(X, b)$	Physical reference string output by algorithm $algo$, for logical reference string X, and a buffer pool of b pages, page 24
$cost(Y)$	Cost of serving the sequence of physical requests Y, page 24
m_{FTL}	Address mapping table maintained by FTL, page 93

m_{LPD}	LPD mapping table, page 98
m_{NFA}	NFA mapping table, page 100
$pr(c)$	The priority of a cluster c, page 48
2TA	Two-tier architecture, page 39
3TA	Three-tier architecture, page 39
BTA	Bottom-tier addresses (page numbers), page 98
CASA	Cost-aware self-adaptive, page 20
CFDC	Clean-first dirty-clustered, page 19
CPM	Cold-page migration, page 95
CSC	Cluster-switch count, a metric for spatial locality, page 50
DRAM	Dynamic random-access memory, page 14
FLA	FTL logical address, page 93
FPA	FTL physical address, page 93
FRW	Flash random write, page 36
FTL	Flash translation layer, page 17
GC	Garbage collection, page 31
HDD	Hard disk drive, disk drive, magnetic disk, or disk, page 14
IOPS	I/O per second, page 14
IPD	Inter-page distance, page 48
LRU	Least recently used, page 25
MLC	Multi-level cell, multiple bits per memory cell, page 16
MRU	Most recently used, page 42
SAWC	Self-adaptive write-clustered, page 70
SLC	Single-level cell, one bit per memory cell, page 16
SSD	Solid-state disk, page 15

List of algorithms

List of figures

List of tables

Chapter 1

Introduction

1.1 Motivation

The worldwide data volume is growing at an astonishing speed. According to IDC, the amount of information created, captured, or replicated in digital form was 281 EB (exabyte) in 2007, which corresponds to 45 GB (gigabyte) of data in average for each person in the world. The amount of data produced in 2011 is estimated to be 1800 EB—a ten-fold growth in merely five years starting from 2006 [IDC 08]. The growth of data volume at such a speed is vividly described as *data explosion*.

At the same time, data and data management are becoming *increasingly important* to both normal users and organizations. Data storage and processing are involved in nearly every aspect of our daily life such as communication, banking, shopping, etc. For many organizations, data have become the core assets that their business relies on and the efficiency of data management directly impacts their business success.

Efficient management of data particularly depends on the efficiency of storing and retrieving data, which are the basic functionalities required by any data-intensive system, e. g., a *database system* (DBS). Conceptually, a DBS consists of a collection of hardware, a collection of data, referred to as the *database* (DB), and a software system, referred to as the *database management system* (DBMS), that manages the data. In this dissertation, the term *(database) storage system*, refers to the software and hardware components of a DBS that are responsible for storing and retrieving data.

The storage systems today are primarily based on two technologies: HDD (hard disk drive, disk drive, or magnetic disk) and DRAM (dynamic random-access memory). Being introduced by IBM in the 1950's, HDDs are still the dominant storage device in desktop and server computers today. However, due to their mechanical nature, HDDs suffer from a high access latency. Even for modern enterprise-class HDDs, the latency is typically several milliseconds. Therefore, it is impossible to achieve a throughput higher than 1000 IOPS (I/O per second) using

a single disk drive. This also explains why 1000 TPS (transactions per second) was an important milestone for transaction processing systems.

DRAM (or RAM for brevity), the dominant technology for main memory, is about 100,000 times faster than modern HDDs. However, compared with HDDs, the performance advantage of DRAM comes at a higher price—by two orders of magnitude per GB. Therefore, there is a large gap between these two technologies in terms of both performance and price.

The latency of HDDs and cost of DRAM are two of the most important limiting factors of today's storage systems, but not the only ones. Another critical issue is energy consumption. DRAM-based system memory and disk drives contribute as much as 50% to the overall power consumption of a data center [Roberts 09]. Moreover, this percentage is expected to increase at a rapid pace as we need more DRAM modules and disk drives to improve throughput and to accommodate more data.

In 2005, the total power used by servers in the U.S. represented about 0.6% of its total annual electricity consumption. When cooling and auxiliary infrastructure are included, that percentage becomes 1.2%, an amount comparable to the capacity of five 1000 MW power plants. The total electricity bill for operating those servers and associated infrastructure in that year was 2.7 billion dollars for the U.S. (7.2 billion dollars for the world) [Koomey 07].

Although servers are becoming more powerful and their capacity keeps being improved, the number of server installations has been rapidly increasing[1]. Consequently, the energy cost for running and cooling them grows rapidly, with a clear trend of exceeding server purchase cost, as shown in Figure 1.1.

Addressing these issues requires a memory technology that is *cheaper* than DRAM but *faster* than HDD, and, ideally, it shall be more *energy efficient* than both of them. The search for such a technology has not been successful, until recently *flash memory* gained popularity and, driven by the market demand, achieved great improvements both in price and performance.

1.2 Flash memory

Flash memory is a kind of *non-volatile semiconductor* memory, therefore, no power is required to keep the data stored on it and no mechanical component is required to access those data. This allows for a series of further attractive properties of flash memory and flash devices (storage devices based on flash memory), e. g., small form factor, shock resistance, zero noise, and energy efficiency. These properties have made flash memory particularly popular in mobile devices, e. g., USB flash drives, digital cameras, portable media players, and smart phones. Among various

[1] This can be viewed as a perfect example of the *Parkinson's Law*.

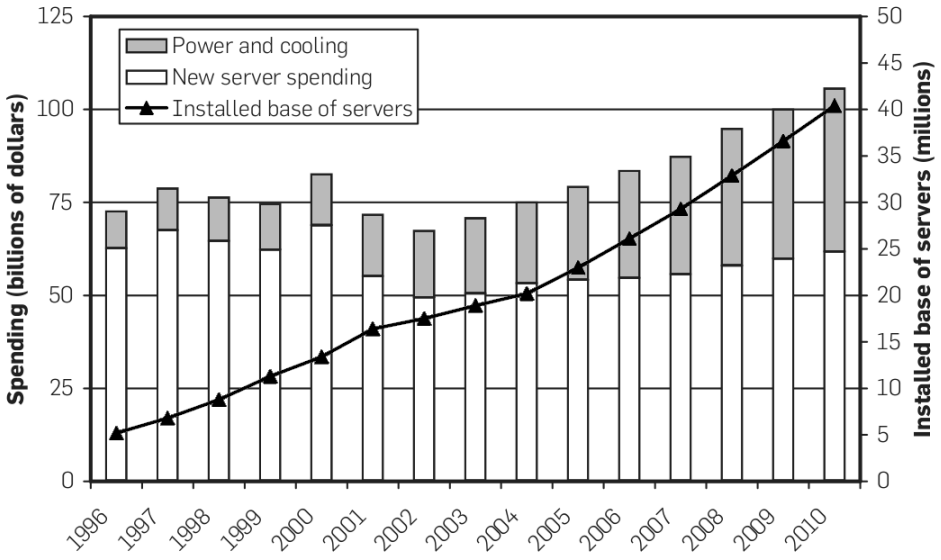


Figure 1.1: IDC estimates for worldwide annual cost spent on powering and cooling servers and purchasing new servers [Roberts 09]

flash devices, flash SSDs (flash-based solid-state disks) are particularly interesting for storage systems, because they have the same host interface as HDDs.

Compared with DRAM, flash memory has a great cost advantage. Its cost has dropped dramatically in recent years, as the market for flash memory has grown and its fabrication has become more efficient. Its density has improved due to the introduction of better processes and additional bits per cell. For example, from 1995 to 2005, the density of flash memory chips has doubled every year[2], implying a corresponding cost reduction of about 50% per year [Samsung 05, Gray 06].

Compared with magnetic HDDs, flash SSDs have a much lower latency (by two orders of magnitude!), because mechanical movements are not required for accessing data. Therefore, it is now possible for a single flash SSD to achieve record-breaking I/O throughputs, which were earlier only possible with dozens of HDDs working in parallel, e. g., in a RAID (redundant array of inexpensive disks) configuration. In fact, compared on a price-per-IOPS basis, flash SSDs are already "cheaper" than HDDs.

Figure 1.2 compares the typical performance and price figures of three major types of storage devices: DRAM, HDD, and flash memory, based on the data of Table A.1 in the Appendix.

In terms of power consumption, flash is more efficient than both DRAM and HDD. For DRAM, a large portion of power is consumed for frequently refreshing its capacitors due to charge leakage. For HDD, accessing data requires energy-consuming mechanical movements. Even if there is no data request to be served,

[2]Faster than an expectation based on the *Moore's Law*.

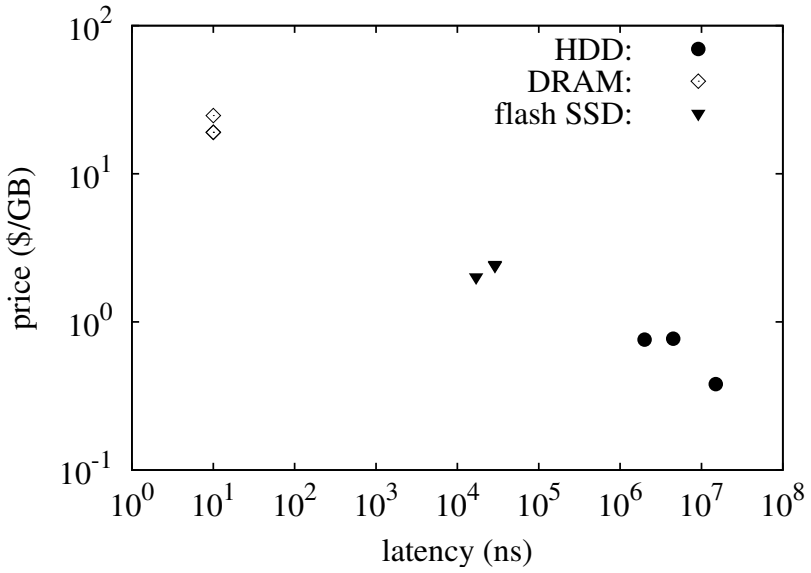


Figure 1.2: Typical performance and price of storage devices

the disks must be kept spinning at a considerable speed.

Due to the aforementioned attractive properties, flash devices are also gaining attention in the primary storage market (for PCs and servers), in form of flash SSDs and flash PCIe cards. Although the per-GB price of those devices is still much higher than that of HDDs, it has a clear trend of becoming more competitive in the coming years.

Depending on the way flash memory cells are arranged, flash memory can be classified into two types: NOR and NAND. NOR flash memory is directly addressable by the processor and it allows for random access in the unit of bytes [Kim 02], whereas NAND flash memory must be accessed via a controller in much larger units. However, NOR flash memory suffers from a lower density (which implies higher price) [Leventhal 08] and higher erase times [Gal 05]. Consequently, NOR flash is typically used for code storage and NAND flash is more appropriate for data storage. Therefore, the use of NAND flash in storage systems is the focus of this dissertation and, hereafter, the terms flash memory and NAND flash memory are used interchangeably.

There are again two types of NAND flash memory: *single-level cell (SLC)* and *multi-level cell (MLC)*, distinguished by the number of bits that can be stored per memory cell. SLC NAND flash stores a single binary value per cell, while MLC NAND flash stores multiple binary values per cell. MLC has a higher density but suffers from a shorter lifespan and higher latencies. Therefore, SLC is more suited for enterprise solutions and MLC is more appropriate for consumer-grade usages [Leventhal 08].

Flash memory's prospects are tantalizing, however, its efficient use is technically challenging, due to some of its special properties and limitations such as *erase-before-write* and *write endurance* (see Section 2.3.1). Therefore, it is often accessed via an intermediate layer called *FTL (flash translation layer)*, which hides the limitations of flash memory and typically supports an HDD-like interface, so that changes to the OS or applications are minimized or not required for using flash devices. However, the performance characteristics of those devices are largely different from those of HDDs, due to the characteristics of flash memory and its limitations. For example, random writes on those devices are generally slower than sequential writes and read operations. We call this problem the *FRW (flash random write) problem*, which will be discussed in more detail in Section 2.3.3.

1.3 Research issues

Due to the exciting recent improvement of flash memory, many believed in its dominance for primary storage in the future, e. g., Gray predicted in [Gray 06]:

Tape is dead, disk is tape, flash is disk, RAM locality is king.

However, a literal interpretation of that prediction is not appropriate, because efficient use of flash memory in database systems is more than simply replacing HDDs by flash SSDs. Existing database systems and algorithms have been designed with conventional HDDs with rotating media in mind, which is largely different from flash memory, therefore, many fundamental principles and techniques of data storage and management must be reconsidered, because, to fully exploit the potential of flash memory, its characteristics must be taken into account in system and algorithm design. We can not elaborate on all these aspects, let alone to discuss them in detail, but the issues sketched below are particularly interesting both from the research and practical perspective.

Overcoming flash memory limitations. The intrinsic limitations of flash memory have a great impact on the performance of flash devices as well as their lifespan. Those limitations must be handled at the device level by improved FTL algorithms[3], e. g., [Kim 02, Lee 07a, Gupta 09], at the file system level by flash-specific file systems, e. g., [Kawaguchi 95, Woodhouse 01], or at the storage system level.

Understanding flash device performance. Many flash devices, e. g., flash SSDs, deal with the aforementioned limitations at the device level. However, due to the complex (and often proprietary) techniques hidden inside the devices, their performance behavior is quite different than that of HDDs, and it varies

[3]Some device-level techniques are introduced in Section 2.3.2, because they are fundamental for comprehension.

severely from device to device. Tailor-made benchmarks and methodology, e. g., those of [Bouganim 09, Chen 09a], are required to investigate the performance behavior of flash devices, because a thorough understanding of their behavior provides a foundation for the optimization of the system based on them.

Speeding up flash I/O. Exemplified in Figure 1.2, access latency of flash memory is yet much higher than that of DRAM, by two orders of magnitude. Therefore, flash devices are typically regarded as I/O devices. Among the major components of database systems, the buffer management component has the most critical impact on I/O performance, because it directly controls when and how to access the storage devices. Conventional buffer algorithms only consider hit ratio as the primary optimization goal, which is not appropriate anymore due to the large difference in performance characteristics between flash devices and HDDs. Some initial efforts, e. g., [Park 06, Jo 06, Jung 08, Seo 08], have been made in this area.

Speeding up transaction processing. To guarantee the ACID (atomicity, consistency, isolation, and durability) properties of transactions [Härder 83b], database systems employ a set of protocols and algorithms for concurrency control and logging & recovery, e. g., [Mohan 92]. These techniques often require a considerable amount of (potentially expensive) accesses to persistent media to guarantee a certain level of data consistency or to ensure well-defined states of the database and transactions even in case of system crash. How to leverage the special properties of flash memory to simplify or to speed up transaction processing is an interesting issue. In [Chen 09b], an approach is proposed which utilizes multiple low-cost flash devices to speed up logging & recovery. Contribution [Gottstein 11] proposed a variant of snapshot isolation that collocates tuple versions created by a transaction in adjacent pages and minimizes random writes at the cost of random reads.

Flash-specific storage scheme and query processing. There are a great deal of possibilities to influence how the database engine accesses storage devices: changing page layout, indexing, and applying novel query processing algorithms. In [Lee 07b], Lee et al. proposed an in-page logging scheme, which stores data pages and their corresponding log records in the same flash block to improve write efficiency of flash-based databases. Contribution [Nath 07] proposed a self-tuning indexing method for a flash-based storage manager. The work described in [Tsirogiannis 09] studied data structures and algorithms that leverage fast random reads to speed up selection, projection, and join operations for relational query processing.

Architecture of flash-based database systems. For an effective use of flash memory in database systems, algorithmic improvements alone are not sufficient, topics regarding architectural issues must also be examined. First, a predominant use of flash devices instead of HDDs for primary storage is not to

happen any time soon, as expected by many expects, e. g., [Kim 12, Grupp 12], mainly due to cost considerations. Therefore, issues regarding hybrid systems where HDDs and flash devices co-exist are of practical value. Such a system with a single flash SSD and a single HDD is studied in [Koltsidas 08]: read-intensive data are placed on the SSD to leverage its superior read performance and write-intensive data on the HDD to avoid expensive random writes to flash. Second, it is not yet clear, in a typically layered database architecture, where flash memory can be used most cost-effectively (in terms of both performance and energy efficiency). For example, using flash memory for the primary storage is the most straightforward approach, however, it requires a considerable amount of flash memory, which is still much more expensive than HDDs. Third, it is also yet to be explored, in which form (e. g., as raw flash memory or in form of flash devices) flash memory shall be used in a database system: using flash devices requires less changes to the database software, but the characteristics of flash memory can only be fully utilized when it is managed directly by the database management system.

The aforementioned research issues span a spectrum that is too broad to be covered in a single dissertation. Some of them are mutually exclusive, e. g., if most of flash I/O can be ideally served by a main-memory buffer pool, the improvements achieved by flash-specific query processing techniques become insignificant, because the characteristics of flash memory is hidden by the buffer pool. Therefore, most of these research issues can be viewed as different solution spaces to the same basic problem—how to improve performance and energy efficiency of database systems that incorporate flash memory. The essence of all possible solutions is to leverage flash-specific characteristics. This dissertation addresses a subset of the aforementioned issues with a focus on database storage systems.

1.4 Outline

The remainder of this dissertation is organized as follows.

Chapter 2 introduces preliminary concepts and techniques that are relevant to our study. Most of them are discussed in the context of database storage systems and flash devices.

Chapter 3 to Chapter 7 present my contribution to this research area. Their content is partially based on a selection of my research papers published between 2009 and 2012. Their main contributions and relations to this dissertation are outlined as follows.

- Aiming at speeding up flash I/O, Chapter 3 discusses representative buffer algorithms previously proposed for flash-based systems, identifies their weaknesses and presents our proposal: the CFDC (clean-first dirty-clustered) algorithm,

based on [Ou 09, Ou 10c]. The basic idea of the algorithm was outlined in [Ou 09], which is one of the earliest works addressing the FRW problem in a database buffer management context. The work described in [Ou 10c] presented an extensive empirical study comparing CFDC with various previously proposed algorithms.

- Extending the study of Chapter 3, Chapter 4 studies the device sensitivity of various flash-aware buffer algorithms and the relationship between their performance and energy efficiency, based on the results of [Ou 10b].
- To ease the practical use of buffer algorithms for flash-based systems, Chapter 5, based on [Ou 10a], examines the parameter tuning problem, which many previously proposed approaches suffer from, and proposes the *cost ratio* concept, emphasizing that the extent of R/W asymmetry is to be taken into account by the buffer algorithms. The chapter also presents the CASA (cost-aware self-adaptive) buffer algorithm, which utilizes the cost ratio and adapts itself at runtime to changing workloads.
- Chapter 6, based on [Ou 11], compares the performance and energy efficiency of two-tier and three-tier storage systems, following a discussion of key concepts and techniques of a three-tier storage system incorporating flash memory in the middle tier. The results of that study reveal that for a variety of workloads, three-tier storage systems are more efficient (both in terms of performance and energy consumption), because cheap and energy-efficient flash memory can be used to replace a considerable amount of energy-hungry DRAM, without sacrificing performance.
- Based on the conceptual and architectural frameworks presented in Chapter 6, Chapter 7 looks into the internals of flash devices used for mid-tier caching. It identifies the CPM (cold-page migration) problem and proposes two effective solutions. Chapter 7 is based on [Ou 12], which, to the best of our knowledge, is the first work that studies native management of flash memory by the mid-tier cache manager.

This dissertation concludes with Chapter 8 which summarizes the work and outlines future research directions.

Chapter 2

Preliminaries

For a discussion on improving the efficiency of flash-based database storage systems, a basic understanding of database storage systems and flash devices is necessary. To this end, this chapter first discusses a well-known DBMS reference architecture, which specifies the responsibilities of all major components of a DBS, before presenting the basic concepts relevant to database storage systems. Then, it introduces the major components and techniques used in flash devices from a device-design perspective, followed by a discussion of their performance characteristics from a device-use perspective. Finally, it outlines the implications of flash memory and flash devices on algorithm and system design.

2.1 DBMS reference architecture

DBMSs are certainly among the most complex software systems, due to various functional and non-functional requirements they have to fulfill, e. g., support of various user interfaces and APIs, guarantee of transactional properties, and performance requirements. Even after decades of evolution, new features are still being continuously added to their "must-have" list, e. g., native support of XML data and XQuery in recent years [Beyer 05, Mathis 06, Haustein 07]. Developing, testing, maintaining, or even understanding such a system is difficult without making considerable efforts concerning its architectural design.

As guidelines for such efforts, a DBMS *reference architecture* is proposed in [Härder 83a], based on the basic ideas of *structured programming* [Dijkstra 68] and *information hiding* [Parnas 75]. The reference architecture defines a hierarchy of five layers (L1 to L5 in Table 2.1) and specifies the major responsibilities and objects of each layer (level of abstraction), from the level of physical storage up to the DBMS interface. This well-specified system architecture provides a perfect context for our discussion. In the following, we briefly interpret the reference architecture, based on our understanding of [Härder 05].

For this dissertation, the most important levels are file management (L1) and

Table 2.1: Five-layer reference architecture for DBMSs [Härder 05]

	level of abstraction	objects	auxiliary data
L5	non-procedural or algebraic access	tables, views, tuples	logical schema description
L4	record-oriented, navigational access	records, sets, hierarchies, networks	logical and physical schema description
L3	record and access-path management	physical records, access paths	free space tables, DB-key translation tables
L2	propagation control	segments, pages	DB buffer pool, page tables
L1	file management	files, blocks	directories, etc.

propagation control (L2). The bottom layer, *file management*, is responsible for reading and writing *DB pages* to and from non-volatile storage devices. Those storage devices are typically the so-called *block devices*, e. g., HDDs and flash SSDs, for which *sectors* (also called blocks) are the unit of transfer, which are fixed-length byte arrays. DB pages are equi-length partitions of a linear address space available to the higher layers. One (DB) page is typically mapped into one or a fixed number of blocks. L1 encapsulates number, type, and location of storage devices so that the higher layers do not have to deal with these details. L1 can be implemented using the file management interface provided by the OS, or directly using the raw device interface[1] for better performance and direct control of the devices.

The next higher layer, *propagation control*, is mainly responsible for accelerating the page-oriented accesses to L1, by buffering pages in main memory, which allows much faster accesses than storage devices. From the L2 perspective, accesses to L1 are *physical*, which often require accesses to the physical storage devices. In contrast, page accesses to L2 are *logical*, because the higher layers do not have to care about where the currently being accessed pages are loaded from (they can be already in the main memory buffer pool or are just fetched from L1) and they only have to *request* the pages from L2 using their logical page addresses.

Similarly, we distinguish between *logical (page) requests* and *physical (page) requests*. A logical request is the intention to read or write a page to and from L2, and a physical request is the intention of L2 to read or write a page to and from L1. Although often used in the context of L2, the distinction between logical and physical concepts is not limited to L2 and can be generally applied to each layer of the reference architecture, because each layer is a level of abstraction, which provides a logical interface to the higher layer and encapsulates the physical details of the lower layer.

The third layer, *record and access management (L3)*, implements a more

[1] On some systems, devices are also abstractly accessed as *files*.

complex interface that allows read, insert, delete, and update of physical records. Moreover, it maintains access path structures of different types, e. g., B-trees, for fast lookup of physical records based on the order of certain physical fields. This layer works on the page-oriented interface offered by L2 and maintains the mapping from physical records to pages. It has to care about the page boundaries and manage the free space inside pages. Fortunately, such physical details do not have to be handled by the next higher layer, the *navigational access layer (L4)*, which allows accesses to logical records via navigational operations or scans in various orders, e. g., table scan, index scan, or an order provided by sorting at runtime. Finally, the top layer, *non-procedural access layer (L5)*, is able to provide logical data structures such as tables and tuples and allows accesses to them via declarative languages such as SQL.

The reference architecture strictly adheres to the information hiding principle and a hierarchical design, which offer the following benefits [Härder 05]:

- It simplifies the implementation of higher-level system components by using the functionality provided by the lower-level system components.
- Changes to the higher-level system components do not require modification of lower-level system components.
- Lower-level system components can be tested without the presence of higher-level components[2].

However, a real DBMS implementation may have to deviate from a strict layered design by introducing some cross-layer or reverse dependencies, called *vertical information channels*, for the purpose of performance optimization (e. g., using query semantics to improve I/O performance [Sacco 82]) or to implement specific system properties (e. g., guarantee of transactional properties often requires collaboration of multiple layers). It is also possible to consolidate some layers to increase the performance optimization potential and to reduce the runtime overhead introduced by a growing number of layers.

Following the reference architecture, the notion of *storage system*, which was introduced in Section 1.1, can be now refined as follows. It consists the bottom-most two layers of a DBMS, i. e., L1 and L2 of the reference architecture, and the hardware components managed by them, in particular, the storage devices managed by L1, and the main memory managed by L2. In the literature, L2 is often referred to as the *buffer manager*, and we use the term *file manager* for L1 in a similar fashion.

The buffer manager plays a key role in the performance optimization of a storage system. Therefore, in the past, numerous research efforts have been made on the topic of buffer management algorithms. In the following, we review some of

[2]By using mock-up (fake) objects, developing and testing of higher-level and lower-level system components in parallel are possible.

the concepts and techniques for buffer management in the context of conventional (disk-based) storage systems.

2.2 Conventional buffer management

Considering the large performance gap between main memory and storage devices (HDDs), it is obvious that, to achieve an optimal performance, the number of page requests that can be directly served from the buffer pool shall be maximized and the number of accesses to the storage devices shall be minimized. However, compared to the DB size (and the capacity of storage devices where the DB resides), the capacity of available main memory is typically much smaller due to economic reasons [Gray 87]. According to [Härder 05], a size ratio of 1 : 1000 is often a good estimation. Consequently, only a subset of the pages from the DB can be kept in the buffer pool at any point in time. Therefore, efficient algorithms must be employed by the buffer manager to effectively utilize the comparatively small amount of main-memory resource.

2.2.1 Basic concepts

If we denote the sequence of n logical requests $(x_0, x_1, \ldots, x_{n-1})$ as X, a buffer management algorithm *algo* is a function that maps X and a buffer pool of b pages into a sequence of m physical requests $Y := (y_0, y_1, \ldots, y_{m-1})$, $m \leq n$, i. e., $algo(X, b) = Y$.

Let $cost(Y)$ denote the accumulated time necessary for a storage device to serve Y, we have $cost(Y) = cost(algo(X, b))$. We call *cost* the cost function and $cost(Y)$ the cost of serving Y. For a disk-based storage system, *cost* is often assumed to be:

$$cost(Y) = |Y| \times C_H \tag{2.1}$$

where the constant C_H is the average cost of a disk access. Therefore, the assumption basically says that the cost is dominated by the number of physical requests $|Y|$.

Given a sequence of logical requests X, a buffer pool with b pages, and a buffer management algorithm *algo*, we say *algo* is *optimal*, *iff* for any other algorithm $algo'$, $cost(algo(X, b)) \leq cost(algo'(X, b))$.

The MIN algorithm proposed by Belady [Belady 66] is such an optimal algorithm. However, optimal algorithms as per this definition require as input the complete sequence of logical requests, which is impossible for online algorithms.

Based on the definition of optimality and the assumption of Equation 2.1, the major optimization goal of buffer management algorithms can be translated into: to

maintain a high *hit ratio*, which is a measure of how often a page request is satisfied without requiring a physical access, defined as: hit ratio = $(|X| - |Y|)/|X|$.

Each *buffer hit*, i. e., the page being requested is already in the buffer pool, saves a physical request, because in case of a *buffer miss* (also called buffer fault), i. e., the requested page is not in the buffer pool, it must be loaded from disk. To achieve a high hit ratio, the buffer manager shall keep the "hottest" pages in the buffer pool, and evict the "coldest" pages to make room for pages that are requested but not yet in the buffer pool.

Which page is the coldest one that can be evicted is decided by a *replacement policy*, which is the core logic of buffer management. Therefore, replacement polices are sometimes also referred to as *buffer algorithms*. The to-be-evicted page selected by the replacement policy is called the *victim* page.

To effectively utilize the main-memory resource, buffer management algorithms have to exploit the (temporal and spatial) *locality* of page accesses [Denning 05]. *Temporal locality* is the phenomenon that an object referenced at one point in time will probably be referenced again sometime in the near future, whereas *spatial locality* refers to the phenomenon that the probability of an object's being referenced is higher if an object stored near it was just referenced.

2.2.2 Exploiting temporal locality

The classical LRU (Least Recently Used) algorithm is one of the most widely-used buffer algorithms. It orders the buffer pages by their reference recency and always selects the least-recently referenced page as the victim, using the *recency* of reference as an indicator for temporal locality. In other words, it assumes that more recently referenced pages have a higher probability of being re-referenced than those that were referenced earlier.

The CLOCK [Corbato 69] algorithm and the Second Chance [Tanenbaum 87] algorithm are two more classical buffer algorithms. They can be viewed as two implementation variants of the same basic idea, because they are functionally identical and always achieve the same hit ratio, which is, in turn, often very close to that of LRU.

LRU, CLOCK, and Second Chance all have a time complexity of $O(1)$ and are relatively simple to implement, which is highly desirable in a high-performance system. However, they are not resistant to *scans* [Sacco 82], which are sequential accesses to a long list of pages (sometimes larger than the buffer pool). Scans typically exhibit a low temporal locality, i. e., pages accessed by a scan have a low probability of being re-accessed (at least not before the scan is finished). However, for algorithms based on simple heuristics such as LRU and CLOCK, pages having higher re-reference probabilities have to be pushed out of the buffer pool to make rooms for the pages scanned, with decreased hit ratio as a result. The property of a buffer algorithm's being resistant to scans is called *scan resistance*.

LRU-k is a classical algorithm specific for DB buffer management [O'Neil 93]. It maintains a history of page references which keeps track of the recent k references to each buffer page. When an eviction is necessary, it selects the page whose *k-th recent* reference has the oldest timestamp. Parameter k is tunable, for which the value 2 is recommended. When $k = 1$, LRU-k corresponds to the classical LRU algorithm.

Both *recency* and *frequency* of references are considered in its victim selection decisions. By considering the k-th recent reference instead of only the 1st recent reference (recency) as in LRU, LRU-k can discriminate between frequently and infrequently referenced pages. Therefore, it is more resistant to scans than LRU, for which *recency* is the only concern.

Another difference of LRU-k to LRU is that it *explicitly* keeps a history of page references, whereas LRU only *implicitly* keeps a reference history limited to the pages that are currently in the buffer pool (by ordering the pages by reference recency). The reference history in LRU-k only records the logical page address and the timestamps of references. Therefore, the algorithm can remember the reference history for a much larger number of pages than the buffer pool can accommodate, without requiring too much space overhead. In general, better replacement decisions can be made by utilizing a longer reference history.

However, LRU-k has a time complexity of $O(\log n)$, which discourages its use in real applications. This issue is elegantly addressed by more recent algorithms such as LIRS [Jiang 02] and ARC [Megiddo 03], which follow the same basic principles: utilizing reference history information and using both reference recency and frequency to exploit temporal locality.

2.2.3 Exploiting spatial locality

Although scans are considered harmful for algorithms such as LRU, they exhibit a strong spatial locality, i. e., page references in a scan are predictable to some extent, which can be utilized to improve disk I/O throughput. For HDDs, the throughput of sequential accesses can be an order of magnitude higher than that of random disk accesses.

Therefore, once such an access pattern is detected, together with the pages that are currently being requested, pages predicted to be referenced as next can be sequentially loaded into the buffer pool in advance, without requiring a separate disk access when they are actually requested later. This technique is commonly referred to as *prefetching*. Some detailed examples are, e. g., [Pai 04, Butt 05, Ding 07]. Prefetching can also be implemented at the file system or device level, where it is more commonly referred to as *read-ahead*.

While prefetching only works for sequential reads, the DULO (Dual Locality) algorithm proposed in [Jiang 05] increases the effectiveness of both I/O scheduling and prefetching, by exploiting both temporal and spatial locality. The basic idea of

DULO is to influence the physical request streams and make them more sequential, by leveraging the filtering effect of the buffer pool (physical requests can be viewed as the output of filtering logical requests using the buffer pool).

2.2.4 Relation to transaction management

The guarantee of transactional properties (ACID) is impossible without the collaboration between the buffer manager and the transaction manager (responsible for concurrency control) and log manager (for logging & recovery).

If a victim page selected by the replacement policy is a *clean page*, i. e., it has not been modified after entering the buffer pool, its memory-resident content can be simply discarded. In contrast, if the victim page is a *dirty page*, i. e., it has been modified after entering the buffer pool, its content has to be written back to persistent storage (called *physical write* or *page flush*), before the corresponding memory area can be reused (otherwise the updates to the victim page are lost and *consistency* can not be guaranteed). Hence, if the replacement victim is dirty, the process or thread requesting a page must wait until page flush completion. This is potentially a performance problem and shall be considered by the buffer manager, as noticed by some researchers [Sacco 82, Effelsberg 84]. However, the clean/dirty page status is commonly ignored in previous works on conventional buffer algorithms, whose main focus is the hit ratio.

The discussion so far has implied that dirty pages are flushed to disk only when they are chosen as replacement victims and their memory area is to be reused. This seems to be a completely autonomous decision of the buffer manager, based on the replacement policy. In fact, the propagation of dirty pages is closely related to the guarantee of *atomicity* and *durability*. If the buffer manager flushes a dirty page modified by an incomplete transaction, atomicity can be violated; if it does not flush the modifications of a committed transaction, durability is not guaranteed. On the other hand, if (cold) pages modified by uncommitted transactions are not allowed to be flushed and replaced, the effectiveness of the replacement policy is reduced; and if commit of a transaction requires flushing all the pages modified by it, the commit response time will become unacceptable.

To have maximum degree of flexibility for the buffer manager, the NoForce/Steal policy for buffer management is necessary [Härder 83b]. *NoForce* means that pages modified by a transaction do not have to be forced to disk at its commit, but only the redo logs. *Steal* means that modified pages can be replaced and their contents can be written to disk even when the modifying transaction has not yet committed, provided that the undo logs are written in advance, observing the WAL (write ahead log) principle. With these options together, the buffer manager has great flexibility in its replacement decision, because the decision is, to a large extent, decoupled from transaction management. Hence, it comes as no surprise that NoForce/Steal is the standard solution for existing DBMSs.

Another aspect related to logging & recovery is *checkpointing*, which limits redo recovery in case of a system failure, e. g., a crash. To create a checkpoint at a "safe place", earlier solutions flushed all modified buffer pages thereby achieving a transaction-consistent or action-consistent firewall for redo recovery on disk. Such *direct checkpoints* are not practical anymore, because—given large DB buffer sizes—they would repeatedly imply limited responsiveness of the buffer for quite long periods. While checkpoints are written, which often occurs in intervals of few minutes, systems are restricted to read-only operations. Assume that many GB (or even TB) of data would have to be propagated to multiple disks using random writes (in parallel). Reaction times for update operations could reach a considerable number of seconds or even minutes. Today, the method of choice is *fuzzy checkpointing* [Mohan 92], where only metadata describing the checkpoint is written to the log, but flushing of dirty pages is performed via asynchronous I/O actions not linked to any specific point in time.

Although the buffer manager is not directly responsible for the *isolation* of concurrent transactions, which is the responsibility of the concurrency control logic, it has to provide a data structure, e. g., a semaphore or a mutex for each buffer page, to facilitate the protection against concurrent modifications to the same page [Gray 93]. The data structure allows multiple transactions to concurrently read the page, but restricts access to a single transaction for writing to the page. Such an extra protection (in addition to the concurrency control) is necessary, because the entire page is to be locked, whereas the granularity of locking used by the concurrency control can be smaller than a page. In addition, with concurrent execution of transactions, such a protection is necessary for the guarantee of *elementary action consistency*, i. e., actions confined to a single page always leave the page in a consistent state after their execution. Elementary action consistency is a layer-specific consistency to be guaranteed at L3, and a building block required by the high-level, transactional *consistency* [Härder 05]. Therefore, page accesses via the buffer manager typically follow a *fix-use-unfix* protocol [Sacco 82, Gray 93].

In summary, the use of NoForce/Steal polices and fuzzy checkpointing is the most efficient combination, which is also the basic assumption on transaction management components for the discussion on buffer management in this dissertation.

2.3 Flash devices

Flash memory is not directly addressable to the processor. Flash devices, e. g., flash SSDs and flash-based PCIe cards, are accessed via typical host interfaces, e. g., SATA, USB, FiberChannel, or PCI Express. The most important building blocks of flash devices are flash memory and FTL. FTL is an intermediate layer between the host interface and flash memory. It is required due to the limitations of flash memory such as erase-before-write and write endurance (see Section 2.3.1).

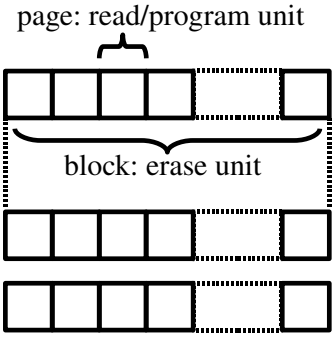


Figure 2.1: Flash memory basic operations

A flash device can have one or multiple flash memory *packages* (chips). A package contains one or multiple *dies*. Each die is organized into multiple *planes* (flash memory arrays). Each plane has a small number of page-sized registers, which are required for the data transfer to and from the flash memory array. Data transfer to and from multiple planes can be carried out in parallel [Agrawal 08].

External to the flash memory packages, flash devices typically have a small amount of DRAM, which is used by the FTL (e. g., for storing the address mapping table) or used as a device cache (on-drive cache) for performance reasons, similar to that of HDDs.

The remainder of this section first introduces the primitive operations supported by flash memory, then it outlines the major responsibilities of FTL and introduces some related key concepts. Finally, it discusses the performances characteristics of flash devices.

2.3.1 Flash memory primitives

Flash memory is a type of electrically *erasable programmable read-only* memory (EEPROM), which supports three basic operations: *read*, *program*, and *erase*. Read and program (also known as write) operations can be performed in the unit of *flash pages*, while erase operations have to be done in a larger unit called *flash block (block, or erase unit)*, which contains a multiple of (e. g., 128) flash pages, as illustrated in Figure 2.1.

Read operations have a very small latency (in the sub-millisecond range). However, program operations are much slower than read operations, typically by an order of magnitude. Erase operations are even slower than program operations, by another order of magnitude. Such a relation for two major types of NAND flash memory is illustrated in Figure 2.2. Let C_{fr}, C_{fp}, and C_{fe} be the costs of read, program, and erase operations, respectively, we have:

$$C_{fr} < C_{fp} < C_{fe} \tag{2.2}$$

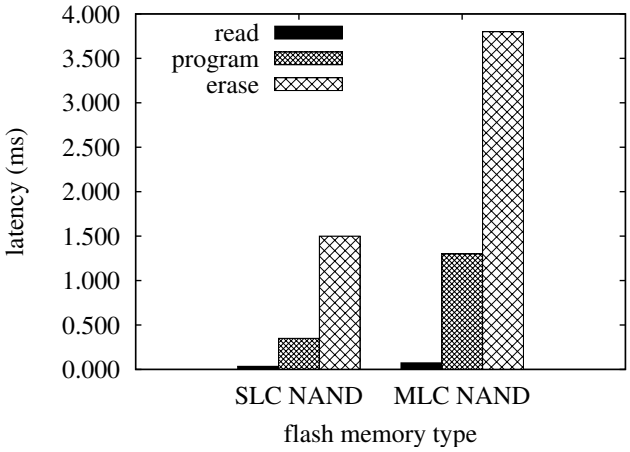


Figure 2.2: Flash memory operational costs

An erase operation turns a block into a *free block* and, consequently, each of its flash pages into a *free flash page*, which is a flash page that has never been programed after the block erasure.

A *program* operation on flash memory can only *clear* bits, i. e., changing their values from 1 to 0. To *set* a (cleared) bit, i. e., changing its value from 0 to 1, an *erase* operation is required, which has to erase an array of memory cells (a flash block) and set all their values to 1 [Woodhouse 01, Gal 05]. This means that only free flash pages can be programed, i. e., a non-free flash page can become programmable again—but only after an erase of the entire block. Therefore, an erase-program combination is required to physically update a flash page, as illustrated by the state diagram in Figure 2.3. This limitation, referred to as *erase-before-write*, implies that an *in-place update* of flash pages would be very expensive. Once a flash page is written, the only way to update it in-place is to erase the entire containing block, followed by a program operation with the updated page content. Furthermore, a backup of user data from other flash pages of the same block prior to the erase, and a restoration of them after the erase

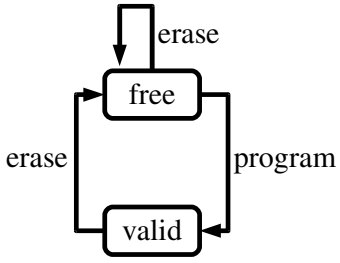


Figure 2.3: Flash page state diagram

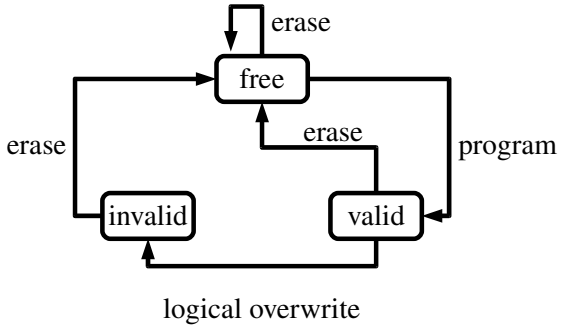


Figure 2.4: FTL page state diagram

are necessary. Such backup and restoration actions require additional read and program operations.

Moreover, flash memory cells can *endure* only a limited number (ranging from 10,000 to 1,000,000 [Gal 05][3]) of program/erase (P/E) cycles, before they wear out and become unreliable. Consequently, after enduring a limited number of P/E cycles, a block becomes highly susceptible to bit errors, i. e., it becomes a bad block. This limitation is called *write endurance*.

A direct use of flash memory is, therefore, challenging. To facilitate the implementation of mechanisms that help to overcome these limitations, a flash page normally consists of a *main area* of several KBs for storing user data, and a *spare area* of a few bytes for storing Error Correcting Code (ECC) or other management information required by those mechanisms.

2.3.2 Flash translation layer

Instead of exposing flash pages and blocks to the device user, FTL supports *logical* page operations, i. e., reading and writing of *logical pages* (corresponding to the size of the main area), and it typically implements an *out-of-place update* scheme to avoid the expensive in-place updates. The out-of-place update works as follows: each logical page update is served using a free flash page prepared in advance, which becomes associated with the logical page address, and the previously associated flash page just needs to be invalidated by updating the management information, as illustrated by the state diagram in Figure 2.4.

Consequently, multiple writes to a logical page can result in multiple page versions co-existing in flash memory. The term *valid page* refers to the latest version, while *invalid pages* refer to the older versions. Similarly, a *valid flash page* is the physical page where the latest version resides. When free flash pages are in short supply, some space taken by invalid pages has to be reclaimed. This is

[3]The number of cycles depends on density, vendor, and flash memory type. SLC NAND flash memory is typically rated for ∼100,000 P/E cycles.

done by a procedure called *garbage collection (GC)*, which reduces the number of invalid pages and increases the number of free flash pages.

To keep track of the valid flash page of a logical page, an address mapping table is maintained. Depending on the map-entry granularity in the concrete implementation of this mapping table, FTL algorithms can be classified into three categories: *page-level* mapping [Ban 95, Birrell 07], *block-level* mapping [Ban 99, Estakhri 99], and *hybrid* mapping [Kim 02, Lee 07a].

Given the endurance constraint, for an ideal device lifespan, the P/E cycles shall be uniformly distributed to all available blocks. Note, if a subset of blocks becomes bad blocks more quickly, the entire device will soon become unusable due to an insufficient number of (usable) blocks, although the remaining blocks are not yet worn out. However, program and erase operations can be "skewed", i. e., a subset of blocks are erased more often than average. In that case, FTL must employ some mechanism, called *wear-leveling*, to "shuffle" blocks that have been erased more often with blocks that are less frequently erased, in order to maximize the device lifespan.

The responsibilities of FTL can be summarized from two perspectives. From the user perspective, it has to support logical page operations, i. e., it shall give the illusion that flash memory is a persistent array of logical pages that can be read and overwritten. From the implementation perspective, it has three major tasks:

- Implementing an out-of-place update scheme.
- Garbage collection.
- Wear leveling.

The three address mapping methods are directly related to the implementation of the out-of-place update scheme. A brief discussion from a performance perspective is given as follows, where we assume that a block contains M pages.

Page-level mapping

Page-level mapping can effectively deal with the erase-before-write limitation. A logical write requires one program operation as well as, in average, one read and $1/M$ erase operations for the GC. Among the three mapping methods, page-level mapping has the greatest performance potential, but it also has the highest resource requirements, mainly due to the mapping table size, which can be prohibitive due to cost reasons and reliability issues (e. g., it is more technically challenging to make a large mapping table resistant to power failures). However, recent studies have shown that the resource problem of page-level mapping can be effectively dealt with using methods such as demand paging of the mapping table [Gupta 09] or new hardware such as PCM (phase-change memory) as the mapping table storage media [Kim 08].

Block-level mapping

Using *block-level mapping*, the mapping table is much smaller, because it only maps a logical block address to its physical location. However, this implies that the offset of a page in the physical block must be identical to its offset in the logical block. To update a page, the new content of the page, due to the offset constraint, must be written to the same offset in a free flash block and the remaining pages of the old block have to be copied to the new block, before the old block can be freed, resulting in $M - 1$ read, up to M program, and one erase operations.

Hybrid mapping

The problems of page-level and block-level mappings can be addressed by hybrid mapping schemes, e. g., [Kim 02, Lee 07a]. In such an approach, a dynamic set of flash blocks, called *log blocks*, is maintained to serve the write requests. The page addresses in a log block are mapped at the *page level*, thus frequent block erasures can be avoided. The remaining flash blocks, called *data blocks*, are managed at the *block level*. Data blocks generally use a much larger flash area than the log blocks. Therefore, the size of the mapping table is not a big issue for hybrid mapping methods.

The approach of [Kim 02] is *block associative*, i. e., a log block is associated with a single data block, and it only serves page writes to the associated data block. If it becomes full, i. e., each page in it has been written once, it is *merged* with the associated data block. If there is no free log block available that can serve the write request to a (not yet associated) data block, one of the (potentially under-utilized) log blocks must be freed, i. e., its content must be propagated to a data block. For each page, its valid version—either in the data block or in the log block—is copied to a third, free block. Then, the third block becomes the new data block. The log block and the old data block are freed and are erased for later use as log block or data block. Thus, a merge operation involves two erasures. An ideal situation happens if a log block contains all valid pages of a data block and their offsets are identical to those of their corresponding pages in the data block, then the log block can be simply marked as the new data block and there is only one erasure necessary to free the old data block. This is called a *switch merge*. This approach may suffer from low space utilization in log blocks, because they often have to be merged before fully utilized.

Lee et al. proposed an approach called *fully-associative* sector translation [Lee 07a], which allows a log block to serve page writes targeting at multiple data blocks, thus achieving higher space utilization in the log blocks and less frequent merge operations. However, if a log block is associated with multiple, say n, data blocks, a merge operation involves all the associated data blocks. For each of them, the valid pages are copied to an empty block. In this case, $n + 1$ erase operations (n erasures for data blocks and one for the log block) are necessary.

2.3.3 Performance characteristics

Flash devices have a set of complex performance characteristics, which are subject to both the device and the workload. Most of these characteristics can be ascribed to the internal design of the devices. The following three aspects of device internals are of particular importance to understand the performance behavior of flash devices.

Flash memory type. SLC and MLC are different in performance, endurance limit, and price. This partially explains why high-end flash devices, which are typically based on SLC flash, often outperform their low-end or mid-range contemporaries, which are typically MLC-based due to cost reasons.

Device cache. DRAM is about two orders of magnitude faster than flash memory. Therefore, the device performance is subject to the presence of a device cache (based on DRAM) and the size of such a cache. A relatively large device cache is often the case for high-end flash devices.

FTL algorithms. FTL plays a key role for flash devices, because it dictates, e. g., how the physical pages and blocks are logically organized, how often the expensive operations such as block erasure and GC are performed, how the metadata are stored and managed, and how the device cache is utilized. The sophisticated mechanisms adopted by FTLs and their broad design space are the major reason for the complex performance behavior of flash devices and the large variety of device-specific performance characteristics.

Although the basic principles of flash device design are no secret, the specific algorithms and components used in a particular device are typically proprietary and transparent to the user. To gain some insights into the performance behavior of those devices, efforts have been made to systematically benchmark the performance of flash SSDs [Gray 08, Chen 09a, Bouganim 09]. The most important findings of these works are summarized and discussed as follows.

Impact of operation type

Most flash-device benchmark results confirm that read workloads have a throughput higher than that of write workloads, sometimes by two orders of magnitudes. This can be attributed to the cost difference among the flash memory primitive operations (see Formula 2.2). Without the device cache, a logical read requires at least one flash read, whereas a logical write requires at least one flash write and sometimes one or multiple flash erases. The presence of a device cache complicates the situation, e. g., a logical read may require the write back of cached data, which can further trigger block erases and GC, so that the observed read latency is even higher than write ones. In that case, a large portion of the latency shall be ascribed to the previous write requests that wrote the cached data.

The significant impact of operation type (read/write) on flash device performance is commonly referred to as *read/write asymmetry* or *R/W asymmetry*. In fact, even for HDDs, the latencies of a read operation and a write operation are slightly different, because the way they handle a read request is different from that of a write one. For example, for a read request, an access to the spinning disk is necessary before the request can be acknowledged, if the requested sector is not in the device cache, whereas, for a write request, the request can be acknowledged as soon as the data has reached the device cache. However, for HDDs, the access pattern dictates how often the disk arm has to be moved (seek) and the latency of moving the disk arm is the dominating factor of HDD performance. Therefore, for HDDs, the performance impact of the operation type is typically ignored.

Interplay of access pattern

Since flash devices do not have mechanical parts, it is commonly believed that access pattern, i. e., whether the access is random or sequential, is not related to their *read* performance. However, it is not always the case, again, due to the device cache. According to [Chen 09a], sequential reads can benefit from a read-ahead mechanism and achieve a higher throughput than random reads. Nevertheless, the access pattern does have a relatively small impact on flash-device read performance, especially compared with its significant impact on HDD performance.

The more serious issue is the impact of access pattern on flash-device write performance. Let us consider block-level mapping methods: for instance, if a block has M pages, due to the erase-before-write limitation, a random write may require up to $M - 1$ read and M program operations and one erase operation, whereas a sequential write only requires one program and $1/M$ erase operations in average.

Hybrid-mapping methods can mitigate the problem to some extent in that a small number of blocks (log blocks) are mapped at the page level, which are conceptually similar to a (write) cache which exploits spatial locality. For the block-associative variant, the probability of a "cache hit", i. e., the data block targeted by a random write is already associated with a log block, is much lower than a sequential write. Consequently, under random write workloads, merge operation must be frequently performed and the being-merged log blocks are more under-utilized. In the worst case, each random write triggers a merge operation. For the fully-associative variant, a merge operation of a randomly filled-up log block involves a relatively high number of data blocks and, consequently, requires a larger number of operations than the merging of a sequential one—for which a switch merge is even possible.

Page-level mapping methods are less sensitive to access patterns. However, to confine the DRAM used for the mapping table to an acceptable limit, modern page-level mapping methods store a large portion of the mapping table on the flash memory. A small portion of it is cached in DRAM on demand [Gupta 09]

and, in case of updates, its must be synchronized with the copy stored in flash memory sometime later. The effectiveness of such a cache and the efficiency of maintaining the mapping table is, again, sensitive to the access pattern.

Therefore, most benchmark results confirm that access patterns have a significant impact on flash device write performance. The throughput of flash devices under random write workloads can be significantly lower than under sequential workloads, sometimes by orders of magnitude. More generally speaking, their random-write throughput can be significantly lower than that of writes with strong spatial locality (sequential write is just a extreme case), due to various cache mechanisms in the device (e. g., device cache and log blocks). We refer to this issue as the *flash random write (FRW)* problem.

Some benchmarks show that flash SSDs can handle random writes with larger *request size*s more efficiently. For example, the bandwidth of random writes using requests of flash block size can be more than an order of magnitude higher than writing requests of page size. In fact, such a write is internally handled as multiple sequential program operations at the page level. Some benchmark results report that some high-end flash devices are seemingly insensitive to access patterns and their random-write performance is comparable to the sequential-write one. There are several reasons for this observation. First, such experiments are often single-threaded so that they did not saturate the sequential-write throughput. Second, the address space of the random requests is too small (e. g., smaller than the capacity of log blocks, or even smaller than the device cache), such that the problems of random writes are hidden.

Further performance behavior

To understand the complex performance behavior of flash devices, operation type and access pattern are not the only factors to be kept in mind, although they are the major ones. For example, under a workload with mixed reads and writes, the latency of reads can be negatively affected by background operations caused by writes, e. g., writing back of dirty pages, and writes can also be affected by reads, which may compete for device cache. Background operations such as GC can also compete for resources with foreground jobs and cause increased latencies and indeterministic response times. Furthermore, the internal state of the flash device, e. g., the number of invalid flash pages and their distribution on the entire flash memory, can also greatly impact performance [Chen 09a]. More importantly, flash SSDs suffer from performance degradation when used in a RAID configuration [Petrov 10], which is often a must in enterprise deployments.

2.4 Flash implications

The distinguished performance characteristics of flash devices provide hints to both algorithm design and system design, which are discussed in the context of storage systems as follows.

2.4.1 Buffer management

Although flash devices offer performance improvements over HDDs of up to a factor 100, buffer management is still an important issue, because there is another two orders of magnitude performance difference between flash memory and DRAM. Data-intensive systems can not rely on the device cache, which does not scale with the device capacity for similar reasons that kept HDD device cache small: cost, volatility, and energy efficiency.

R/W asymmetry

Buffer management for flash-based system shall consider the R/W asymmetry. The assumption of Equation 2.1 is acceptable for disk-based storage systems, because for HDDs under (dominantly) random workloads (which is the case in a typically OLTP DB environment), the number of physical accesses dominates I/O performance. However, such an assumption is not valid for flash SSDs, because their performance behavior is quite different from that of HDDs, especially under random workloads. Their random read performance is typically two orders of magnitude higher than that of HDDs, whereas random writes on flash SSDs can be even slower than those on magnetic disks. As an example, the MTRON MSP-SATA7525 flash SSD achieves 12,000 IOPS for random reads and only 130 IOPS for random writes of 4 KB units [Mtron 08], whereas high-end magnetic disks typically have 200 IOPS for random I/O [Gray 08].

Therefore, the formalization given in Section 2.2.1 can not be directly used to describe the buffer management problem for flash-based storage systems, for which the following extension and modification are necessary.

Each page request, either logical $x_i \in X$, or physical $y_i \in Y$, has to be represented as a tuple of the form $(op, pageId)$, where $op \in \{R, W\}$ is either a read request R or a write request W. For a page request p, its operation type is denoted as $op(p)$. Moreover, *cost* for a flash device can be assumed as:

$$cost(Y) = |\{y \in Y | op(y) = R\}| \times C_{FR} + |\{y \in Y | op(y) = W\}| \times C_{FW} \quad (2.3)$$

where the constants C_{FR} and C_{FW} are the average costs of a read and a write operation on a flash device. The optimization goal is, still, minimize $cost(Y)$.

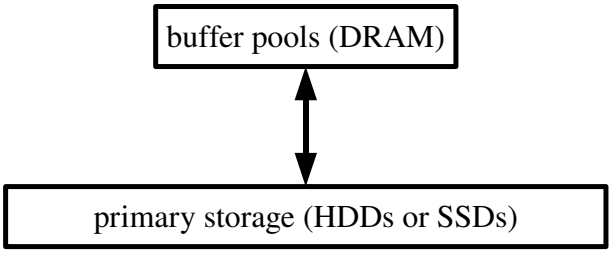


Figure 2.5: Two-tier archtecture

The FRW problem

The FRW problem is not only a performance issue, but also a reliability and cost issue, because behind the performance problem are the higher numbers of flash P/E operations, which shorten the device lifespan. An intuitive idea to address the FRW problem is to increase the DB *page size*, which is the unit of data transfer between the buffer manager and the file system (or the raw device directly) in most database systems. It would be an attractive solution if the overall performance could be improved this way, because only a simple adjustment of a single parameter would be required. However, a naive increase of the page size generally leads to more unnecessary I/O (using the same amount of DRAM for the buffer pool), especially for OLTP workloads, where random accesses dominate. Furthermore, in multi-user environments, large page sizes favor thread contentions. Hence, simply increasing the I/O request size is not feasible, and, a more sophisticated solution is needed.

Read performance

Prefetching of pages plays an important role for conventional disk-based buffer management: It is not hindered by flash devices. But, because of their random-read performance, prefetching becomes much less important, because pages can be randomly fetched on demand without (hardly) any penalty in the form of access latency. Because prefetching always includes the risk of fetching pages later not needed, it is even better for flash-aware buffer algorithms to not use this conventional optimization technique.

2.4.2 Architectural variants

Flash memory not only opens up a broad range of opportunities for algorithmic improvements, its effective use also require architectural considerations. To integrate flash memory into database storage systems, there are basically *three architectural variants*:

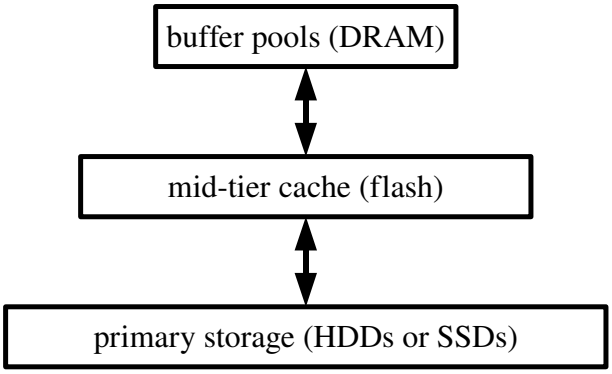


Figure 2.6: Three-tier archtecture

Two-tier architecture (2TA) Flash SSDs support the common host interfaces, therefore, using them as the primary storage and completely replacing HDDs (as shown in Figure 2.5) is the most intuitive usage, which basically follows the reference architecture introduced in Section 2.1. However, since the context of discussion is only the storage system instead of the entire DBMS, we refer to L1 of the reference architecture as the *bottom tier* (based on HDDs or flash SSDs), and L2 as the *top tier* (based on DRAM), for an increased readability.

Three-tier architecture (3TA) The performance of flash memory and flash devices also allow their use as a means to enhance existing disk-based storage systems. Compared with a disk-based storage system following the reference architecture, the three-tier architecture introduces a flash-based *middle tier*, between the DRAM-based *top tier* and the *bottom tier* based on HDD storage, as shown in Figure 2.6. In such a three-tier storage system, there are two caches: a faster but smaller one (the top-tier *buffer* pool) and a slower but larger one (the mid-tier flash-based cache). For increased clarity, we use the term *buffer* or *buffering* to refer to the top-tier buffer pool, and the term *cache* or *caching* to refer to the mid-tier *cache*, whenever appropriate.

Hybrid storage Hybrid storage is another variant of using flash to enhance disk-based two-tier storage systems. In this configuration, flash SSDs are used together with HDDs in the bottom tier as the primary storage. The capacity of flash storage is typically smaller than the HDD storage due to cost reasons. Therefore, ideally the "hot" data shall be placed on the flash SSDs which are faster than the HDDs. The difference to the aforementioned 3TA is that in a hybrid storage the data is only partially stored on HDDs.

For the hybrid storage, the data placement, i. e., when and how to place which portion of data to the SSDs, is a complex issue [Koltsidas 08, Schiefer 10]. Furthermore, the buffer management complexity is increased, because it has to

deal with hybrid storage devices. Both complexities discourage its use in many cases. Therefore, this dissertation focus on the first two architectural variants: 2TA and 3TA. A storage hierarchy following these two architectures is referred to as *two-tier storage hierarchy* or *three-tier storage hierarchy*, respectively.

2.5 Evaluation methodology

All the experiments in this dissertation are performed using our prototype implementation of a database storage engine, which has been evolving during the course of this research work and supports both 2TA and 3TA.

From the user perspective, two-tier and three-tier storage systems have the same interface, i. e., the buffer manager interface. Therefore, the performance seen at the buffer manager interface, e. g., IOPS or response time, represents the overall storage system performance. We use *buffer traces*, i. e., page reference strings as a history recording of a buffer manager's work, as input to the system (See Appendix B). A test program communicates with the buffer manager by sending the logical page requests delivered by the input. While the page requests are being served by the buffer manager, performance metrics such as execution time (wall-clock time elapsed for running a trace) or number of operations are collected for our performance study. Similarly, we also use *mid-tier traces*, which are reference strings to the mid-tier, to examine the behavior of the mid-tier cache manager in some studies related to 3TA. The workload is I/O-intensive, because only the storage engine is involved for storing and retrieving data and no data processing is involved.

Most of the performance metrics interesting to us, e. g., hit ratio and number of device accesses, are independent of hardware. However, some of the performance measurements, e. g., execution time, are indeed specific to the hardware. These measurements are still important for our performance study, because they allow a comparison among various approaches on the same hardware platform. In such a case, we always report the relevant hardware setup to facilitate the repeatability of the experiments.

2.6 Summary

This chapter presented the preliminary knowledge relevant to other parts of the dissertation. It gave an overview of database storage systems, including their role in a DBMS with the help of the reference architecture, an introduction to the two most performance-critical components, the buffer manager and the storage device. Based on the characteristics of the storage device, implications on system and algorithm design are outlined. Finally, the basic approach for performance evaluation is sketched.

Chapter 3

Flash-aware buffer management

Flash SSDs are considered an important alternative to HDDs for storage systems following the two-tier architecture introduced in Section 2.1. In such systems, the secondary storage (bottom tier) exclusively consists of flash SSDs. The decision when and how to access the storage devices is made by the buffer manager (top tier). Therefore, we focus on the problem of buffer management for such systems.

The major contribution of this chapter is the presentation and evaluation of the CFDC algorithm, which is originally proposed in [Ou 09] and evaluated in [Ou 10c]. The basic idea of CFDC is delaying the eviction of dirty pages to reduce the number of flash writes and exploiting the spatial locality to improve page flushing efficiency.

The remainder of this chapter is organized as follows. First, Section 3.1 gives a brief survey of the state-of-the-art buffer algorithms for flash-based systems. Then, Section 3.2 presents the CFDC algorithm. Finally, the experiments evaluating its performance are reported in Section 3.3.

3.1 Flash-aware algorithms

To achieve maximized performance on flash-based devices, performance characteristics of flash-based devices, e. g., the R/W asymmetry and the FRW problem, must be taken into account. Buffer management algorithms tailored to flash-based devices are referred to as *flash-aware* algorithms.

3.1.1 The clean-first strategy

Considering the significant R/W asymmetry of flash-based devices, it is straightforward that flash-aware buffer algorithms shall try to minimize the number of physical writes, even at the expense of some increased number of physical reads if necessary, because the former is much more expensive than the latter.

This criterion is now much more important in our context, because, for flash SSDs, the cost of a page write may be two orders of magnitude higher than that of a page read. Assume, we have to select one of these two pages as the replacement victim: a clean page p and a dirty page q. Furthermore, it is known in advance that, in subsequent requests, p is to be re-read n times, q is to be re-modified m times, and n and m are comparable (i. e., $n \sim m$). Then, it is straightforward to select and replace p in favor of q, because the cost of n flash reads is much lower than the benefit of saving m flash writes by serving them directly in the buffer. We call such a strategy of advancing the eviction of clean pages and delaying the eviction of dirty pages the *clean-first strategy*. Besides a potential performance gain, another strong argument for the clean-first strategy is that saving flash writes extends the device lifespan and improves system availability.

The success of this strategy depends not only on the clean/dirty page state, but also on the page access statistics. The dirty pages, whose evictions are delayed, shall be updated frequently enough, to justify the cost of correspondingly early eviction of clean pages, because the latter could contribute to buffer hits if they were kept in the buffer pool. In the aforementioned example, if $m = 0$, there is no benefit of keeping q in the buffer pool. Similarly, if p is a hot clean page and q a cold dirty page (i. e., $n \gg m$), there will be hardly any performance gain for the clean-first strategy. Although flash SSDs allow much faster random read accesses than HDDs, maintaining a high hit ratio—the primary goal of conventional buffer algorithms—is still important, because memory access is still much faster (the bandwidth of main memory is at least an order of magnitude higher than the interface bandwidth provided by flash SSDs). Therefore, a careful trade-off must be made between increased number of flash reads and decreased number of flash writes.

The clean-first strategy is first proposed by Park et al. in [Park 06], where the CFLRU (Clean-First LRU) algorithm is presented. As the name suggests, the algorithm is based on the LRU replacement policy, and, similar to LRU, it also maintains a list structure where all the buffer pages are ordered by their access recency. However, as a flash-specific improvement, the list in CFLRU is divided into two regions: the *working region* at the MRU (most recently used) end of the list, and the *clean-first region* at the LRU end. In the clean-first region, clean pages are always selected as victims over dirty pages. Only when clean pages are not present in the clean-first region, the dirty page at the LRU tail is selected as victim. The size of the clean-first region is determined by a parameter w called the *window size*. By evicting clean pages first, the buffer area for dirty pages is effectively increased—thus, the number of flash writes can be reduced.

CFLRU is one of the earliest proposals of flash-aware buffer algorithms. It is of great importance from a research perspective. However, there are several problems that hinder a practical use of the algorithm:

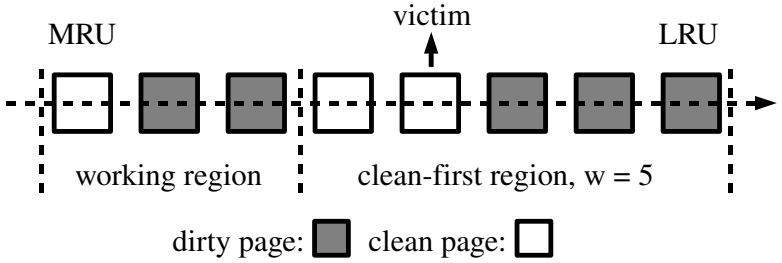


Figure 3.1: Example of the CFLRU algorithm, after [Park 06]

- First, the algorithm often has to walk a potentially very long list in case of a buffer fault, because it has to search backwards from the LRU end for a clean page, and such a page is not always at the LRU tail. Furthermore, such a page tends to stay close to the working region (see Figure 3.1), because clean pages are always selected over dirty pages in the clean-first region.
- Second, under the LRU assumption, the dirty pages in the clean-first region have a lower probability of being re-accessed than the clean pages, thus managing those dirty pages in an LRU fashion may not be an optimal way of utilizing the valuable main-memory resource.
- Third, CFLRU has the same problem as LRU: it becomes inefficient when the workload is mixed with scans, because the hot pages cached so far are pushed away by sequences of cold pages referenced by scans.

Another major problem of CFLRU is its tuning parameter w, which is critical to performance but its optimal value is difficult to determine. The optimal value depends on the extent of R/W asymmetry of the storage device and the update intensity of the workload, which may vary over time. Although its authors have mentioned a dynamic version of CFLRU, which automatically adjusts the parameter "based on periodically collected information about flash read and write operations" [Park 06], its control logic is not presented. However, in this chapter, we assume that the characteristics of both the storage device and the workload do not change frequently, so that an empirically determined "optimal" value for w is acceptable. We will re-visit the parameter-tuning problem in Chapter 5, where the problem is dealt with as the central topic.

As potential alternatives to the CFLRU algorithm, we discuss in the following some representative flash-aware buffer algorithms proposed in recent years. These algorithms can be roughly classified into two categories. Algorithms in the first category, e. g., LRUWSR, CCFLRU, and ADLRU, share the same basic idea (clean-first strategy) with CFLRU and deal with the R/W asymmetry, while algorithms in the second category, e. g., FAB and REF, address the FRW problem.

3.1.2 Other clean-first algorithms

LRUWSR (Write Sequence Reordering) [Jung 08] is a flash-aware algorithm based on LRU and Second Chance, using only a single list as auxiliary data structure. The idea is to evict clean and cold-dirty pages and keep the hot-dirty pages in buffer as long as possible. When a victim page is needed, it starts search from the LRU end of the list. If a clean page is visited, it will be returned immediately (LRU and clean-first strategy). If a dirty page is visited and is marked "cold", it will be returned; otherwise, it will be marked "cold" (Second Chance) and the search continues.

The authors of CCFLRU (Cold-Clean-First LRU) [Li 09] further refine the idea of LRUWSR by distinguishing between cold-clean and hot-clean pages. Cold pages are distinguished from hot pages using the Second Chance algorithm. They define four types of eviction costs: cold-clean, cold-dirty, hot-clean, and hot-dirty, with increasing priority, thus cold-clean pages are first considered for eviction, then cold-dirty, and so on.

The ADLRU (Adaptive Double LRU) algorithm [Jin 12] considers three dimensions in its eviction decision: recency, frequency, and cleaness (i. e., the dirty/clean status of a page). It uses two LRU lists to distinguish frequently and less-frequently accessed pages and to identify least-recently-used and least-frequently-used clean pages, which are then first considered for eviction.

3.1.3 Addressing the FRW problem

FAB (Flash-Aware Buffer) [Jo 06] is a buffer management policy designed for personal media players. FAB manages buffer pages in two dimensions: pages are grouped according to their flash-block membership, the page groups are ordered by recency. If the buffer pool is full, FAB selects a page group as victim and flushes all pages of the group, which also belong to the same flash block. The victim in the FAB method is the page group which contains the largest number of pages and, in case such groups are not unique, the least-recently referenced one among them.

REF (Recently-Evicted First) [Seo 08] is a flash-aware replacement policy that addresses the FRW problem based on the LRU algorithm. It also maintains an LRU list and has a *victim window* at the MRU end of the list, similar to the clean-first region of CFLRU. Victim pages are only selected from the so-called *victim blocks*, which are blocks with the largest numbers of pages in the victim window. From the *set* of victim blocks, pages are evicted in LRU order. When all pages of the victim blocks are evicted, a *linear search* within the victim window is triggered to find a new set of victim blocks. This way, REF ensures that during a certain period of time, the pages evicted are all accommodated by a small number of flash blocks, thus improving the efficiency of FTL.

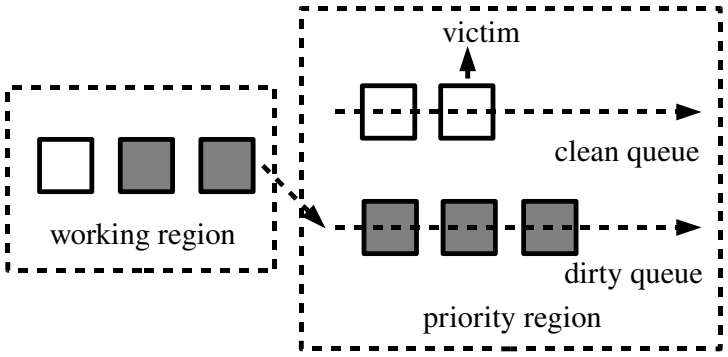


Figure 3.2: Example of the generalized two-region scheme

FAB and REF both address the FRW problem by flushing pages in groups, according to their flash-block membership. Such page flushes can be more efficiently handled by flash-based devices than random page flushes. However, they make no special efforts on minimizing the number of page flushes and ignore the clean/dirty page state in the victim-selection decision. In contrast, the clean-first algorithms focus on reducing the number of page flushes by giving dirty pages higher priorities, but they do not improve the efficiency of page flushing.

3.2 The CFDC algorithm

The problems of existing approaches motivated the design of the CFDC (Clean-First Dirty-Clustered) algorithm, aiming at the following goals:

G1 Minimize the number of physical writes.

G2 Improve the efficiency of page flushing.

G3 Keep a relatively high hit ratio.

3.2.1 Overview

To minimize the number of physical writes (G1), we adopt the clean-first strategy of CFLRU. We address the first problem of CFLRU by introducing *two* queues for the clean-first region: one for the clean pages and one for the dirty pages. A page evicted from the working region goes to the clean queue if it is clean, otherwise to the dirty queue. Upon a buffer fault, if the clean queue is not empty, the tail of the clean queue is returned as the victim, otherwise a page from the dirty queue will be selected as the victim. As a side note, this improvement can also be applied to the CFLRU algorithm. The improved CFLRU behaves the same as the original algorithm in terms of hit ratio and number of page flushes, but search costs for clean pages are entirely eliminated.

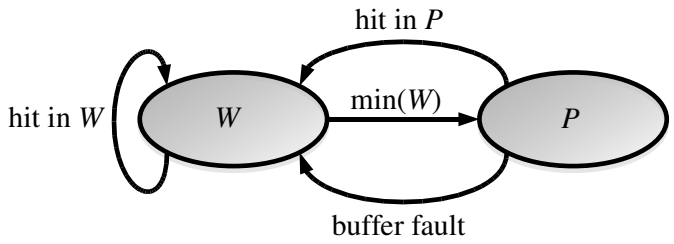


Figure 3.3: Page flow in the two-region scheme

To improve the write efficiency (G2), we propose a technique called *clustered write*, which enforces a strong spatial locality of page flushes and improves the efficiency of writing to flash. A *cluster* is a set of pages physically located in proximity and having the same *cluster number*. We derive the cluster number by dividing their page number by a constant *cluster size*. Though page numbers are logical addresses, because of the space allocation in most DBMSs and file systems, the pages in the same cluster have a high probability of being physically neighbored, too. The byte size of a cluster should correspond, but does not have to be strictly equal to the size of a flash block, thus information about exact flash block boundaries are not required.

To keep a high hit ratio (G3), we propose a *generalized two-region scheme*, where the buffer pool is managed in two regions: a *working region*, similar to CFLRU, for keeping *hot* pages, and a *priority region* responsible for optimizing replacement costs by assigning varying priorities to pages. This way, we can delegate the task of maintaining a high hit ratio to well-studied classical algorithms, without re-inventing the wheels. Such a scheme also makes it possible to study a variety of hybrid buffer algorithms whose working region and priority region are managed by different strategies.

Figure 3.2 shows the generalized two-region scheme using the improved CFLRU method as an example. In this example, the working region uses LRU, whereas the priority region assigns higher priorities to dirty pages. Upon a buffer fault, a victim is selected in the priority region to make room for a page currently in the working region. After this page displacement, the requested page can enter the working region.

3.2.2 Page flow

Following the generalized two-region scheme, CFDC manages the buffer pool in two regions: the *working region* W and the *priority region* P. A parameter $\lambda \in [0, 1]$, called *priority window*, determines the size ratio of P relative to the total buffer. Therefore, if the buffer has B pages, then P contains $\lambda \cdot B$ pages and the remaining $(1 - \lambda) \cdot B$ pages are managed in W. Note, W does not have to be bound to a specific replacement policy. Various conventional replacement policies can be used

Algorithm 1: CFDC

data: request for page p, buffer pool B with b pages, working region W and priority region P, $|W| = (1 - \lambda) \cdot b \wedge |P| = \lambda \cdot b$

```
1  if p ∈ B then
2      if p ∈ W then
3          adjust W as per W's policy ;
4      else
5          demote min(W), promote p ;
6  else
7      page q ← select a victim from P ;
8      if q is null then
9          q ← select victim from W as per W's policy ;
10     if q is dirty then
11         physically write q ;
12     clear q and read content of p from external storage into q ;
13     p ← q ;
14     if p ∈ P then
15         demote min(W), promote p ;
16 return p ;
```

to maintain high hit ratios in W and, therefore, prevent hot pages from entering P. Figure 3.3 illustrates the page flow in our two-region scheme.

The code to be executed upon a page request is sketched in Algorithm 1. If a page in W is hit (line 3), the base algorithm of W should adjust its data and structures accordingly. For example, if LRU is the base algorithm, it should move the page that was hit to the MRU end of its list structure. If a page in P is hit (line 5), a page $min(W)$ is determined by W's victim selection policy and moved (demoted) to P, and the hit page is moved (promoted) to W. In case of a buffer fault, the victim is always first selected from P (line 7). Only when no victim page is available from P (e. g., all pages in P are fixed), we select the victim from W (line 9). Considering recency, the newly fetched page is first promoted to W (line 15).

3.2.3 Priority region

Priority region P maintains three structures: an LRU list L_c of clean *pages*, a priority queue Q of *clusters* where dirty pages are accommodated, and a hash table H with cluster numbers as keys for efficient cluster lookup.

The victim selection logic in P is shown in Algorithm 2. Clean pages are always

Algorithm 2: Select a victim from P

```
data: priority region P, consisting of a list of clean pages L_c in LRU order
      and a priority queue of dirty-page clusters Q
1 if L ≠ ∅ then
2     v ← the LRU page of L_c ;
3 if v = null then
4     cluster c ← cluster of lowest priority in Q ;
5     if c ≠ null then
6         v ← the LRU page in c ;
7         if v ≠ null then
8             c.ipd ← 0 ;
9 return v ;
```

selected over dirty pages (line 1–2). If there is no clean page available, a cluster c having the lowest priority is selected from Q and the LRU page in c is selected as victim (line 3–6). Once a victim is selected from a cluster, its priority is set to minimum (line 8) until all dirty pages in this *victim cluster* are consumed by subsequent page evictions, resulting in strong spatial locality of page evictions.

For a cluster c with n pages ($n > 1$) in Q, with page numbers $p_0, ..., p_{n-1}$, ordered by their time of entering Q, we define a metric, *IPD (inter-page distance)*, to represent its "randomness":

$$ipd(c) = \sum_{i=1}^{n-1} |p_i - p_{i-1}| \tag{3.1}$$

IPD is used to distinguish between randomly accessed clusters and sequentially accessed clusters (IPDs of clusters containing only one page are set to 1). Apparently, we prefer to keep a randomly accessed cluster in the buffer for a longer time than a sequentially accessed cluster. For example, a cluster with pages $\{0, 1, 2, 3\}$ has an IPD of 3, while a cluster with pages $\{7, 5, 4, 6\}$ has an IPD of 5.

The priority of c, denoted as $pr(c)$, is defined as:

$$pr(c) = \frac{ipd(c)}{n^2 \times (globaltime - timestamp(c))} \tag{3.2}$$

The algorithm tends to assign large clusters a lower priority for two reasons:

1. Flash SSDs are efficient in writing such clustered pages.
2. The pages in a large cluster have a higher probability of being sequentially accessed.

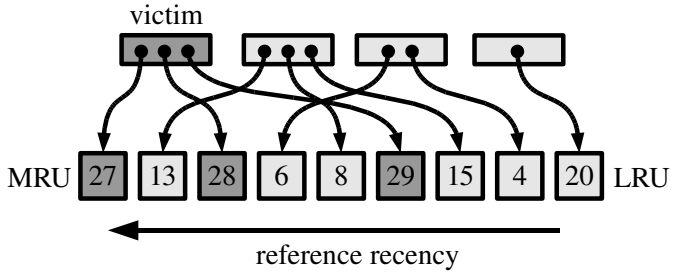


Figure 3.4: Example of clustered write

Both spatial and temporal factors are considered by the priority function. The purpose of the time component in Formula 3.2 is to prevent randomly, but rarely accessed small clusters from staying in the buffer forever. The cluster timestamp $timestamp(c)$ is the value of *globaltime* at the time of its creation. Each time a dirty page is inserted into the priority queue ($min(W)$ is dirty), *globaltime* is incremented. We derive its cluster number and perform a hash lookup using this cluster number. If the cluster does not exist, a new cluster containing this page is created with the current *globaltime* and inserted to the priority queue. Furthermore, it is registered in H. Otherwise, the page is added to the existing cluster and the priority queue is maintained if necessary. If page $min(W)$ is clean, it simply becomes the new MRU node in the clean list.

After demoting $min(W)$, the page to be promoted, say p, will be removed from P and inserted to W. If p is to be promoted due to a buffer hit, we update its cluster IPD including the timestamp. This will generally increase the cluster priority according to Formula 3.2 and cause c to stay in the buffer for a longer time. This is desirable since the remaining pages in the cluster will probably be revisited soon due to locality. In contrast, when adding demoted pages to a cluster, the cluster timestamp is not updated.

An example of clustered write is illustrated in Figure 3.4, where the algorithm selects, from a set of random dirty pages, a cluster with sequential pattern.

The time complexity of CFDC depends on the complexity of the base algorithm in W and the complexity of the priority queue. The latter is $O(\log m)$, where m is the number of clusters. This should be acceptable, since $m \ll \lambda \cdot B$, where $\lambda \cdot B$ is the number of pages in P.

3.3 Experiments

Our test machine has an AMD Athlon Dual Core Processor, 512 MB of main memory, is running Ubuntu Linux with kernel version 2.6.24, and is equipped with a magnetic disk and a flash disk, both connected to the SATA interface used by the file system EXT2. Both OS and the storage engine are installed on the magnetic

disk. The test data (as a DB file) resides on the flash SSD—a 32 GB MTRON MSP-SATA7525 based on NAND flash memory [Mtron 08]. All the pages to be referenced are pre-allocated in the DB file in the file system. Thus, a page fault will not cause an extra page-allocation operation.

We deactivated the file system prefetching and set the DIRECT_IO flag when accessing the flash SSD, so that the influences of file system and OS were minimized. All experiments started with a *cold* DB buffer. Except for the native code responsible for file access, the DB engine and the algorithms are completely implemented in Java. CFDC and competitor algorithms are used by the buffer manager via an interface.

In the following, we use CFDC-k to denote the CFDC instance running LRU-k ($k = 2$) and use CFDC-1 for the instance running LRU in its working region. Both of them are referred to as CFDC if there is no need to distinguish. We cross-compared seven buffer algorithms, which can be classified in two groups: the flash-aware algorithms including CFLRU, LRUWSR, REF, CFDC-k, and CFDC-1; the classical algorithms including LRU and LRU-k ($k = 2$). For better clarity, CFDC-k and LRU-k are not included in the experiments using the real-life traces, because they are of lower practical importance due to a higher complexity ($O(\log n)$).

The *block size* parameter of REF, which should correspond to the size of a flash block, was set to 16 pages (DB page size = 8 KB, flash block size = 128 KB). To be comparable, the *cluster size* of CFDC was set to 16 as well. The VB parameter of REF was set to 4, based on the empirical studies of its authors. Furthermore, we used the improved version of CFLRU as discussed in 3.2.1, which is more efficient at runtime, yet functionally identical to the original algorithm.

The spatial locality of page requests can be quantified by the metric *CSC (cluster-switch count)*. Let $S := \{q_0, q_1, \ldots, q_{m-1}\}$ be a sequence of page requests, $pn(q_i)$ a function that extracts the page number addressed by request q_i, and $nb(p_1, p_2)$ a boolean function that tells if two page numbers p_1 and p_2 are spatially close, i. e., in the same cluster, the metric $csc(S)$ reflects the spatial locality of S:

$$csc(S) = \sum_{i=0}^{m-1} \begin{cases} 0, & \text{if } q_{i-1} \text{ exists and } nb(pn(q_{i-1}), pn(q_i)) \\ 1, & \text{otherwise} \end{cases} \tag{3.3}$$

In our experiments, the function nb is defined as:

$$nb(p_1, p_2) = \begin{cases} true, & \text{if } p_1/16 == p_2/16 \\ false, & \text{otherwise} \end{cases} \tag{3.4}$$

Compared to clustered writes, the sequence of dirty pages evicted by the algorithm REF generally has a much higher CSC, because it selects victim pages from a *set* of victim blocks and the victim blocks can be addressed in any order. However, this kind of write requests can also be efficiently handled by flash SSDs,

if the parameter VB is properly set. Because the sequence of dirty pages evicted can be viewed as multiple sequences of clustered writes that are interleaved with one another, we denote the REF approach as *semi-clustered writes*.

Let $R := \{p_0, p_1, \ldots, p_{n-1}\}$ be the sequence of logical page requests fed to the buffer manager, we further define the metric *cluster-switch factor* (CSF) to reflect its efficiency in performing clustering:

$$csf(R, S) = csc(S)/csc(R) \tag{3.5}$$

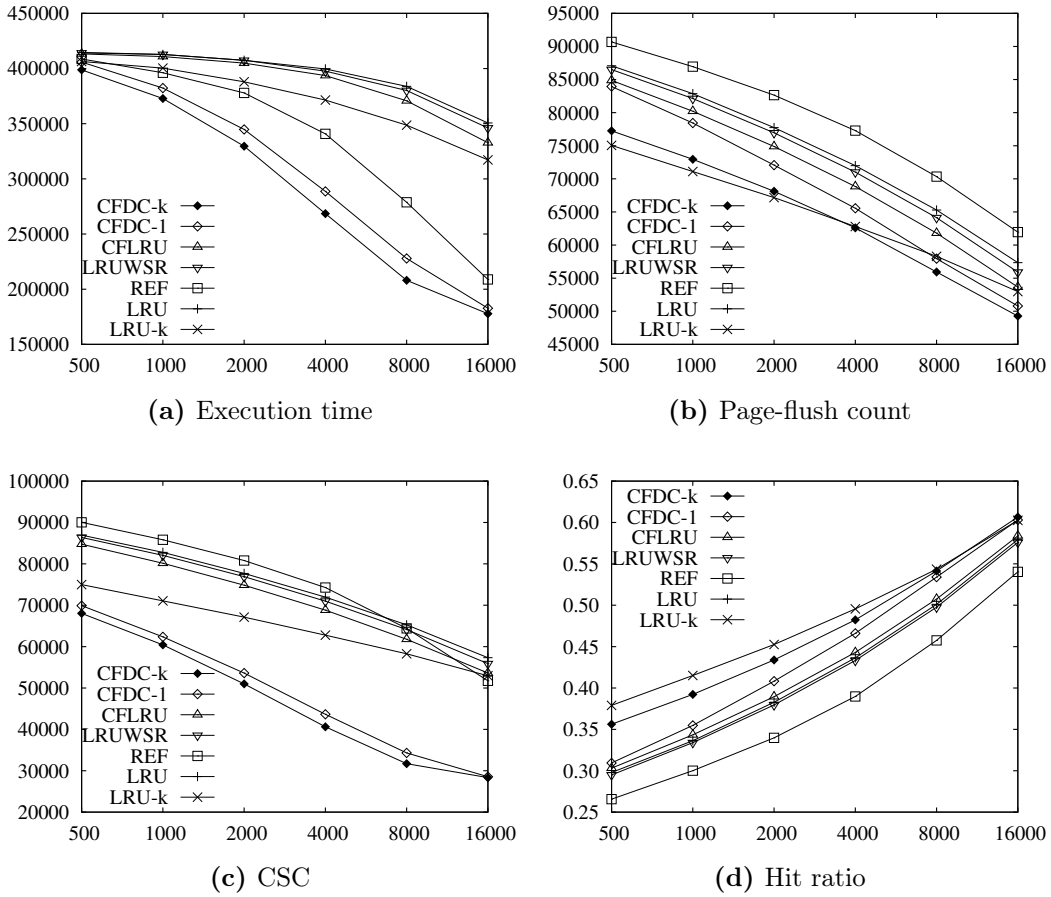
Note, if R is constant, it is sufficient to consider the CSC metric alone. If $d(S)$ is the set of distinct clusters addressed by a sequential access pattern S, we have $csc(S) = |d(S)|$. In this section, CSC is used to study the spatial locality of page flushes, i. e., sequences of physical writes.

3.3.1 Synthetic workload

The synthetic trace simulates typical DB buffer workloads with mixed random and sequential page requests. Four types of page references are contained in the trace: 100,000 single page reads, 100,000 single page updates, 100 scan reads, and 100 scan updates. A *single page read* requests one page at a time, while a *single page update* further updates the requested page. A *scan read* fetches a contiguous sequence of 200 pages, while a *scan update* further updates the requested sequence of pages. The page number of the single page requests are randomly generated between 1 and 100,000 with an 80–20 self-similar distribution. The starting page number of the scans is uniformly distributed in $[1, 10^5]$.

The DB size is 764 MB. Parameter k was set to 2 for both CFDC-k and LRU-k. Parameter λ was set to 0.5. We varied the buffer size from 500 to 16,000 pages (or 4–125 MB) and plotted the performance metrics of running this trace in Figure 3.5. The running times (shown in Figure 3.5a) of CFDC-k and CFDC-1 are very close, with CFDC-k being slightly better. Both CFDC variants clearly outperform all other algorithms compared. For example, with a buffer of 4,000 page frames, the performance gain of CFDC-k over REF is 26%.

Detailed performance breakdowns are presented by Figure 3.5b, 3.5c, and 3.5d, corresponding to the three metrics of interest: number of page flushes, spatial locality of page flushing, and hit ratio. REF suffers from a low hit ratio and a high write count, but is still the third-best in terms of execution times due to its semi-clustered writes. LRU-k performs surprisingly good on flash SSDs—even better than the flash-aware algorithms CFLRU and LRUWSR. This emphasizes that *hit ratio* is still an important metric for flash-aware buffer algorithms.



(a) Execution time

(b) Page-flush count

(c) CSC

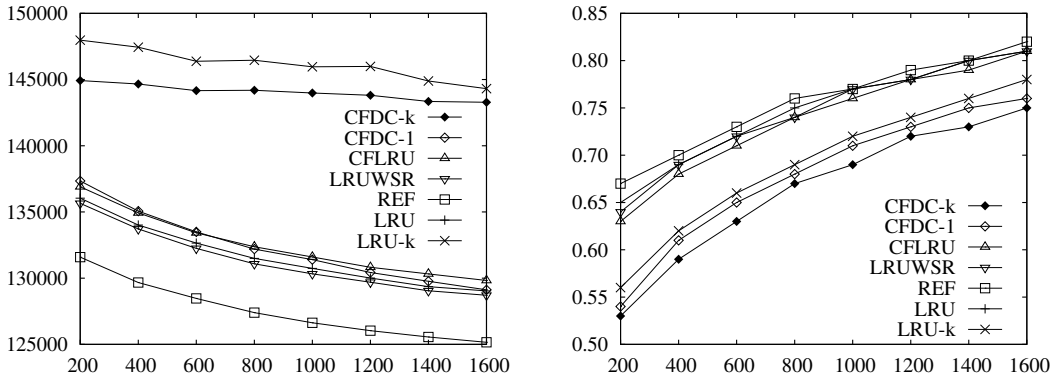
(d) Hit ratio

Figure 3.5: Synthetic trace performance

3.3.2 Scan resistance

To examine scan resistance, we generated a set of traces by changing the locality of the single page requests of the synthetic trace to a 90–10 distribution and varying the number of scan reads and scan updates from 200 to 1,600. The starting page numbers of the scans are moved into the interval $[100001, 150000]$. The buffer size configured in this experiment equals the length of a scan (200 pages). Thus, we simulate the situation where sequential page requests push the hot pages out of the buffer.

The buffer hits in this experiment are compared in Figure 3.6a. While most algorithms suffer from a drop in the number of hits between 5% to 7%, the hits of CFDC-k only decrease by 1% (from 144,926 to 143,285) and those of LRU-k only decrease about 2.5%. This demonstrates that CFDC-k gracefully inherits the merits of LRU-k. Another advantage of CFDC is demonstrated by Figure 3.6b: it has always the lowest CSF, i. e., its page flushes are effectively clustered.



(a) Hit count **(b)** Cluster-switch factor

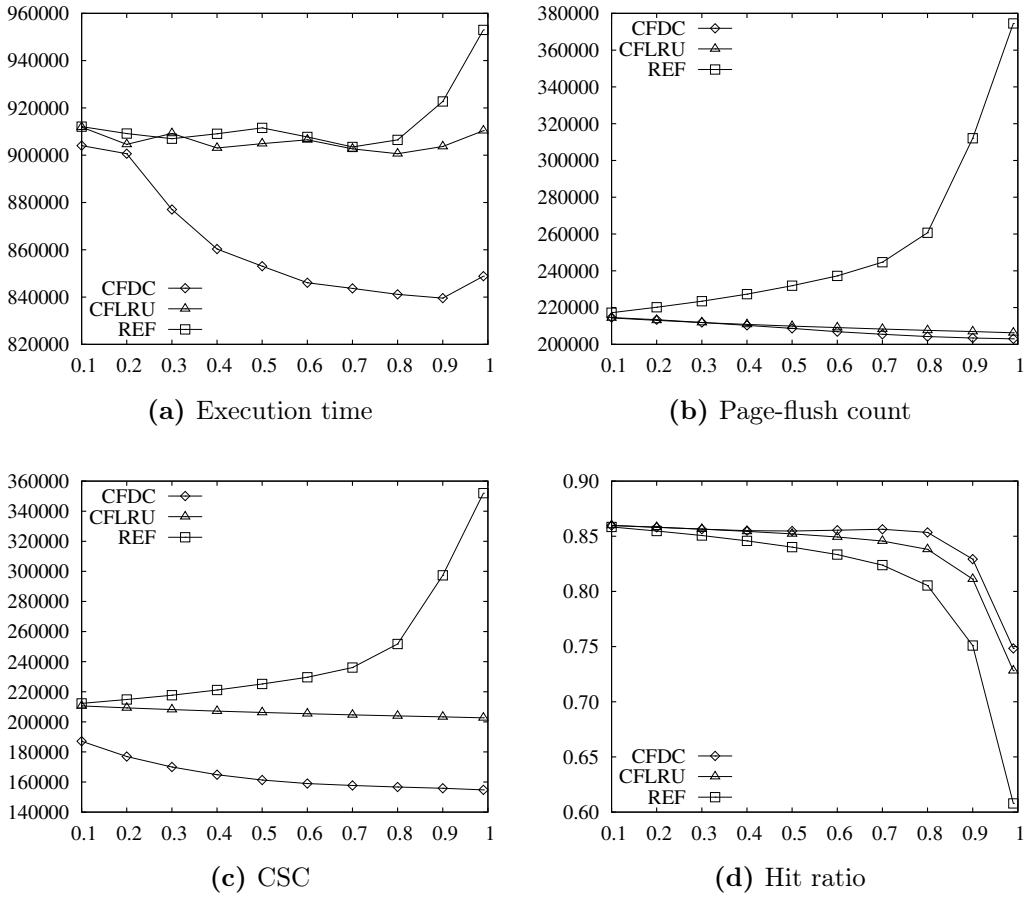
Figure 3.6: Increasing the number of scans (x-axis)

3.3.3 Impact of the window size

In our two-region scheme, the size of the priority region is configurable with the parameter λ, similar to the parameter *window size* (w) of CFLRU. The algorithm REF has a similar configurable *victim window* as well. For simplicity, we refer to them uniformly with the name "window size". In the experiments discussed so far, this parameter is not tuned—it was set to 0.5 for all related algorithms. To examine its impact under a real workload, we ran the TPC-C trace (tpcc in Section B.1.1) for each of the algorithms CFDC, CFLRU, and REF, scaling the respective window size from 0.1 to 0.99 relative to the total buffer size (1,000 pages in this experiment). The performance metrics are shown in Figure 3.7.

The performance of CFDC generally benefits from an increasing window size. However, its runtime goes slightly up after a certain window size is reached (0.9 in this case). This is because, with the size of the working region approaching zero, the loss of the hit ratio is too significant to be covered by the benefit of reducing physical writes and performing clustered writes in the priority region. Similar behavior is also observed at a window size of 0.8 for CFLRU. Based on our experience, CFDC often has the best performance when λ is in the range of $[0.5, 0.8]$.

For CFDC and CFLRU, a larger window size leads to smaller number of writes. In contrast, the number of physical writes generated by REF grows quickly with an increase of the window size (Figure 3.7b), resulting in a sharp runtime increase beginning at window size 0.8. This is due to two reasons: First, in REF's victim window, the sizes of the blocks are the only concern when selecting a victim block, whereas temporal factors such as recency and frequency of references are ignored. Second, REF does not distinguish between clean and dirty pages such that an increase of the window size does not lead to more buffer hits of dirty pages.



(a) Execution time

(b) Page-flush count

(c) CSC

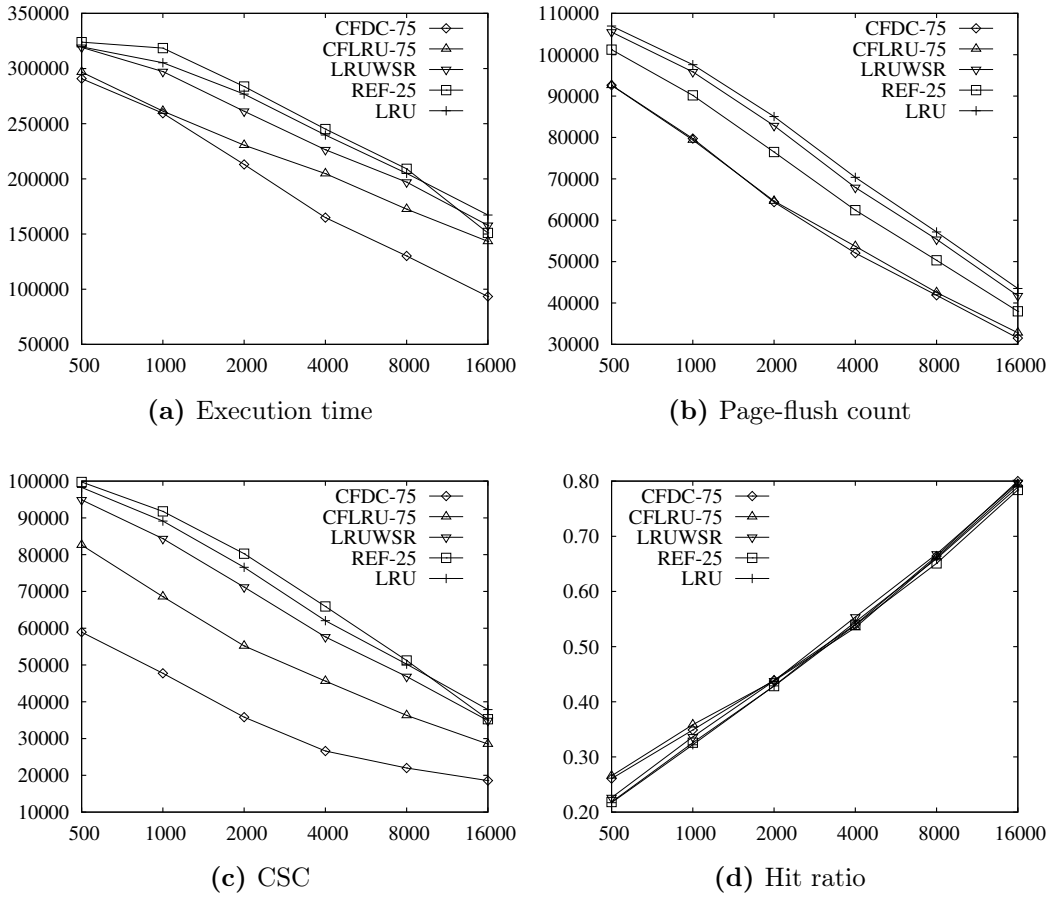
(d) Hit ratio

Figure 3.7: Impact of window size under TPC-C workload

3.3.4 Real-life workload

We also compared the related algorithms using the real-life OLTP workload: the bank trace (see Section B.2). For each of the algorithms CFDC, CFLRU, and REF, we ran all experiments three times with the window size parameter set to 0.25, 0.50, and 0.75 respectively, denoted as REF-25, REF-50, REF-75, etc., and chose the setting that had the best performance.

With the results shown in Figure 3.8, it is clear that CFDC is superior to the other algorithms under the real-life workload that is highly skewed. The performance gain of CFDC over CFLRU is, e. g., 53% for the 16,000-page setting and 33% for the 8,000-page setting. Under the skewed workload, most of the hot pages are retained in a large-enough buffer. Therefore, the differences in hit ratios become insignificant as the buffer size is beyond 2000 pages.



(a) Execution time

(b) Page-flush count

(c) CSC

(d) Hit ratio

Figure 3.8: Performance under real-life workload

3.4 Summary

This chapter focuses on flash-aware buffer management. In Section 3.1, we studied the representative flash-aware buffer algorithms and identified their potential problems. Our proposal, the CFDC algorithm, is presented in Section 3.2. Section 3.3 is an extensive empirical study comparing all relevant algorithms in an identical environment, under a variety of parameters and workloads.

The experimental results clearly show that CFDC's performance is superior among the compared algorithms. The basic idea of CFDC is delaying the eviction of dirty pages to reduce the number of flash writes and exploiting the spatial locality to improve page flushing efficiency. By following a flexible generalized two-region scheme, CFDC delegates the task of maintaining a high ratio of buffer hits to the base algorithm in one of its buffer-pool regions.

Some of the flash-aware algorithms, including CFDC, have a tuning parameter to control the trade-off between hit ratio and flash-specific optimization. In this

chapter, we assumed that the characteristics of both the storage device and the workload do not change frequently, so that we can use a rule-of-thumb value for these parameters. Chapter 5 drops this assumption and proposes an algorithm that can adapt itself to such changes.

Chapter 4

Energy efficiency and performance

Previous research on flash-aware buffer management focused on flash-specific performance optimizations. However, an important question remains open: how sensitive is the performance of flash-aware algorithms to the storage device? This question is important due to two reasons. First, the performance characteristics of flash-based devices varies greatly from device to device. Second, due to the lower $/GB cost, magnetic disks will certainly remain dominant in the near future, therefore enterprises have to deal with the situations where both kinds of devices co-exist.

Another interesting aspect ignored in all previous works is the energy efficiency of related algorithms, which is often critical in environments where flash devices are deployed in the first place, e. g., mobile data management. Energy efficiency is also becoming increasingly important in server environments, due to rapidly rising energy costs and environmental concerns.

Based on these observations, this chapter extends the performance study of Chapter 3 and, at the same time, examines the energy consumption issue. The major contributions of this chapter are:

- An extensive performance study of related algorithms using a variety of flash SSDs and magnetic disks. This device sensitivity study is missing in all previous works, where evaluation was often performed on a single simulated flash device.
- An examination of the energy consumption of the system running these algorithms using an energy measurement device. The examination reveals a strong correlation between system performance and energy consumption and demonstrated the great energy-saving potential of flash SSDs.

The remainder of this chapter is organized as follows. Section 4.1 introduces the hardware system used for both performance and energy measurements. Section 4.2 reports our experimental results. Section 4.3 summarizes our findings.

Table 4.1: Disk drives used in the test

name	device type	idle (W)	peak (W)	interface
HDD1	WD WD800AAJS 7200 RPM	5.3	6.3	SATA
HDD2	WD WD1500HLFS 10000 RPM	4.5	5.7	SATA
HDD3	Fujitsu MBA3147RC 15000 RPM	8.4	10.0	SAS
SSD1	SuperTalent FSD32GC35M	1.3	2.1	SATA
SSD2	MTRON MSP-SATA-7525-032	1.2	2.0	SATA
SSD3	Intel SSDSA2MH160G1GN	0.1	1.2	SATA

4.1 A tailor-made system

The experiments in this chapter uses a tailor-made hardware system consisting of a system under test (SUT) and a set of measurement and monitoring components. The SUT has an Intel Core2 Duo processor and 2 GB of main memory. Both the OS (Ubuntu Linux with kernel version 2.6.31) and the DB engine are installed on an IDE magnetic disk (system disk). The test data (as a DB file) resides on a separate magnetic/flash SSD (data disk). The data disks, listed in Table 4.1, are connected to the system one at a time.

A power measurement device [Schall 09], consisting of ten voltage and current meters, connects the power supply and the system's hardware components, as depicted in Figure 4.1. The device does not tamper the voltages at the power lines, because the hardware components are sensitive to voltage variations. Instead, it measures the current using current transformers with inductive measurement, and the voltage using voltage dividers on a shunt circuit. Both measurements are forwarded over a data bus to the A/D-Converter, which allows the signals being processed by a monitoring PC in real-time.

Using this setup, we are able to precisely measure the energy consumption of the SUT's major parts of interest: the data disk (denoted as SATA, although HDD3 is measured over the SAS power lines), the system disk (denoted as IDE), and the remaining components on the mainboard (denoted as ATX) including CPU and RAM. For a time period T, the average power $\bar{P}(T)$ is given by Formula 4.1:

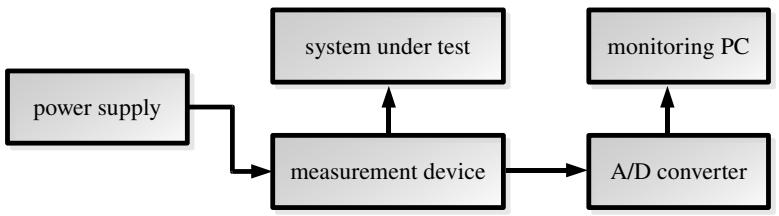


Figure 4.1: Power measurement setup

Table 4.2: Power profile of SUT

power line	components	idle (W)	peak (W)
SATA/SAS	data disk	(see	Table 4.1)
IDE	system disk (Maxtor 6Y080P0)	6.3	12.2
ATX	CPU, RAM, and mainboard chips	19.6	33.5

$$\bar{P}(T) = \frac{1}{T}\int_0^T (v(t) \cdot i(t))dt \tag{4.1}$$

where $v(t)$ and $i(t)$ are the voltage and current as functions of time, and $\int_0^T (v(t) \cdot i(t))dt$ is the work, which is equal to the energy consumption $E(T)$.

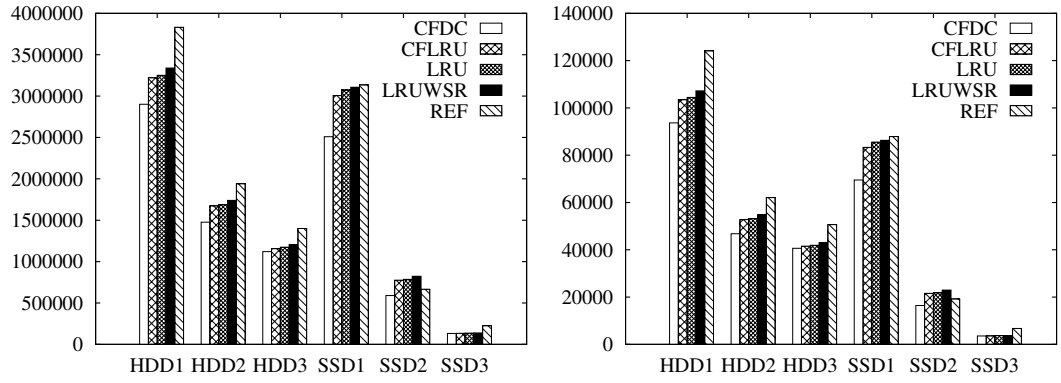
Table 4.2 lists the power profile of the major components of SUT. The idle column refers to the power values when the components are idle (0% utilization) but ready for serving requests, i. e., not in a service-unavailable state such as stand-by or hibernate, while the peak column refers to the power values when the components are under 100% utilization. An interesting observation can be made from this profile: ignoring the data disk, the idle power of the system is 57% of the peak power (25.9 W / 45.7 W). In other words, a lion's share of the power is consumed only to keep the system in a ready-to-service state. Note this "idle share" is even larger in practice, because the peak power can only be reached in some extreme conditions, where all the hardware components are fully-stressed at the same time. With the data disk, this observation still holds, because similar ratios exist between the idle and peak power of HDDs, while SSDs had too low power values to have a large impact on the idle/peak power ratio of the overall system. Furthermore, this observation is not specific to the test machine discussed here, because most state-of-the-art servers and desktop computers have similar idle/peak power ratios.

4.2 Experiments

Our experiments are driven by two OLTP traces: the TPC-C trace (tpcc in Section B.1.1) and the bank trace (see Section B.2).

4.2.1 TPC-C workload

We ran the TPC-C trace for each of the five algorithms and repeated this on each of the devices listed in Table 4.1. The parameter "window size" was *not* tuned—it was set to 0.5 for all related algorithms. The recorded execution times and energy consumptions are shown in Figure 4.2.



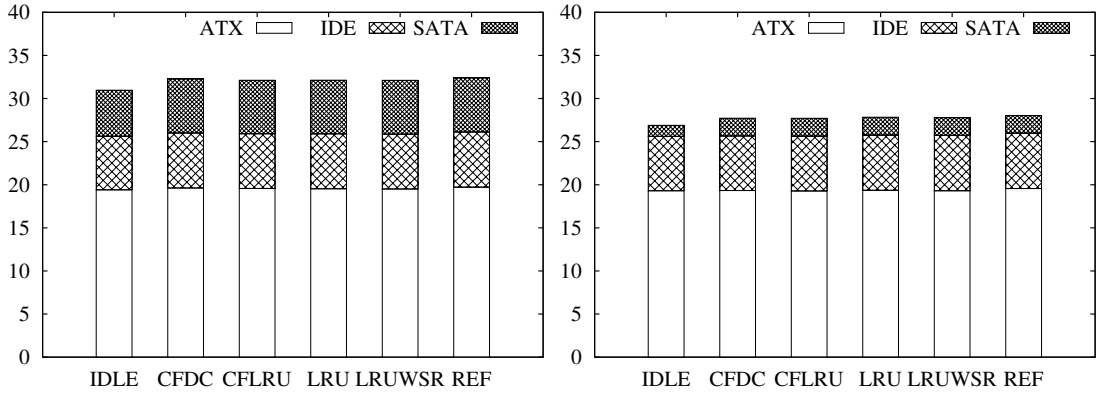
(a) Execution times (ms) **(b)** Energy consumption (J)

Figure 4.2: TPC-C trace performance and energy consumption

Since our workload is I/O-intensive, the device performance has a strong impact on the overall system performance, e. g., SSD3 reduced the average runtime (average over the algorithms) by a factor of 21 compared with HDD1 and by a factor of 19 compared with SSD1, while the corresponding energy-saving factors are 25 and 19, respectively. The CFDC algorithm had the best performance on all of the devices, with a maximum performance gain of 22% over CFLRU on SSD1. Interestingly, even on the magnetic disks, CFDC and CFLRU had a better performance than LRU, which, in turn, outperformed LRUWSR even on the flash SSDs. In most configurations, REF had the longest execution times due to its lower hit ratio and higher number of page flushes, with the exception on SSD2, where its semi-clustered writes are best accommodated by that specific device.

Similar to magnetic disks, it is common for flash SSDs to be equipped with a device cache. Very often, it can not be deactivated or re-sized by the user or the OS, and its size is often undocumented. Obviously, the DB buffer in our experiment should be larger than the device caches. Otherwise, the effect of the DB buffer would be hidden by them. On the other hand, if the DB buffer is too large, the difference between our algorithms would be hidden as well, since a large-enough buffer would hardly provoke any I/O. Based on these considerations, we used a buffer size of 8000 pages (64 MB) for this experiment, because the largest known device cache size is 16 MB of HDD3. The difference between the execution times of the algorithms becomes smaller on SSD3 (see Figure 4.2) due to two reasons: 1. The I/O cost on SSD3 is much smaller than on other devices, yielding the buffer layer optimization less significant; 2. This device has supposedly the largest device cache, since it is the newest product among the devices tested.

Comparing Figure 4.2a with Figure 4.2b, we can see a strong correlation between execution times and energy consumption. In particular, the *best-performing algorithm was also the most energy-saving algorithm.* For example, CFDC reduced



(a) HDD1 **(b)** SSD1

Figure 4.3: Breakdown of average power (W)

energy consumption by 32% on HDD1 and by 71% on SSD3 compared with REF.

This effect is further explained by Figure 4.3, which contains a breakdown of the average working power of major hardware components of interest, compared with their idle power values[1]. Ideally, the power consumption of a component (and the system) should be determined by its utilization. But for both configurations, there is no significant power variation when the system state changes from idle to working. Furthermore, no clear difference can be observed between the various algorithms, although they have different complexities and, in fact, also generate different I/O patterns. This is due to the fact that, independent of the workload, the processor and the other units of the mainboard consume most of the power (the ATX part in the figure) and these components are not *energy proportional*, i. e., their power is not proportional to the system utilization caused by the workload. However, due to the missing energy-proportional behavior of most system components, the elapsed time T of processing the workload almost completely determines the energy consumption $E(T)$ (note, $E(T) = \bar{P}(T) \cdot T$).

4.2.2 Real-life workload

Similar observations were made in our experiment using the bank trace. We ran this trace for each of the devices listed in Table 4.1 and, for each of them, we scaled the buffer size from 500 to 16000 pages. In addition, for each of the algorithms CFDC, CFLRU, and REF, we always ran the trace three times with the window size parameter set to 0.25, 0.50, and 0.75, respectively. Besides a strong correlation

[1]The figures shown for the configurations HDD1 and SSD1 are indicative. Similar observations can be made for the other data disks (IDE and ATX remain constant) and they are, therefore, omitted for brevity.

between performance and energy consumption, we also found that the optimal window size depends on both the buffer size and the device characteristics, for all the three related algorithms.

As an indicative example, we discuss the performance and power figures measured on SSD2 in more detail. For brevity, we choose one best-performing window-size configuration for each of the related algorithms and compare them with LRU and LRUWSR in Figure 4.4. For CFDC and CFLRU, it was 0.75 and 0.25 for REF.

The linear complexity of REF resulted in a higher CPU load compared with other algorithms having constant complexity. This is captured by Figure 4.4c, where the working power of the system is illustrated. The power value of REF goes up with an increasing buffer size, while the power values caused by the other algorithms slightly decrease, because the larger the buffer sizes the more physical I/Os were saved. Note, handling a logical I/O from the buffer is more energy-efficient than doing a physical I/O, because fewer CPU cycles are required and device operations are not involved.

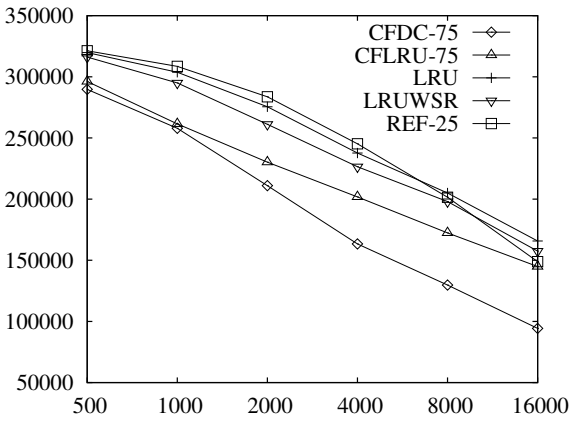
The algorithms' complexity did have an impact on the power of the system, but the time factor had a stronger impact on the overall system energy consumption. For example, enlarging the buffer from 500 to 16000 pages augmented the power value of REF by 2.1% (from 28.04 W to 28.63 W), while the corresponding execution time (Figure 4.4a) decreased by 46.3%. In fact, the effect of the increased CPU load was hidden by the "idle share" of the working power (27.08 W, not shown in the figure). Therefore, the energy consumption curves in Figure 4.4b largely mirror the performance curves of Figure 4.4a. In particular, at a buffer size of 16000 pages, the relative performance gain of CFDC over CFLRU is 54%, while the corresponding energy saving is 55%.

4.3 Summary

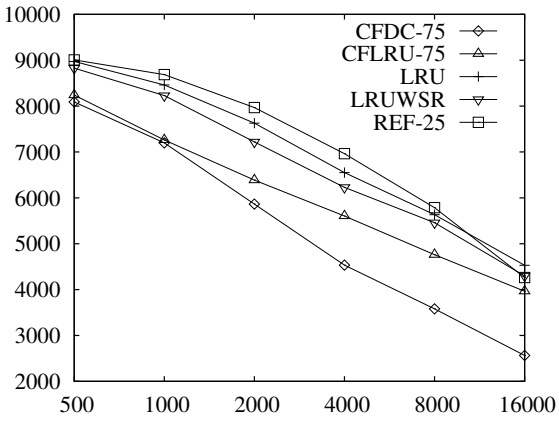
Our experiments reveal a great performance potential of flash SSDs. The use of flash-aware buffer algorithms can further significantly improve system performance on these devices. The CFDC algorithm clearly outperforms the other algorithms in most settings. According to our device sensitivity study, its flash-specific optimizations do not exclude its application in systems based on magnetic disks.

The performance gain can be translated into a significant energy-saving potential due to the strong correlation between performance and system energy consumption. Furthermore, faster I/O reduces the overall system runtime and leaves more opportunities for the system to go into deeper energy-saving states, e. g., stand-by or even off-line.

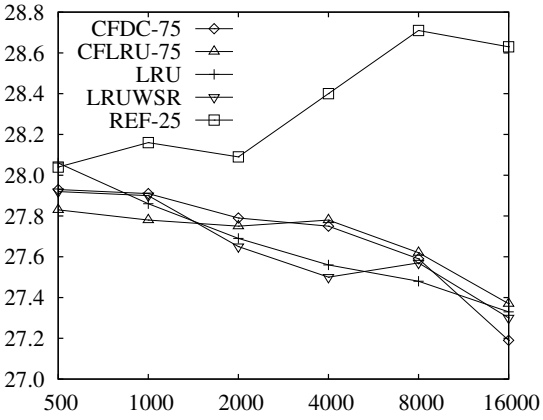
As an important future task of hardware designers and device manufacturers, all system components other than flash SSDs should be developed towards stronger



(a) Execution times (ms)



(b) Energy (J)



(c) Power (W)

Figure 4.4: Performance and energy consumption running the bank trace on SSD2

energy-proportional behavior. Then, the speed or runtime reduction gained by flash use for I/O-intensive applications could be directly translated into further substantial energy saving. As a consequence, energy efficiency due to flash SSD use would be greatly enhanced as compared to magnetic disks.

Chapter 5

Cost-aware buffer management

Reading a page from a flash SSD is extremely fast, whereas a page update can be one or two orders of magnitude slower. For this reason, existing flash-aware buffer algorithms, usually trade physical reads for physical writes to some extent in order to improve the overall I/O performance, as we have already seen in Chapter 3. However, the optimal performance of those algorithms may highly depend on the *extent* of R/W asymmetry of the storage device and the workload, or on the tuning of a parameter, whose optimal value, again, depends on the storage device and workload.

The major contribution of this chapter is the presentation and evaluation of a cost-aware self-adaptive (CASA) buffer management algorithm (originally proposed by us in [Ou 10a]), which makes the trade-off between physical reads and physical writes in a controlled fashion, depending on the extent of R/W asymmetry of the storage device, and automatically adapts itself to varying workloads. CASA is designed for two-tier storage systems based on *homogeneous* storage devices with *possibly* asymmetric R/W costs, i. e., 3TA and hybrid storage (see Section 2.4.2) are not the focus of this chapter.

The remainder of this chapter is organized as follows. Section 5.1 introduces two important concepts, cost ratio and cost awareness, following the discussion of the problems of existing flash-aware algorithms. Section 5.2 presents the CASA algorithm. The related performance study is reported in Section 5.3.

5.1 Introduction

Ideally, the buffer layer should be aware of the characteristics of the underlying storage devices and adapt itself to changes of such characteristics automatically. This is not the case both for classical buffer algorithms and for flash-aware algorithms. Classical buffer algorithms, e. g., LRU, assume that a physical read has about the same cost as a physical write and, therefore, do not apply differing decision policies when read-only or modified pages are replaced in the buffer [Effelsberg 84].

This assumption is reasonable for conventional magnetic disks, but not valid for flash-based devices. On the other hand, existing flash-aware algorithms are not applicable for magnetic disks, because their saving in physical writes usually comes at the expense of a higher number of physical reads or lower hit ratios. In other words, classical buffer algorithms become suboptimal on flash-based devices and flash-aware algorithms can not be used for conventional disk-based systems. This observation reveals an important problem, because magnetic disks are expected to co-exist with flash SSDs for a long period of time due to the lower $/GB cost of magnetic disks, their dominant market position is not likely to be taken over by flash SSDs in the near future.

5.1.1 The parameter tuning problem

Most of the existing flash-aware buffer algorithms assume that a physical write is *much more* expensive than a physical read. However, in fact, the extent of R/W asymmetry of flash-based devices varies from device to device, even among the devices from the same manufacturer. For example, an Intel X25-V SSD achieves 25 K IOPS for reads and 2.5 K IOPS for writes [Intel 10b], while an Intel X25-M SSD reaches 35 K IOPS for reads and 8.6 K IOPS for writes [Intel 10a].

Another common problem of existing flash-aware buffer algorithms is that they ignore the possibly changing *update intensity* (i. e., the percentage of write requests) in the workload while making the replacement decision. Some of them, e. g., CFLRU, leave a parameter for the user to make such an important but difficult performance tuning.

Although other flash-aware algorithms, e. g., LRUWSR and CCFLRU, do not require parameter tuning, their clean-first strategy is carried out only based on the coarse assumption of R/W cost asymmetry and hot-cold detection using the Second Chance algorithm, which, in turn, only approximates LRU. As a consequence, it is difficult for them to reason, when should a cold-dirty page be first considered for eviction over a hot-clean page, and vice versa.

5.1.2 Cost ratio

We use the term *R/W cost ratio*, or *cost ratio* for short, defined as the ratio between the long-term average time used by physical reads and the same used by physical writes, to express the R/W asymmetry of storage devices, i. e., for a storage device, it tells *how much more* expensive is a write compared with a read, and vice versa. For example, the cost ratios of the above mentioned devices are $1 : 10$ and $1 : 4$, respectively, based on their data sheets. Cost ratios deliver important information for making trade-offs between physical reads and physical writes, but they are ignored by existing flash-aware algorithms, i. e., these algorithms are *not aware of the cost*.

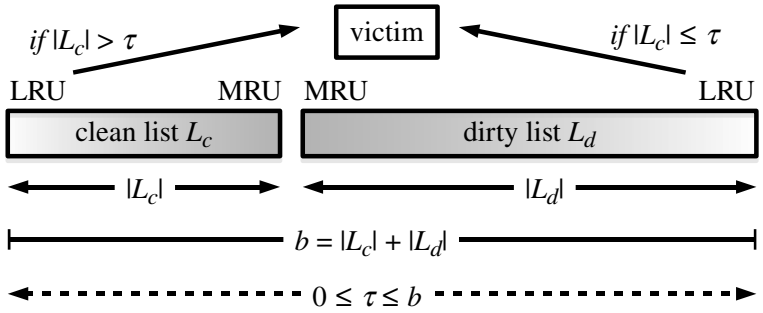


Figure 5.1: CASA dynamically adjusts the size of the clean list and the dirty list

We assume that the cost ratio is known, e. g., it may be derived from IOPS figures or average response times. In Section 5.2.3, we demonstrate an efficient technique that can detect the cost ratios online.

5.2 The CASA algorithm

5.2.1 Overview

CASA manages the buffer pool B of b pages using two dynamic lists: the *clean* list L_c for keeping *clean* pages, that are not modified since being read from secondary storage, and the *dirty* list L_d accommodating *dirty* pages that are modified at least once in the buffer. Pages in either list are ordered by reference recency. Both lists are initially empty, while in the stable state (no empty pages[1] available) we have the following invariants: $|L_c| + |L_d| = b, 0 \leq |L_c| \leq b, 0 \leq |L_d| \leq b$, as illustrated in Figure 5.1.

The algorithm continuously adjusts parameter τ, which is the dynamic target size of L_c, $0 \leq \tau \leq b$. Therefore, the dynamic target size of L_d is $b - \tau$. The logic of adjusting τ is simple: we invest in the list that is more cost-effective for the current workload. If there is a page hit in L_c, we heuristically model the current *relative cost effectiveness* of L_c with $|L_d| \div |L_c|$. Similarly, in case of a page hit in L_d, we model its current relative cost effectiveness with $|L_c| \div |L_d|$. The relative cost effectiveness is considered when determining the magnitude of adjustment for τ (see Section 5.2.2). The simple heuristics used here has low overhead and is effective, as shown by our experiments in Section 5.3.

[1]Empty page refers to the buffer area for a page that has not been used since the start of the buffer manager. Following the convention in the literature, we avoid using the term *buffer frame* (data structure holding a page).

Algorithm 3: CASA

init. : buffer pool B with capacity b; list E of empty pages, $|L_e| = b$; lists L_c and L_d, $|L_c| = 0, |L_d| = 0$; $\tau \in \mathcal{R}$, $\tau \leftarrow 0$

data : request for page p in the form $(op, pageId)$, where $op \in \{R, W\}$; normalized costs $\tilde{C}_R$ and $\tilde{C}_W$ such that $\tilde{C}_R + \tilde{C}_W = 1$ and $\tilde{C}_R \div \tilde{C}_W =$ cost ratio

```
1  if p ∈ B then
2      if p ∈ L_c ∧ op = R then
3          τ ← min(τ + C̃_R × (|L_d| ÷ |L_c|), b) ;
4          move p to MRU position of L_c ;
5      else if p ∈ L_c ∧ op = W then
6          move p to MRU position of L_d ;
7      else if p ∈ L_d ∧ op = W then
8          τ ← max(τ − C̃_W × (|L_c| ÷ |L_d|), 0) ;
9          move p to MRU position of L_d ;
10     else
           // p ∈ L_d ∧ op = R
11         move p to MRU position of L_d ;
12 else
13     victim page v ← null ;
14     if |L_e| > 0 then
15         v ← remove tail of L_e ;
16     else if |L_c| > τ then
17         v ← LRU page of L_c ;
18     else
19         v ← LRU page of L_d ;
20         physically write v;
21     physically read p into v ;
22     p ← v ;
23     if op = R then
24         move p to MRU position of L_c ;
25     else
26         move p to MRU position of L_d ;
27 return p ;
```

5.2.2 The algorithm

Besides the page request, the algorithm (see Algorithm 3) requires as input also the normalized read and write costs, $\tilde{C}_R$ and $\tilde{C}_W$, of the storage device, such that $\tilde{C}_R + \tilde{C}_W = 1$ and $\tilde{C}_R \div \tilde{C}_W =$ cost ratio. They can be derived from the cost ratio,

i. e., important to the algorithm is the extent of the R/W asymmetry, not the exact costs of physical reads and writes.

The magnitude of the adjustment in τ is determined both by the cost ratio and by the relative cost effectiveness. The adjustment is performed in two cases:

Case 1 A logical R-request is served in L_c (line 3 of Algorithm 3);

Case 2 A logical W-request is served in L_d (line 8 of Algorithm 3)

In Case 1, we increase τ by $\tilde{C}_R \times (|L_d| \div |L_c|)$. Note $|L_c| \neq 0$, since it is a buffer hit. The increment combines the "saved cost" of this buffer hit $\tilde{C}_R$ and the relative cost effectiveness $|L_d| \div |L_c|$. Similarly, in Case 2, we decrease τ by $\tilde{C}_W \times (|L_c| \div |L_d|)$.

In case of a buffer fault (line 12–26), if there is no empty page available, τ guides the decision from which list to select the victim page (line 16 and 19). The actual sizes of both lists are also influenced by the clean/dirty state of requested pages. The clean/dirty state of a requested page p is decided by its previous state in the buffer, i. e., in which list it resides, and the current request type (R or W). If the state of the requested page p is clean (after serving the request), p will be moved to L_c (line 4 and 24), otherwise to L_d (line 6, 9, 11, and 26). Therefore, the sizes of L_c and L_d are dynamically determined by τ and the update intensity. Under a workload with mixed R-requests and W-requests, a starvation of one list will never happen, even when $\tau = 0$ or $\tau = b$, whereas under R-only (or W-only) workloads a starvation of L_d (or L_c) is desired and the starved list recovers as soon as the workload becomes mixed again.

5.2.3 Dynamic cost-ratio detection

Being fundamentally different from flash-aware algorithms that require parameter tuning or discriminate clean pages based on fixed rules, the CASA algorithm automatically optimizes itself at runtime, given the knowledge concerning cost ratios. So far, we have assumed that this knowledge is available to the algorithm. It can be provided, e. g., by the device manufacturer or by the administrator. It would be even better if, in the future, devices provide an interface for querying the cost ratio online.

In fact, the elapsed time serving each physical I/O request can be measured online. Therefore, it can be used to derive the cost-ratio information. However, these measurements are subject to severe fluctuations. For example, the latency of a physical read on magnetic disks depends on the position of the disk arm. On flash SSDs, a physical write may trigger a much more expensive flash erase operation or even a garbage collection process involving multiple flash erase operations [Bouganim 09]. Therefore, we use an n-point moving average of the measured values to smooth out short-term fluctuations, because only the long-term average cost is of interest. Hence, the average cost of the last n physical reads

Algorithm 4: Clustered write with frequency-based filtering

```
   data: physical-write request for page p, cluster table T_c
 1 physically write p ;
 2 cluster c ← lookup p's cluster in T_c ;
 3 if c exists then
 4     foreach q ∈ c ∧ q ≠ p do
 5         if freq(q) ≤ freq(p) then
 6             physically write q ;
 7             remove q from c ;
 8     if |c| = 0 then
 9         remove c from T_c ;
```

(or writes) is used as the basis for the normalized cost $\tilde{C}_R$ (or $\tilde{C}_W$) required by Algorithm 3. Note, no change to the algorithm is needed to use the dynamically detected costs.

Maintaining the moving average requires the last n measurements to be remembered. Assuming that two bytes[2] are used to store a measured value to remember, e. g., 32,768 values, we need only eight pages (page size = 8 KB) and, in total, 16 pages for both reads and writes. To optimize the moving-average procedure, the measured values can be stored in an array (managed as a FIFO queue). Then, the time complexity of maintaining the moving average is independent of n: for each new measurement, only an array-element update and a few arithmetic operations are involved.

5.2.4 Integrating clustered writes

So far, our discussions in this chapter have not yet addressed the FRW problem, which is important for many types of flash-based devices. As shown in Chapter 3, exploiting efficient write patterns can significantly improve the performance of the buffer manager. Therefore, we develop an approach that integrates the clustered-write technique with the CASA algorithm.

To enable clustered writes in CASA, a cluster table T_c is maintained, which keeps track of the dirty pages and, for a given page p, returns the set of dirty pages that are in the same cluster of p. To integrate the write clustering technique with CASA, we only have to replace Line 20 of Algorithm 3 by Algorithm 4. For distinction, we call the resulting algorithm SAWC (Self-Adaptive Write-Clustered).

[2]Two bytes can store timings ranging from 0 to 65,535 microseconds, which is sufficient to cover all the possible physical I/O cost values. Furthermore, burst values out of this range can be safely ignored.

For a physical-write request with regard to page p, Algorithm 4 not only writes page p, but also identifies and writes some pages in the same cluster of p. This improves the spatial locality of the flash write requests, but, on the other hand, it is obvious that, for the same workload, SAWC requires a larger number of physical writes than CASA does. For this reason, we filter the candidate pages based on their *update frequency* and skip the dirty pages that are more frequently updated than p. Function $freq()$ gives, for page p, its update frequency, which is the number of logical write requests for p while p is in the buffer. This can be simply implemented with a counter associated with each buffer page.

The intuition behind the *frequency-based filtering* is, if page q is more frequently updated than p, flushing q together with p will have no performance gain, because q will likely be updated again soon. In contrast, pages that are less frequently updated will not likely be updated again, flushing them together with p improves the performance due to higher spatial locality.

Unlike the priority queue Q of CFDC, table T_c does not maintain an ordering of the clusters. In our current implementation, it has an ignorable space overhead and its maintenance requires only a constant number of operations.

5.2.5 Implementation issues

Algorithm 3 requires a request in the form of $(op, pageId)$, i. e., the request type must be present. This may not be the case in some systems, where a page is first requested without explicitly claiming the request type and it is read or updated some time later. However, most DBMSs use the classical pin-use-unpin (or fix-use-unfix) protocol [Gray 93] for pages requests. It is easy to use an update flag, which is cleared upon the pin call and set by the actual page update operation. Upon the unpin call, the buffer manager knows the request type by checking this flag.

In practice, page flushes are normally not coupled with the victim replacement process—most of them are performed by background threads. For better clarity, the presented algorithms do not include the asynchronous page-flushing logic, although they are compatible with this technique. For example, line 2 to line 9 of Algorithm 4 can be executed in an asynchronous fashion. Similarly, in the case of CFDC, the background routines can directly use CFDC's dirty queue, where the dirty pages are already collected and clustered.

As a final remark: the time complexity of the algorithms CASA and SAWC is $O(1)$ and both of them require minimal auxiliary data structures. As our experiments will show, they are also very efficient.

Table 5.1: Statistics of the OLTP trace and DSS trace

trace attribute	OLTP	DSS
number of page requests	1,420,613	3,250,972
number of distinct pages	59,782	104,308
min page number	0	220,000
max page number	219,040	325,639
number of reads	1,156,795	3,250,972
number of updates	263,818	0
update percentage	18.57%	0
locality (number of the hottest pages vs. number of requests for them)	11,957 vs. 1,224,613 (20% vs. 86%)	20,862 vs. 2,875,664 (20% vs. 88%)

5.3 Experiments

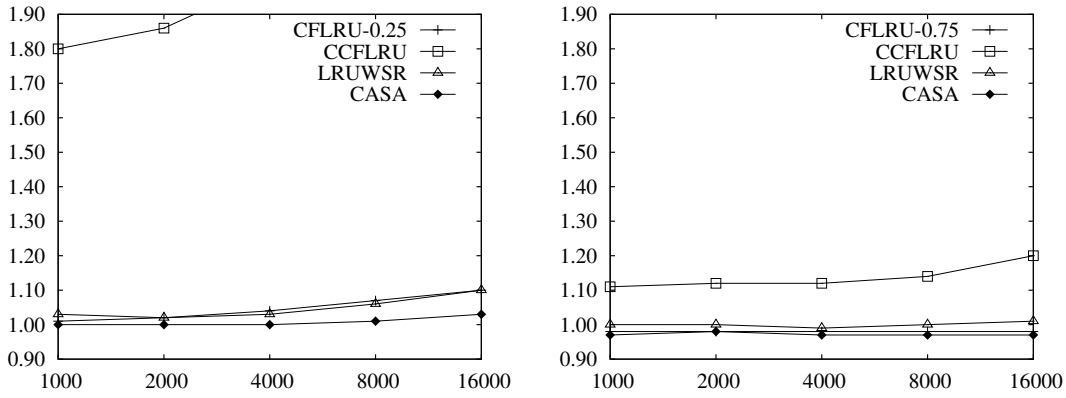
The experiments in this section are organized in two parts. The first part, Section 5.3.1 to Section 5.3.3, focuses on the evaluation of CASA, where it is compared to four algorithms: three representative flash-aware algorithms: CFLRU, CCFLRU, and LRUWSR, and one representative conventional algorithm LRU. The second part, Section 5.3.4, focuses on the comparison among CASA, SAWC, and CFDC.

5.3.1 Changing workload

To examine the behavior of the algorithms under changing workloads, we used a tailor-made trace, called CONCAT. It was built from an OLTP trace and a DSS trace. The OLTP trace recorded a TPC-C workload (the first half of tpcc in Section B.1.1), whereas the DSS trace captured a read-only TPC-H query workload (the first half of tpch in Section B.1.2). The pages referenced by both traces did not overlap. Table 5.1 lists specific statistics of these traces recorded. To simulate changing workloads, we concatenated the first halves of the OLTP and the DSS traces and attached a copy of them at the end, i. e., as final result, the trace CONCAT had the four phases OLTP–DSS–OLTP–DSS and an overall update percentage of 5.6%.

We ran this trace for each algorithm and recorded the number of physical reads and physical writes necessary to serve the logical request stream. Hit ratios can be derived from the number of physical reads and the total number of requests, but of primary concern is the overall I/O cost which, in our simulation, can be presented by the *virtual execution time* t_v:

$$t_v = n_R \times \tilde{C}_R + n_W \times \tilde{C}_W \tag{5.1}$$



(a) t_v for cost ratio 1:1 **(b)** t_v for cost ratio 1:64

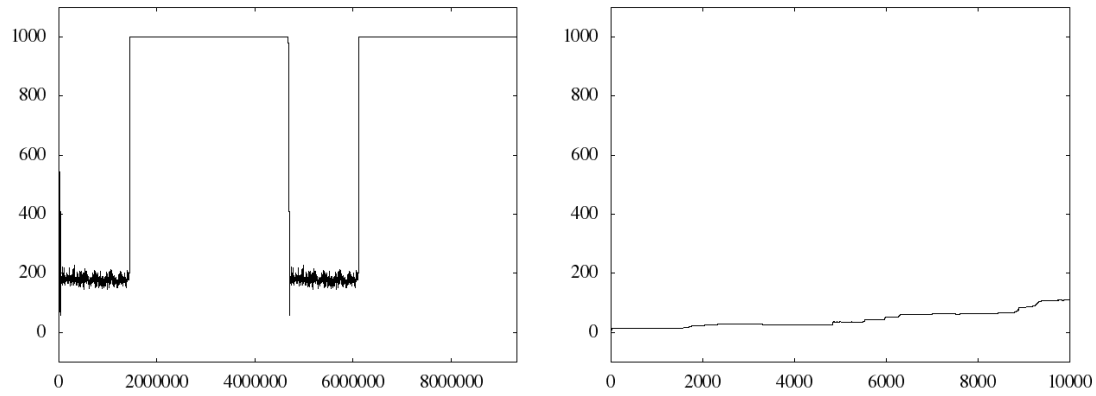
Figure 5.2: Virtual execution times relative to LRU running the CONCAT trace, for R/W cost ratios 1:1 and 1:64. Buffer size scaled from 1,000 to 16,000 pages.

where n_R and n_W are the number of physical reads and physical writes, respectively, and $\tilde{C}_R \div \tilde{C}_W =$ cost ratio.

Figure 5.2 shows the virtual execution time t_v of CASA and the flash-aware algorithms relative to LRU, for cost ratio 1:1 and 1:64. For CFLRU, we repeated the experiment for window sizes $w = 0.25$, $w = 0.50$, and $w = 0.75$, relative to the buffer size, and denoted as CFLRU-0.25, CFLRU-0.50, and CFLRU-0.75. For improved clarity of the visual presentation, we chose to plot its curve only for the best-performing w-setting.

The cost ratio 1:1 (Figure 5.2a) simulates the case of magnetic disks. Here, LRU exhibited the best performance, because it is immune to variations of update intensities in the workload. While the flash-aware algorithms are clearly outperformed by LRU, t_v of CASA closely approaches that of LRU. Although read and write costs are symmetric, the flash-aware algorithms still discriminate clean pages and try to keep dirty pages as long as possible, resulting in an unjustified high n_R and consequently a higher t_v. For example, with 16,000 buffer pages, n_R of CCFLRU is higher than that of LRU by factor three ($4,444,570$ vs. $1,433,996$), while its n_W savings is only about 12% ($111,891$ vs. $128,456$). As a result, its t_v is two times higher than that of LRU (out of the plot area). For CFLRU, the setting $w = 0.25$ had the best performance, because, with the other two settings, the higher numbers of physical reads were not paid off by the savings in physical writes.

With the cost ratio 1:64 (Figure 5.2b), the window size $w = 0.75$ of CFLRU achieved a better performance than its other two settings, because it more greedily trades reads for writes and this behavior paid off here, because physical writes are now much more expensive than physical reads. Having a highly read-intensive



(a) The complete trace **(b)** The first 10,000 requests

Figure 5.3: The size of the clean list changes with the virtual time (request number), reflecting the workload characteristics. The buffer size was 1,000 pages and the R/W cost ratio was 1:64.

workload, the achievable savings in physical writes are rather small. Therefore, the performance advantages of CASA and CFLRU over LRU are not significant and LRUWSR was just comparable to LRU. Nevertheless, CASA outperforms CFLRU, even when the latter used its best setting. Although not significant, its performance advantage is clear without ambiguity. Note, the performance figures are obtained from (discrete-event) simulation, therefore, no measurement error and runtime overhead were introduced.

In Figure 5.3a, we plot the size of the clean list L_c of CASA versus the virtual time (request number). The curve clearly reflects the four phases of the workload: it fluctuates around 200 in the OLTP phases and stays at $1,000$ in the DSS (read-only) phases. The violent oscillation in the OLTP phases is only a visual effect. For example, the slowly climbing curve in Figure 5.3b, reflecting the stage when the clean list gains pages from the empty list during the first 10,000 requests, is squeezed into nearly a vertical line in Figure 5.3a.

5.3.2 Cost awareness

We now study CASA's behavior with various cost ratios, whereas the experiments in Section 5.3.1 focused on the changes in the workload.

For CASA, we ran the real-life trace (see Section B.2) with cost ratios scaling from 1:1 to 1:128. For the remaining algorithms, there is no need to repeat the test for cost ratios 1:4 to 1:128, because only CASA is aware of different cost ratios and can adjust its behavior accordingly. As illustrated by the n_R and n_W figures listed in Table 5.2, CASA used increasingly more physical reads and, in turn, saved more

Table 5.2: Number of physical reads and physical writes running the bank trace using 1,000 and 10,000 buffer pages

	1,000 pages		10,000 pages	
	n_R	n_W	n_R	n_W
CCFLRU	427,012	71,103	237,518	35,642
CFLRU-0.25	393,914	85,818	175,448	43,096
CFLRU-0.50	389,653	79,408	181,223	39,428
CFLRU-0.75	388,917	74,688	195,160	37,188
LRU	403,056	95,849	177,861	51,157
LRUWSR	409,469	90,183	186,306	46,076
CASA 1:1	389,350	77,884	175,985	44,975
CASA 1:4	398,249	72,190	180,558	39,947
CASA 1:16	417,715	71,359	192,149	37,427
CASA 1:64	425,993	71,138	211,015	36,089
CASA 1:128	426,932	71,116	221,344	35,666

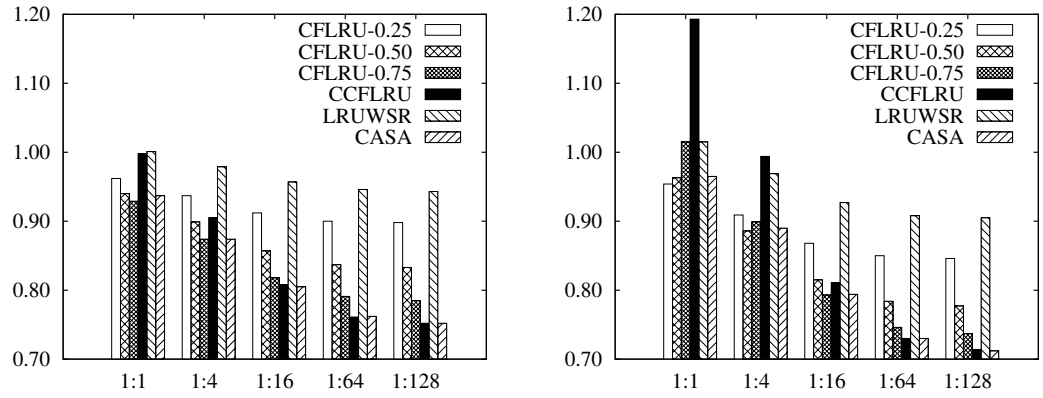
physical writes, while the relative cost of physical writes was increased.

We calculated the t_v's according to Formula 5.1 and show their ratios relative to those of LRU in Figure 5.4. The workload has a relatively high percentage of update requests and, as a result, the flash-aware algorithms could demonstrate their performance advantage over LRU—nearly all of them are below the 1.0 mark in the chart. CASA had clearly the best performance for nearly all settings. For cost ratio 1:128 and buffer size 10,000 pages, it is about 30% faster than LRU. In several cases, it was slightly outperformed by CFLRU with its best w-settings (which were manually optimized). But in real application scenarios, the best w-setting of CFLRU is hard to find: it depends not only on the cost ratio and the update percentage of the workload, but also on the buffer size (This is clear, e. g., by comparing Figure 5.4b for cost ratios 1:1 or 1:4.).

5.3.3 Cost-ratio detection

To test the cost-ratio detection technique presented in Section 5.2.3, we ran the same trace evaluated in Section 5.3.2 using real device accesses to a WD WD1500HLFS HDD (magnetic disk) and an Intel SSDSA2MH160G1GN flash SSD. The physical R/W costs were measured and updated online as described above. We chose $n = 32768$ for the n-point moving average, because it is large enough to smooth out the short-term fluctuations and its space overhead is small.

The test machine is equipped with an AMD Athlon Dual Core Processor, 3 GB of main memory, and is running Linux (kernel version 2.6.24) residing on a magnetic disk. To avoid the influence of the down-stream caches along the cache

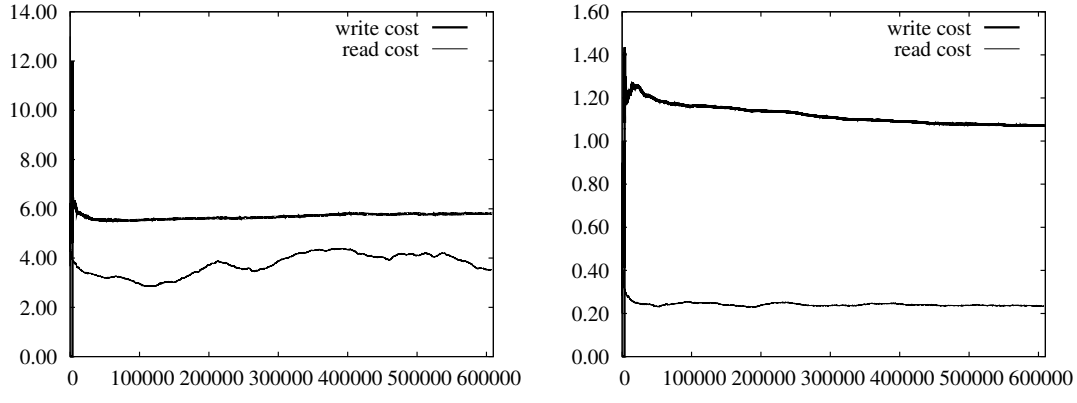


(a) 1,000 pages **(b)** 10,000 pages

Figure 5.4: Virtual execution times relative to LRU running the bank trace for buffer sizes of 1,000 and 10,000 pages. The R/W cost ratio was scaled from 1:1 to 1:128.

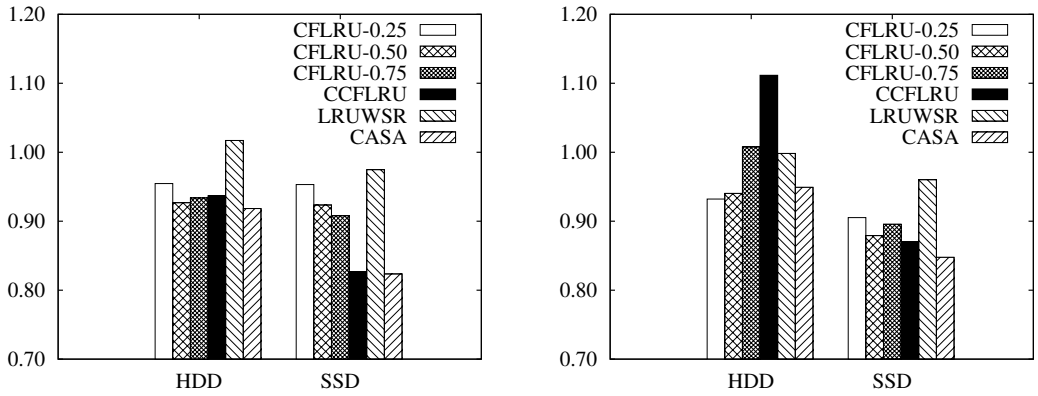
hierarchy, we deactivated the file system prefetching and the on-device write cache, and set the DIRECT_IO flag while accessing the device under test.

Figure 5.5 plots the detected R/W costs for the HDD and SSD devices. Our approach effectively hided the bursts in the measured values and amortized them in the cost ratio, which is about 1:1.5 for the HDD and 1:4.5 for the SSD. As an extra advantage, it captures the cost ratio's long-term variations, which are



(a) R/W costs on HDD (ms) **(b)** R/W costs on SSD (ms)

Figure 5.5: Dynamically detected physical R/W costs vs. the virtual time, using an n-point moving average ($n = 32768$), for running the bank trace using 1,000 buffer pages



(a) 1,000 pages **(b)** 10,000 pages

Figure 5.6: Real execution times relative to LRU running the bank trace on the HDD and the SSD, for buffer sizes of 1,000 and 10,000 pages

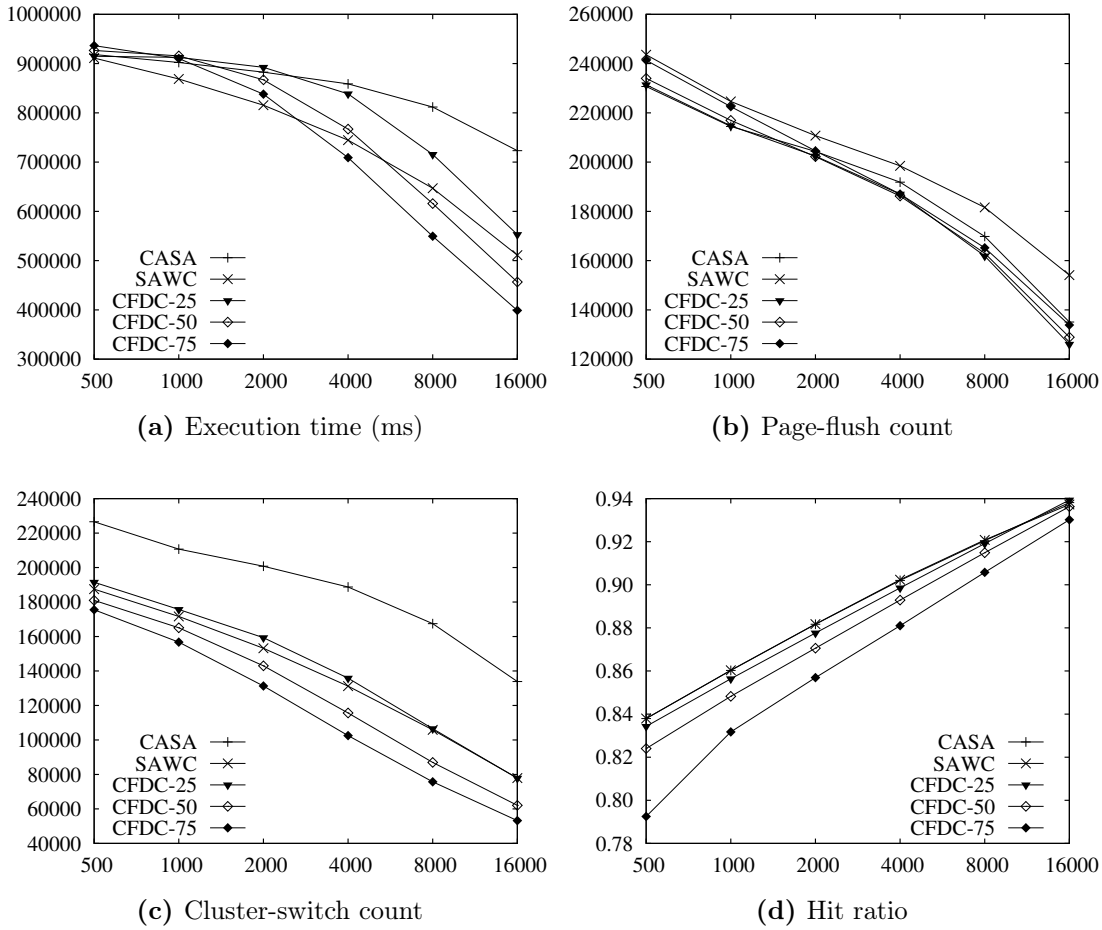
caused by, e. g., the change of read/update patterns (random vs. sequential) in the workload.

We scaled the buffer size from 1,000 to 10,000 pages and measured the real execution times. On the SSD, CASA had the best performance whereas, on the HDD, it was comparable to CFLRU with the best w-settings, but better than the remaining algorithms. Figure 5.6 plots the measured execution times relative to that of LRU for buffer sizes of 1,000 and 10,000 pages. The relative *real* performance shown in Figure 5.6 is roughly comparable with the relative *virtual* performance shown in Figure 5.4 for cost ratios 1:1 and 1:4.

In summary, our experiments so far covering varying workload and various cost ratios have demonstrated the problems of existing flash-aware algorithms: their performance advantage over conventional algorithms heavily depends on the update intensity in workloads and the R/W cost ratio of storage devices. A remarkable example is CCFLRU: under the typical update-intensive OLTP workload (Figure 5.4), it achieved very good performance for highly skewed cost ratios (e. g., 1:64 and 1:128), but suffered from a drastic performance degradation for symmetric R/W costs (1:1). Furthermore, its performance becomes unacceptable under the workload with varying update intensities (Figure 5.2). In contrast, CASA exhibits a consistently good performance for various configurations, both in the simulation and in the real system.

5.3.4 Comparison with CFDC

After comparing CASA with the state-of-the-art flash-aware algorithms, an interesting question remains to be explored: where does CASA (or SAWC) stand



(a) Execution time (ms)

(b) Page-flush count

(c) Cluster-switch count

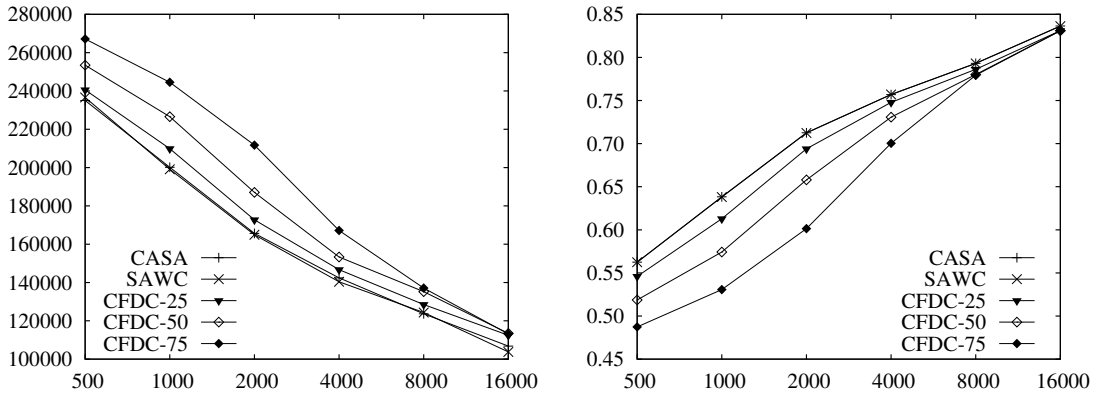
(d) Hit ratio

Figure 5.7: Performance under update-intensive workload

compared with CFDC? To answer this question, we performed experiments using a TPC-C trace (tpcc in Section B.1.1) workload and a TPC-E trace (tpce20 in Section B.1.3). The execution times reported here are measured on a 32 GB MTRON MSP-SATA7525 NAND flash SSD. The remaining metrics, hit ratio, CSC, and page-flush count are independent of devices.

TPC-C workload

Figure 5.7 shows the results of running the TPC-C trace. In general, SAWC achieved a performance comparable to CFDC (Figure 5.7a), without any tuning effort. Compared with CASA, SAWC required a moderately higher number of physical writes (up to 14% for the 16000-page setting in Figure 5.7b). However, as shown in Figure 5.7c, the corresponding cluster-switch count is only 58% of that of CASA, indicating a much stronger spatial locality of the write requests generated



(a) Execution time (ms) **(b)** Hit ratio

Figure 5.8: Performance under read-intensive workload

by SAWC. The clustered writes greatly improved the performance of CASA: with a reduction of execution time by $\sim 30\%$ for the 16000-page setting (Figure 5.7a). In terms of hit ratio, CASA and SAWC are very close (Figure 5.7d).

TPC-E workload

In contrast to the TPC-C trace which has an update percentage of 18.7%, the workload represented by the TPC-E trace is highly read-intensive: it has only 17 k logical writes out of 2.6 million page requests (update percentage is less than one percent). Therefore, the performance impact of the number of physical writes and the cluster-switching count is ignorable and we only show the execution time and hit ratio in Figure 5.8, because the hit ratio (Figure 5.8b) dominates the execution time (Figure 5.8a), whereas CSC and page-flush count have ignorable impact on the execution time. The advantage of self-adaptiveness of CASA and SAWC is clear here: they achieve a higher hit ratio than CFDC, whose three window-size settings are all sub-optimal for this nearly read-only workload (For a read-only workload, the optimal window size should be zero.).

5.4 Summary

The problem of buffer management for storage devices with asymmetric I/O costs is of great importance with emerging flash SSDs. In this chapter, we focused on a common problem of typical flash-aware buffer algorithms: the parameter tuning problem. As solution, we proposed to use the R/W cost ratio to capture the R/W asymmetry of those devices and presented a cost-aware self-adaptive algorithm called CASA.

Our experiments have shown that CASA is efficient and can adapt itself to various cost ratios and to changing workloads, without requiring manual tuning. Our solution is not limited to flash-based storage devices, but should be generally applicable to block-oriented storage devices with asymmetric I/O costs.

Our experiments also included a comparison between CASA (and SAWC) and the CFDC algorithm presented in Chapter 3, which reveals some rules for choosing between these two algorithms in the practice. The CFDC algorithm is suitable for environments where the workload is relatively write-intensive and its characteristics does not change frequently. In such cases, the administrator can better optimize the buffer performance by tuning the parameter λ. For environments with frequently changing workloads, the self-adaptive algorithm CASA or SAWC is a better choice.

Chapter 6

Energy efficiency and architecture

Chapter 3 and Chapter 5 focused on buffer management algorithms, which are a key issue of a two-tier storage system. Efficient use of flash memory requires not only algorithmic improvements, but also a reconsideration of the architecture of storage systems. In this chapter, we study the use of flash memory from an architectural perspective.

For a conventional storage system following the two-tier architecture (2TA, see Section 2.4.2), where a buffer pool at the top tier accelerates page requests to and from the HDD-based bottom tier, the capacity of the expensive and energy-inefficient RAM-based buffer pool often becomes the bottleneck of scaling, with an increasing amount of data to be accommodated at the bottom tier. Despite fast random access, flash devices, e. g., flash SSDs, are still too expensive to be the primary storage solution.

In terms of performance and per-GB price, flash memory and flash devices perfectly bridge the gap between DRAM and magnetic HDDs, as indicated by the figures shown in Figure 1.2. These figures strongly suggest a *three-tier architecture* (3TA, Section 2.4.2), where flash is used in the intermediate tier as a page cache of considerable size, while inexpensive HDDs (or even low-end flash SSDs) are employed at the bottom tier to accommodate our ever-increasing demand of storage capacity. With such a storage hierarchy, the capacity of the RAM-based top tier could be kept relatively small, because a larger amount of pages can be cached on the flash media, which is still much faster than HDDs. To justify the move from a (disk-based) 2TA to such a (flash-incorporated) 3TA, a few questions need to be answered:

Q1 Will the cost of adding the intermediate tier be justified by performance improvements?

Q2 Can we achieve the goal of improving performance while saving energy at the same time?

The major contribution of this chapter is giving answers to the questions

Q1 and Q2 by performing an extensive empirical study, where the performance and energy consumption of 2TA and 3TA are compared under various kinds of workload. In addition, we define the basic interfaces for the three tiers and present a prototype design of such a storage system.

The remainder of this chapter is organized as follows. Section 6.1 discusses related works. Section 6.2 makes some basic assumptions on the design of three-tier storage systems. Section 6.3 presents and discusses baseline algorithms used in our empirical study, which is reported in Section 6.4. Our findings are summarized in Section 6.5.

6.1 Related work

Multi-level caching has been intensively studied in the past. Zhou et al. [Zhou 04] characterized second-level buffer access patterns and proposed a set of algorithms for managing the second-level buffer. Those algorithms are not flash-specific, therefore, their major performance metric is the hit ratio. One of them is implemented in our prototype system and included in our experiments.

Koltsidas and Viglas [Koltsidas 09] identified three page-flow schemes in a three-level caching hierarchy and proposed flash-specific cost models for those schemes. While addressing both theoretical problems and important implementation issues, their focus is the validation of the cost models and the comparison among those schemes. Energy efficiency and a comparison between 2TA and 3TA are not covered in their work.

Narayanan et al. [Narayanan 09] addressed both complete replacement of disks by SSDs as well as use of SSDs as an intermediate layer between disks and DRAM. They compare these architectural variants with 2TA using an offline tool which, given a block-level trace of a workload, suggests the least-cost storage configuration that supports the workload's requirements. They found that replacing disks by SSDs is not a cost-effective option for any of their workloads, due to the higher dollar-per-GB cost of flash SSDs.

Although our goal partially overlaps with that of [Narayanan 09], there are several aspects that distinguish our work fundamentally from theirs: 1. Their traces represent workloads of the bottom tier (block-level traces), whereas our traces represent those of the buffer manager (buffer traces). 2. Our observations are quite different from theirs. For example, they found that fewer than 10% of their workloads can benefit from an intermediate tier based on flash, while in our experiments, 3TA is superior to 2TA in most configurations. 3. Our observations are expected to be more accurate, because traces were not executed in their experiments, but just analyzed by the tool, whereas our traces are actually run in the real systems.

6.2 Basic assumptions

As introduced in Section 2.4.2, a three-tier storage system consists of:

1. The top tier (or buffer layer) T_t managing the RAM-based buffer pool with a capacity of $|T_t|$ pages,
2. The middle tier (or cache layer) T_m managing the flash-based page cache with a capacity of $|T_m|$ pages,
3. The bottom tier (or storage layer) T_b based on HDDs with a total capacity of $|T_b|$ pages.

Considering the relative price and performance ratios of the three types of storage media, e. g., those listed in Table A.1, we assume that:

$$|T_t| \leq |T_m| \leq |T_b| \tag{6.1}$$

Due to these capacity constraints and performance ratios, the hottest pages should be kept in T_t, and T_m should try to keep the hot pages that can not be kept in T_t. As a consequence, replacement policies are required both for T_t and T_m.

T_t supports a typical buffer pool interface, e. g., that of the classical fix-use-unfix protocol [Gray 93]. Both T_m and T_b provide the interface of reading or writing a page, identified by its logical page number. Each tier only uses the interface provided by the tier directly below it, i. e., there is no cross-tier dependency. In particular, in a three-tier storage system, T_t never accesses T_b directly.

However, because T_m and T_b basically have the same interface, T_m can be implemented as an optional tier. When T_m is not present, T_t directly accesses T_b. In that case, 3TA degenerates to 2TA. Such a degeneration is practically used for our experiments in Section 6.4.

For both architectures, we assume that T_t follows two basic principles: *demand paging* and *write back*. Consequently, we have the following two *invariants*, which are independent of the algorithm and implementation of T_m and valid for both 3TA and 2TA:

I1 T_t calls the read(p) function at the tier directly below it, if page p is not present in T_t and a page request for p is to be served by T_t (page fault in T_t).

I2 T_t calls the write(p) function at the tier directly below it, if page p is to be evicted from T_t and p is dirty (modified at least once after entering T_t).

6.3 Baseline algorithms

T_t and T_b are basically the same as in the conventional two-tier disk-based storage system. For this reason, we only present the replacement algorithms for the

Algorithm 5: LOC read page from T_m

```
data: read request for page p, list of cache slots L_s, directory H
1  cache slot c ← lookup p in H ;
2  if c ∈ T_m then
3      read p from cache slot c ;
4      move c to MRU position of L_s ;
5  else
6      victim cache slot v ← LRU position of L_s ;
7      page q ← the page stored at v ;
8      if v is dirty then
9          read q from cache slot v and flush q to T_b ;
10     read p from T_b and store p at v ;
11     move v to MRU position of L_s ;
12     update H by replacing entry (q, v) with entry (p, v) ;
13 return p ;
```

management of T_m in the following: the *Local* (LOC) algorithm and the *Global* (GLB) algorithm.

In both algorithms, a list of cache slots L_s with $|L_s| = |T_m|$ is maintained in an LRU fashion. A *cache slot* represents a portion of the cache, which corresponds to the size of a page. A cache slot uses a bit to represent the clean/dirty status of the page. Furthermore, a directory H is maintained, mapping currently cached pages to their corresponding cache slots.

6.3.1 The LOC algorithm

In the LOC algorithm, T_m is managed *locally* in an LRU fashion, without requiring extra knowledge from T_t. The procedure of reading a page from T_m is shown in Algorithm 5. An important difference to a main-memory LRU cache is that flushing a page involves first reading the page from flash and then writing it to the storage. Writing a page p to T_m involves finding its cache slot c via H and storing p at c. If p is not found in T_m, it will be written to T_b immediately.

Because $|T_t| \leq |T_m|$ and LOC only has local knowledge, it is possible that some or even all pages in T_t are doubly cached in T_m. However, pages in T_t are not necessarily in T_m, due to different page reference behavior at different tiers. Note, references to T_m are consequences of buffer faults in T_t.

Algorithm 6: GLB evict page to T_m

data: request for evicting page q, list of cache slots L_s, directory H, bottom tier T_b

```
victim cache slot v ← LRU position of L_s ;
page s ← the page stored at v ;
if v is dirty then
    read s from cache slot v and flush s to T_b ;
store q at v ;
move v to MRU position of L_s ;
update H by replacing entry (s, v) with entry (q, v) ;
```

6.3.2 The GLB algorithm

The GLB algorithm is first introduced in [Zhou 04]. Here, we examine this algorithm in a flash context. The GLB algorithm follows the exclusive scheme [Koltsidas 09], i. e., no page is ever cached in T_t and T_m at the same time. For better comprehension, we assume the replacement policy in T_t is also LRU, without loss of generality. Based on this assumption, we can think of a *global* logical LRU list L_g, consisting of the LRU list of T_t at its MRU end, and the LRU list of T_m at its LRU end.

Reading a page p from T_m is requested upon a page fault in T_t (see I1). In case of a cache hit in T_m, p is moved from T_m to T_t (H and L_s are updated accordingly). In case of a cache miss, p is read directly from T_b to T_t, avoiding doubled caching in T_m. In both cases, a page q is evicted from T_t to T_m. After being read, p becomes the MRU page in T_t (also in L_g).

To "maintain" the logical list L_g, page q currently evicted from T_t should become the MRU page in T_m. Therefore, we have to extend the interface of T_m by a new function *evict* called by T_t for passing evicted clean pages to T_m. Note, a write request is called on T_m, only when the evicted page is dirty (see I2). The procedure of processing a write or evict request for page q is the same (shown in Algorithm 6): if the LRU slot v of L_s is dirty, flush the page pointed by it, store q at v, move v to the MRU position, mark v dirty if q is dirty, and update H.

6.3.3 Discussion

Given the same workload, the global cache hit count (total number of buffer hits in T_t and T_m) of GLB is expected to be higher than that of LOC, because the effective cache size of the latter is smaller, due to doubled caching in T_t. However, in GLB, the number of flash writes equals the number of T_t page evictions. This is OK for a RAM-based second-level buffer, but it is an issue for flash media in terms of both performance (see Section 6.4) and lifespan [Gal 05].

For both algorithms in our current implementation, the dirty pages in T_m (whose cache slots are marked dirty) are flushed to T_b when the system is shutdown, for the sake of consistency. A simple improvement leveraging the non-volatility of flash can be made here: we can just materialize the content of H at shutdown and rebuild H at startup[1], without flushing the "dirty" pages in T_m (Note T_m is non-volatile). This technique not only speeds up the shutdown procedure, but also shortens the warm-up phase of the system, because the hot portion of the pages are likely already in T_m, ready for immediate access. For the LOC algorithm, pages in T_m are up-to-date at restart, *iff* the dirty pages of T_t are flushed before the shutdown of T_m starts. For the GLB algorithm, page sets T_m and T_t are disjunct, therefore, pages in T_m are automatically up-to-date at restart.

6.4 Experiments

To answer the questions Q1 and Q2, we did an extensive empirical study based on a fair comparison between 2TA and 3TA, using buffer traces recorded under various workloads. We first present our simulation-based study using TPC-E, TPC-C, and TPC-H traces (tpce in Section B.1.3, tpcc in Section B.1.1, and tpch in Section B.1.2) before we discuss the experiments ran on real devices using the trace from a real-life application (see Section B.2). Our study on energy consumption is based on the following assumption:

A1 The acquisition cost and power consumption of storage media are linear to their capacity in use.

Assumption A1 might not be valid at a fine granularity, however, it is reasonable when observed at a coarser granularity. For example, if the power of a 4-GB DRAM module is 10 W, according to A1, 0.4 GB of DRAM would consume 1 W, which is not valid, because, as long as the module is working, it consumes 10 W, no matter the remaining 3.6 GB are in use or not. But we can safely say that $4n$ GB of DRAM based on the same model consume $10n$ W.

All experiments were done using our prototype implementation of the 3TA storage system, which can also be easily configured to function as a 2TA system, as described in Section 6.2. For both architectures, our test program only communicates with T_t by sending the logical page requests delivered by the traces to its buffer manager, which manages the T_t buffer pool using the replacement policy LRU. All experiments start with *cold* T_t and T_m buffers. The time used to flush the dirty pages at shutdown is included in the measurements.

In our experiments, we scaled the size parameter b (in number of pages) logarithmically. For 2TA, b is the size of the buffer pool, i. e., $|T_t| = b$, while for 3TA, we set $|T_t|$ and $|T_m|$ as follows:

[1]The byte size of H is much smaller compared to that of T_t and T_m.

$$|T_m| = b \times s \tag{6.2}$$

and

$$|T_t| = max(1, \lfloor b - |T_m| \times (M_f/M_r + S_d/S_p) \rfloor) \tag{6.3}$$

where M_f/M_r is the per-GB price ratio of flash to RAM, S_d is the byte size of a directory entry of H, and S_p the page size in bytes. The term $|T_m| \times M_f/M_r$ gives the number of RAM pages that should be reduced to achieve a cost-neutral investment for $|T_m|$ pages of flash memory. The term $|T_m| \times S_d/S_p$ is the number of RAM pages consumed by the directory H for $|T_m|$ flash pages. We call Formula 6.2 and 6.3 the *equi-cost constraints*, because it enforces a fair basis for the comparison among the 2TA and 3TA configurations, i. e., having the same acquisition cost.

The parameter s is used to examine the behavior of 3TA when the size of T_m is scaled. Because $|T_t|$ can not be negative, we have $b - |T_m| \times (M_f/M_r + S_d/S_p) > 0$, which resolves to $s < (M_f/M_r + S_d/S_p)^{-1}$. Together with the constraint in Formula 6.1, we have the practical range of s:

$$1 \leq s < (M_f/M_r + S_d/S_p)^{-1} \tag{6.4}$$

If we ignore S_d/S_p, which is relatively small[2], then we obtain $1 \leq s < M_r/M_f$. We chose the price ratio $M_f/M_r = 0.10$, which is very close to the real price ratios according to Table A.1, the practical range of s is approximately $[1, 10)$. Note, s does not have to be an integer. For a given b, the value of s actually controls how much RAM is traded for flash, observing the equi-cost constraints.

6.4.1 Simulations

For the simulation-based experiments, the *Virtual Execution Time* (t_v) is used as the major performance metrics, defined as:

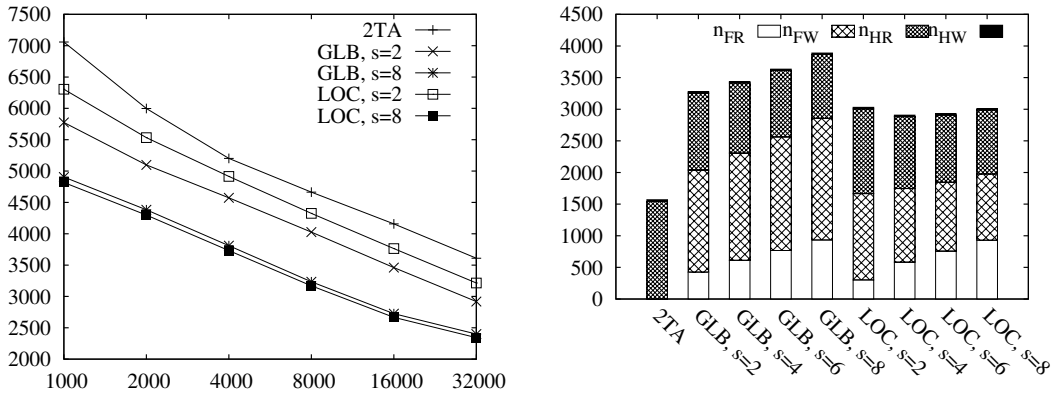
$$t_v = t_m + t_b \tag{6.5}$$

Here, t_m and t_b are the simulated device access times elapsed in T_m and in T_b, respectively. t_m is defined as:

$$t_m = t_{FR} + t_{FW} = n_{FR} \times C_{FR} + n_{FW} \times C_{FW} \tag{6.6}$$

t_{FR} and t_{FW} are the accumulated times for reading from and writing to a flash media, n_{FR} is the number of flash reads, C_{FR} the average cost of a flash read, n_{FW} the number flash writes, and C_{FW} the average cost of a flash write. The flash reads and flash writes here refer to the physical reads from and writes to a flash

[2]In our experiments, the page size S_p is 8192 bytes and the directory entry size S_d is 4 bytes.



(a) t_v (sec) for each $b \in \{1000, ..., 32000\}$ **(b)** Device accesses (x1000) for $b = 1000$ (pages)

Figure 6.1: TPC-E trace performance

device. They are not to be confused with the read and write requests sent to the T_m software. Similarly, t_b is defined as:

$$t_b = n_H \times C_H \tag{6.7}$$

where n_H is the number of disk accesses and C_H the average latency of disk accesses. t_b can be further split into the read portion t_{HR} and write portion t_{HW} such that $t_b = t_{HR} + t_{HW}$. The definition of t_v only considers the costs of accessing the storage media and ignores the CPU cost, because all the algorithms involved have a constant complexity. The inter-tier communication costs are ignored as well, because the dominating cost in the system is the cost of page access, not page transfer. In our simulation, we used the average read and write costs close or equal to those of the middle-class devices in Table A.1, i. e., $C_{FR} = 0.030$, $C_{FW} = 0.120$, and $C_H = 4.5$ (ms).

Figure 6.1a illustrates t_v of running the TPC-E trace using 2TA and 3TA. All 3TA configurations tested significantly outperform the 2TA configuration. For better clarity of the chart, we only show the curves for $s = 2$ and $s = 8$. For the $s = 8$ configuration, LOC reduced the virtual execution time by 32% to 35% (for $b = 1000$ to $b = 32000$), compared with 2TA.

The behavior of 3TA is better explained by Figure 6.1b, where the number of device accesses[3] is compared for $b = 1000$. For 2TA, there is no flash device access, whereas a significant amount of flash device accesses is required for 3TA (Figure 6.1b). For both GLB and LOC, with s scaled from 2 to 8 (thus an increasing $|T_m|$ and decreasing $|T_t|$), the number of flash reads climbs up, indicating a growing number of hits in T_m, and, consequently, the number of disk reads goes down. The

[3]In the simulation, no real device access occurs.

Table 6.1: Energy consumption of the TPC-E trace for $b = 1000$

algo	s	$\lvert T_m \rvert$	$\lvert T_t \rvert$	P_m (mW)	P_t (mW)	$P_m + P_t$ (mW)	t_v (s)	E (J)
2TA		0	1000	0.000	4.121	4.121	7059	29.09
GLB	2	2000	799.02	0.014	3.292	3.307	5776	19.10
GLB	4	4000	598.05	0.029	2.464	2.493	5304	13.22
GLB	6	6000	397.07	0.043	1.636	1.679	5061	8.50
GLB	8	8000	196.09	0.057	0.808	0.865	4905	4.24
LOC	2	2000	799.02	0.014	3.292	3.307	6305	20.85
LOC	4	4000	598.05	0.029	2.464	2.493	5372	13.39
LOC	6	6000	397.07	0.043	1.636	1.679	5024	8.44
LOC	8	8000	196.09	0.057	0.808	0.865	4818	4.17

latter is equal to the number of *global cache misses* (i. e., a page is neither in T_t nor in T_m). Because of the speed difference of flash to disk, the flash accesses introduced at T_m are paid off in terms of overall performance (Figure 6.1a).

As shown in Figure 6.1b, the number of flash writes performed by GLB increases with an increasing $|T_m|$ and a decreasing $|T_t|$, because it depends on the latter, as discussed in Section 6.3.3. In contrast, the increasing $|T_m|$ reduces the number of flash writes performed by LOC. This is due to the reduction of T_m cache misses, because each cache miss requires a flash write (line 10 of Algorithm 5).

Table 6.1 compares the energy efficiency of 2TA and 3TA for $b = 1000$. The $|T_m|$ and $|T_t|$ values in the 3rd and 4th column are calculated according to Formula 6.2 and 6.3. Using these values, we can compute the power value of T_t, based on assumption A1, as follows:

$$P_t = |T_t| \times S_p \times \dot{P}_R \tag{6.8}$$

where $\dot{P}_R$ is the unit power of RAM, having the value 0.503×10^{-9} (W/B) here, derived from the data sheet of RAM2 in Table A.1. The power value of T_m, denoted as P_m, is calculated in a similar way, with $\dot{P}_F = 0.873 \times 10^{-12}$ (W/B), derived from the data sheet of SSD4. Having $P_t + P_m$ and the virtual execution times (t_v), we can then calculate the energy consumption values in the last column. Note that the top tier of 2TA consumed much more energy than those of 3TA (by a factor of six for $s = 8$). Enterprise-class DRAM modules can have a much higher power consumption than RAM2 (by up to a factor of five), i. e., if we use their figures, the energy-saving factor of 3TA will be even higher. Bottom-tier values are not included in the table, because they are of the same size in both architectures.

The results of running the buffer traces of the TPC-C and TPC-H workloads are shown in Figure 6.2 and Figure 6.3. In general, these results confirm our observation concerning the performance advantage of 3TA. For both traces, with b beyond 16000 pages and $s = 8$, the flash cache of 3TA is large enough to

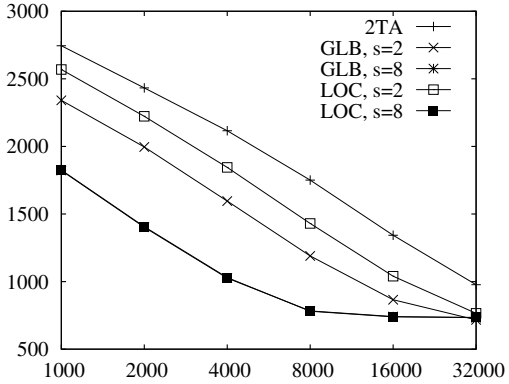


Figure 6.2: TPC-C trace performance: t_v (sec) for each $b \in \{1000, ..., 32000\}$

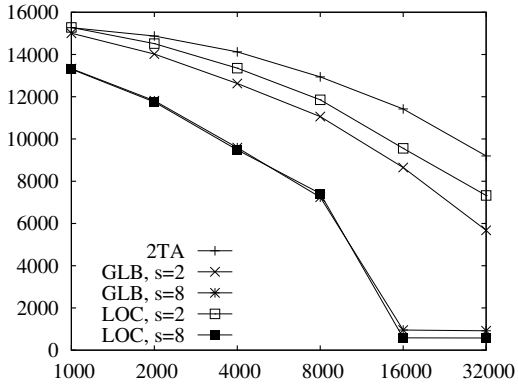


Figure 6.3: TPC-H trace performance: t_v (sec) for each $b \in \{1000, ..., 32000\}$

accommodate all pages of the working sets, which are much smaller than that of the TPC-E trace, therefore, no performance improvement can be observed when b is increased to 32000 pages. The TPC-H trace is highly read intensive, with only 256 page updates out of 6.5 million page requests. That is the reason why the performance of 3TA improves much faster with the growing buffer sizes under the TPC-H workload (Figure 6.3), compared to the TPC-E and TPC-C cases.

6.4.2 Running a real-life trace on real devices

As complement to our simulation-based study, we also experimented with a trace from a real-life application on real devices. Our test machine is equipped with an AMD Athlon Dual Core Processor, 1 GB of main memory, and is running Linux (kernel version 2.6.24). HDD2 from Table A.1 is used as the storage device in T_b, and SSD4 is used as the flash device in T_m. Both devices are accessed as raw devices, i. e., no file system or OS caching is involved, and our storage system has control over the access to the devices.

The measured execution times (wall-clock times) are shown in Figure 6.4. The curves have a shape very similar to that of Figure 6.2, confirming the accuracy of our simulation. An interesting observation can be made here: for $b = 32000$, the execution time in 3TA increases with s, instead of decreasing with it as in most cases tested. In our case here, the 51880 distinct pages addressed by the trace can be completely accommodated by T_t and T_m, for $s = 2$. Therefore, in such a situation, trading RAM for more flash does not further avoid any access to the bottom tier, but reduces the number of buffer hits in T_t and introduces higher numbers of flash accesses, as indicated by Figure 6.5a, where a breakdown of device I/O is presented, with measured values of t_{FR}, t_{FW}, t_{HR}, and t_{HW}. Nevertheless, the energy consumption decreases with an increasing s, as shown in Figure 6.5b,

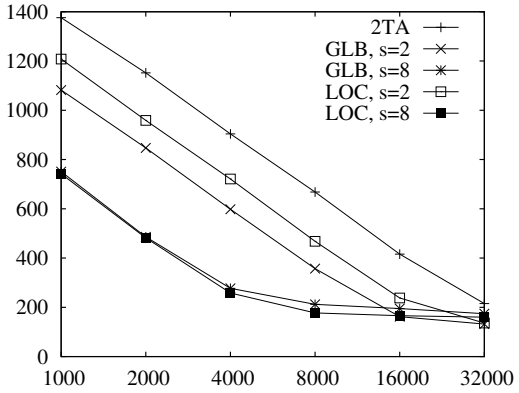


Figure 6.4: Real-life trace performance: execution time (sec) for each $b \in \{1000, ..., 32000\}$

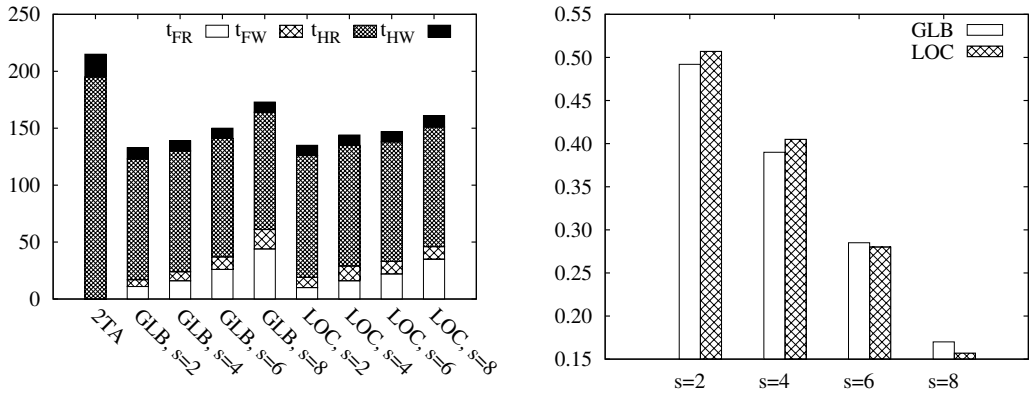
which illustrates the energy consumption figures, similarly obtained as those of Table 6.1.

A question arises here: how much RAM should be traded for flash? Or, in our context, what is the break-even point for s? Analytically determining the optimal value for s is a very difficult problem. However, based on our empirical research, we know that for workloads having a small working set that can be kept in the RAM buffer pool, there is no performance benefit of trading RAM for flash, whereas for workloads with larger working sets that do not fit into main memory, a larger s generally improves performance as well as energy efficiency. Of course, when s closely approaches M_r/M_f, $|T_t|$ becomes 1 (Formula 6.3), i. e., the RAM buffer pool has only one page. Such extreme cases should obviously be avoided in a system configuration. Together with Formula 6.4, our observations can be used as *rules of thumb* in practical applications.

Based on our experiments discussed so far, we can summarize the characteristics of GLB and LOC as follows. For small $|T_m|$, i. e., $|T_m| \sim |T_t|$, GLB achieves higher hit ratios, whereas for large $|T_m|$, i. e., $|T_m| \gg |T_t|$, LOC is generally better, because GLB's advantage in hit ratios becomes insignificant and is eaten up by its higher number of flash writes, which is much more expensive than flash reads. A configuration with $|T_m| \gg |T_t|$ is closer to our goal of managing extremely large amounts of data with high performance and low power consumption.

6.5 Summary

In this chapter, we looked at the flash-based storage systems from an architectural perspective. Our empirical study considered the most important aspects of TCO (Total Cost of Ownership) of a storage system: the acquisition cost and the



(a) Device I/O breakdown (sec) **(b)** Energy consumption relative to 2TA

Figure 6.5: Statistics running the real-life trace for $b = 32000$

operating cost (power cost). Our study gives positive answers to the questions Q1 and Q2 and reveals that we can build a 3TA system which is much faster and much more energy efficient than a 2TA system built with the same acquisition cost, meeting the goals of performance and energy efficiency, which are often considered conflicting, at the same time. With a flash-based page cache accelerating the accesses to the disk-based storage, the amount of expensive and energy-inefficient RAM required by storage systems can be reduced.

In practice, with improved storage system performance, the number of disks, which is sometimes higher than necessary to boost disk I/O throughput (e. g., in a RAID configuration), can generally be reduced, resulting in further operational cost savings due to reduced floor space and cooling requirements.

The performance advantage of 3TA comes from the superior performance/price ratio of flash devices compared with HDDs[4]. This ratio will steadily increase in the next years, while the performance/price ratio of HDDs will remain relatively stable. As a consequence, the performance advantage of 3TA will be even more significant in the future.

LOC and GLB served as the baseline algorithms. No flash-specific optimizations are yet integrated. Techniques such as using different page size at different tiers as those discussed in [Koltsidas 09] could further improve the performance of 3TA. It could also be interesting to examine hybrid configurations of algorithms, e. g., a frequency-based algorithm at one tier and a recency-based algorithm at the other tier. However, future improvements expected for performance and energy-efficiency of 3TA do not conflict with our observations made in this chapter.

[4]Similar observations are made in our experiments using the values of HDD3, the high-end HDD in Table A.1.

Chapter 7

A closer look at flash-based mid-tier caching

As indicated by the study in Chapter 6, using flash memory in the middle tier of a three-tier storage hierarchy is a cost-effective approach for a broad range of workloads. That study was based on cost models and per-capacity prices of DRAM and flash devices. There we assumed the existence of an FTL so that flash memory can be accessed simply as block device and the conventional caching algorithms can be directly applied without modification. Flash-based mid-tier caching is still the focus of this chapter, however, we take a different perspective and look deeper into the internals of flash devices, to exploit their full potential.

7.1 Introduction

Previous studies on flash-based mid-tier caching only considered the *indirect* use of flash memory, i. e., the use of flash memory via an FTL, which makes the native interface of the flash memory transparent to the mid-tier cache manager, as shown in Figure 7.1.

To distinguish a logical page address supported by the top tier and a logical address supported by FTL (see Section 2.3.2), we refer to the latter as *FTL logical address*. Similarly, the logical pages exposed via the FTL interface are referred to as *FTL logical pages*. To keep track of the valid flash page of an FTL logical page, FTL maintains an address-mapping table $m_{\mathrm{FTL}} : A_F \mapsto A_f$, where A_F represents the set of FTL logical addresses (FLAs), i. e., logical page numbers supported by the FTL, and A_f the set of FTL physical addresses (FPAs), i. e., flash page addresses available on the device.

As introduced in Section 2.3.2, there are three categories of FTLs, depending on the map-entry granularity of their m_{FTL} implementations: page-level, block-level, and hybrid mapping. In this chapter, we focus on page-level mapping in favor

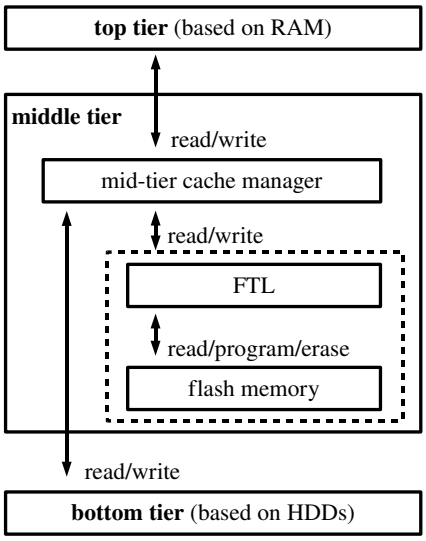


Figure 7.1: Three-tier storage system with indirect use of flash memory by the mid-tier cache. The two components of a flash device, surrounded by the dashed line, appear as a "black box".

of its performance potential, although the problems studied and the basic ideas leading to our solutions are not specific to any FTL implementation.

7.1.1 Problem

Although simplifying the use of flash memory, the indirect approach has some fundamental problems. FTL implementations are usually vendor-specific and proprietary [Gal 05, Chung 09]. The proprietary FTL logic makes it impossible to accurately model or predict the performance of flash-based devices. This is not acceptable for performance-critical applications, because their optimization is often based on the cost model of the underlying storage devices. Furthermore, without direct control over potentially expensive procedures such as GC, the response time becomes indeterministic for the application. It has been reported that GC can take up to 40 seconds[Chang 04], which is not only an issue for applications with real-time requirements, but also intolerable for normal use cases.

For flash-based mid-tier caching, the indirect approach has an even more serious problem related to GC. This problem is explained in the following with the help of a simplified GC procedure, which involves three steps:

1. Select a set of *garbage blocks*, which are blocks containing some invalid pages.
2. Move all valid pages from the garbage blocks to another set of (typically free) blocks and update the corresponding management information.
3. Erase the garbage blocks, which then become free blocks.

If a block has M pages and Step 1 selects only one garbage block, which has v valid pages, then Step 2 consumes v free flash pages, and the procedure increases the total number of free flash pages by $M-v$, at a total cost of $(C_{fr}+C_{fp})\times v+C_{fe}$, where $(C_{fr}+C_{fp})\times v$ is caused by Step 2 and C_{fe} caused by Step 3. The ratio v/M is called *block utilization*. Obviously, GC is more effective and also more efficient for smaller values of v/M, because more free flash pages are gained at a lower cost. Therefore, v/M is an important criterion to be considered for the garbage block selection in Step 1. If the entire flash memory is *highly utilized*, i. e., v/M is statistically close to 1, GC becomes relatively expensive, ineffective, and has to be invoked frequently.

Although for a cache only hot pages should be kept and cold pages should be evicted, FTL must guarantee that each valid page is accessible no matter the page is cold or hot. This means that, during GC processing, cold pages have to be moved along with hot pages (Step 2), while the cold ones, which make v/M unnecessarily high, could actually be discarded from the cache manager perspective. We call this problem the *CPM (cold-page migration)* problem.

More specifically, the CPM problem negatively impacts mid-tier performance in two aspects: 1. The cost of GC, due to the (unnecessary) CPM; 2. The frequency of GC, because, if cold pages are regarded valid, fewer pages can be freed by one invocation of GC and, as a result, the subsequent GCs have to be invoked earlier. Furthermore, the GC frequency is proportional to the number of block erases, which is inversely proportional to the device lifespan due to the endurance limitation.

A similar problem exists when flash SSDs are used as the external storage under a file system. File deletion is a frequent operation, but the information about deleted files is normally kept in OS and not available to the SSD. The latter has to keep even the deleted data valid, at a potentially high operational cost. As solution, a *Trim* attribute for the Data Set Management command has been recently proposed and became available in the ATA8-ACS-2 specification [INCITS 07]. This attribute enables disk drives to be informed about deleted data so that their maintenance can be avoided.

However, no sufficient attention has been paid to the CPM problem, which actually impacts the performance in a more serious way. First, when used in the mid-tier cache, flash devices experience a much heavier write traffic than that of file systems, because pages are more frequently loaded into and evicted from the cache. To flash devices, heavy write traffic means frequent GCs. Second, the capacity utilization of a mid-tier cache is always full (i. e., the flash memory is highly utilized), which makes GC expensive and ineffective (especially for heavy write workloads). In contrast, the GC issue is less critical to file systems, because typically a large portion of their capacity is unused.

7.1.2 Solution

To solve the CPM problem, we develop two approaches, which share the same basic idea: drop cold pages proactively such that the garbage collector sees a reduced block utilization.

1. The first approach, LPD (logical page drop), accesses flash memory *indirectly* via an *extended FTL*, which can be informed about proactively evicted cold pages, and ignores them during GCs.
2. The second approach, NFA (native flash access), manages flash memory in a *native* way, i. e., it implements the out-of-place update scheme and handles GC by the cache manager, without using an FTL.

According to our experiments, both approaches significantly outperform the normal indirect approach, by improving the GC effectiveness and reducing its frequency. For example, NFA reduces the GC frequency by a factor of five, which not only contributes to data access performance, but also implies a greatly extended device life time. In terms of overall performance (IOPS), NFA achieves an improvement ranging from 15% to 66%, depending on the workload.

7.1.3 Contribution

To the best of our knowledge, our work is the first that identifies the CPM problem. Our work is also the first that considers managing flash memory natively in the mid-tier cache. Our further major contributions are:

- We propose two novel approaches for flash-based mid-tier caching: LPD and NFA, both of them effectively deal with the CPM problem.
- Our study shows that, for a flash-based mid-tier cache, our native approach significantly improves the storage system performance while reducing the resource requirements at the same time.
- More importantly, the results of our study urge the reconsideration of the architectural problem of optimally using flash memory in a DB storage system, i. e., whether it should be managed natively by the DBMS or indirectly via the proprietary FTL implementations.

The remainder of this chapter is organized as follows: Section 7.2 discusses related works. Section 7.3 presents and discusses our approaches. Section 7.4 reports our experiments for the evaluation of both approaches. The concluding remarks are summarized in Section 7.5.

7.2 Related work

Before flash memory became a prevalent, disruptive storage technology, many studies, e. g., [Zhou 04, Chen 05, Jiang 07, Gill 08], addressed the problem of multi-level caching in the context of client-server storage systems, where the first-level cache is located at the client side and the second-level (mid-tier) cache is based on RAM in the storage server. However, these studies did not consider the specific problems of a *flash-based* mid-tier cache. Our proposals are orthogonal to and can be combined with their approaches, because their primary goal is to reduce the disk I/O of the storage server, whereas our approaches primarily focus on the operational costs of the middle tier.

In one of the pioneer works on flash-aware multi-level caching [Koltsidas 09], Koltsidas et al. studied the relationships between page sets of the top tier and the mid-tier caches and proposed flash-specific cost models for three-tier storage systems.

Not only academia, but also industry has shown great interest in flash-based mid-tier caching. Canim et al. [Canim 10] proposed a temperature-aware replacement policy for managing an SSD-based middle tier, based on access statistics of disk regions. In [Do 11], the authors studied three design alternatives of an SSD-based middle tier, which mainly differ in the way how to deal with dirty pages evicted from the first tier, e. g., write-through or write-back.

Although flash-specific cost models and their difference to those of traditional storage devices have been taken into account by previous works on flash-based mid-tier caching [Koltsidas 09, Ou 11, Canim 10, Do 11], they commonly only consider the indirect approach, whereas hardly any efforts have been made to examine the internals of flash devices when used as a mid-tier cache. Such efforts fundamentally distinguish our work from the previous ones.

7.3 Our approaches

As introduced in Section 7.1.2, our basic idea is to drop cold pages proactively and ignore them during GCs. A question critical to the success is: to what extent valid but cold pages are dropped? Note, if we drop valid pages too greedily, the benefit will not be covered by the cost of increased accesses to the bottom tier.

Which pages are cold and can be dropped is the decision of the cache manager, whereas the decision, when and how to do GC, is typically made by the FTL—if we strictly follow the architecture of Figure 7.1. Therefore, another important question is how to bring these two pieces of information together.

7.3.1 LPD

The LPD approach is basically an indirect approach, which follows the architecture shown in Figure 7.1. However, to make the basic idea working, we propose, as an extension to the FTL interface, a *delete* operation, in addition to the read and write operations. Similar to the read and write operations, the delete operation is also a logical operation. Upon such a delete request, FTL should mark the corresponding flash page invalid (and update other related management information properly) so that it can be discarded by subsequent GCs.

LPD has some typical cache manager data structures. To tell whether and where a page is cached, it maintains an address mapping table $m_{\mathrm{LPD}} : A_b \mapsto A_F$, where A_b denotes the set of bottom-tier addresses (BTAs) and A_F the set of FLAs. A *cache slot* is a volatile data structure corresponding to exactly one FLA. In addition to the FLA, the cache slot uses one bit to represent the clean/dirty state of the cached page. A *dirty page* contains updates not yet propagated to the bottom tier. Therefore, evicting such a page involves writing it back to the bottom tier. A *free*[1] cache slot is a cache slot ready to cache a new page. Such a slot is needed when a read or write cache miss occurs, so that the missing page can be stored at the corresponding FLA. Storing the page turns a free cache slot into an *occupied* slot, which becomes free again when the page is evicted.

For the mid-tier cache manager to make use of the extended FTL, the procedure of allocating a free cache slot has to be enhanced by some additional code as shown in Algorithm 7. The piece of code (Line 6 to 10) evicts up to the d coldest pages and instructs FTL to delete them, i. e., *dropping a page* involves evicting it from the cache and deleting it logically via the extended FTL. Page dropping happens after the standard logic of cache replacement (Line 4 to 5), which is only required when there is no free cache slot available.

An example of LPD is shown in Figure 7.2, where the cache slot with FLA = 1 was just dropped and became free. The corresponding flash page, although containing the latest version of page A (A2 in the figure), was invalidated (shown in grey). If later block 1 is garbage-collected, A2 can be simply discarded.

The tuning parameter d controls how greedily cold pages are dropped. When $d = 0$, LPD degenerates to the normal indirect approach without using the extension. In contrast, when $d > 0$, the d coldest pages are dropped and the same number of cache slots are turned into free slots, ready to be used for the subsequent allocations of free cache slots (Line 1 to 2).

The LPD approach is orthogonal to the cache replacement policy responsible for the victim selection (Line 4 and Line 7), which shall identify the coldest page as per its own definition. In other words, LPD is compatible with other cache management techniques, which can be used to further improve the hit ratio.

[1] There is no connection between free cache slot and free flash page, although both concepts use the word “free” by convention.

Algorithm 7: Allocation of a free cache slot by LPD

data: parameter d, set F of free slots, set S of occupied slots

```
1  if F ≠ ∅ then
2      remove and return one element from F ;
3  else
4      cache slot v ← select and remove a victim from S ;
5      evict the page cached in v ;
6      for 0 to d and S ≠ ∅ do
7          cache slot s ← select and remove a victim from S ;
8          evict the page cached in s ;
9          FTL.delete(s) ;
10         add s to F ;
11     return v ;
```

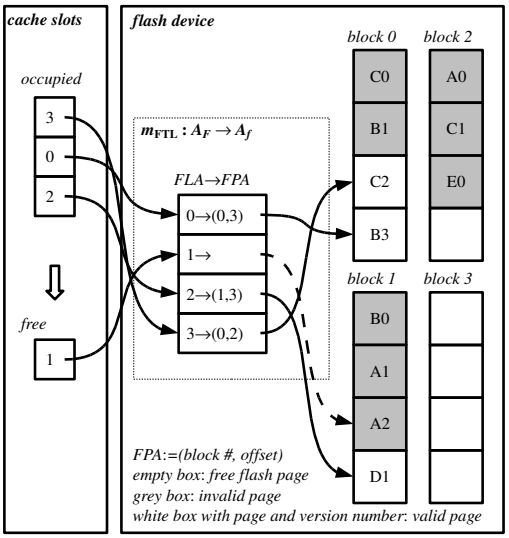


Figure 7.2: Example of logical page drop. Note m_{LPD} is not shown in the figure.

7.3.2 NFA

In contrast to the indirect approaches, NFA does not require an FTL. Instead, it manages flash memory natively. As shown in Figure 7.3, the operations available to the NFA cache manager are read and program of *flash pages*, and erase of *blocks*. Besides the common cache management functionality, NFA has to provide the implementation of an out-of-place update scheme and GC.

For the cache management functionality, NFA maintains a mapping table $m_{\text{NFA}} : A_b \mapsto A_f$, where A_b denotes the set of BTAs and A_f the set of FPAs. Note in LPD (and other indirect approaches), two mapping tables are required: m_{LPD} for cache management and m_{FTL} maintained by FTL as introduced in Section 7.1.

A volatile data structure, *block management structure (BMS)* represents the state of a block. BMS contains two bit vectors, *validity* and *cleanness*, which mark the valid/invalid and clean/dirty states for each flash page in the block. Validity is used by GC processing, whereas cleanness is checked when dropping a page. Furthermore, BMS stores, for each of its valid flash pages, the corresponding BTA to speed up reverse lookups and the corresponding last access time, which is used by the page-dropping logic. The memory consumption of BMS is very low, e. g., using 4 bytes per BTA and another 4 bytes per access time, for a block of 128 pages (each 8 KB), the memory overhead of BMS is 0.1% at maximum.

Following the out-of-place update scheme, both serving a write request and caching a (not yet cached) page consume a free flash page, which is allocated

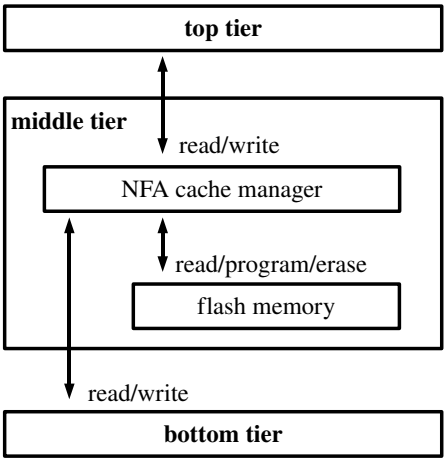


Figure 7.3: NFA architecture

according to Algorithm 8. The algorithm maintains a write pointer wp, which always points to the next free flash page to be programed. After the program operation, wp moves to the next free flash page in the same block, until the block is fully written—in that case, wp moves to the begin of a new free block.

Because GC is a relatively expensive procedure, it is typically processed by a separate thread. NFA uses a low watermark w_l and a high watermark w_h to control when to start and stop the GC processing. GC is triggered, when the number of free blocks is below or equal to w_l, and stops when it reaches w_h, so that multiple garbage blocks can be processed in one batch. The available number of blocks and the high watermark determine the logical capacity of the cache. If we have K blocks with M pages per block, the logical capacity of the cache is: $(K - w_h) \times M$. We say that w_h blocks are *reserved* for GC processing.

Note that the out-of-place update scheme and the use of reserved blocks for GC processing shown in Algorithm 8 are common FTL techniques. They are presented here for comprehension and completeness, because they are now integral to the NFA approach.

The NFA GC procedure (shown in Algorithm 9) is similar to that of a typical FTL in some steps (Line 1, 9, and 10 roughly correspond to Step 1, 2, and 3 of the simplified GC discussed in Section 7.1.1). The difference is due to the dropping of *victim blocks* and cold pages. Victim blocks are selected by a *victim-selection policy* based on the temporal locality of block accesses. In contrast, garbage blocks are selected by a *garbage-selection policy*, for which block utilization is typically the most important selection criterion. Except for these basic assumptions, the NFA approach is neither dependent on any particular garbage selection policy (Line 1) nor on any particular victim selection policy (Line 3).

Dropping of a victim block happens when the selected garbage block is fully utilized, i. e., all its pages are valid. Garbage-collecting such a block would not gain

Algorithm 8: Allocation of a free flash page by NFA

```
   data: pointer wp, set F of free
         blocks, watermarks w_l, w_h
 1 if current block is fully written
   then
 2     wp ← the first flash page of a
       free block ;
 3     if |F| ≤ w_l then
 4         while |F| < w_h do GC ;
 5     return wp ;
 6 else
 7     return wp ← wp + 1 ;
```

Algorithm 9: NFA GC

```
   data: page-dropping threshold t
 1 block b ← select a garbage block ;
 2 if all pages in b are valid then
 3     b ← select a victim block ;
 4     t ← the last access time of b ;
 5 foreach page p ∈ b do
 6     if last access time of p ≤ t then
 7         drop(p) ;
 8     else
 9         move p to a free flash page ;
10 erase b and mark it a free block ;
```

any free flash page. Furthermore, such a garbage block signals that the overall flash memory utilization is full or close to full (otherwise the garbage selection policy would return a block with lower block utilization). Therefore, instead of processing the garbage block, a *victim* block is selected by the victim selection policy (Line 3). The last access time of the block is used to update the page-dropping threshold t. This has the effect that all pages of the victim block are then dropped immediately (Line 6 to 7). The dynamically updated threshold t is passed on to subsequent GC invocations, where the threshold makes sure that valid pages accessed earlier than t are dropped as well.

A flash page managed by NFA has the same set of possible states (shown in Figure 7.4) as those managed by an FTL: free, valid, and invalid. However, NFA has a different set of possible state transitions, e. g., a read or write page miss in an NFA cache can trigger a program operation (for storing the missing page) which changes the state of a free flash page into valid, whereas for FTL, serving a read

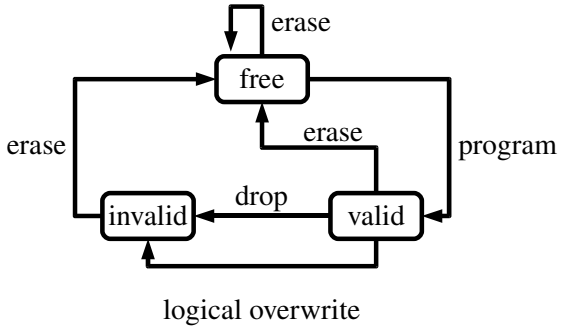


Figure 7.4: NFA flash page states

request does not require a program operation. Obviously, the drop transition is not present in any FTL, either. The semantics of NFA page dropping is similar to that of LPD: the page is evicted (removing the corresponding entry from m_{NFA}, and, if the page is dirty, it is written back to the bottom tier), and then the corresponding flash page is marked invalid.

From the NFA cache manager perspective, the free flash pages for storing pages newly fetched from the bottom tier (due to page faults) are completely provided by the GC procedure in units of blocks. Therefore, NFA does not require page-level victim selection and eviction, which are common in classical caching.

7.3.3 Discussion

Although sharing the same basic idea, the presented approaches, LPD and NFA, are quite different from each other. While NFA directly integrates the drop logic into the GC processing, LPD can only select the drop candidates and delete them logically. LPD can not erase a block due to the indirection of FTL—the intermediate layer required by an indirect approach. Therefore, contiguously dropped logical pages may be physically scattered over the flash memory and LPD has no control when these pages will be garbage-collected, which is again the responsibility of FTL. Such dropped pages can neither contribute to the mid-tier cache hit ratio nor contribute to the reduction of GC cost, until the space taken by them is eventually reclaimed by some GC run. In contrast to the LPD approach, the pages dropped by NFA immediately become free flash pages.

To control how greedily pages are dropped, LPD depends on the parameter d, for which an optimal value is difficult to find, while the "greediness" of NFA is limited to M pages (one block at maximum). However, due to the victim selection based on block-level temporal statistics, the NFA hit ratio could be slightly compromised and a few more accesses to the bottom tier would be required.

7.4 Experiments

To evaluate our approaches, we implemented a three-tier storage system simulator supporting both architectures depicted in Figure 7.1 and Figure 7.3. The simulated flash memory and HDD modules used in our experiments were identical for both architectures.

The workloads used in our experiments originate from three buffer traces, which contain the logical page requests received by DB buffer managers under the TPC-C, TPC-H, and TPC-E benchmark workloads (tpcc100, tpch10, and tpce in Section B.1). Therefore, the buffer traces represent typical, strongly varying workloads at the top tier and our results are expected to be indicative for a broad spectrum of applications.

Table 7.1: Size ratios of the top tier and middle tier relative to the DB size

trace	top tier	middle tier	DB size (max. page number)
TPC-C	2.216%	13.079%	451,166
TPC-H	0.951%	5.611%	1,051,590
TPC-E	0.002%	0.013%	441,138,522

The logical page requests recorded in the buffer traces were sent to the top tier to generate the *mid-tier traces* running the experiments. The top tier, which had a buffer pool of 10,000 pages managed under an LRU replacement policy, served the requests directly from the buffer pool whenever possible. In cases of buffer faults or eviction of dirty pages, it had to read pages from and write pages to the middle tier. The sequences of read and write requests received by the middle tier were recorded and served as the mid-tier traces used in the experiments. We used them to stress the systems containing the middle and bottom tiers. As a result, the access statistics to the flash memory and HDD modules were collected for the performance study.

Three approaches were under comparison: NFA, LPD (with $d = 1024$ unless otherwise specified), and a baseline (BL), which is a mid-tier cache with indirect flash access (but without the delete extension). Our FTL implementation uses page-level mapping, which is the ideal case for the indirect approaches LPD and BL. For all three approaches, the LRU replacement policy was used for selecting victim cache pages (LPD and BL) or victim blocks (NFA), and the *greedy policy* [Rosenblum 92, Kawaguchi 95] is used for selecting garbage blocks, which always selects the block having the least number of valid pages.

For each approach, the flash memory module was configured to have 512 blocks of 128 pages. Similar to [On 11] and [Prabhakaran 08], the low and high watermarks for GC were set to 5% and 10%, respectively. Due to this setting, the logical size of the mid-tier cache is 59,008 pages $((512 - 51) \times 128)$ for all approaches. In Table 7.1, we list the ratios of the top-tier buffer pool size and the logical size of the mid-tier cache relative to the DB size (using the maximum page number as an estimate[2]).

7.4.1 Overall performance

We use the throughput of the middle tier, i. e., the throughput seen by the top tier, as the overall performance metric, which is defined as: throughput $= N/t_v$, where N is the number of page requests in a trace and t_v its execution time, which

[2]For the TPC-E trace, the DB size estimation is coarse because the trace was converted from a proprietary format addressing more than 20 DB files whose sizes and utilization were unavailable to us.

Table 7.2: Operation costs

operation	cost (ms)
C_{fr}	0.035
C_{fp}	0.350
C_{fe}	1.500
C_H	5.500

is further defined as:

$$t_v = t_m + t_b \tag{7.1}$$

t_m represents the total operational cost in the flash memory and t_b the total disk I/O cost. t_m is defined as $t_m = n_{fr} \times C_{fr} + n_{fp} \times C_{fp} + n_{fe} \times C_{fe}$, where n_{fr}, n_{fp}, and n_{fe} are the numbers of flash read, program, and erase operations, and the corresponding costs of flash memory operations are denoted as C_{fr}, C_{fp}, and C_{fe}, as introduced in Section 2.3.1. t_b is similarly defined as $t_b = n_H \times C_H$, with C_H being the cost of a disk access and n_H the number of disk accesses. Therefore, the trace execution time t_v is the weighted sum of all media access operations performed in the middle tier and bottom tier while running the trace.

For the costs of flash operations, C_{fr}, C_{fp}, and C_{fe}, we used the corresponding performance metrics of a typical SLC NAND flash memory of a leading manufacturer, while the disk access cost corresponds to the average latency of a WD1500HLFS HDD [WDC 11]. These costs are listed in Table 7.2.

Figure 7.5 compares the overall performance of the three approaches under the TPC-C, TPC-H, and TPC-E workloads. Our two approaches, NFA and LPD, significantly outperformed BL, and NFA had a clear performance advantage over LPD. For the TPC-C workload, NFA achieved an improvement of 43% and 66% compared with LPD and BL, respectively.

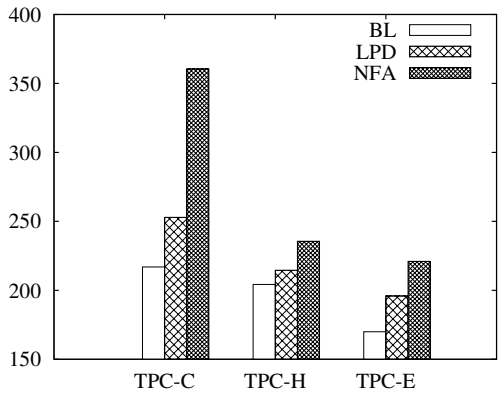
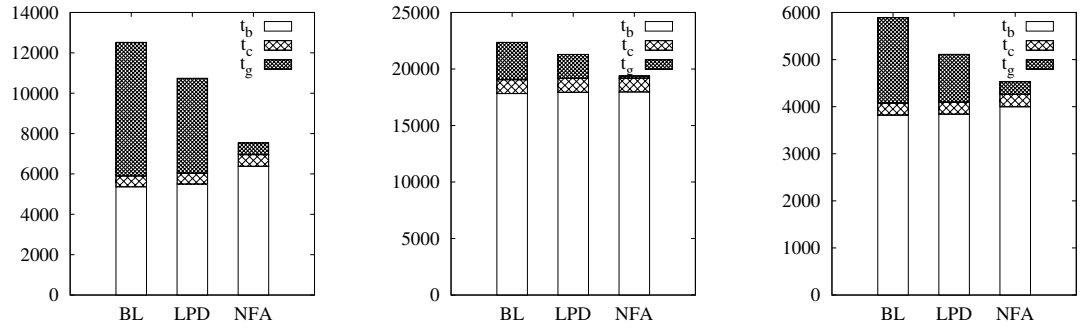


Figure 7.5: Throughput (IOPS)



(a) TPC-C **(b)** TPC-H **(c)** TPC-E

Figure 7.6: Breakdown of the trace execution time (seconds) into the fractions of GC t_g, cache overhead t_c, and disk accesses t_b

The performance improvement of our approaches can be entirely credited to the cost reduction in the middle tier, because both of our approaches do not focus on minimizing disk accesses. In fact, they even had a slightly higher number of disk accesses due to proactive page dropping. It is expected that a small fraction of the dropped pages are re-requested shortly after the dropping, which increases disk accesses. However, this is the small price we have to pay in order to achieve the overall performance gain.

Figure 7.6 confirms our expectation, where we provide a breakdown of the execution times according to Formula 7.1. The mid-tier cost t_m is further broken down into two fractions: the fraction caused by GCs, denoted as t_g, and the fraction caused by normal caching operations (e. g., read operations due to cache hits and program operations due to cache replacements), denoted as t_c, such that

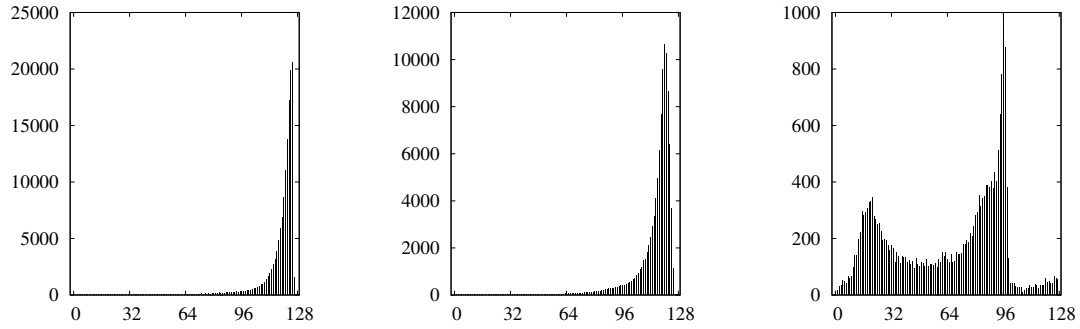
$$t_v = t_m + t_b = (t_g + t_c) + t_b$$

As clearly shown in Figure 7.6, both our approaches effectively improved the GC fraction, without significantly increasing the cost of other two fractions.

The remainder of this section is a detailed analysis of the experimental results. For brevity, we only focus on the performance metrics collected under the TPC-C workload and omit those of the TPC-H and TPC-E workloads, from which similar observations were made.

7.4.2 Detailed analysis

To further understand why our approaches improved the GC efficiency and reduced the number of its invocations, we plotted, in Figure 7.7, the distribution of the number of valid pages in garbage-collected blocks. The majority of blocks garbage-collected in the BL configuration had a number of valid pages very close to 128,



(a) BL **(b)** LPD **(c)** NFA

Figure 7.7: Distribution of the number of valid pages in garbage-collected blocks. A bar of height y at position x on the x-axis means that it happened y times that a block being garbage-collected contains x valid pages. Note the different scales of the y-axis.

which resulted in a poor efficiency of GC. Compared with Figure 7.7a, the dense region in Figure 7.7b is located slightly farther to the left, meaning fewer valid pages in the garbage blocks. For NFA, the majority of garbage-collected blocks had less than 96 valid pages per block, i. e., more than 32 pages could be freed for each garbage block.

Interestingly, in Figure 7.7c, the region between 96 and 127 is very sparse. This is due to the filtering effect (Line 6 to 7 of Algorithm 9). The valid pages in a block either become invalidated due to logical overwrites or are filtered out when they become cold. Therefore, the probability that a block has full or close-to-full utilization is artificially reduced.

For LPD, we ran the trace multiple times scaling d from 0 up to 65,536, which controls how greedily pages are dropped from the cache. For $d = 0$, LPD is equivalent to BL, which does not use the extended FTL and does not drop any pages. For $d = 65536$, it drops all pages from the cache whenever a cache replacement occurs (Line 5 to 11 of Algorithm 7).

Under the same workload, NFA processed 22,106 GCs and achieved a hit ratio of 0.7438 (independent of d). Relative to these values, Figure 7.8 plots the number of GCs and the hit ratio of LPD, with d scaled from 0 to 65,536. For $d = 0$, LPD (and BL, due to equivalence) obtained a slightly higher hit ratio than NFA (by 5.84%), however, its number of GCs was much higher than that of NFA (by a factor of five). For $d = 65536$, although LPD's number of GCs was greatly reduced (still higher than that of NFA by 21%), its hit ratio drastically dropped and became only 63.1% of the NFA hit ratio. Note, we could not find a value for $d \in [0, 65536]$ for LPD, such that the number of GCs is lower and the hit ratio is higher than those of NFA at the same time.

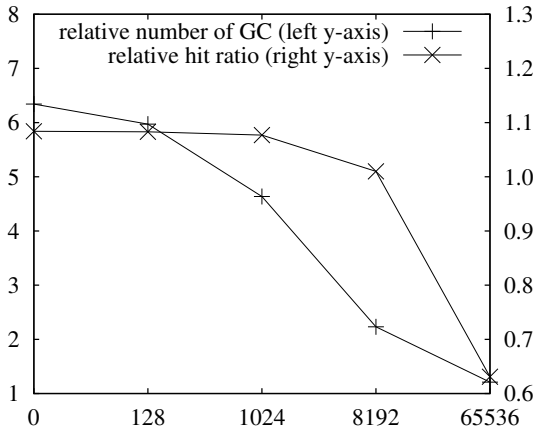


Figure 7.8: Number of GCs and hit ratio of LPD relative to NFA, when d is scaled from 0 to 65,536

7.4.3 Wear leveling

So far, we have not discussed other aspects of flash memory management such as wear leveling and bad block management, which are not the focus of our current work, because they can be dealt with using standard techniques proposed in previous works related to FTL. However, fortunately, our approaches seem to have automatically distributed the erases uniformly to the blocks, as shown in Figure 7.9, where the number of erases for each of the 512 blocks is plotted for all three approaches under comparison.

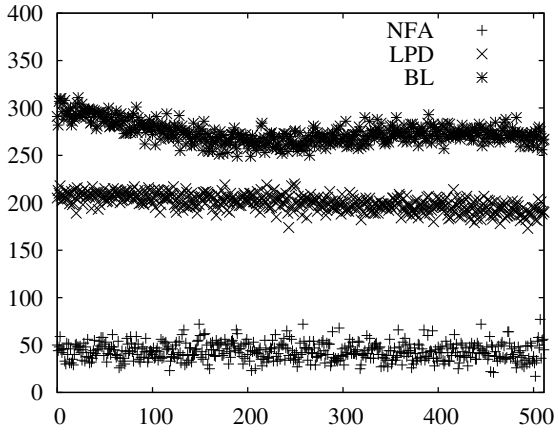


Figure 7.9: Number of erases for each block. Each position on the x-axis refers to a block.

7.5 Summary

In this chapter, we studied the problem of efficiently using flash memory for a mid-tier cache in a three-tier storage system. We identified the problems of using flash memory indirectly, which is the common approach taken by previous works. Among these problems, the most important one is the CPM problem, which not only greatly impacts performance, but also shortens the lifespan of flash devices used in the cache. Our basic idea to solve this problem is to drop cold pages proactively and ignore them during GCs. Based on this basic idea, we proposed two approaches, an indirect one and a native one, that effectively handle the problem, as shown by our experiments. The experiments also demonstrated the gravity of the CPM problem, which is ignored so far by typical indirect approaches represented by the baseline. The cache-specific knowledge (e. g., which pages can be dropped) and the direct control over the flash memory (e. g., when is the GC to be started) is the key to the significant performance gain achieved by NFA, the native approach.

We believe that the optimal use of flash memory in a mid-tier cache can only be achieved when the flash memory is managed natively by the cache management software. For similar reasons, system designers should seriously consider how to natively support flash memory in the database software.

Chapter 8

Conclusion and outlook

8.1 Conclusion

This dissertation studied various problems concerning the design and implementation of flash-based storage systems and proposed effective solutions. It addressed buffer management issues for two-tier storage systems (caching for flash-based databases), where flash-based devices are used as the primary storage. It also examined the efficient use of flash memory as a mid-tier page cache (flash-based caching for databases) to speed-up access to the primary storage, in the context of three-tier storage systems.

All the experiments in this dissertation were performed on our prototype implementation of a database storage engine, which has evolved into a highly configurable system supporting various algorithms at three tiers for both simulations and real-device accesses. We also implemented a flash-device simulator, which can be integrated into the storage engine, either into the middle tier or into the bottom tier, depending on the configuration.

As a basic approach, our studies are based on the cost models of related storage media and devices. Although it is impossible to improve the efficiency of a storage system without making *device-type*-specific optimizations, we kept our cost models as concise as possible to achieve a high degree of *device model* independence, which shall ease the practical application of the proposed methods. The effectiveness of the proposed methods are demonstrated in various empirical studies. The major conclusions based on our studies can be summarized as follows.

- Significant improvements over conventional methods can be achieved by applying flash-specific algorithms and architectures.
- Cost models of flash memory and flash devices are essential for performance optimizations.
- Careful tradeoff must be made between the level of abstraction and performance.

8.2 Outlook

A lot of interesting and challenging topics related to flash memory and emerging new memory technologies are yet to be explored, especially in the context of data storage and management.

Although studied intensively for years, the performance behavior of flash devices still remains partially unpredictable or unexplainable. The major reason responsible for that is the complexity of FTL implementations and its proprietary nature. Tailor-made micro-benchmarks, e. g., those of [Chen 09a, Bouganim 09], are necessary measures towards a better understanding of their sometimes mysterious performance behavior. However, trying to explain those behavior without looking into the internals of the devices is difficult. In our opinion, benchmarking with flash devices alone is not sufficient. To gain a deeper insight into the performance characteristics of flash devices, accurate flash device simulators are necessary as well. The flash device simulator implemented for this dissertation is an initial step towards this direction.

One of the major differences of a flash-based cache to a RAM-based cache is non-volatility. We have discussed a technique leveraging this property to shorten the warm-up phase of the system in Section 6.3.3, which is to be empirically evaluated in the future. The non-volatility of flash memory should be further exploited to speed-up processing of transactions. An interesting work in this area is [On 11], where a novel transaction commit scheme, "flag commit", is proposed, which exploits the unique characteristics of SLC flash memory to improve the performance of transaction processing.

Our work focused on typical database storage systems supporting page-oriented accesses. Flash memory technology can certainly be leveraged also for new types of data management systems emerged in recent years. For example, [Debnath 10] proposed to use flash memory in a high-throughput persistent key-value store; [Bernstein 11] presented a distributed log-structured multi-versioned transactional record manager based on flash memory.

Another important future research direction is related to database storage systems using *Phase Change Memory (PCM)*. PCM is a promising next-generation memory technology, with a range of features interesting to system designers. Similar to flash memory, PCM is *non-volatile* and can *endure* only a limited number of writes. However, unlike flash memory, whose update is subject to the erase-before-write constraint, PCM is *bit alterable* and its update does not require a separate erase operation. Similar to DRAM, PCM is *byte addressable*. However, due to its *higher density*, the cost of PCM can potentially be much lower than DRAM. Therefore, PCM is often considered an alternative to DRAM or an extension of DRAM.

There have been some pioneer works on the use of PCM in database systems. For example, Gao et al. presented a novel logging scheme that exploits the non-

volatility and bit alterability of PCM for efficient transaction logging in disk-based databases [Gao 11]. The use of PCM for logging and recovery is a challenging research topic, because PCM has the potential of drastically changing the design of the logging component, which is one of the most performance-critical and complex components of database engines. At the same time, changes to the logging scheme may not be compatible with existing concurrency control and indexing schemes. Therefore, the impact of PCM on core database technology can be even greater than flash memory.

Appendix A

Storage devices

Table A.1 lists, for each storage media type, the prices and performance figures of a few devices (from low-end to high-end)[1].

Table A.1: Price and performance of storage devices

device	model number	price (EUR/GB)	latency (ms)
RAM1	Kingston KVR667D2D8P5/2G	19.00	$\sim$ 10 ns
RAM2	Kingston KHX1600C9D3B1K2/4GX	19.11	$\sim$ 10 ns
RAM3	Kingston KVR1333D3D4R9S/4G	24.70	$\sim$ 10 ns
SSD1	SuperTalent FSD32GC35M	N/A	0.1
SSD2	MTRON MSP-SATA-7525-032	N/A	0.083
SSD3	Intel SSDSA2MH160G1GN	2.40	0.029
SSD4	Intel SSDSA1MH160G2GN	2.44	0.029
SSD5	Crucial CTFDDAC256MAG-1G1	2.01	0.017
HDD1	WD WD800AAJS 7200 RPM	0.38	15.000
HDD2	WD WD1500HLFS 10000 RPM	0.77	4.500
HDD3	Fujitsu MBA3147RC 15000 RPM	0.76	2.000

[1]We used the sales prices of Internet stores as of November 2010. Prices for SSD1 and SSD2 were already not available (N/A) at that time due to rapid improvements of flash devices. Performance figures are read from or derived from the device data sheets for randomly accessing pages of 4 KB.

Appendix B

Workloads

This appendix describes the most important buffer traces used in our empirical research. The statistics of those traces are listed in Table B.1. They recorded the sequences of logical page requests received by the buffer managers of both open-source and commercial DBMSs. Two categories of workloads were used to collect these traces: TPC[1] benchmarks (Section B.1) and a real-life workload (Section B.2). The TPC benchmarks include TPC-C, TPC-H, and TPC-E. The TPC-C and TPC-H traces were obtained using a PostgreSQL DBMS: our code integrated into its buffer manager enables it to record the logical page requests. The TPC-E traces originate from a trace collected by IBM using DB2 and delivered in binary format.

Either based on these traces or using a random generator, further traces were built to study some special issues. Those special-purpose traces are introduced in the text where they are used.

Table B.1: Buffer traces used in the experiments

trace	number of requests	distinct pages	update intensity	locality[2]
tpcc	2,841,225	88,698	18.72 %	88 %
tpcc100	12,316,654	111,337	16.72 %	90 %
tpch	6,502,242	105,640	0.00 %	72 %
tpch10	10,000,000	1,051,591	7.41 %	87 %
tpce	3,341,738	460,064	1.03 %	87 %
tpce20	2,613,847	150,065	0.67 %	92 %
bank	607,390	51,880	22.51 %	72 %

[1] http://www.tpc.org/

[2] Percentage of requests addressing the top 20% hottest pages.

B.1 TPC benchmarks

B.1.1 TPC-C traces

tpcc This trace recorded the buffer reference string of a 20-minute TPC-C workload with a scaling factor of 50 warehouses and one terminal.

tpcc100 This trace recorded the buffer reference string of a 30-minute TPC-C workload with a scaling factor of 100 warehouses and 100 terminals.

B.1.2 TPC-H traces

tpch This trace recorded a workload generated by read-only TPC-H queries with a scaling factor of one and executed in one stream.

tpch10 This trace recorded the first ten million page requests of a TPC-H workload with a scaling factor of ten and executed in three concurrent streams.

B.1.3 TPC-E traces

tpce This trace is parsed and converted from the binary TPC-E trace using the metadata provided by IBM.

tpce20 This trace is based on the aforementioned TPC-E trace (tpce), but only contains the references to the top-20 hottest files.

B.2 Real-life workload

bank The real-life trace is a one-hour page reference string of the production OLTP system of a bank. It was also used in experiments of, e. g., [O'Neil 93, Johnson 94, Lee 01, Megiddo 03, Li 09]. It contains 607,390 references to 51,880 distinct pages of 8 KB in a DB having a size of 22 GB, addressing 51,870 distinct page numbers. Moreover, this trace exhibits an extremely high access skew, e. g., 40% of the references access only 3% of the DB pages used in the trace [O'Neil 93], and 20% (10,376) of the hottest pages are referenced by 72% (434,702) of the requests. About 23% of the references update the page requested.

Bibliography

[Agrawal 08] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse & R. Panigrahy. *Design tradeoffs for SSD performance.* In USENIX ATC'08, pages 57–70. USENIX Association, 2008.

[Ban 95] A. Ban. *Flash file system*, 4 1995. US Patent 5,404,485.

[Ban 99] A. Ban. *Flash file system optimized for page-mode flash technologies*, 10 1999. US Patent 5,937,425.

[Belady 66] L. Belady. *A study of replacement algorithms for a virtual-storage computer.* IBM Systems journal, vol. 5, no. 2, pages 78–101, 1966.

[Bernstein 11] P. Bernstein, C. Reid & S. Das. *Hyder–A Transactional Record Manager for Shared Flash.* In 5th Biennial Conf. on Innov. Data Syst. Research (CIDR), Asilomar, CA, USA, pages 9–20, 2011.

[Beyer 05] K. Beyer, F. Özcan, S. Saiprasad & B. Van der Linden. *DB2/XML: designing for evolution.* In SIGMOD, pages 948–952. ACM, 2005.

[Birrell 07] A. Birrell, M. Isard, C. Thacker & T. Wobber. *A design for high-performance flash disks.* SIGOPS Oper. Syst. Rev., vol. 41, no. 2, pages 88–93, 2007.

[Bouganim 09] L. Bouganim, B. Jónsson & P. Bonnet. *uFLIP: understanding flash IO patterns.* In CIDR'09, 2009.

[Butt 05] A. R. Butt, C. Gniady & Y. C. Hu. *The performance impact of kernel prefetching on buffer cache replacement algorithms.* In SIGMETRICS, pages 157–168, New York, NY, USA, 2005. ACM.

[Canim 10] M. Canim, G. Mihaila, et al. *SSD bufferpool extensions for database systems.* In VLDB, pages 1435–1446, 2010.

[Chang 04] L.-P. Chang, T.-W. Kuo & S.-W. Lo. *Real-time garbage collection for flash-memory storage systems of real-time embedded systems.* ACM Trans. Embed. Comput. Syst., vol. 3, no. 4, pages 837–863, November 2004.

[Chen 05] Z. Chen, Y. Zhang, Y. Zhou, H. Scott & B. Schiefer. *Empirical evaluation of multi-level buffer cache collaboration for storage systems.* In SIGMETRICS, pages 145–156, New York, NY, USA, 2005. ACM.

[Chen 09a] F. Chen, D. A. Koufaty & X. Zhang. *Understanding intrinsic characteristics and system implications of flash memory based solid state drives.* In SIGMETRICS/Performance, pages 181–192, 2009.

[Chen 09b] S. Chen. *FlashLogging: exploiting flash devices for synchronous logging performance.* In SIGMOD'09, pages 73–86. ACM, 2009.

[Chung 09] T. Chung, D. Park, S. Park, D. Lee, S. Lee & H. Song. *A survey of flash translation layer.* Journal of Systems Architecture, vol. 55, no. 5, pages 332–343, 2009.

[Corbato 69] F. J. Corbato. *A paging experiment with the multics system.* In In Honor of Philip M. Morse, page 217. MIT Press, Cambridge, Mass, 1969.

[Debnath 10] B. Debnath, S. Sengupta & J. Li. *FlashStore: High throughput persistent key-value store.* PVLDB, vol. 3, no. 1-2, pages 1414–1425, 2010.

[Denning 05] P. J. Denning. *The locality principle.* Commun. ACM, vol. 48, no. 7, pages 19–24, July 2005.

[Dijkstra 68] E. Dijkstra. *Letters to the editor: go to statement considered harmful.* Communications of the ACM, vol. 11, no. 3, pages 147–148, 1968.

[Ding 07] X. Ding, S. Jiang, F. Chen, K. Davis & X. Zhang. *DiskSeen: exploiting disk layout and access history to enhance I/O prefetch.* In USENIX ATC, pages 20:1–20:14, Berkeley, CA, USA, 2007. USENIX Association.

[Do 11] J. Do, D. DeWitt, D. Zhang, J. Naughton, et al. *Turbocharging DBMS buffer pool using SSDs.* In SIGMOD'11, pages 1113–1124. ACM, 2011.

[Effelsberg 84] W. Effelsberg & T. Härder. *Principles of database buffer management.* ACM TODS, vol. 9, no. 4, pages 560–595, 12 1984.

[Estakhri 99] P. Estakhri & B. Iman. *Moving sequential sectors within a block of information in a flash memory mass storage architecture*, July 1999. US Patent 5,930,815.

[Gal 05] E. Gal & S. Toledo. *Algorithms and data structures for flash memories.* ACM Computing Surveys, vol. 37, no. 2, pages 138–163, 2005.

[Gao 11] S. Gao, J. Xu, B. He, B. Choi & H. Hu. *PCMLogging: reducing transaction logging overhead with PCM.* In CIKM'11, pages 2401–2404, 2011.

[Gill 08] B. Gill. *On multi-level exclusive caching: offline optimality and why promotions are better than demotions.* In USENIX, pages 1–17. USENIX Association, 2008.

[Gottstein 11] R. Gottstein, I. Petrov & A. Buchmann. *SI-CV: Snapshot Isolation With Co-Located Versions.* In TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC'11), in conjunction with VLDB'11, August 2011.

[Gray 87] J. Gray & F. Putzolu. *The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time.* In SIGMOD, pages 395–398, New York, NY, USA, 1987. ACM.

[Gray 93] J. Gray & A. Reuter. Transaction processing: Concepts and techniques. Morgan Kaufmann, 1993.

[Gray 06] J. Gray. *Tape is dead, disk is tape, flash is disk, RAM locality is king.* Storage Guru Gong Show, December 2006.

[Gray 08] J. Gray & B. Fitzgerald. *Flash disk opportunity for server applications.* ACM Queue, vol. 6, no. 4, pages 18–23, 2008.

[Grupp 12] L. M. Grupp, J. D. Davis & S. Swanson. *The bleak future of NAND flash memory.* In USENIX FAST'12. USENIX Association, 2012.

[Gupta 09] A. Gupta, Y. Kim & B. Urgaonkar. *DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings.* In Proceedings of the 14th international conference on architectural support for programming languages

and operating systems, ASPLOS '09, pages 229–240, New York, NY, USA, 2009. ACM.

[Härder 83a] T. Härder & A. Reuter. *Concepts for implementing a centralized database management system.* In Int. Computing Symp. on App. Sys. Devel., Nürnberg, pages 28–61. Teubner-Verlag, 1983.

[Härder 83b] T. Härder & A. Reuter. *Principles of transaction-oriented database recovery.* ACM Computing Surveys, vol. 15, no. 4, pages 287–317, 12 1983.

[Härder 05] T. Härder. *DBMS Architecture – Still an Open Problem.* In BTW'05, LNI 65, pages 2–28, Karlsruhe, Germany, March 2005. GI.

[Haustein 07] M. P. Haustein & T. Härder. *An efficient infrastructure for native transactional XML processing.* Data Knowl. Eng, vol. 61, no. 3, pages 500–523, 2007.

[IDC 08] IDC. *The diverse and exploiding digital universe – (an IDC white paper sponsored by EMC).* `www.emc.com/collateral/analyst-reports/diverse-exploding-digital-universe.pdf`, March 2008.

[INCITS 07] INCITS. *Data Set Management commands proposal for ATA8-ACS2 (revision 6).* `http://t13.org/Documents/UploadedDocuments/docs2008/e07154r6-Data_Set_Management_Proposal_for_ATA-ACS2.doc`, 2007. INCITS T13.

[Intel 10a] Intel. *X25-M SSD datasheet.* `http://download.intel.com/design/flash/nand/mainstream/322296.pdf`, 2010. Intel Corp.

[Intel 10b] Intel. *X25-V SSD datasheet.* `http://download.intel.com/design/flash/nand/value/datashts/322736.pdf`, 2010. Intel Corp.

[Jiang 02] S. Jiang & X. Zhang. *LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance.* In SIGMETRICS, pages 31–42, New York, NY, USA, 2002. ACM.

[Jiang 05] S. Jiang. *DULO: An effective buffer cache management scheme to exploit both temporal and spatial localities.* In USENIX FAST'05, pages 101–114. USENIX, 2005.

[Jiang 07] S. Jiang, K. Davis, et al. *Coordinated multilevel buffer cache management with consistent access locality quantification.* IEEE Transactions on Computers, pages 95–108, 2007.

[Jin 12] P. Jin, Y. Ou, T. Härder & Z. Li. *AD-LRU: an efficient buffer replacement algorithm for flash-based databases.* Data & Knowledge Eng., vol. 72, pages 83–102, 2012.

[Jo 06] H. Jo, J. Kang, S. Park, J. Kim & J. Lee. *FAB: flash-aware buffer management policy for portable media players.* Trans. on Cons. Electr., vol. 52, no. 2, pages 485–493, 2006.

[Johnson 94] T. Johnson, D. Shasha, et al. *2Q: a low overhead high performance buffer management replacement algorithm.* In VLDB, pages 439–450, 1994.

[Jung 08] H. Jung, H. Shim, et al. *LRU-WSR: integration of LRU and writes sequence reordering for flash memory.* Trans. on Cons. Electr., vol. 54, no. 3, pages 1215–1223, 2008.

[Kawaguchi 95] A. Kawaguchi, S. Nishioka & H. Motoda. *A flash-memory based file system.* In Proceedings of the USENIX 1995 Technical Conference, TCON'95, pages 13–13, Berkeley, CA, USA, 1995. USENIX Association.

[Kim 02] J. Kim, J. M. Kim, S. H. Noh, S. L. Min & Y. Cho. *A space-efficient flash translation layer for CompactFlash systems.* IEEE Transactions on Consumer Electronics, vol. 48, no. 2, pages 366–375, May 2002.

[Kim 08] H. Kim & S. Ahn. *BPLRU: a buffer management scheme for improving random writes in flash storage.* In USENIX FAST'08, pages 239–252. USENIX, 2008.

[Kim 12] B. Kim. *Commercial SSD products – status quo and next (invited talk).* In DASFAA'12, 2nd Int'l Workshop on FlashDB, LNCS. Springer, 4 2012.

[Koltsidas 08] I. Koltsidas & S. D. Viglas. *Flashing up the storage layer.* VLDB Endow. Arch., vol. 1, no. 1, pages 514–525, 2008.

[Koltsidas 09] I. Koltsidas & S. D. Viglas. *The case for flash-aware multi-level caching.* Rapport technique, University of Edinburgh, 2009.

[Koomey 07] J. Koomey. *Estimating total power consumption by servers in the US and the world.* `http://sites.amd.com/de/Documents/svrpwrusecompletefinal.pdf`, February 2007.

[Lee 01] D. Lee, J. Choi, et al. *LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies.* Trans. on Computers, vol. 50, no. 12, pages 1352–1361, 2001.

[Lee 07a] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park & H.-J. Song. *A log buffer-based flash translation layer using fully-associative sector translation.* ACM Trans. Embed. Comput. Syst., vol. 6, no. 3, July 2007.

[Lee 07b] S. Lee & B. Moon. *Design of flash-based DBMS: an in-page logging approach.* In SIGMOD'07, pages 55–66. ACM, 2007.

[Leventhal 08] A. Leventhal. *Flash storage memory.* Communications of ACM, 2008.

[Li 09] Z. Li, P. Jin, et al. *CCF-LRU: a new buffer replacement algorithm for flash memory.* Trans. on Cons. Electr., vol. 55, pages 1351–1359, 2009.

[Mathis 06] C. Mathis, T. Härder & M. Haustein. *Locking-aware structural join operators for XML query processing.* In SIGMOD, pages 467–478. ACM, 2006.

[Megiddo 03] N. Megiddo & D. S. Modha. *ARC: a self-tuning, low overhead replacement cache.* In USENIX FAST'03. USENIX, 2003.

[Mohan 92] C. Mohan, D. J. Haderle, et al. *ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging.* ACM Trans. Database Syst., vol. 17, no. 1, pages 94–162, 1992.

[Mtron 08] Mtron. *Solid state drive MSP-SATA7525 product specification*, 2008. MTRON Ltd.

[Narayanan 09] D. Narayanan, E. Thereska, et al. *Migrating server storage to SSDs: analysis of tradeoffs.* In EuroSys, pages 145–158. ACM, 2009.

[Nath 07] S. Nath & A. Kansal. *FlashDB: dynamic self-tuning database for NAND flash.* In Int. Conf. on Information Processing in Sensor Networks, pages 410–419, 2007.

[On 11] S. T. On, J. Xu, B. Choi, H. Hu & B. He. *Flag Commit: supporting efficient transaction recovery on flash-based DBMSs.* IEEE Transactions on Knowledge and Data Engineering, vol. 99, 2011.

[O'Neil 93] E. J. O'Neil, P. E. O'Neil, et al. *The LRU-K page replacement algorithm for database disk buffering.* In SIGMOD, pages 297–306, 1993.

[Ou 09] Y. Ou, T. Härder & P. Jin. *CFDC: a flash-aware replacement policy for database buffer management.* In SIGMOD Workshop DaMoN, pages 15–20, 2009.

[Ou 10a] Y. Ou & T. Härder. *Clean first or dirty first? a cost-aware self-adaptive buffer replacement policy.* In IDEAS'10, Montreal, QC, Canada, 2010.

[Ou 10b] Y. Ou, T. Härder & D. Schall. *Performance and power evaluation of flash-aware buffer algorithms.* In DEXA, 2010.

[Ou 10c] Y. Ou & T. Härder. *CFDC: a flash-aware buffer management algorithm for database systems.* In ADBIS'10, volume 6295 of *LNCS*, pages 435–449. Springer, 9 2010.

[Ou 11] Y. Ou & T. Härder. *Trading memory for performance and energy.* In DASFAA'11, 1st Int'l Workshop on FlashDB, pages 241–253. Springer-Verlag, 2011.

[Ou 12] Y. Ou, J. Xu & T. Härder. *Towards an efficient flash-based mid-tier cache.* In DEXA'12 (accepted to appear in LNCS). Springer, September 2012.

[Pai 04] R. Pai, B. Pulavarty & M. Cao. *Linux 2.6 performance improvement through readahead optimization.* In Proceedings of the Linux Symposium, volume 2, 2004.

[Park 06] S. Park, D. Jung, et al. *CFLRU: a replacement algorithm for flash memory.* In CASES, pages 234–241, 2006.

[Parnas 75] D. Parnas & D. Siewiorek. *Use of the concept of transparency in the design of hierarchically structured systems.* Communications of the ACM, vol. 18, no. 7, pages 401–408, 1975.

[Petrov 10] I. Petrov, G. Almeida, A. Buchmann & U. Gräf. *Building large storage based on flash disks.* In ADMS'10 (in conjunction with VLDB'10), September 2010.

[Prabhakaran 08] V. Prabhakaran, T. L. Rodeheffer & L. Zhou. *Transactional flash.* In Proceedings of the 8th USENIX conference on operating systems design and implementation, OSDI'08, pages 147–160, Berkeley, CA, USA, 2008. USENIX Association.

[Roberts 09] D. Roberts, T. Kgil, et al. *Integrating NAND flash devices onto servers.* Communications of the ACM, vol. 52, no. 4, pages 98–103, 2009.

[Rosenblum 92] M. Rosenblum & J. K. Ousterhout. *The design and implementation of a log-structured file system.* ACM Trans. Comput. Syst., vol. 10, pages 26–52, February 1992.

[Sacco 82] G. Sacco & M. Schkolnick. *A mechanism for managing the buffer pool in a relational database system using the hot set model.* In VLDB, pages 257–262, 1982.

[Samsung 05] Samsung. *Memory technology and solutions roadmap.* `http://www.samsung.com/us/aboutsamsung/ir/ireventpresentations/analystday/downloads/analyst_20051104_0800.pdf`, 2005. Samsung Analyst Day, Samsung Electronics Co., Ltd.

[Schall 09] D. Schall. Energieeffizienz in Datenbanksystemen - Entwurf einer Meß- und Auswertungsumgebung. Diplomarbeit, Technische Universität Kaiserslautern, September 2009.

[Schiefer 10] B. Schiefer. *DB2 / ISAS and SSD – overview, performance insights, challenges.* DB2 Community Meeting, 2010.

[Seo 08] D. Seo & D. Shin. *Recently-evicted-first buffer replacement policy for flash storage devices.* Trans. on Cons. Electr., vol. 54, no. 3, pages 1228–1235, 2008.

[Tanenbaum 87] A. S. Tanenbaum. Operating systems, design and impl. Prentice-Hall, 1987.

[Tsirogiannis 09] D. Tsirogiannis, S. Harizopoulos, M. Shah, J. Wiener & G. Graefe. *Query processing techniques for solid state drives.* In SIGMOD, pages 59–72. ACM, 2009.

[WDC 11] WDC. *Specifications for the 150 GB SATA 3.0 Gb/s VelociRaptor drive (model WD1500HLFS, WD1500BLFS).* `http://wdc.custhelp.com/app/answers/detail/search/1/a_id/2716`, retrieved on 14th Mar. 2012, 2011. Western Digital Corp.

[Woodhouse 01] D. Woodhouse. *JFFS: the journalling flash file system.* In The Ottawa Linux Symp., 2001.

[Zhou 04] Y. Zhou, Z. Chen, et al. *Second-level buffer cache management.* IEEE Transactions on Parallel and Distributed Systems, vol. 15, no. 6, pages 505–519, 2004.

Curriculum Vitae

Personal information

First name	Yi
Last name	Ou
Date of birth	Oct. 13, 1974
Place of birth	Changsha, Hunan, China

Research

Mar. 2008–today	Doctoral candidate, research group Databases and Information Systems (DBIS), University of Kaiserslautern

Education

Apr. 2003–Jan. 2008	Study of *Computer Science* with emphasis on Enterprise Information Systems, University of Kaiserslautern Acquired academic degree: *Diplom-Informatiker*
Oct. 2002–Jan. 2003	German language courses, Goethe-Institut Peking
Sep. 1993–Jun. 1997	Study of *Industrial Foreign Trade*, Hunan University, China Acquired academic degree: *Bachelor of Engineering*
Sep. 1981–Jun. 1993	Primary and secondary education, Hunan, China

Work experience

Jan. 2007–Apr. 2007	Internship, DFKI GmbH, Kaiserslautern
Nov. 2005–Dec. 2006	Programmer, XTC project, University of Kaiserslautern
Sep. 2004–Dec. 2006	Network administrator, Institut für Technologie und Arbeit, Kaiserslautern
Jul. 1997–Jul. 2002	Assistant (in the first year) and account manager, China Electronics Shenzhen Company, Shenzhen, China