Integrating Flash Memory into the Storage Hierarchy

A DISSERTATION
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Biplob Kumar Debnath

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Advisor: David J. Lilja
Co-Advisor: Mohamed F. Mokbel

October, 2010

Acknowledgements

I would like to thank my advisor Prof. David J. Lilja for his invaluable guidance, encouragements, timely advice and mental support, which greatly helped me to make thesis into a reality. I would like to thank Prof. Mohamed F. Mokbel and Prof. David Du for their invaluable guidance and numerous discussions during the course of my PhD studies.

I would also like to thank Dr. Sudipta Sengupta and Prof. Chris Kim for their willingness to serve on my thesis committee and for their insightful comments and criticisms. In addition, I am very grateful to Dr. Sudipta Sengupta and Dr. Jin Li from Microsoft Research for helping me to demonstrate the innovative applications of the flash-based storage devices.

I thank all faculties and students of the CRIS group for their comments, criticisms, and support which plays an important role to improve various parts of this thesis. Finally, I would like to thank my parents, relatives, and friends for their constant support and encouragements which help me to successfully finish this thesis.

Dedication

To my parents

Abstract

With the continually accelerating growth of data, the performance of storage systems is increasingly becoming a bottleneck to improving overall system performance. Many applications, such as transaction processing systems, weather forecasting, large-scale scientific simulations, and on-demand services are limited by the performance of the underlying storage systems. The limited bandwidth, high power consumption, and low reliability of widely used magnetic disk-based storage systems impose a significant hurdle in scaling these applications to satisfy the increasing growth of data. These limitations and bottlenecks are especially acute for large-scale high-performance computing systems.

Flash memory is an emerging storage technology that shows tremendous promise to compensate for the limitations of current storage devices. Flash memory's relatively high cost, however, combined with its slow write performance and limited number of erase cycles requires new and innovative solutions to integrate flash memory-based storage devices into a high-performance storage hierarchy. The first part of this thesis develops new algorithms, data structures, and storage architectures to address the fundamental issues that limit the use of flash-based storage devices in high-performance computing systems. The second part of the thesis demonstrates two innovative applications of the flash-based storage.

In particular, the first part addresses a set of fundamental issues including new write caching techniques, sampling-based RAM-space efficient garbage collection scheme, and writing strategies for improving the performance of flash memory for write-intensive applications. This effort will improve the fundamental understanding of flash memory, will remedy the major limitations of using flash-based storage devices, and will extend the capability of flash memory to support many critical applications. On the other hand, the second part demonstrates how flash memory can be used to speed up server applications including Bloom Filter and online deduplication system. This effort will use flash-aware data structures and algorithms, and will show innovative uses of flash-based storage.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background and Motivation

Magnetic disk drives continue to be the primary medium for storing data. Over the last three decades, the access time of disks has improved at an annual rate of about 8%, while the processor speed has improved at an astounding rate of 60% per year. Moreover, disk capacity has doubled every 18 months so that the same amount of storage now requires fewer spindles. As a result, processors see significant delays for all disk I/O operations. The time required to read data is very small when the data is found in the DRAM cache typically inserted between the processor and the disk. The latency for a cache miss, however, is dictated by the disk's rotational speed. To reduce latency, the disk platters can be spun faster, such as using 15K RPM disk drives instead of the more conventional 7200 RPM drives. However, this higher rotational speed comes at substantially higher cost and increased power consumption. For example, a 10 TB data set stored on 15K RPM drives would cost approximately $22K and consume 473 watts, while the same data set stored on 7200 RPM drives would cost only about $3K and consume 112 watts [95]. The additional cost and power consumption needed to cut the latency in half is not acceptable for most applications.

With recent technological advances, it appears that flash memory has enormous potential to overcome the shortcomings of conventional magnetic media. In fact, flash memory has rapidly become the primary non-volatile data storage medium for mobile devices, such as cell phones, digital cameras, and sensor devices. Flash memory is

popular for these devices due to its small size, low weight, low power consumption, high shock resistance, and fast read performance [31, 63, 67, 69, 72, 80, 81, 90, 92, 106, 107, 132]. The popularity of flash memory has also extended from embedded devices to laptops, PCs, and enterprise-class servers with flash-based Solid State Disk (SSD) widely considered as a future replacement for magnetic disks. [81, 88, 90, 91, 95, 103]. Market giants Dell, Samsung, and Apple have already launched laptops with only SSDs [11, 27, 70]. Samsung, STec, SimpleTech, Intel, and others have launched SSDs with higher performance than traditional 15000 RPM enterprise disk drives [9, 28, 29, 102]. Enterprise-class SSDs provide unique opportunities to boost the performance of I/O-intensive applications in large datacenters [119, 125].

Flash memory-based storage has several unique features that distinguish it from conventional disk:

- The read/write performance of flash-based storage is much better than magnetic disks. For example, the read and write times for a 2KB data block in a Seagate Barracuda 7200.7 ST3800011A disk are 12.7 ms and 13.7 ms, respectively, while it is 80 μs and 200 μs in a typical flash memory.

- In conventional magnetic disks, the access time is dominated by the time required for the head to find the right track (seek time) followed by a rotational delay to find the right sector (latency time). As a result, the time to read a random data block from a magnetic disk depends primarily on the physical location of that block. In contrast, flash memory-based storage provides uniformly fast random access to all areas of the device independent of its address or physical location.

- In conventional magnetic disks, the read and write times are approximately the same. In flash memory-based storage, in contrast, writes are substantially slower than reads. Furthermore, all writes in a flash memory must be preceded by an erase operation that requires about 1.5 ms, unless the writes are performed on a clean (previously erased) block. Read and write operations are done at the page level while erase operations are done at the block level, which typically has 32 to 64 pages.

- Finally, flash suffers from the *wear-out* problem in which a block can be erased only a limited number of times (typically 100K erase cycles) after which it acts

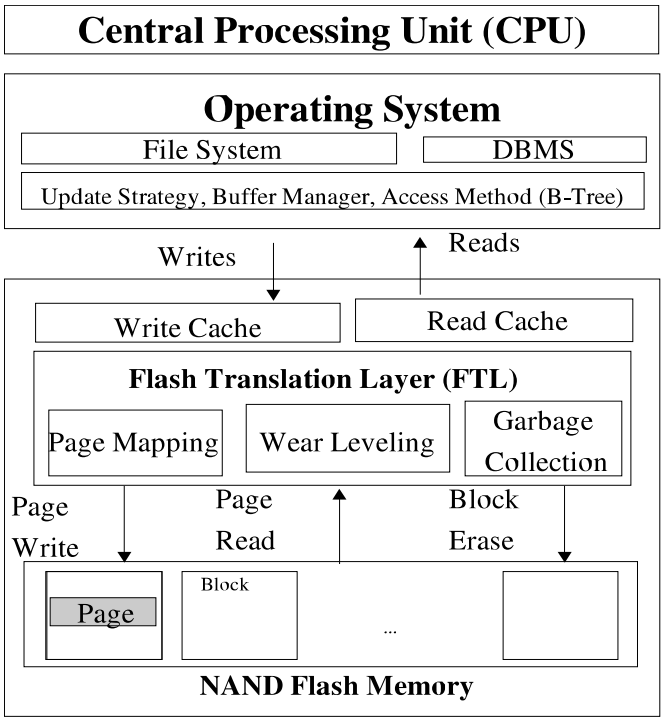


Figure 1.1: SSD Storage Hierarchy

like a read-only device.

The slow write performance and wear-out problems of flash devices are two of the key issues preventing the widespread adoption of flash memory-based storage systems. Figure 1.1 shows a typical storage hierarchy consisting of the CPU, the DRAM containing the operating system and other applications, and a flash-based storage device. Existing file systems and applications, such as database management systems (DBMSs), are heavily optimized to the characteristics of magnetic disk drives. To take full advantage of the performance improvements promised from using flash-based storage devices, these systems will have to adopt new update strategies, buffer management algorithms, and data access methods that take into account the unique characteristics of flash memory described above. On the other hand, to overcome the fundamental performance issues of flash-based storage devices, new architectures, especially new caching algorithms need to be developed. These contributions will greatly help existing applications, as well as new applications, extract the greatest benefit from flash-based storage systems.

1.2 Contributions

This thesis explores new architectures, algorithms, data structures, and innovative solutions for the fundamental issues that limit the use of flash memory-based storage devices in high-performance computing systems. In addition, it demonstrates two innovative applications of the flash-based storage. Providing efficient solutions to flash's fundamental performance issues - i.e., slow write performance and early wear-out - will greatly help legacy applications take advantage of the faster read performance provided by flash memories and their lower energy consumption compared to conventional magnetic disk drives. These applications will be able to use flash memory-based storage systems with no modifications. From the software engineering perspective, this will be a huge gain as it will save tremendous development costs and time. Solving the fundamental performance issues described below will additionally help new applications enjoy higher performance with lower energy costs than is possible using conventional disks. As storage performance is always critical and energy costs are increasing in importance, solving the fundamental issues described below will positively impact every layer of the storage hierarchy. On the other hand, innovative applications demonstrates how flash-memory can be used to overcome some of the I/O bottlenecks faced by magnetic disk drives.

The first part of this thesis will focus on several fundamental issues required to successfully incorporate flash memory into the storage hierarchy: 1) caching algorithms optimized for flash; 2) RAM-space efficient metadata management, and 3) writing strategies. The second part of this thesis will demonstrate two innovative applications of the flash-based storage in server-side applications. These applications take advantage of the unique characteristics of the flash-based storage devices.

1.2.1 Write Caching Algorithm

The performance of the FTL mapping layer and the wear-leveling algorithms are dominated by the access patterns of the write operations (and, to a lesser extent, the read patterns). A non-volatile cache can be used to buffer incoming writes and filter them into a more flash-friendly pattern. This can improve not only the write performance, but can also reduce the number of erasures needed. An important research question is developing appropriate policies to maintain this cache. Traditional caching algorithms are

optimized for the characteristics of magnetic disks. However, since flash-based storage devices exhibit completely different characteristics, current caching policies and algorithms need to be revisited and possibly redesigned completely. This thesis presents a novel write caching algorithm, the Large Block CLOCK (LB-CLOCK) algorithm, which considers 'recency' and 'block space utilization' metrics to make cache management decisions. LB-CLOCK dynamically varies the priority between these two metrics to adapt to changes in workload characteristics [58].

1.2.2 RAM-space Efficient Metadata Management

For page address mapping and garbage collection, FTL needs to maintain various metadata, for example, erase count, block utilization, and age etc. Usually, for the faster access, these metadata are stored on an SRAM-cache in the current flash-based storage device controller [63, 69]. However, due to higher price per byte of SRAM, it is unlikely that SRAM will scale proportionately with the increase of flash capacity [69]. Thus, scaling out metadata that are stored and processed for a large-capacity flash storage, in the limited amount of SRAM becomes very challenging. Since SRAM space is a scarce resource, recent research works focus on reducing metadata stored in the SRAM. Most of the research works [63, 69, 93, 94, 83, 77, 130, 111, 112] try to save SRAM space for storing the page address mapping. For example, recently DFTL [69] proposes a demand based page mapping scheme, which significantly reduces the SRAM space for page mapping [69]. However, very few research works [63, 76] have been conducted to reduce the metadata stored and processed in SRAM for the garbage collection. This thesis addresses metadata management issue by using a sampling-based algorithm [53].

1.2.3 Deferred Update Methodology

While flash-based storage is particularly good for the read-intensive type workloads, such as a decision support system, for example, it can produce poor performance when used for workloads that require frequent updates. Examples of such workloads include online transaction processing, mobile applications, and spatio-temporal applications. To accommodate for such write intensive workloads, we aim to improve write performance and longevity of flash-based storages by designing an efficient scheme for handling data

updates. In particular, this thesis propose a new hierarchical *deferred update* methodology that significantly improves the update processing overhead with the cost of only a few flash blocks [54]. The main idea is to process all update transactions through an intermediate two-level hierarchy consisting of an update memo and log pages. The update memo is a set of pre-erased blocks which is used as a scratch space for writing updated records. Once the memo is full, we flush it to corresponding log pages. Finally, we flush the log pages into corresponding data pages. Data retrieval is performed in the opposite manner by visiting data blocks, followed by log pages and then update memos. By doing so, we avoid in-place updates that would result in erase operations, which are not only high latencies operations, but also decrease the longevities of flash-based storage devices.

1.2.4 Flash-based Bloom Filter

The bloom filter is a probabilistic data structure that provides a compact representation of a set of elements. However, the size of the bloom filter in some applications is too big to fit in main memory. When sufficient RAM space is not available, it may be necessary to store the bloom filter in magnetic disk-based storage. However, as bloom filter operations are randomly spread over its length, the slow random access (seek) performance of disks, of the order of 10 milliseconds, becomes a huge bottleneck. The use of independent hash functions destroys any locality that may be present in the element space, hence, even using a RAM based cache does not help to improve bloom filter performance. Since flash memory has faster read performance, of the order of 10-100 μsecs in currently available Solid State Drives (SSDs), it is a viable storage option for implementing bloom filters and striking tradeoffs between high cost of RAM and fast bloom filter access times. Flash memory, however, provides relatively slow performance for random write operations (vs. reads and sequential writes). This thesis proposes a flash-based bloom filter which is aware of flash memory characteristics and exhibits superior performance. Currently, this work is under review for the publication [57].

1.2.5 Flash-based Index Storage for Data Deduplication

Storage deduplication has received recent interest in the research community. In scenarios where the backup process has to complete within short time windows (say, half-a-day), *inline* deduplication can help to achieve higher backup throughput. In such systems, the method of identifying duplicate data, using disk-based indexes on chunk hashes, can create throughput bottlenecks due to disk I/Os involved in index lookups. RAM prefetching and bloom-filter based techniques used by Zhu et al. [134] can avoid disk I/Os on close to 99% of the index lookups. Even at this reduced rate, an index lookup going to disk contributes about 0.1msec to the *average* lookup time – this is about 10^3 times slower than a lookup hitting in RAM. We propose to reduce the penalty of index lookup misses in RAM by orders of magnitude by serving such lookups from a flash-based *index*, thereby, increasing inline deduplication throughput. Flash memory can reduce the huge gap between RAM and hard disk in terms of both cost and access times and is a suitable choice for this application. This thesis designs a *flash-assisted* inline deduplication system using ChunkStash, a chunk metadata store on flash [55].

1.3 Outline

The rest of the thesis is organized as follows.

- Chapter 2 gives an overview of the flash-based storage device.
- Chapter 3 describes a new write caching algorithms specially designed for flash-based storage.
- Chapter 4 describes a RAM-efficient garbage colletion metadata management scheme.
- Chapter 5 describes a update processing technique for the flash-based storage.
- Chapter 6 describes the design and implementation of a flash-based Bloom Filter.
- Chapter 7 demonstrates how to speedup a inline data deduplication system using flash-based storage.
- Chapter 8 concludes the discussion and lists some future extensions.

Chapter 2

Overview of the Flash-Based Storage

Flash memory is available in two different technologies: NOR and NAND typed [20]. NOR type is byte-addressable and is very suitable for storing and executing program code. However, it is relatively costly and exhibits slower write performance. On the other hand, NAND type is sector-addressable, cheaper, exhibits better write performance, and is more suitable for use in file storage. In this thesis, we are focusing on the NAND flash-based storage technologies as we are dealing with massive data.

Figure 2.1 gives a block-diagram of flash-based storage. Flash Translation layer (FTL) is an intermediate software layer inside SSD, which makes linear flash memory device act like a virtual disk drive. FTL receives logical read and write commands from the applications and converts them to the internal flash memory commands. A *page* (also known as *sector*) is the smallest addressable unit in an SSD. A set of pages form a *block*. Typical page size is 2KB and block size is 128KB [72]. Flash memory supports three types of operations: *read*, *write*, and *erase*. Typical access time for read, write, and erase operations are 25 microseconds, 200 microseconds, 1.5 milliseconds, respectively [72]. *Read* is a page level operation and can be executed randomly in the flash memory without incurring any additional cost. *Erase* operation can be executed only at a block level and results in setting of all bits within the block to 1. *Write* is also a page level operation and can be performed only once a page has been previously

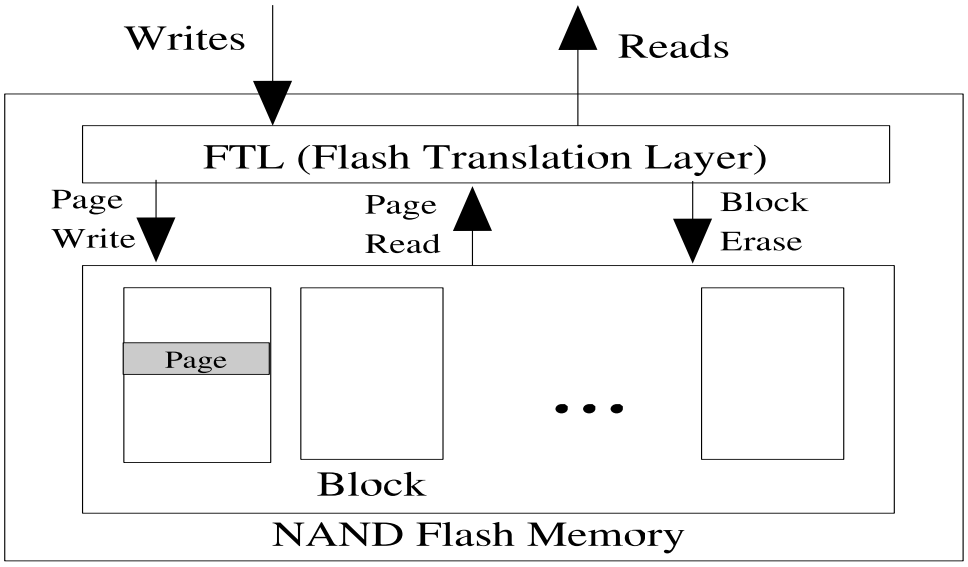


Figure 2.1: NAND Flash-based Storage

erased since it selectively sets the required bits to 0. A flash memory block can only be erased for a limited number of times. For example, a single layer cell (SLC) and multi layer cell (MLC) block can be erased only 100K and 10K times, respectively, after which it acts like a read-only device [72]. FTL also uses various wear-leveling techniques to even out the erase counts of different blocks in the flash memory to increase its overall longevity.

In order to mitigate the effect of erase operations, FTL redirects the overwrite of page operations to a small set of spare blocks called *log blocks* [84]. These log blocks are used as a temporary storage for the overwrite operations. If there is no free log block, then FTL selects one of the log blocks as a victim and performs a *merge* operation [84] also termed as *full merge* in [81]. As *full merge* is a very expensive operation, it is desirable to minimize this operation and instead to have a *switch merge* [84] operation. *Switch merge* is a special case of *full merge* operation, wherein the victim log block is found to have the complete sequence of the pages in the right order from the first page to the last. Therefore the log block can be exchanged with the original data block by just updating the mapping table information and the operation can be completed with only one erase of the data block. Log-based FTL schemes are very effective for the

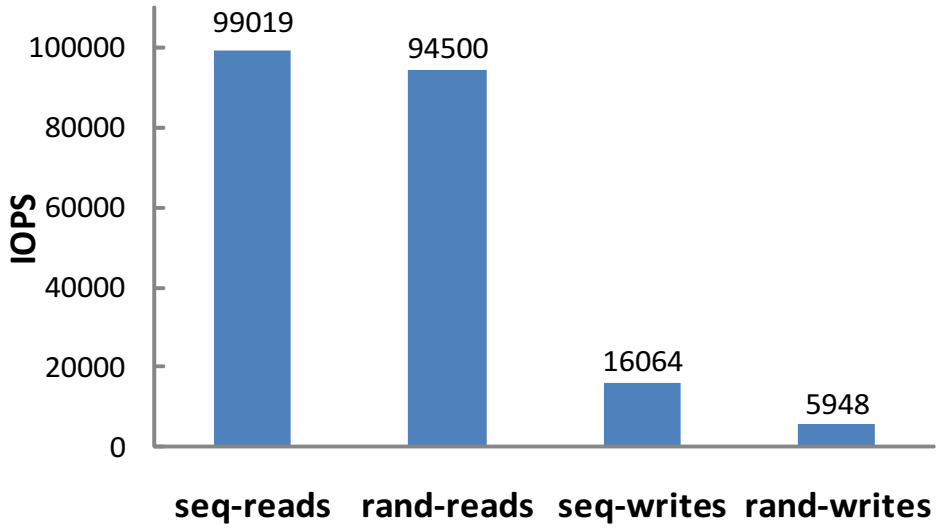


Figure 2.2: IOPS for sequential/random reads and writes using 4KB I/O request size on a 160GB fusionIO drive.

workloads with sequential access patterns. However, for the workloads with random access patterns, these schemes show very poor performance [81, 92, 88].

To validate the performance gap between sequential and random writes on flash, we used Iometer [10], a widely used performance evaluation tool in the storage community, on a 160GB fusionIO SSD [7] attached over PCIe bus to an Intel Core 2 Duo E6850 3GHz CPU. The number of worker threads was fixed at 8 and the number of outstanding I/Os for the drive at 64. The results for IOPS (I/O operations per sec) on 4KB I/O request sizes are summarized in Figure 2.2. Each test was run for 1 hour. The IOPS performance of sequential writes is about 3x that of random writes and worsens when the tests are run for longer durations (due to accumulating device garbage collection overheads). We also observe that the IOPS performance of (random/sequential) reads is about 6x that of sequential writes. (The slight gap between IOPS performance of sequential and random reads is possibly due to prefetching inside the device.)

In this thesis, we have two main design goals. First goal is to improve random write performance, while second goal is to avoid random write as much as possible. We achieve first goal by placing a nonvolatile cache above FTL. Our proposed write caching policy makes the write requests more FTL friendly, which consequently helps

to reduce the total number of erase operations. Reduction in the number of erase operations helps to improve write performance as well as increases lifetime of the SSDs. We achieve second goal by designing data structures and algorithms which use flash in log structured manner in order to avoid random write operations. Chapter 3 focuses on the first goal, while Chapter 5, 6, and 7 focus on the second goal.

Chapter 3

Write Cache for Solid State Disks

3.1 Introduction

Flash memory has rapidly increased in popularity as the primary non-volatile data storage medium for mobile devices, such as cell phones, digital cameras, and sensor devices. Flash memory is popular for these devices due to its small size, low weight, low power consumption, high shock resistance, and fast read performance [31, 63, 81, 80, 90, 92, 106, 107, 132]. The popularity of the flash memory has also extended from embedded devices to laptops, PCs, and enterprise-class server domains [81, 90, 88, 91, 95, 103, 40]. Flash based Solid State Disk (SSD) is regarded as a future replacement for the magnetic disk drive. Market giants like Dell and Samsung have already launched laptops with only SSDs [27, 70]; Samsung, STec, and SimpleTech have launched SSDs with better performance compared to the traditional 15000 RPM enterprise disk drives [28, 29, 102]. Enterprise class SSDs provide unique opportunities to boost up the I/O-intensive applications performance in the datacenters [119, 125]. Compared to hard disks, flash based SSDs are very attractive in the high-end servers of datacenters due to their faster read performance, lower cooling cost, and higher power savings.

Unlike the conventional magnetic disks, where read and write operations exhibit symmetric speed, in flash based SSDs, the write operation is substantially slower than the read operation. This asymmetry arises as flash memory does not allow overwrite operations and write operations in a flash memory must be preceded by an erase operation. This problem becomes further complicated because write operations are performed

at the page granularity, while erase operations are performed at the block granularity. Typically, a block spans 32-64 pages and live pages from a block need to be moved to new pages before the erase is done. Furthermore, a flash memory block can only be erased for a limited number of times, after which it acts like a read-only device. This is known as *wear out* problem. These slow write performance and wear-out issues are the two most important concerns restricting SSDs widespread acceptance in data-centers [119]. Existing approaches to overcome these problems through modified flash translation layer (FTL) are effective for the sequential write access patterns; however, they are inadequate for the random write access patterns [81, 92, 88]. In this thesis, our primary focus is to improve the random write performance of the flash based SSDs.

Nonvolatile RAM cache inside an SSD is useful in improving the random write performance [81]. This cache acts as a filter and makes random write stream close to the sequential before forwarding these requests to an FTL. Since overwrite of a page in the flash memory would require an entire block to be erased, to exploit this behavior the caching algorithms working inside the SSDs operate at the logical block granularity rather than the traditional page granularity [81, 75]. In these algorithms, resident pages in the cache are grouped on the basis of their logical block associations. In case of evictions, all pages belonging to the evicted block are removed simultaneously. Overall, this strategy helps to reduce the number of erase operations. In the page-level algorithms, evicting a page makes a free page space available in the cache. However, in case of the block level algorithms, since different block can contain different number of pages, therefore the number of free pages available depends on the number of pages in an evicted block. Due to this behavior, direct application of the existing disk-based well known caching policies like LRU, CLOCK [51], WOW [65], CLOCK-Pro [73], DULO [74], and ARC [101], so on, are inappropriate for flash based SSDs. In addition, existing flash-optimized write caching algorithms are not very effective for the diverse workloads. We have described the drawbacks of these algorithms in Section 3.2. For the flash based SSDs, we need a completely new set of block level write caching algorithms which will be effective under diverse set of workloads.

In this thesis, as our main contribution, we propose a new block-level write cache management algorithm, which we call the Large Block CLOCK (LB-CLOCK) algorithm, for flash based SSDs. This algorithm considers *recency* as well as *block space utilization*

(i.e., number of pages currently resident in a logical block) to select a victim block. The LB-CLOCK algorithm is based on the CLOCK [51] algorithm, which has been widely used in operating systems to simulate LRU. However, LB-CLOCK algorithm has two important differences from the traditional CLOCK algorithm. First, it operates at the granularity of logical blocks instead of pages. Second, instead of selecting the first page with recency bit zero as the victim, it uses a greedy criterion to select the first block with the largest *block space utilization* from the *victim candidate set*. The *victim candidate set* contains all the blocks with recency bit set to zeros. The most crucial aspect of the LB-CLOCK algorithm lies in the creation of this *victim candidate set* which dynamically adapts to the workload characteristics. We apply two optimizations, *sequential block detection* and *preemptive victim candidate set selection*, to decide when a specific block becomes part of the *victim candidate set*.

We evaluate the performance of the LB-CLOCK algorithm using typical datacenter application benchmarks, including write intensive online transaction processing (OLTP) type Financial Workload trace and a Web Download trace. Based on these experimental results, the LB-CLOCK algorithm is shown to outperform best known existing flash based write caching schemes, BPLRU [81] and FAB [75]. For the OLTP type Financial workload trace, LB-CLOCK performs 245%-493% fewer block evictions and provides 261%-522% more throughput compared to FAB, while performing 4%-64% fewer block evictions and providing 4%-70% more throughput compared to BPLRU. It is important to mention here that OLTP type applications running in datacenters are particularly stressful workloads for the SSDs. Our experimental results also confirm that LBCLOCK is the most effective algorithm under diverse workloads. This work has been published in the MASCOTS 2009 conference [58].

The remainder of this chapter is organized as follows. Section 3.2 surveys the existing caching schemes for flash memory. Section 3.3 explains the LB-CLOCK algorithm in detail. Section 3.4 describes the experimental setup and workload traces. Section 3.5 explains the simulation results. Finally, Section 3.6 concludes the discussion.

	CFLRU	FAB	BPLRU	LB-CLOCK
Granularity Level	Page	Block	Block	Block
Target	Host	Host SSD	SSD	SSD
Read / Write	Both	Both	Write	Write
Recency	Yes	No	Yes	Yes
Block Space Utilization	No	Yes	No	Yes

Table 3.1: Comparison of different existing caching algorithms for the flash memory

3.2 Related Work

Existing caching algorithms for flash memory can be classified into two main categories. One category operates on the *page-level* and another category operates on the *logical block-level*. The first category includes CFLRU algorithm [114] and the second category includes BPLRU [81] and FAB [75] algorithms. Our proposed algorithm Large Block CLOCK (LB-CLOCK), also operates at the block-level. In the rest of this section, we will discuss each algorithm in detail.

Clean First LRU (CFLRU) [114] buffer management scheme exploits the asymmetric read and write speed of the flash memory. It divides the host buffer space into two regions: *working region* and *eviction region*. Victim buffer pages are selected from the eviction region. The size of these two regions vary based on the workload characteristics. CFLRU reduces number of write operations by performing more read operations. It chooses a clean page as victim instead of a dirty page, as write operation is more expensive than the read operation. When all the pages in the eviction region are clean, the victim is selected in the Least Recently Used (LRU) order. As in this thesis we are considering only write operations in the SSD device, CFLRU is not relevant to us.

Block Padding LRU (BPLRU) [81] and Flash Aware Buffer (FAB) [75] cache management schemes group the cached pages that belong to the same erase block in the SSDs into single block. BPLRU uses a variant of the LRU policy named Block Padding LRU (BPLRU) to select a victim in the cache. BPLRU manages theses blocks in a LRU list. Whenever any single page inside a block gets hit, the entire block is moved to

the Most Recently Used (MRU) end of the list. When there is not enough space in the cache, the victim block is selected from the LRU end of the list. In contrast to BPLRU, FAB considers *block space utilization*, which refers to the number of resident cached pages in a block, as the sole criteria to select a victim block. FAB evicts a block having the largest number of cached pages. In case of tie, it considers the LRU order. Since BPLRU only considers *recency* (i.e., LRU order) to select a victim block, for some write workloads where there are no or marginal recency, for example completely random write workload, FAB outperforms BPLRU. However, in general BPLRU outperforms FAB for the workloads which have even moderate temporal localities. Here, our goal is to design a caching algorithm that will perform well for diverse set of workloads. Table 3.1 gives a comparison among all existing flash caching algorithms and our proposed LB-CLOCK algorithm.

3.3 Algorithm Description

The Large Block CLOCK (LB-CLOCK) algorithm is inspired from the CLOCK page replacement algorithm [51] used widely in Operating Systems (OS). In this section, at first, we briefly describe the CLOCK algorithm. Then, we explain LB-CLOCK algorithm in detail. Finally, we describe two optimizations in the core LB-CLOCK algorithm.

3.3.1 CLOCK Algorithm

CLOCK algorithm is used as an efficient way to approximate the working of Least Recently Used (LRU) algorithm on memory pages [124]. Since it is very expensive to maintain the LRU lists of memory pages on each memory access, CLOCK algorithm maintains one reference bit per page. Whenever a page is accessed (i.e., read or written), the reference bit is set by the hardware. CLOCK algorithm resets this bit periodically to ensure that a page will have reference bit set only if it has been accessed at least once from the duration of the last reset. CLOCK algorithm maintains all the physical memory pages in a circular list with a clock pointer currently pointing to a page. Reference bits of all the pages in the clock pointer's traversal path are serially checked to select the victim page for eviction. If there is a page with reference bit set (i.e., 1), it is reset (i.e.,

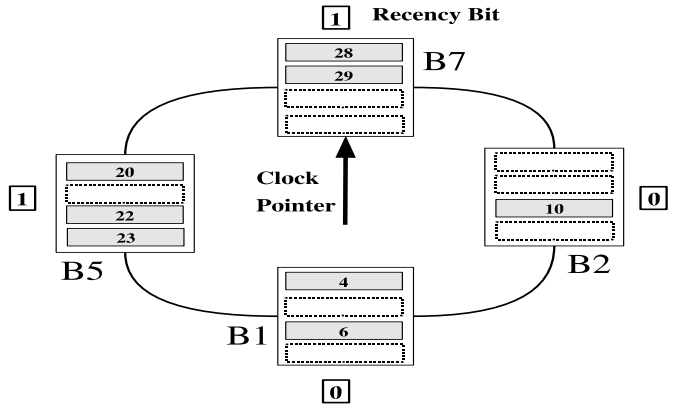
set to 0) and the clock pointer is advanced by one position. This step is repeated until the clock pointer encounters a page with reference bit zero. The corresponding page is chosen as the victim and the clock pointer is moved to the next position.

One of the main reasons why LB-CLOCK algorithm is designed based on the CLOCK algorithm is due to its dynamically created sets of recently used and not used pages. In CLOCK algorithm, at any point in time we can consider pages in the memory to be part of one of these following two sets: *(a) Recently used set:* consisting of pages which have current reference bit values set to 1s. *(b) Not recently used set:* consisting of pages having current reference bit values set to 0s. Therefore, if we need to consider more than one victim selection criteria exploiting recency of page accesses, we can apply other criteria on all the pages which are part of not recently used set and this set gets dynamically updated based on the workload's temporal locality.

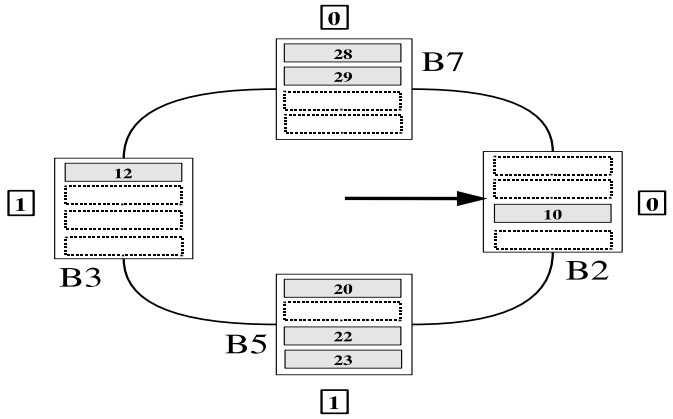
3.3.2 LB-CLOCK Algorithm

In case of the Large Block CLOCK (LB-CLOCK) algorithm, all pages in the cache are divided into logical groups (*block*s) based on the physical erase block they belong to in an SSD. A block can be comprised of up to N pages, where N is block size in pages. Every logical block currently in cache is associated with a *reference bit*. This bit is set whenever there is an access to any page of that block. Similar to the CLOCK algorithm, LB-CLOCK algorithm also uses a clock pointer which traverses in one direction over the circular list of logical blocks currently in cache. Whenever cache is full and an eviction decision needs to be made to accommodate a new write request, LB-CLOCK algorithm selects the victim group of pages at the granularity of a block. Thus, when a block is selected as a victim, all the pages which are part of that block, are simultaneously evicted out from the cache.

In order to select a block for eviction, LB-CLOCK algorithm considers two important metrics: *recency* and *block space utilization*. *Recency* is important for workloads which exhibit temporal locality. On the other hand, *block space utilization*, which refers to the numbers of cached pages in a block, is also an important metric as evicting a block containing many pages results in more cache space available for the future write requests. Thus, evicting larger blocks will consequently help to reduce the total number of block evictions. During the victim selection process, LB-CLOCK checks the recency bit the



(a) Initial cache state, when cache is full



(b) Cache state to accommodate a new page request

Figure 3.1: Working of the LB-CLOCK algorithm

block pointed by the clock pointer. If currently recency bit is set to 1, it is reset to 0, and the clock pointer is advanced to the next block. This step is repeated until the clock pointer comes across a block with 0 recency bit value. In other words, the clock pointer stops at a block, which had its recency bit set to 0 prior to current round of the victim selection. The set of all blocks, which currently have their recency bit set to 0 is considered to be part of the *victim candidate set*. A *victim block* is the block with the largest *block space utilization* value (i.e., containing the highest number of pages) in this set.

3.3.3 An Illustration of LB-CLOCK

Figure 3.1 illustrates the working of the LB-CLOCK algorithm for a cache size of eight pages and block size of four pages. Block n, B_n, contains at most four pages: $4n$, $4n+1$, $4n+2$, and $4n+3$. The initial cache state with eight pages belonging to four different blocks is shown in Figure 3.1(a). The shaded pages indicate the currently resident pages in the cache. Pages 4 and 6 belong to block B_1; page 10 belongs to block B_2; pages 20, 22, and 23 belong to block B_5; and page 28 and 29 belong to block B_7. The clock pointer is currently pointing at B_7. The pointer moves in the clock-wise direction. The recency bit of every block is indicated by the value in the adjacent small box. Now, the cache is full. To accommodate any new page request which incurs a page miss, LB-CLOCK needs to evict a block.

Suppose, there is a new write request for page 12 belonging to block B_3. Since the cache is full the clock pointer is traversed to find a block with recency bit 0. In this traversal, if pointer encounters a block with recency bit 1, it resets the bit to 0 and moves pointer to the next block. In the running example, clock pointer resets recency bit of block B_7, and moves to block B_2. As the recency bit of B_2 is 0, clock pointer stops traversing. Now, the *victim candidate set* contains $\{B_1, B_2\}$. LB-CLOCK selects block B_1 as the victim block since B_1 contains the largest number of pages. LB-CLOCK evicts all the pages currently residing in B_1 at the same time to make free space in the cache. Finally, the new write request of page 12 belonging to B_3 is inserted behind B_7 as the CLOCK pointer initially points to B_7. Finally, the recency bit of B_3 is set to 1. Figure 3.1(b) shows the new cache state.

3.3.4 Optimizations in LB-CLOCK

We apply two optimizations in the basic LB-CLOCK algorithm. In the rest of this section, we discuss these two optimizations.

Preemptive Victim Candidate Set Selection

Depending on the workload, varying priority should be given to *recency* and *block space utilization*. If a workload has moderate temporal locality, then an algorithm like BPLRU, which gives priority to *recency*, would lead to fewer block evictions. On

the other hand, if the workload does not have much temporal locality (random) and have blocks of varying sizes, then prioritizing *block space utilization* would lead to fewer block evictions. In an SSD, each block eviction would incur an block erase operation. Since, erase is a very expensive operation, fewer number of erase operations will lead to better performance derived out of SSDs. Moreover, fewer erase operations will eventually increase the lifetime (i.e., reliability) of SSDs. Hence, a balance between the priority given to *recency* and *block space utilization* is required. However, due to lack of prior information about the workloads, this balance has to be achieved dynamically.

LB-CLOCK achieves this balance by using the following heuristic: every time a block is selected for eviction, LB-CLOCK keeps track of the number of cached pages in that evicted block. Whenever a page is written in a block, LB-CLOCK checks if this is the last page of that block. If the last page of a block is written, then there is a low probability that it is going to be accessed again in the near future. On the other hand, if the current page written on a block is not the last page, then there is still a chance that the block is going to be accessed again in the near future, therefore the corresponding block is not considered as a part of the *victim candidate set*. If this first check succeeds, LB-CLOCK further checks whether that block is completely full (contains all pages) or not. If this second check also succeeds, LB-CLOCK considers the corresponding block as a part of the *victim candidate set*. On the other hand, if the second test fails, LB-CLOCK checks if *block space utilization* value of the block is greater than that of the previously evicted block. If this third test succeeds, LB-CLOCK includes the corresponding block as part of the *victim candidate set* by resetting its reference bit to 0. The second test gives priority to the blocks, which have passed the first heuristic test (i.e., blocks considered as less likely to be written again in the near future) and have larger number of pages than the block evicted in the previous phase of the victim selection. At any point of time, if there are only small sized blocks (i.e., blocks with very less number of pages) in the *victim candidate set* and there is a sudden burst of large sequential writes, then this heuristic will make sure that these recently written large blocks are considered for eviction earlier than the existing relatively small-sized blocks. To accommodate one large write requests, core LB-CLOCK has to evict out many small blocks from cache which results in a large number of block erase operations. The second test would prevent this problem.

Operation	Time (microseconds)
128-KB Block Erase	1500
2KB-Page Read	25
2KB-Page Write	200
2KB-Data Transfer	13.65
Memory Access Latency	0.0025

Table 3.2: Timing parameters values used in the simulation. The first three values are taken from [72], while the fourth value is calculated based on the SATA1 burst rate of 150 MB/s [16]

Sequential Write Detection

If a block is being written sequentially, then it is assumed that the same block is unlikely to be accessed in the near future. The idea is similar to the one discussed in BPLRU [81]. When the write page request correspond to the last page of a block and rest of the pages of that block are already in the cache, such a block is considered as *sequentially written block*. For example, if a block can contain 64 pages and first 63 pages are already in the cache, then the cache receives the 64-th page write request, in this case the corresponding block is considered as sequentially written block. BPLRU places such block at the tail of the LRU list, thus making sure that it is being immediately selected as the victim in case of eviction becomes necessary. LB-CLOCK simply resets the corresponding block's reference bit to zero, thus making sure that such block is a part of the *victim candidate set*.

3.4 Experimental Setup

This section gives a detail overview of the simulator and traces used to evaluate LB-CLOCK algorithm.

3.4.1 Simulator Detail

A cache simulator is implemented as a stand alone tool in C language on the Linux platform. The simulator takes trace file as input with each write request represented by a pair of values <*start_page, length*>. Other parameters include the cache size and one of the following algorithms to run: 1) LB-CLOCK, 2) BPLRU [81], and 3) FAB [75]. The

simulator executes the entire write requests in the trace file by simulating the working behavior of the designated algorithm.

Our simulation environment and flash translation layer (FTL) settings are similar to the BPLRU [81]. We assume a 80-GB NAND flash memory with block size of 128KB and page size of 2KB. We calculate the write throughput and the total number of block evictions while varying the RAM buffer size from 1MB to 256MB. For generating traces, we create a 80GB NTFS partition in a system running Windows XP Operating System.

To evaluate LB-CLOCK algorithm, we use total number of *block evictions* and *write throughput* as performance metrics. We calculate *write throughput* using the following formula: *total amount of data written from the cache / (total time for block erases + total time to read missing pages in the evicted blocks + total time to write evicted number of full data blocks + total time taken as a result of memory accesses from algorithmic overhead).* The parameter values used for the *write throughput* calculation are listed in Table 3.2. The main reason for choosing total number *blocks evictions* as a metric is that write cache is logically placed above FTL and the blocks evictions from the write cache have direct correlations with the erase operations. The more blocks evictions from the write cache boils down to more tasks for the FTL in terms of wear leveling and garbage collection, which cause frequent erase operations. Therefore, if we can reduce the number of blocks evictions, it will lead to fewer number of erase operations. Since erase is the most expensive operation, therefore reducing the number of block erase operations will help to improve write performance. On the other hand, since a block can only be erased for a limited number of times, reduction in the number of block erase operations will also help to increase lifetime of flash based SSDs.

3.4.2 Workload Traces Used

We use the following traces covering different types of applications.

SPC Financial Trace: SPC [18] traces are collected from the UMass Storage Repository [21], which describes the trace data as being from the online transaction processing (OLTP) applications running at two large financial institutions. OLTP type of application running in datacenters is one of the stressful workloads for the SSDs due its write-intensive nature. We have extracted only the writes requests and ignored all the read requests in these traces.

MS OFFICE 2007 Installation Trace: We use Diskmon utility [6] from Microsoft to gather disk writes on NTFS file system while installing MS Office 2007 on an empty 80GB partition running Windows XP operating system.

MP3 Files Copy Trace: We copy over 4GB of MP3 files to an empty 80GB empty NTFS disk partition and collected the disk write requests using Diskmon utility [6].

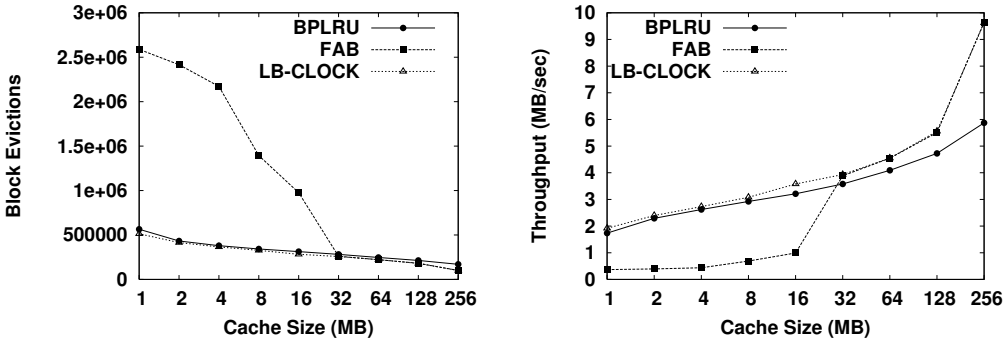
Web Download Trace: About 7GB of internet data is downloaded from the amazon's website [2] using the Wget utility [8] to an empty 80GB NTFS partition. The write requests to the disk are collected by using Diskmon utility [6].

Synthetic Trace: To perform a comprehensive study, we also evaluate LB-CLOCK for a synthetic workload which is 100% random and has no temporal locality. We use Iometer utility [10] to generate this workload trace. The Iometer generated disk write requests are collected by using the Diskmon utility [6].

3.5 Result Analysis

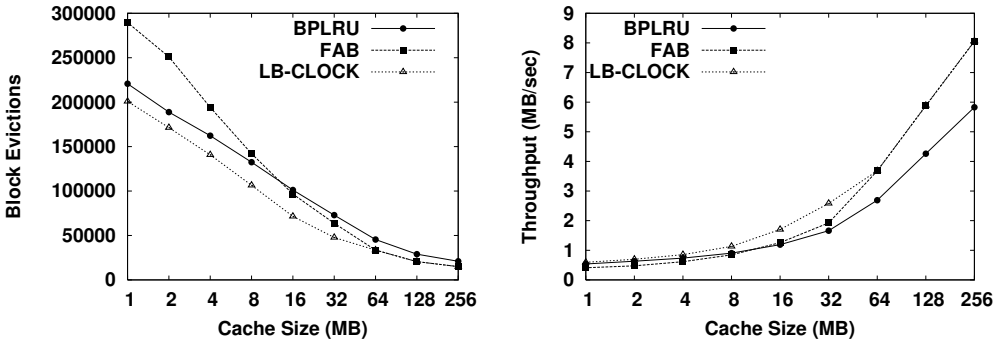
We compare our proposed algorithm Large Block CLOCK (LB-CLOCK) with the current best known algorithms BPLRU [81] and FAB [75] for both real workload traces and synthetic traces. We compare the resultant number of block evictions and write throughput for the cache size varying from 1MB to 256MB.

OLTP type SPC Financial_1 Trace: Figure 3.2 shows that LB-CLOCK algorithm performs 245%-493% less block evictions and provides 261%-522% more throughput compared to FAB for the cache size upto 16MB. Beyond 16MB FAB does almost same as LB-CLOCK. Although it not clear from Figure 3.2, but Figure 3.8 shows that LB-CLOCK performs 4%-70% less block evictions compared to BPLRU. In terms of throughput, it provides 4%-64% more throughput compared to BPLRU. Careful analysis of this workload revealed that it exhibits a very high degree of both spatial and temporal locality. Since FAB is not suited for such a workload, it performs poorly for cache size up to 16MB. Beyond 16 MB, cache size is large enough to accommodate the temporal locality window, so FAB performs as good as LB-CLOCK. It is also not surprising to find that LB-CLOCK is better than BPLRU for workloads of this nature. BPLRU does not perform well since it considers only *recency*, while by having dynamically varying priority for *recency* as well as *block space utilization* LB-CLOCK performs



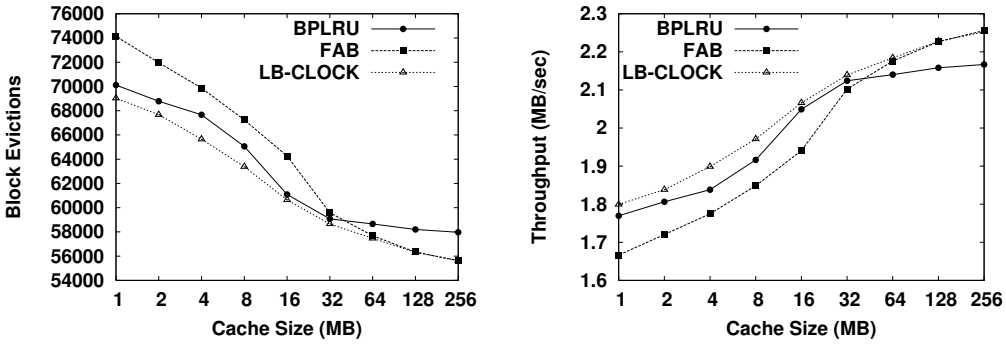
(a) (b)

Figure 3.2: OLTP type SPC Financial_1 Trace



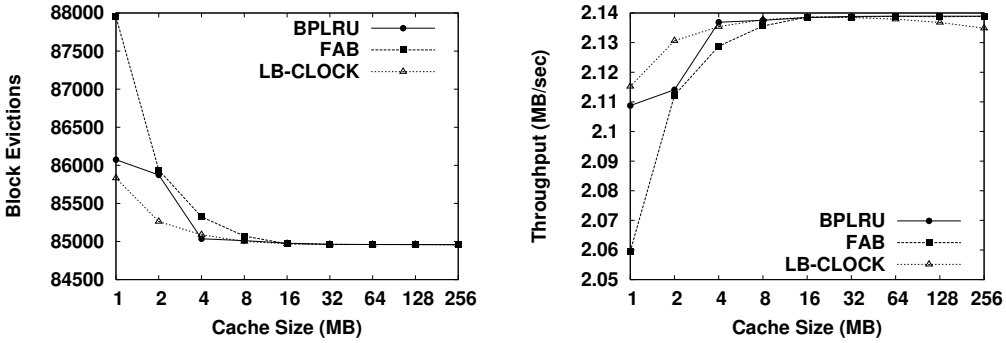
(a) (b)

Figure 3.3: OLTP type SPC Financial_2 Trace



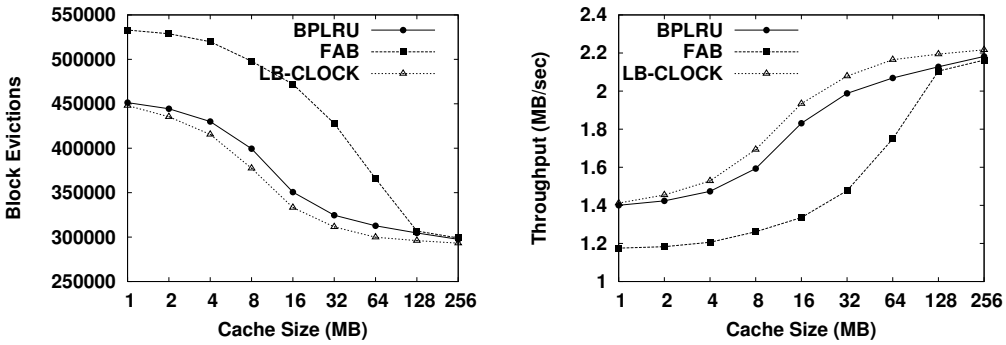
(a) (b)

Figure 3.4: MS Office 2007 Installation Trace



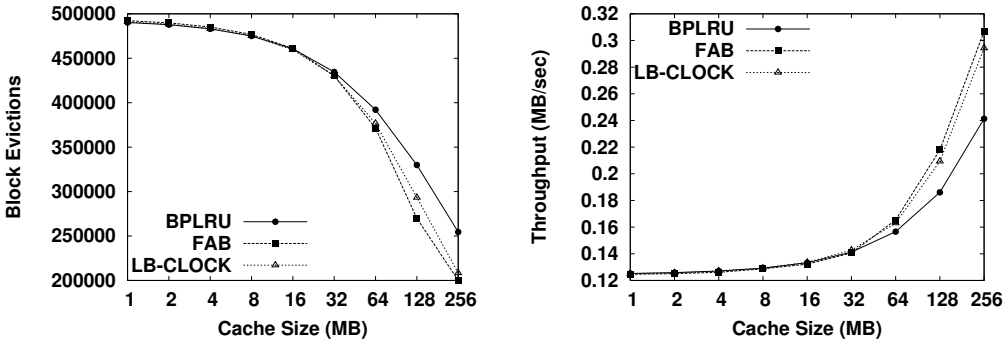
(a) (b)

Figure 3.5: MP3 Files Copy Trace



(a) (b)

Figure 3.6: Web Download Trace



(a) (b)

Figure 3.7: Synthetic trace containing only 100% random single page write requests

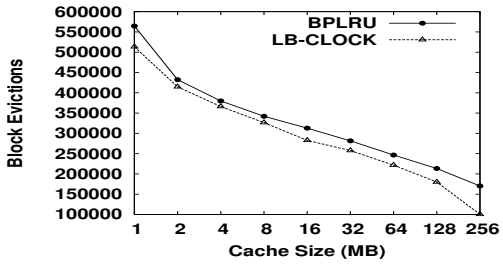


Figure 3.8: OLTP type SPC Financial_1 Trace: BPLRU vs. LB-CLOCK algorithm (a closer view)

better than BPLRU.

OLTP type SPC Financial_2 Trace: Figure 3.3 shows that, LB-CLOCK again performs the least number of block evictions. FAB again performs poorly for smaller cache sizes and does almost same as LB-CLOCK for cache size of 64MB and more. LB-CLOCK performs 33%-46% less block evictions and provides 34%-47% more througput compared to FAB for cache size less than 64MB. On the other hand, LB-CLOCK performs 10%-53% less block evictions compared to BPLRU, while provides 10%-56% more throughput. These results are very similar to the previous ones and further proves that for a workload with a high degree of temporal and spatial locality, LB-CLOCK is the best algorithm compared to the FAB and BPLRU algorithm.

MS OFFICE 2007 Installation Trace: Figure 3.4(a) show that LB-CLOCK outperforms both FAB and BPLRU for all cache sizes varying from 1MB-256MB. LB-CLOCK performs 2%-4% less block evictions compared to BPLRU. While LB-CLOCK performs 2%-7% less block evictions compared to the FAB for the cache size varying from 1MB-32MB, beyond 32MB cache size FAB performs same as LB-CLOCK. LB-CLOCK also provides 1%-4% throughput improvement compared to BPLRU, while it provides 2%-8% throughput improvement compared to FAB algorihtm.

MP3 Files Copy Trace: Figure 3.5(a) shows that beyond 8MB cache all three algorithms LB-CLOCK, FAB, and BPLRU result in almost same number of block evictions. Even with smaller cache size LB-CLOCK and BPLRU perform very similarly. However, for the cache size less than 8MB, LB-CLOCK and BPLRU outperform FAB. For cache size less than 4MB, LB-CLOCK performs 3% less block evictions and provides 3% more throughput compared to FAB. Since MP3 files copy workload is fairly

sequential, moderate cache size is good enough to capture its workload window. LB-CLOCK and BPLRU outperform FAB in the smaller cache sizes, as FAB prematurely evicts some of the larger blocks which are still being to be sequentially written.

Web Download Trace: From the Figure 3.6(a), it is clear that LB-CLOCK and BPLRU follow a similar trend. However, LB-CLOCK performs lesser number of block evictions. FAB performs poorly for this workload trace for cache sizes less than 128MB. In general, LB-CLOCK performs the best in this case and beats FAB by a big margin. Compared to BPLRU, LB-CLOCK does 2%-6% less block evictions and provides 2%-6% more throughput. Whereas compared to FAB, LB-CLOCK performs 2%-42% less block evictions and provides 2%-45% more throughput.

Synthetic Trace Containing 100% Random Write Requests: This synthetic trace having no or marginal temporal locality, represents one of the worst case write traces for LB-CLOCK and BPLRU, while the best case for FAB. This is because on average the cache will comprise of varying sized blocks and since this is a 100% random workload, none of the pages in the cache are likely to be requested within a very short time. Therefore, the best caching policy in this case is to choose the largest sized (i.e., space utilized) block as the victim ignoring its recency bit. Figure 3.7 shows that for this type of random workload, all three algorithms perform almost same for smaller cache size range, while for the larger cache size both LB-CLOCK and FAB outperform BPLRU. For example, for the cache size 64MB-256MB range, LB-CLOCK performs 4%-12% less block evictions and provides 4%-12% more throughput compared to BPLRU. While LB-CLOCK performs 1%-8% more block evictions and provides 1%-4% less throughput compared to FAB. Although it is not included in Figure 3.7 for consistency with other figures, we find that compared to FAB for the 512MB and 1GB cache size, LB-CLOCK provides 7% and 10% more throughput, respectively, which which illustrates the dynamic adaptability of the algorithm. This type of workload is not very common. From our observation, all real life workloads always have a fair amount of temporal locality and FAB does not perform well for these workloads. We analyzed this trace to perform a comprehensive analysis of our proposed LB-CLOCK algorithm.

Results Summary: Table 3.3 gives a summary of the results for all workload traces. In case of the real workload traces, LB-CLOCK outperforms both BPLRU and

	LB_CLOCK vs. BPLRU		LB_CLOCK vs. FAB	
	Block Evictions	**Throughput**	**Block Evictions**	**Throughput**
Financial_1	4% to 70%	4% to 64%	245% to 493%	261% to 522%
Financial_2	10% to 53%	10% to 56%	33% to 46%	34% to 47%
MS Office	2% to 4%	1% to 4%	2% to 7%	2% to 8%
MP3	0%	0%	3%	3%
Web	2% to 6%	2% to 6%	2% to 42%	2% to 45%
Synthetic	4% to 12%	4% to 12%	-1% to -8%	-1% to -4%

Table 3.3: Summary of the improvement in LB-CLOCK compared to BPLRU and FAB for the cache size 1MB-256MB

FAB. For the synthetic trace, LB-CLOCK outperforms BPLRU, while FAB outperforms LB-CLOCK. However, the synthetic trace result is not important as all the realistic workloads have a fair amount of temporal locality. *We have observed the highest benefit of the LB-CLOCK algorithm in case of the online transaction processing (OLTP) type Financial workloads. This is a very significant result as the write performance of the the OLTP type applications on SSDs is one of the major concerns for the datacenters in widely deploying SSDs.* In addition, since block evictions from write cache have direct correlations with physical flash block erases, less block evictions performed by the LB-CLOCK algorithm also helps to mitigate the wear-out problem of the SSDs.

3.6 Conclusion

Flash based Solid State Disks (SSDs) show substantial promise to improve data throughput, power and heat efficiency of the high-end servers in the datacenters. However, relatively slow write performance and wear-out problem are the two big concerns in widely deploying SSDs in the datacenters. In this thesis, we are mainly focusing on improving write performance of the SSDs. We have proposed a new write back caching algorithm named as Large Block CLOCK (LB-CLOCK), which is especially designed to cope with the physical properties of the flash memory. LB-CLOCK algorithm operates on the block level and considers two key metrics: *recency* and *block space utilization* to manage pages in the cache. Depending on the workload behavior, LB-CLOCK dynamically varies the priorities between these two metrics, which helps it to outperform the previously best known algorithms including BPLRU [81] and FAB [75]. In addition,

LB-CLOCK helps to deal with the wear-out issue by reducing the number of block erase operations.

Our experimental results confirm that LB-CLOCK algorithm consistently outperforms BPLRU algorithm for real as well as synthetic workload traces with up to 70% better performance for an online transaction processing (OLTP) trace, which is one of the most write-intensive workloads running in the datacenters, and 12% for the synthetic trace in terms of throughput. While LB-CLOCK algorithm outperforms FAB algorithm for the real workload traces including as much as 522% for an OLTP trace and 45% for Web Download trace in terms of throughput, it performs very close to FAB algorithm for the synthetic trace. Therefore, overall our proposed LB-CLOCK algorithm is better than the BPLRU and the FAB algorithm.

Chapter 4

Sampling-based Garbage Collection Metadata Management for Flash Storage

4.1 Introduction

With the continually accelerating growth of data, the performance of storage systems is increasingly becoming a bottleneck to improve overall system performance. Many applications, such as transaction processing systems, weather forecasting, large-scale scientific simulations, and on-demand services, are limited by the performance of the underlying storage systems. The limited bandwidth, high power consumption, and low reliability of widely used magnetic disk-based storage systems impose a significant hurdle in scaling these applications to satisfy the increasing growth of data. Flash memory is an emerging storage technology that shows tremendous promise to compensate for the limitations of current magnetic disk-based storage devices. In fact, flash-based Solid State Disks (SSDs) are predicted to be future replacement for the magnetic disk drives. For example, market giants like Dell, Apple, and Samsung have already launched laptops with only SSDs [27, 70, 12]. Microsoft has added features to support SSDs in the new release of Windows OS [23], while Google has announced to support only SSDs in the Chrome OS [100]. Recently, MySpace.com has switched from using magnetic disk drives

in its servers to SSDs as primary storage for its data center operations [22].

Flash memory exhibits different read and write performance characteristics compared to magnetic media. In flash-memory, random read operations are as fast as sequential operations due to lack of mechanical head movement. Whereas, the write operations are substantially slower than the read operations (we discuss the characteristics of flash-based storage device in more detail in Chapter 2). A distinguishing property of flash memory is that it does not allow overwrite operations. Once data is written in a page, to overwrite it, the page must be erased (which is a slow operation) before writing again. This is known as *in-place update* problem. This problem becomes further complicated because read and write operations are performed in the page granularity, while erase operations are done in the block granularity (typically a block spans 32-64 pages). Furthermore, a flash memory block can only be erased for a limited number of times [31]. This is known as *wear out* problem.

To hide limitations of the flash memory, a flash-based storage device (for example, solid state disk) uses Flash Translation Layer (FTL), which helps to acts it like a virtual hard disk drive. To solve the *in-place update* problem, FTL writes the updated pages in a log-manner and maintains a logical-to-physical page address mapping table to keep track of the current location of a logical page. FTL periodically invokes garbage collection to perform cleaning operation and reclaim free space. To solve the *wear out* problem, during garbage collection, FTL employs various *wear leveling* algorithms, which spread out the updates uniformly among all blocks so that individual blocks are erased out evenly. For page address mapping and garbage collection, FTL needs to maintain some metadata. Usually, for the faster access, these metadata are stored on an SRAM-cache in the current flash-based storage device controller [63, 69]. However, due to higher price per byte of SRAM, it is unlikely that SRAM will scale proportionately with the increase of flash capacity [69]. Thus, scaling out metadata that are stored and processed for a large-capacity flash storage, in the limited amount of SRAM becomes very challenging.

Since SRAM space is a scarce resource, recent research works focus on reducing metadata stored in the SRAM. Most of the research works [63, 69, 93, 94, 83, 77, 130, 111, 112] try to save SRAM space for storing the page address mapping. For example, recently DFTL [69] proposes a demand based page mapping scheme, which significantly reduces the SRAM space for page mapping [69]. However, very few research

Block Size	256KB
Metadata Size Per block	8B
Total Blocks Needed for 1GB	4096
Total Metadata Size for 1GB	4096*8B = 32KB
Total Metadata Size for 1TB	1024*32KB = 32MB

Table 4.1: Metadata Size Estimation

works [63, 76] have been conducted to reduce the metadata stored and processed in SRAM for the garbage collection. In this paper, we address this issue by using a sampled-based algorithm.

All the current garbage collection algorithms use score-based heuristics to select a victim block for reclaiming free space and wear leveling. (An overview of these algorithms is given in Section 4.2.2). The block with the highest (or lowest) score is selected as victim. The score is estimated using the metadata formation, i.e., block utilization, age (time since a block has been lastly erased), or erase count. To implement these algorithms, we need to maintain these metadata per block level. To quickly find a victim block, these algorithms maintain a priority queue and choose a block based on the priority score. The SRAM space for the priority queue is $O(K)$ [52], where K is the flash based storage capacity in terms of total number of blocks. As the flash capacity increases, K increases linearly. For example, Table 4.1 shows that for 1GB of flash memory, we need 32KB SRAM to store the metadata, while 32MB for 1TB of flash memory. Clearly, with increase of the flash capacity, scaling out SRAM space for metadata is a problem. Besides the space problem, the priority queue needs to be continuously updated and every updates requires $O(lg(K))$ operations [52], whenever any relevant metadata used to estimate the score has been changed, which incurs lot of computation overhead. Current approaches solely focus on solving the SRAM space problem. To reduce the metadata space, they either logically groups adjacent blocks and maintain metadata per group level, or use coding technique to store per block erase count. These approaches are explained in detail in Section 4.2.3. However, these approaches still require $O(K)$ space.

In this paper, we have taken a radically different approach. We propose to use a sampling-based approach to approximate existing garbage collection algorithms. The

main idea is as follows. Instead of storing metadata for all blocks in the SRAM, we store metadata only for N number of randomly selected blocks, where $N << K$. As the time progresses, these samples also change continuously. Whenever we need to select a victim block for garbage collection, instead of considering metadata of all K blocks, we consider only the metadata of the current pool of N sampled blocks. Our sampling-based approximation algorithm significantly reduces the SRAM consumption. Experimental evaluations show that maintaining only small number of samples is good enough for emulating the current garbage collection algorithms. For example, if 30 samples are good enough, then we need only 240 bytes of SRAM space, in contrast to 32MB needed for 1TB flash storage (as shown in Table 4.1). On the other hand, we calculate scores on demand (i.e., during victim selection), thus our approach imposes very less computation overhead. Here, we do not provide any new garbage collection algorithms, rather propose a space and computation efficient mechanism to implement the existing garbage collection algorithms in limited SRAM of a large-capacity flash storage controller. The basic idea of our proposed sampling-based approach is inspired from the randomized web-cache replacement scheme proposed by Psounis and Prabhakar [115]. Currently, we are working to further improve the sampling-based algorithm proposed in this chapter. A technical report of our sampling-based algorithm is available in online [53].

The main contributions of our work are summarized as follows:

- We propose a sampling-based algorithm to approximate existing garbage collection algorithms. Our algorithm significantly reduces space and computation overhead in the SRAM for the management of garbage collection metadata. To best of our knowledge, we take the first attempt to reduce both space and computation overhead.

- Our experimental results show that sampling-based approximated algorithms always outperforms unsampled algorithms in terms of wear leveling metrics (i.e., erase counts variance and max erase count), which means that sampling-based algorithms inherently helps to solve the wear leveling issues. This is a very significant result as it improves *wear out* problem of the flash memory which is one of the big concerns for the wide deployment of the flash-based storage [119].

The rest of this chapter is organized as follows. Section 4.2 gives an overview of

the architecture and current state-of-art metadata management schemes of a flash-based storage device. Section 4.3 presents a detailed description of our sampling based algorithm. Section 4.4 provides an experimental evaluation. Finally, we conclude the paper in Section 4.5.

4.2 Background and Motivation

In this section, first we describe the various metadata needed for page address mapping and garbage collection. Next, we describe the related works that reduce metadata information stored in the SRAM for garbage collection.

Chapter 2 gives an overview of the flash-based storage. Flash Translation Layer (FTL) is an intermediate software and hardware layer inside an SSD, which makes a linear flash memory device acts like a virtual disk drive. FTL receives logical read and write commands from the file system and converts them to the internal flash memory commands. To avoid a block erase for every page update operation, FTL maintains a logical-to-physical page mapping. Whenever a logical page data is updated, FTL writes data to a new physical location, invalidates data in the old physical location for future garbage collections, and updates address mapping table to store new logical-to-physical page information. Many updates produce numerous invalid pages and reduce the number of clean pages as time goes. Usually, some free space is reserved to support new write operations. When free space becomes less than a threshold, FTL invokes garbage collection operation to reclaim the space which becomes invalid due to earlier update operations. Garbage collection has the following steps: select a victim block, identify the valid pages in that block, move the valid pages to a new block, and erase the victim block and mark it as ready for the future write operations. The valid page movement process is known as *cleaning* operation. We define the overhead of the garbage collection as *cleaning cost*. During cleaning, garbage collector uses various wear-leveling techniques to even out the erase counts of different blocks in the flash memory to increase its overall longevity [31]. Usually, garbage collection operation is done in the background so that page update operations can be performed without any significant performance degradation.

Page address mapping and garbage collection require to maintain metadata information. For example, for page mapping, FTL needs to maintain logical page number and physical page number information. While for garbage collection, it needs to maintain erase count, number of valid pages, number of invalid pages, and age etc. These metadata are stored in the SRAM cache due to faster access latency of SRAM compared to flash memory. With the increase of the flash-based storage capacity, scaling metadata space consumption in the SRAM becomes challenging. In the rest of this section, we describe the state-of-the-art for storing the metadata information in SRAM.

4.2.1 Page Address Mapping

Page address mapping information can be maintained at the page level, block level, or a combination of them (hybrid). In a page level mapping case, FTL maintains a mapping between each logical page number (LPN) and physical page number (PPN). Although a page level mapping scheme has its merits in high block utilization and good read/write performance, but it requires very large memory space to store the entire page mapping table. To overcome this large memory limitation, a block level mapping FTL tries to map each logical block number (LBN) to physical block number (PBN). In addition, inside a block, page offsets are always fixed. An update to a page in a page level mapping may not have to trigger a block erasure, while an update to a page in a block level mapping will trigger the erasure of the block containing the corresponding page. Thus, the performance of a block level mapping for write intensive workloads is much worse than that of a page level mapping.

To take advantage of the both page level and block level mapping, various hybrid schemes [63, 93, 94, 83, 77, 130] have been proposed. Most of these schemes are fundamentally based on a block level mapping with an additional page level mapping restricted only to a small number of log blocks in order to delay the block erasure. However, for write intensive workloads, hybrid FTL schemes still suffer from performance degradation due to the excessive number of block erase operations. Recently, Gupta et al. propose a pure page level mapping scheme called DFTL (Demand-based Flash Translation Layer) with a combination of small SRAM and flash memory [69]. In DFTL, the entire page level mapping is stored in the flash memory and temporal

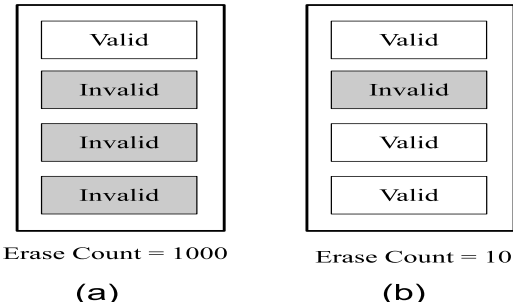


Figure 4.1: Illustration of cleaning and wear leveling centric algorithms

locality is exploited in an SRAM cache to reduce lookup overhead of the mapping table. DFTL scheme helps to significantly reduce the metadata space consumption in the SRAM. Since efficiently storing page mapping information in SRAM has been resolved by DFTL scheme, in this paper we do not consider this issue.

4.2.2 Garbage Collection

A garbage collection scheme has two main goals: **(1)** reduce the cleaning cost and **(2)** even out wear in different blocks as much as possible. Typically, cleaning and wear-leveling algorithms have conflicting goals. A cleaning-centric policy prefers those blocks which contain small number of valid pages, since it helps to minimize the cleaning cost. In contrast, a wear-leveling centric policy chooses the blocks least worn out blocks. These blocks usually store valid (live) pages and contain mostly read-only data. Figure 4.1 shows the difference between these two approaches. Figure 4.1(a) shows a block having three invalid pages and erase count of 1000. While Figure 4.1(b) shows a block having one invalid page and erase count of 10. A cleaning centric policy will select the block in Figure 4.1(a) as victim, since it needs fewer page movement and provide much larger free space. Whereas, a wear-leveling centric policy will select the block in Figure 4.1(b) as victim, since erasing this block helps to mitigate the unevenness in erase counts (although it may not provide larger free space).

Over last decades various garbage collection algorithms have been proposed [31, 63, 45, 35, 26, 43, 47, 44, 123, 129, 78, 98, 128, 82, 49]. All these algorithms require some metadata to select a victim block. In the rest of this section, we describe some of

these algorithms to give an overview of what type of metadata information required to implement them. These algorithms can be broadly classified into three categories.

Cleaning Centric

This category of garbage collection algorithms focus only on cleaning cost. The main goal is to reduce the overhead of cleaning, i.e., the movement of valid pages from a victim block.

The greedy strategy is to select a block with the largest number of invalid pages [78]. This strategy works for the uniformly distributed access patterns, while fails for the workloads which exhibit high degree of localities in the access patterns. On the other hand, the cost-benefit (CB) strategy selects a block with the highest $\frac{1-u}{2u} * age$ value, where u stands for fraction of valid pages in block (i.e, the ratio of valid pages and total capacity in terms of pages of a block), and age is the time since last invalidation [78]. The benefit of reclamation is the number of free pages obtained, which is captured by $1 - u$. While the cost of reclamation is the overhead of the number of valid pages that need to be read and written elsewhere, which is capture by $2u$. This benefit and cost ratio is weighted by age. A smaller value of the age indicates the fact that valid page occupancy is more likely to decrease soon, therefore waiting longer will increase the benefit of reclaiming its space. In contrast, a larger value of age indicates that this block is relatively static block, even if we wait longer, it is less likely that the number valid pages will decrease, therefore this block is a very good candidate for reclamation. DFTL uses CB policy for the garbage collection [69].

Wear Leveling

This category of the garbage collection algorithms focus on cleaning efficient policies most of the time, however conditionally switch back to wear-leveling centric policies.

The greedy strategy is to select one of the least worn out blocks. Some algorithms focus on the switching policy: Lofgren at al. patented that when the difference in erase count between the victim block and least worn block is more than a threshold (i.e., 15000) wear-leveling centric policy is used [98]. Woodhouse proposed that after every 100-th reclamation, one of the blocks containing only valid data pages will be randomly selected for reclamation [129]. The aim is to move static data to relatively more worn

out blocks. Ban proposed that a block will be randomly selected for reclamation after every certain number of reclamation [35]. This interval can be either deterministic or can be randomly determined.

Cleaning + Wear Leveling

This category of the garbage collection algorithms consider both cleaning efficiency and wear status during the reclamation decision.

Wells proposed a policy that uses a score, which is a weighted average of cleaning cost and wear leveling state of block [128]. The block with the highest score is selected as victim. The score of a block j is defined as $score(j) = \alpha * obsolete(j) + (1 - \alpha) * max_i\{erasures(i) - erasures(j)\}$, where $obsolete(j)$ stands number of invalid page in block j and $erasure(j)$ stands for current erase count of block j. First part of the scoring formula captures the cleaning cost, while the second part captures the impact on the wear leveling. The magnitude of the weight determines which part is will be given more priority during making the reclamation decision. Usually, $\alpha = 0.8$ is used to calculated the score. Thus, more importance is given to the cleaning cost. However, if the difference between most and least worn out block becomes more than threshold (i.e., 500), then $\alpha = 0.2$ is used to give more priority wear-leveling. Kim et al. proposed to define the score of a block j as $score(j) = \lambda(\frac{obsolete(j)}{valid(j)+obsolete(j)}) + (1-\lambda)\frac{erasure(j)}{1+max_i(erasure(i)}$, where λ is monotonic function of the difference between the erase count of most and least worn out blocks and $0 < \lambda < 1$ [82]. CAT score is defined as $score(j) = \frac{obsolete(j)*age(j)}{valid(j)*erasure(j)}$ [49]. Here, $age(j)$ is the time since the late erasure of unit j, and $\frac{obsolete(j)*age(j)}{valid(j)}$ part works like cost-benefit (CB) scheme described in Section 4.2.2, while $\frac{1}{erasure(j)}$ part gives priority to the least worn out blocks.

4.2.3 Reducing Garbage Collection Metadata

In this part, we describe the research works that have focused on reducing this metadata space consumption in the SRAM. These algorithms focus on the metadata needed for only wear leveling, i.e., block erase count, while does not consider to reduce computation overhead. In contrast, Our sampling-based approximation algorithm focuses on reducing space as well as computation overhead.

Group-based wear leveling algorithm [76] uses grouping to save SRAM space. It groups logically adjacent blocks into a single group, maintains an average erase count per group, thus it reduces SRAM space by group size times. However, this scheme specially designed for log-based block-level FTL address mapping, and it cannot be applied to DFTL [69], which is state-of-the-art FTL page mapping scheme. In addition, it still requires $O(K)$ space, where K flash capacity in total number of block. In contrast our sampling based approach requires $O(N)$ of SRAM space, where $N << K$.

K-Leveling [113] algorithm attempts to minimize metadata consumption by using coding like technique. It keeps track of the erase count of each block by storing only difference between current erase count and least worn block's erase count. To store an erase count of 100K, we need at least 20 bits of space. Now, if the difference between current erase count and least worn block's erase count can be restricted at max 31, we need only 5 bits to store the erase count information. Thus by saving only the difference, K-Leveling can reduce the metadata size by four times. Still it requires $O(K)$ space, while our sampling-based approach requires $O(N)$ space and $N << K$.

4.3 Sampling-based Algorithm

In this section, first we describe our sampling-based algorithm to approximate existing score-based garbage collection algorithms. Next, we illustrate our proposed algorithm with an example. Furthermore, we discuss how to select different sampling parameters. Finally, we analyze space and computation overhead.

4.3.1 Algorithm

Algorithm 1 gives a pseudo-code of our sampling based victim selection algorithm. At the very beginning (i.e., for the first time a victim block needs to be selected), we randomly read the metadata of the N blocks from the flash memory (Line 1 in Algorithm 1) to SRAM. Then, we calculate scores (the scores are described in Section 4.2.2) and sort these sampled metadata based on these scores (Line 6-7 in Algorithm 1). Finally, we select a block which has max (or min) score as a victim based on the garbage collection policy used (Line 8 in Algorithm 1). The performance of our proposed sampling-based algorithm depends on the quality of the randomly chosen samples. If the samples are

Algorithm 1 Sampling-based Algorithm

```
1: if (First Iteration) then
2:     Randomly select metadata for N fresh blocks
3: else
4:     Randomly select metadata for N − M fresh blocks
5: end if
6: Calculate scores based on metadata for N sampled blocks
7: Sort the scores in descending order
8: Select the block with max (or min) score as a victim
9: Remove metadata for the victim block from the sample
10: Remove metadata for last N − M − 1 blocks in descending order
```

good (i.e., samples contain the blocks which are more likely to be selected as victim), the overall performance is also good. That is why at end of each iteration, we keep M good samples from the current N samples, and throw out $N-M$ bad samples. We throw out the samples which are less likely to be selected as victim in the next iteration. Out of bad $N-M$ samples, one sample is the current victim block, and rest of the $N-M-1$ samples are chosen based on the eviction criteria. We throw out the last $N-M-1$ samples in sorted order (Line 9-10 in Algorithm 1). When we need to perform next victim selection, we again randomly read $N-M$ fresh samples from the flash to SRAM (Line 4 in Algorithm 1), and make eviction decision based on the metadata of these $N-M$ fresh samples and previously retained M samples. This procedure is repeated whenever we need to select a victim block.

4.3.2 A Working Example

To illustrate our sampling-based approach, we assume the victim selection criteria prefers the block with minimum erase count as the candidate for garbage collection. We name this as *Greedy_Wear* scheme. Now, will approximate *Greedy_Wear* scheme by keeping only metadata for $N = 5$ sampled blocks in the SRAM. We assume $M = 2$. At first, we randomly select metadata for five fresh sample blocks from the flash memory and bring them to SRAM. The block erase count of the samples are shown in Figure 4.2(a). Next, we sort the samples based on the erase count as shown in Figure 4.2(b). We select the block having the minimum erase count (i.e., 10) as victim and remove it metadata from SRAM. Now, we need to remove $N-M-1 = 5-2-1 = 2$ more samples from SRAM. Since, our victim selection criteria is the block having minimum erase count, we remove last two blocks in the ascending order of erase counts. The

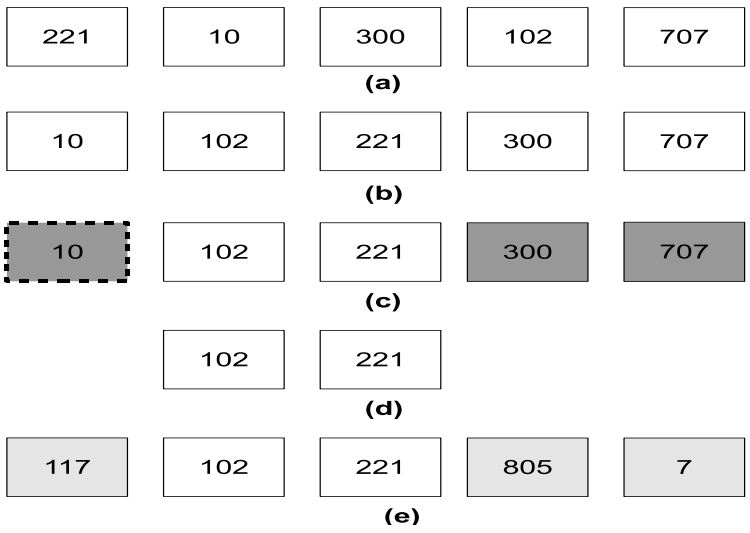


Figure 4.2: Illustration of sampling-based approximation algorithm for $N = 5$ and $M = 2$. Here, the block with minimum erase count is selected as a victim.

victim block is marked in the dashed rectangle and all the removed blocks are marked in gray color, as shown in Figure 4.2(c). After removing these three samples, currently we have only two metadata sampled blocks in SRAM, as shown in Figure 4.2(d). During the next victim selection process, again we randomly select metadata for three fresh sample blocks from flash memory and bring them to SRAM. These fresh samples (marked in light gray color) and two previously retained samples are shown in Figure 4.2(e). Now, from these five samples, we select the block having erase count of seven as a victim, and so on.

4.3.3 Parameter Selection

Usually, smaller values of M help to choose better victim candidates [115]. The main reason behind this behavior is that as the M of increases, fewer number of fresh samples are drawn from the flash memory, while larger number of old samples are retained in the memory to make the eviction decision. The performance degradation occurs due to bad samples. The optimal value of M has been estimated as $M_{opt} = N - \sqrt{(N+1) * \frac{100}{e}}$ [115]. Here, e is desirable error value. An error has been defined as the case, where

selected block does not belong to the e-th percentile of the blocks, which would satisfy the victim selection criteria. The difference $N - M$ contributes to number of times that we need to perform metadata read operations from the flash memory. Since, flash read operations much slower than the the SRAM operation, we need to keep $N - M$ as small as possible to quickly perform victim selection operation.

4.3.4 Space and Computation Overhead

Space Overhead. Sampling-based algorithm requires $O(N)$ space to store samples in SRAM, where N stands for sample size. In contrast, unsampled version of the same algorithm will require $O(K)$ space to store the priority queue [52], where K stands for flash capacity in terms of total number of blocks. Sampling-based algorithm uses a fixed value for N and $N << K$.

Computation Overhead. In unsampled case, irrespective of the victim selection, every metadata update operation requires to recalculate the score, which needs $O(lg(K))$ operation in the priority queue [52] to update it. Therefore to process r number of write requests, in total we need to perform $r * O(lg(K))$ SRAM operations (i.e., computation overhead). On the other hand, in our sampling-based implementation during victim block selection, we need to calculate score for the fresh $N - M$ samples, and sort N samples. Sorting needs $O(N * lg(N))$ operations. Thus, in total we need $O(N - M) + O(N * lg(N)) \approx O(N) + O(N * lg(N)) \approx O(N * lg(N))$ operations. For analysis, we assume that a block contains p number of pages and after every $a * p$ write requests, where $a \geq 1$, we need to perform garbage collection (i.e., need to select victim blocks). Therefore, to process r number of write requests, we need to perform victim block selection operations for $\frac{r}{a*p}$ number of times. We need in total $\frac{r}{a*p} * O(N * lg(N))$ SRAM operations, while unsampled version needs $r * O(lg(K))$ operations. Usually, $\frac{1}{a*p} * O(N * lg(N)) < O(lg(K))$, thus our sampling-based algorithm needs lesser number of SRAM operations. For example, by putting $K = 4096, N = 32, p = 64$, and $a = 1$, for unsampled case, we get $lg(4096) = 12$. While for sampling, we get $\frac{1}{64} * 32 * lg(32) = \frac{1}{64} * 32 * 5 = 2.5 < 12$. As the flash size increases, K increases proportionately, whereas number of samples (N) does not change much (this will be validated in Section 4.4). By putting $K = 4096 * 1024$ and keeping other values fixed, for unsampled case, we get $lg(4096 * 1024) = 22$. In contrast, for sampling, we get $\frac{1}{64} * 32 * lg(32) = 2.5 < 22$.

Therefore, it is evident that sampling-based approximation algorithm will incur less SRAM operations.

4.4 Experimental Evaluation

In this section, we study the performance of our sampling-based approximation algorithms using real-world trace driven simulation. We approximate the garbage collection schemes using our sampling algorithm, and compare their performance according to various garbage collection evaluation metrics.

4.4.1 Experimental Setup

Simulator

To evaluate the performance of our sampling-based scheme, we have used Solid State Drive (SSD) simulator developed by The Pennsylvania State University [1]. This simulator is an extension of Disksim [19], a widely-used simulator to study disk performance. Recently, this extended simulator has been used to evaluate the DFTL [69], which is the state-of-the-art FTL scheme. We have modified the garbage collection modules in this simulator to test our sampling-based schemes.

Workloads

We have used three enterprise class real-world traces to study the impact of the sampling-based implementation of the various garbage collection algorithms.

- **UMASS Financial Traces.** We use two write-intensive block I/O traces from the UMass Storage Repository [21]. These traces data are collected from the online transaction processing (OLTP) applications running at two large financial institutions. We refer these two traces as **Financial-1** and **Financial-2** in the rest of this paper.

- **MSR Cambridge Trace.** We use a block I/O trace, *src1 volume0*, collected at Microsoft Research Cambridge Lab from SNIA IOTTA Repository [17]. We have extracted only the writes requests and ignored all the read requests. This

particular trace is used since it has relatively large mean and peak write requests rate [105]. We refer this trace as **MSR-Trace** in the rest of this paper.

Simulation Setup

We have simulated a large-block NAND flash-based SSD using the parameters shown in Table 4.2. In our evaluation, DFTL [69] is used as underlying page address mapping scheme. We use only part of the flash memory as an active area to store our test workloads, and rest of the space is considered read-only (i.e., cold data). This is similar to the DFTL [69] scheme's evaluation setup. For the three test workloads, we restricted the active regions in the following way: Financial-1 (2GB and 4GB), Financial-2 (2GB), and MSR-Trace(8GB). Using different sized active regions help us to understand the impact of sampling with the variation in flash capacities. For the Financial traces, we remove the requests that are outside active regions, and for MSR-Trace we truncate those requests to fit in the active region. We process X millions requests for XGB case, i.e., 2 millions for 2GB, to ensure that lot of updates operations happen and garbage collection are invoked frequently. We assume that initially full active region is free, and when less than 3% free space available, garbage collection is invoked. To collect statistics, we process each trace twice. First run is used as warm up phase, while second run is used for reporting statistics.

We have used the following three settings for the evaluation of sampling-based algorithm: (a) $N = 3, M = 1$, (b) $N = 8, M = 2$, and (c) $N = 30, M = 5$. We refers these three settings as **N3M1, N8M2**, and **N30M5**, respectively, in the rest of this section. The setting **N3M1** refers that we maintain only $N = 3$ samples in SRAM to select a victim block for garbage collection, and once a victim is selected, we retain $M = 1$ old sample for the next iteration. During next iteration, we draw $N - M = 3 - 1 = 2$ fresh samples, and make a victim selection decision from the two fresh samples and one carried sample from the previous iteration. The $N - M = 2$ samples contribute to the victim selection overhead in terms of flash page read operations.

Policies Evaluated

We approximate four different garbage collection schemes using sampling-based algorithm. Out of four schemes, the first scheme focuses only on cleaning cost, the second

Parameter	Value
Block Size	256 KB
Page Size	4 KB
Data Register Size	4 KB
4KB-Page Read to Register Time	25 μs
4KB-Page Write Time from Register	200 μs
Serial Access time to Register (Data bus)	100 μs
Block Erase Time	1500 μs
Program/Erase Cycles	100000

Table 4.2: Parameters values for NAND flash memory [31]

scheme only focuses on wear leveling, the third scheme is used in the state-of-the-art page mapping scheme, and the fourth scheme focuses on cleaning cost and wear leveling.

- **Greedy_Clean.** This greedy scheme selects a block with largest number invalid pages. It is a cleaning efficient scheme.
- **Greedy_Wear.** This greedy scheme selects a block with the least erase count. It is wear leveling efficient scheme.
- **CB.** The Cost-benefit (CB) scheme [78] selects the block with highest utility $\frac{1-u}{2u} * age$, where u stands for block utilization (i.e., $u = \frac{valid(j)}{valid(j)+obsolete(j)}$, $valid(j)$ and $obsolete(j)$ refer to the number valid and invalid pages, respectively, in block j), and age is the time since last invalidation. The state-of-the-art space-efficient page address mapping scheme, DFTL [69], uses CB scheme.
- **CAT.** This scheme defines score as $\frac{obsolete(j)*age(j)}{valid(j)*erasure(j)}$. Here, $obsolete(j)$ is the number of invalid page, $valid(j)$ is the number of valid pages, $erasure(j)$ is the current erase count, and $age(j)$ it the time since the last time the the block j has been erased [49]. The block with the highest score is selected for garbage collection.

Evaluation Metric

We use the following four metrics to evaluate our sampling-based implementation of the garbage collection policies mentioned in Section 4.4.1 compared to the implementation

of their unsampled versions.

- **Variance of Erase Counts.** Variance of the erase counts indicate the impact of the garbage collection scheme in wear-leveling. A lesser variance indicates that most of the blocks are wear out evenly. The lesser the variance is, the better is the garbage collection scheme, as it helps to increase the lifetime of a flash-based storage device.

- **Max Erase Count.** Since a flash block can be erased for a limited number of times, this metric indicates how fast that limit has been reached. A good garbage collection scheme has a smaller value for this metric.

- **GC Overhead.** GC (garbage collection) overhead metric indicates how many valid pages need to be moved during garbage collection to claim free space from victim blocks. A lesser of value of this overhead indicates that the victim blocks contain lot of invalid pages. Good garbage collection schemes have smaller values for this metric.

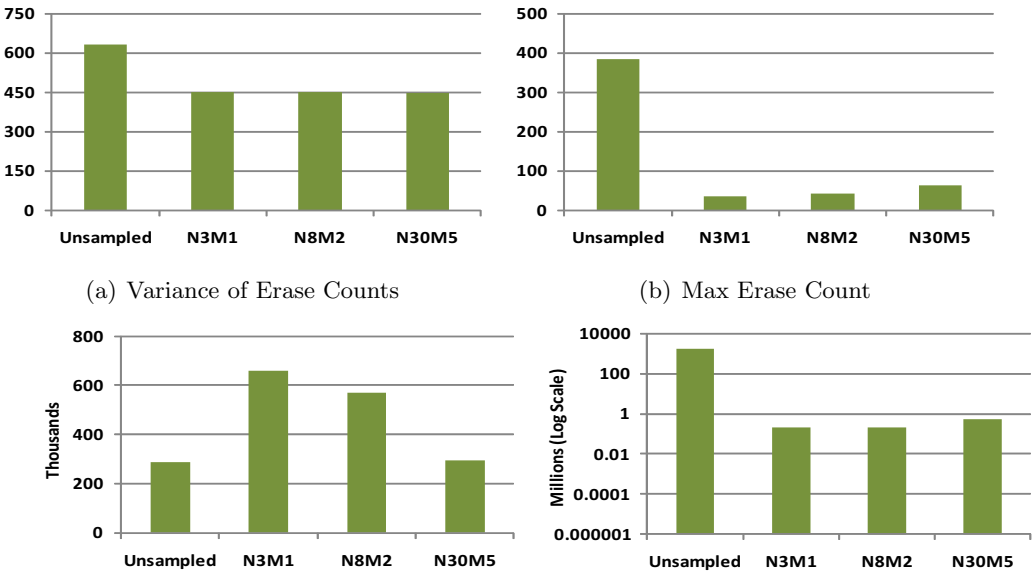
- **Victim Selection Overhead.** This metric indicates the overhead (in terms of number of pages) incurred due to reading of the metadata from the flash memory during the victim selection when the metadata information needed for garbage collection can not stored in SRAM.

4.4.2 Result Analysis

Financial-1 (2GB)

Greedy_Clean. We evaluate how well sampling-based (i.e., randomized) Greedy_Clean algorithm approximates the unsampled (i.e., original) Greedy_Clean scheme. Figure 4.3 shows the performance statistics.

In terms of *erase counts variance* and *max erase count*, sampling-based algorithm works better than unsampled one. Figure 4.3(a) shows that *variance* is almost same for all three sampling settings, while Figure 4.3(b) shows that *max erase count* becomes higher with the increase of sample size. At a first glance, these results seem counter-intuitive, as it is expected that unsampled version will always outperform sampling-based version, after all it is an approximation of the unsampled one. However, these



(a) Variance of Erase Counts

(b) Max Erase Count

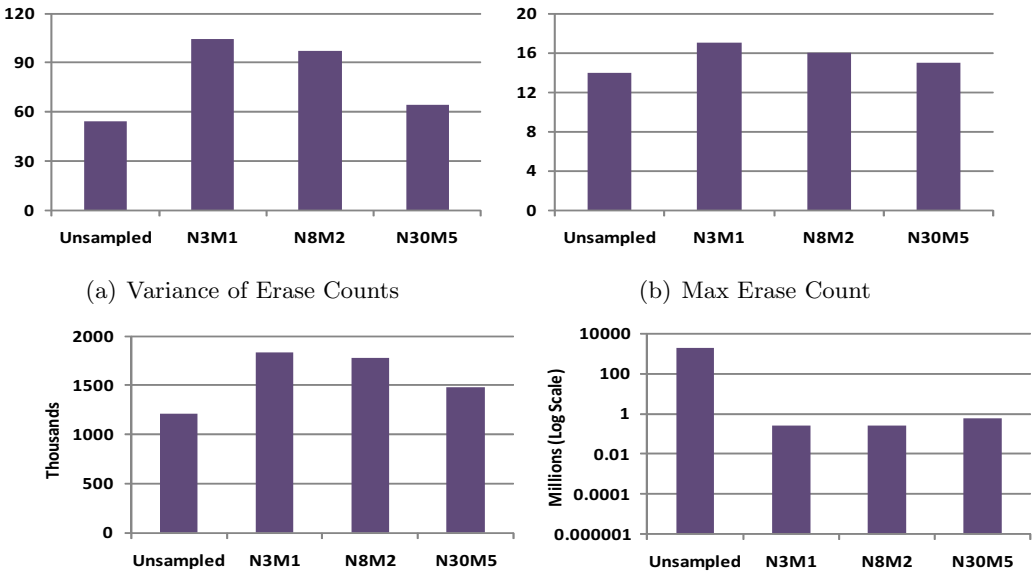
(c) GC Overhead

(d) Victim Selection Overhead

Figure 4.3: Greedy_Clean Scheme for Financial-1 (2GB)

results are correct. The main reason is as follows. Greedy_Clean scheme is a cleaning efficient policy and it does not consider wear leveling factor at all. *Erase count variance* and *max erase count* metrics are biased to wear leveling. Sampling-based algorithm does not enough information (i.e, does not consider all blocks metadata) to make 100% cleaning effcient decision as it is an approximation (i.e, makes decisions from the metadata of a small number of sampled blocks), which makes it inherently biased to the wear leveling metrics. That is why unsampled version performs worse for these two metrics. With more samples, the sampling-based algorithm becomes closer approximation of the unsampled one, and that is why with an increase in sample size, *max erase count* becomes larger and difference between *max erase count* values of the unsampled and sampling-based algorithm decreases.

Figure 4.3(c) shows that in terms of *garbage collection overhead*, unsampled version performs better. As expected, with an increase in sample size, sampling-based algorithm exhibits performance similar to the unsampled one. These results are intuitive, since Greedy_Clean scheme is specially optimized for garbage collection overhead, quite naturally it will perform better. The more the samples, the better the approximation,



(a) Variance of Erase Counts

(b) Max Erase Count

(c) GC Overhead

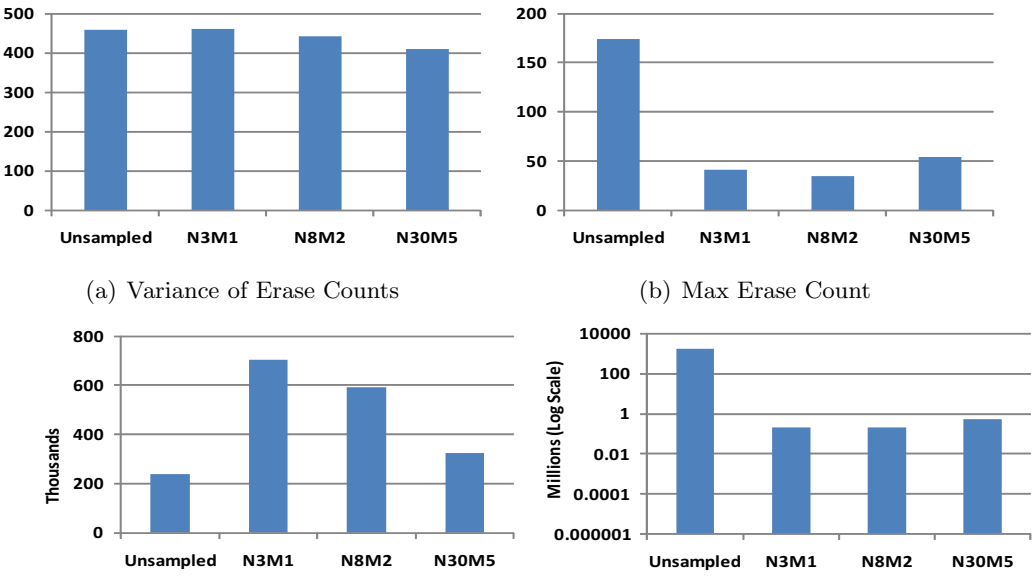
(d) Victim Selection Overhead

Figure 4.4: Greedy_Wear Scheme for Financial-1 (2GB)

the closer is the performance. Figure 4.3(c) shows that for **N30M5** i.e, $N = 30$ and $M = 5$, sampling-based algorithm performs almost as good as unsampled one.

Figure 4.3(d) shows that when metadata is not stored in the SRAM, to select a victim the overhead in terms of number of flash read operations. This overhead is several magnitude lesser for the sampling-based algorithm compared to unsampled one. This is expected, as sampling case, we need to read only fewer number of sampled blocks, while unsampled algorithm scans all blocks to select a victim. As expected, in the sampling-based algorithm, as the difference $N - M$ increases (i.e., number fresh blocks that needs to be read), the overhead also increases proportionately. Overall, $N = 30$ and $M = 5$, shows better performance compared to the other sampling settings.

Greedy_Wear. Figure 4.4 shows the performance of sampling-based Greedy_Wear algorithm compared to the unsampled (i.e., original) version. In terms of *erase counts variance* and *max erase count*, sampling-based algorithm performs worse than sampled one. However, Figure 4.4(a) and Figure 4.4(b) show that as the sample size increases, *variance* and *max erase count* become closer to the unsampled one. This results are quite expected as Greedy_wear is wear leveling centric policy, and these two metrics biased



(a) Variance of Erase Counts

(b) Max Erase Count

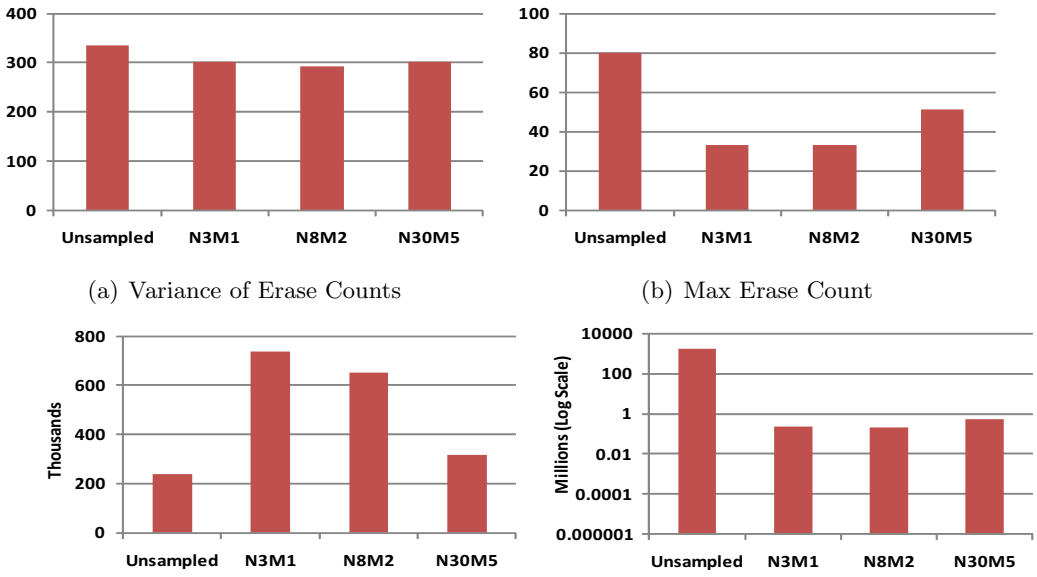
(c) GC Overhead

(d) Victim Selection Overhead

Figure 4.5: CB Scheme for Financial-1 (2GB)

to wear leveling. That is why unsampled version outperforms sampled version. With more samples (i.e., $N = 30$), sampling-based algorithm becomes closer approximation of the unsampled one, and exhibits closer performance. Figure 4.4(c) shows that in terms of *garbage collection overhead* unsampled version performs better. As expected, again with more samples, sampling-based algorithm approximates better, and exhibits better performance. Figure 4.4(d) shows that with the increase in the difference $N - M$, the victim selection overhead becomes larger. Overall, $N = 30$ and $M = 5$, shows better performance compared to the other settings.

CB. Figure 4.5 shows the performance of sampling-based approximated CB algorithm compared to the unsampled version. Like Greedy_Clean algorithm, CB algorithm is also a cleaning centric scheme, while it considers more metadata information to make victim selection decision. In terms of *erase count variance* and *max erase count*, sampling-based algorithm performs better than unsampled one. The reasons are same as the case of Greedy_Clean scheme explained earlier. In terms of *garbage collection overhead* and *victim selection overhead*, Figure 4.5(c) and Figure 4.5(d), respectively, show that CB scheme exhibits similar trends as the Greedy_Clean scheme. Note that, $N = 30$ and



(a) Variance of Erase Counts

(b) Max Erase Count

(c) GC Overhead

(d) Victim Selection Overhead

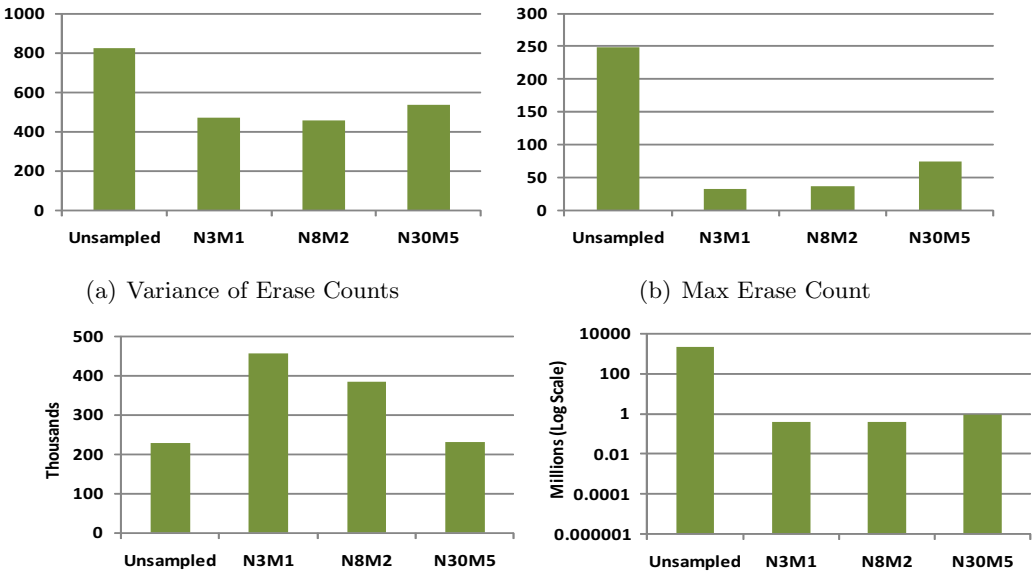
Figure 4.6: CAT Scheme for Financial-1 (2GB)

$M = 5$, exhibits overall better performance compared to the unsampled version.

CAT. Figure 4.6 shows the performance of sampling-based CAT algorithm compared to the unsampled version. CAT algorithm considers both cleaning and wear leveling in order to make victim selection decision. Figure 4.6(a)-Figure 4.6(d) shows that CAT scheme exhibits similar trends as the CB scheme. This is quite expected, as CAT is a combination of CB scheme and wear leveling factor $\frac{1}{erasure(j)}$ as described in Section 4.2.2. Due to this extra wear leveling factor, *variance* and *max erase count* values are relatively smaller for the CAT score compared to the CB scheme. Even for this scheme, $N = 30$ and $M = 5$, shows better performance compared to the other settings.

Financial-1 (4GB)

Now, to understand the impact of the flash capacity of the sampling scheme, we doubled the active region, from 2GB to 4GB. Figures 4.7 - 4.10 show performance numbers for Greedy_Clean, Greedy_Wear, CB, and CAT scheme, respectively, according to various garbage collection evaluation metrics. It is interesting that even with size doubling,



(a) Variance of Erase Counts

(b) Max Erase Count

(c) GC Overhead

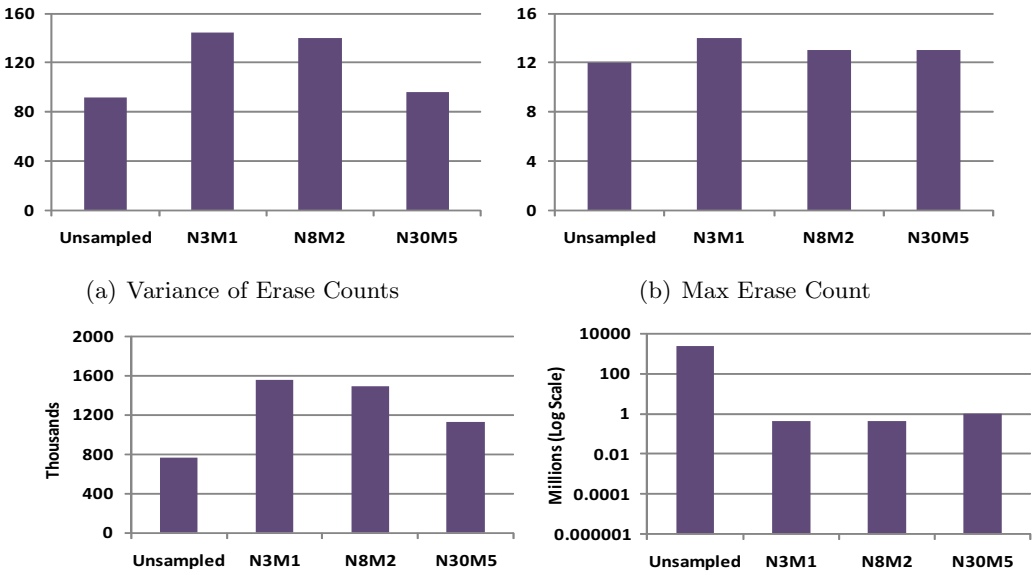
(d) Victim Selection Overhead

Figure 4.7: Greedy_Clean Scheme for Financial-1 (4GB)

trends between sampled and version remains same as Financial-1 (2GB). This is a very important result, since with increase in flash capacity, the space and processing time in SRAM increases linearly as explained in Section 4.3.4, while with the same number of samples, approximation algorithms provide similar performance trends. Thus, space and processing overhead is constant for sampling-based algorithms. For this trace, $N = 30$ and $M = 5$, shows overall better performance compared to the other settings.

Financial-2 (2GB)

Figures 4.11-4.14 show the performance of sampling-based approximated algorithm for Greedy_Clean, Greedy_Wear, CB, and CAT scheme, respectively. Financial-2 workload also shows similar trends as Financial-1 workload. In addition, $N = 30$ and $M = 5$ setting provides overall better performance.



(a) Variance of Erase Counts

(b) Max Erase Count

(c) GC Overhead

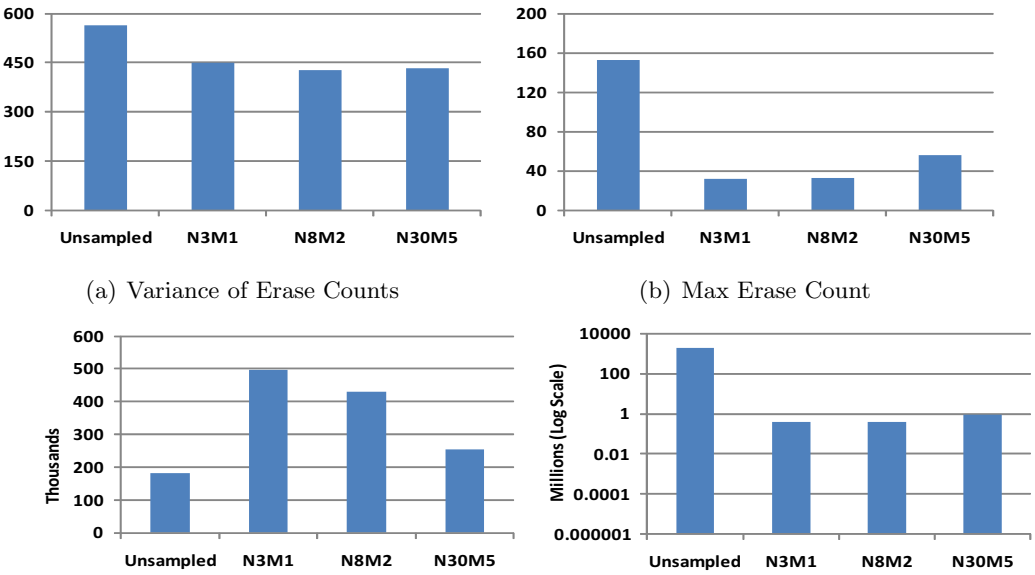
(d) Victim Selection Overhead

Figure 4.8: Greedy_Wear Scheme for Financial-1 (4GB)

MSR-Trace (8GB)

Since both Financial-1 and Financial-2 show similar trends, in order to understand different characteristics of the garbage collection, i.e., when lot of block contains mostly invalid pages, we have processed MSR-Trace differently. In addition, we have made active space larger, to study whether the same sampling settings scale well with increase in the flash capacity. We restricted write operations to 8GB area, any request falling outside this limit, has been modified by performing modulo by 8GB to fit in the 8GB region.

Figures 4.15-4.18 show the performance of sampling-based approximated algorithm for Greedy_Clean, Greedy_Wear, CB, and CAT scheme, respectively. These results also exhibit similar trends as Financial-1 and Financial-2 workloads. Since magnitude of the some statistics are very large, we have used logarithmic scales to represent them so that relatively smaller values will be visible. Except Greedy_Wear, other three schemes show almost same trends. The main reason as follows. These three schemes consider cleaning cost, and since lot of blocks contain only invalid pages, all these schemes mostly select



(a) Variance of Erase Counts

(b) Max Erase Count

(c) GC Overhead

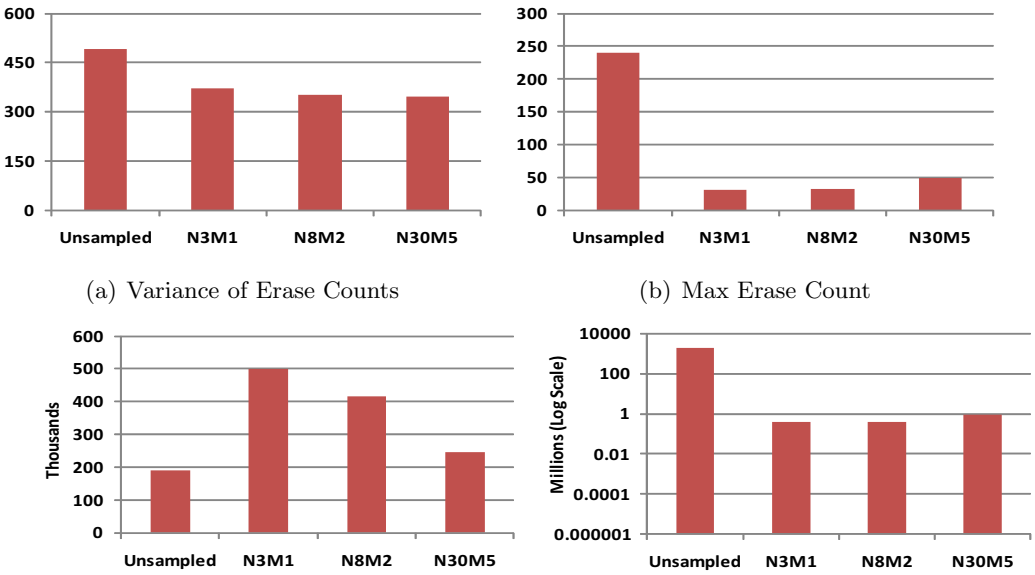
(d) Victim Selection Overhead

Figure 4.9: CB Scheme for Financial-1 (4GB)

those blocks as victims. As expected, with more samples, the approximation becomes closer to the unsampled version. This is specially evident in the *garbage collection overhead* performance results. With more samples ($N = 30$), garbage collection overhead becomes as as low as unsampled version. On the other hand, Greedy_Wear is wear leveling centric policy that is why it exhibits different characteristics, and shows relatively smaller *erase count variance* and *max erase count* values compared to other three schemes. Both of these values are biased to the wear leveling centric scheme. For this trace, overall, $N = 30$ and $M = 5$, shows better performance compared to the other settings.

Result Summary

For the three real-world traces, we find that small number of samples, i.e., $N = 30$ and $M = 5$, can approximate existing garbage collection algorithms very well. More specifically, the number of samples needed for approximation does not increase with the



(a) Variance of Erase Counts

(b) Max Erase Count

(c) GC Overhead

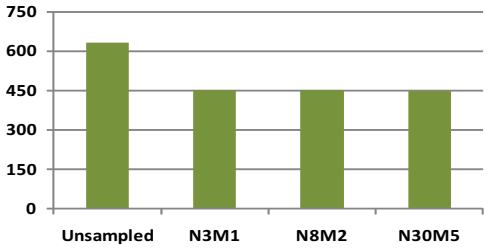
(d) Victim Selection Overhead

Figure 4.10: CAT Scheme for Financial-1 (4GB)

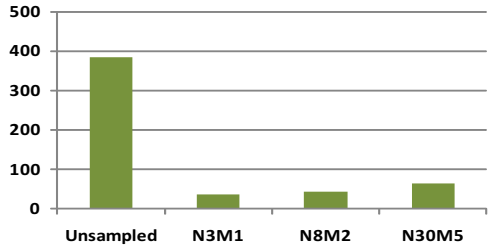
increase in flash-based storage capacity (i.e., 2GB, 4GB, and 8GB). Thus, sampling-based approach looks very promising. However, we need to perform more analysis and experiments in order to verify and validate, whether the number of samples does not depend on the capacity of flash storage or not, in other words, small number of samples can work for any size of flash storage. We will address this issue in the future.

4.5 Conclusion

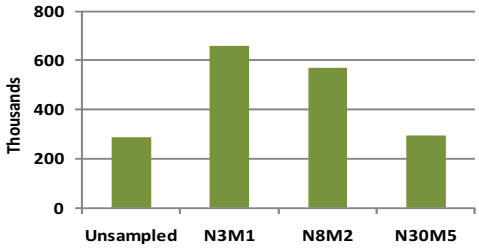
This paper proposes a sampling-based (i.e, randomized) approach to approximate existing garbage collection algorithms. Our approach is very easy to implement and it takes very less space in SRAM as well as incurs less computation overhead. It greatly simplifies the garbage collection management in a flash-based storage controller. Our experimental results show that sampling-based approximation algorithms are wear leveling friendly and they scale very well for the large-capacity flash-based storage.



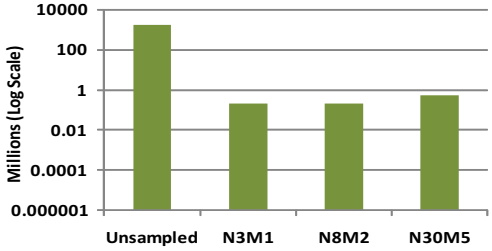
(a) Variance of Erase Counts



(b) Max Erase Count

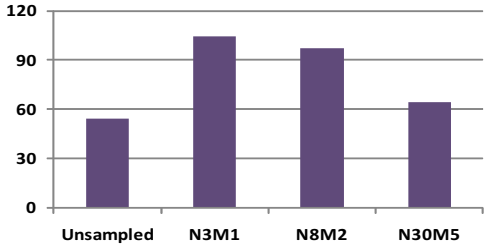


(c) GC Overhead

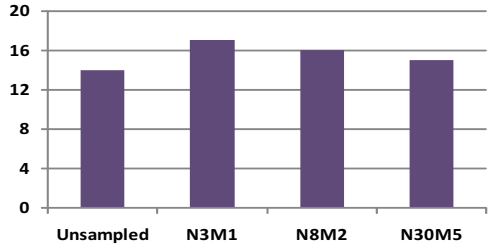


(d) Victim Selection Overhead

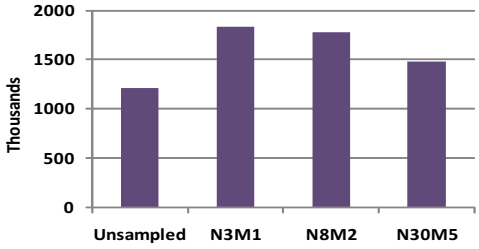
Figure 4.11: Greedy_Clean Scheme for Financial-2 (2GB)



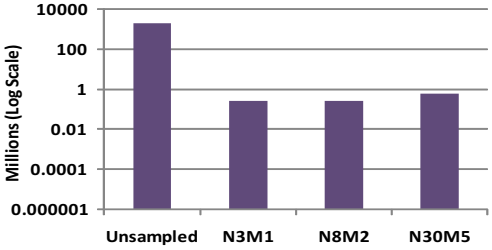
(a) Variance of Erase Counts



(b) Max Erase Count

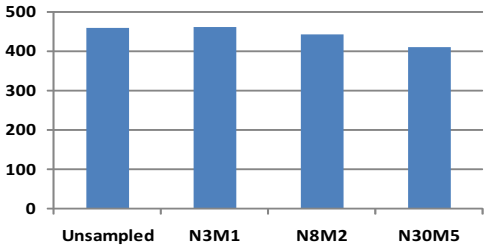


(c) GC Overhead

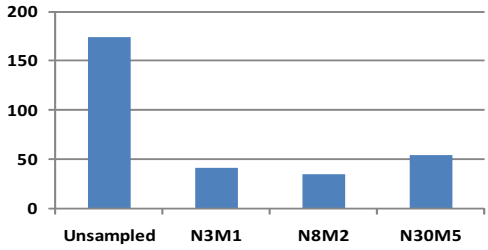


(d) Victim Selection Overhead

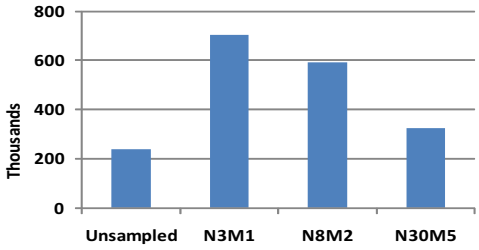
Figure 4.12: Greedy_Wear Scheme for Financial-2 (2GB)



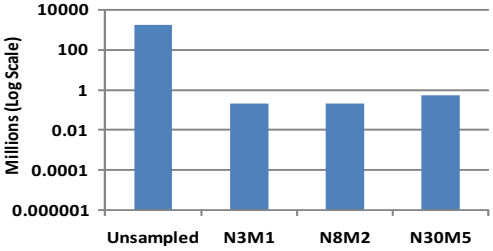
(a) Variance of Erase Counts



(b) Max Erase Count

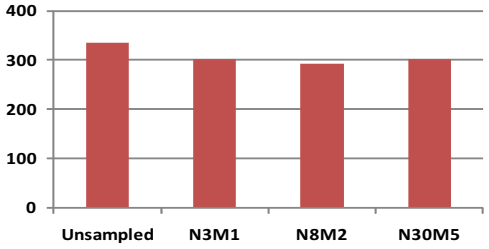


(c) GC Overhead

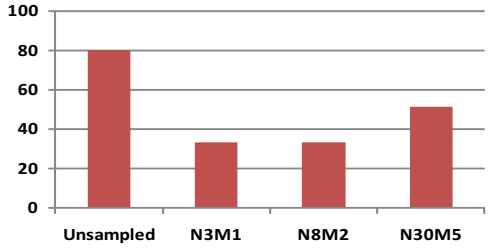


(d) Victim Selection Overhead

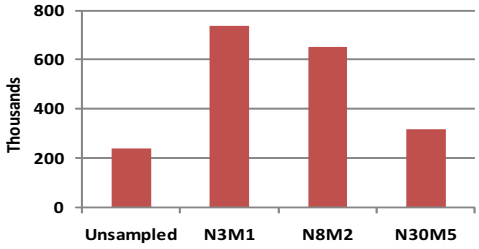
Figure 4.13: CB Scheme for Financial-2 (2GB)



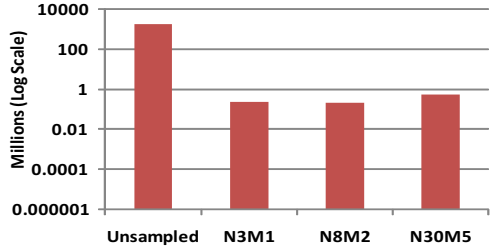
(a) Variance of Erase Counts



(b) Max Erase Count

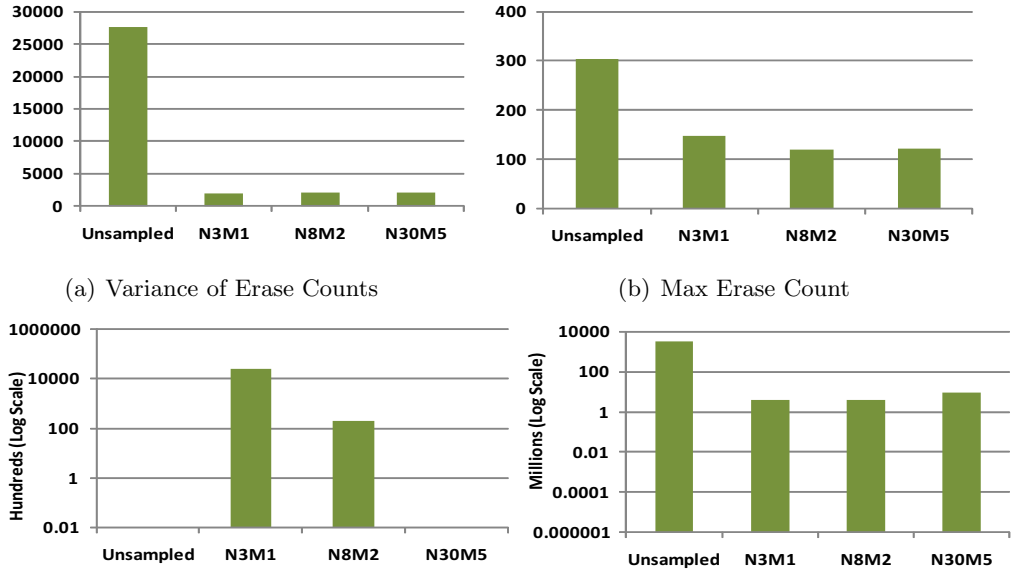


(c) GC Overhead



(d) Victim Selection Overhead

Figure 4.14: CAT Scheme for Financial-2 (2GB)



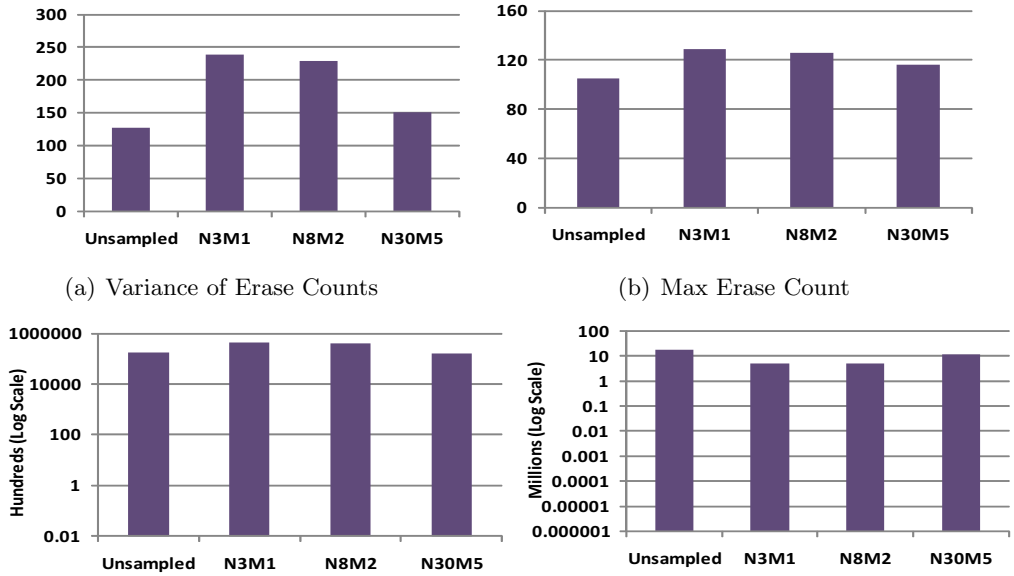
(a) Variance of Erase Counts

(b) Max Erase Count

(c) GC Overhead

(d) Victim Selection Overhead

Figure 4.15: Greedy_Clean Scheme for MSR-Trace (8GB)



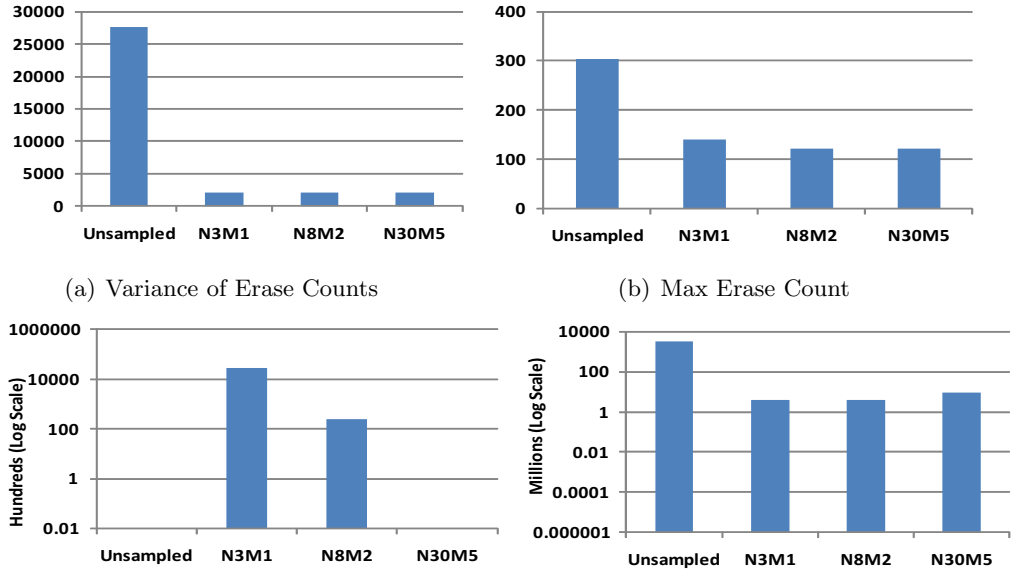
(a) Variance of Erase Counts

(b) Max Erase Count

(c) GC Overhead

(d) Victim Selection Overhead

Figure 4.16: Greedy_Wear Scheme for MSR-Trace (8GB)



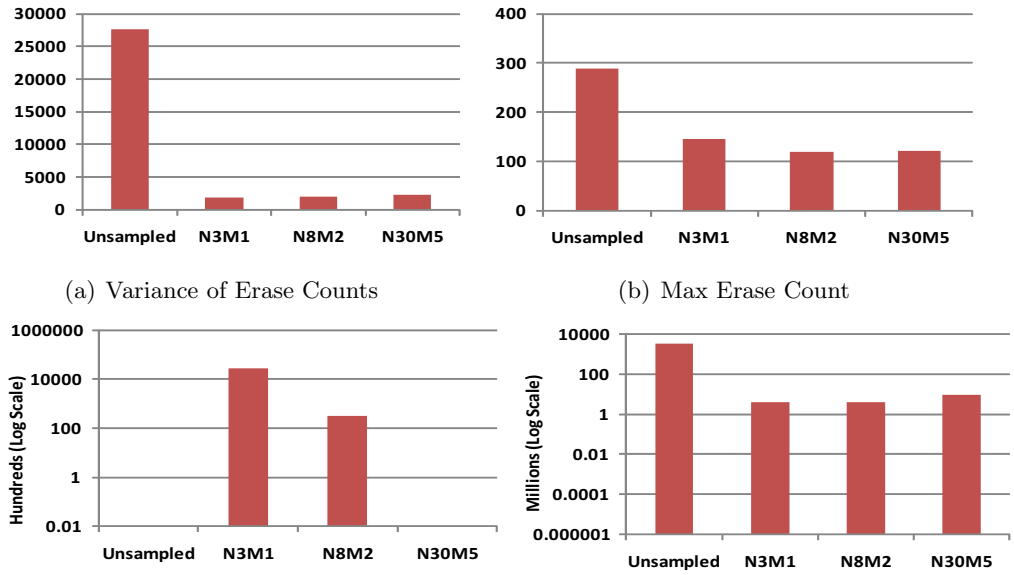
(a) Variance of Erase Counts

(b) Max Erase Count

(c) GC Overhead

(d) Victim Selection Overhead

Figure 4.17: CB Scheme for MSR-Trace(8GB)



(a) Variance of Erase Counts

(b) Max Erase Count

(c) GC Overhead

(d) Victim Selection Overhead

Figure 4.18: CAT Scheme for MSR-Trace (8GB)

Chapter 5

Deferred Updates for Flash-based Storage

5.1 Introduction

NAND flash memory is increasingly adopted as main data storage media in mobile devices, such as PDAs, MP3 players, cell phones, digital cameras, embedded sensors, and notebooks due to its superior characteristics such as smaller size, lighter weight, lower power consumption, shock resistance, lesser noise, non-volatile memory, and faster read performance [64, 106, 90, 107]. Recently, to boost up I/O performance and energy savings, flash-based Solid State Disks (SSDs) are also being increasingly adopted as a storage alternative for magnetic disk drives by laptops, desktops, and datacenters [11, 22, 103, 81]. Due to the recent advancement of the NAND flash technology, it is expected that NAND flash based storages will greatly impact the designs of the future storage subsystems [81, 91, 31, 90].

A distinguishing feature of flash memory is that read operations are very fast compared to magnetic disk drive. Moreover, unlike disks, random read operations are as fast as sequential read operations as there is no mechanical head movement. However, a major drawback of the flash memory is that it does not allow *in-place* update (i.e., overwrite) operations. Chapter 2 gives an overview of a flash-based storage device. In flash memory, data are stored in an array of flash blocks. Each block spans 32-64 sectors, where a sector is the smallest unit of read and write operations. Sectors are

also commonly referred as pages. However, in the rest of this chapter, we use sector in order to distinguish it from a conventional database page. Sector write operations in a flash memory must be preceded by an erase operation. Within a block, sectors must be written sequentially (in low to high address order) [31]. The *in-place* update problem becomes complicated as write operations are performed in the sector granularity, while erase operations are performed in the block granularity. The typical access latencies for *read*, *write*, and *erase* operations are 25 microseconds, 200 microseconds, and 1500 microseconds, respectively [31]. In addition, before an erase operation is being done on a block, the live (i.e., not over-written) sectors from that block need to be moved to pre-erased blocks. Thus, an erase operation incurs lot of sectors read and write operations, which makes it a performance critical operation. Besides the asymmetric read and write latency issue, flash memory exhibits another limitation: a flash block can only be erased for limited number of times (e.g., 10K-100K) [31].

Faster read performance of the flash memory will be particularly good to speed up the read-intensive type workloads, e.g., decision support systems (DSS). However, flash memory can produce poor performance when used for other workloads that require frequent random update operations. Examples of such workloads include online transaction processing (OLTP), mobile applications, and spatio-temporal applications. These update-intensive applications perform a lot of small-to-moderate size random write operations that are much smaller than the flash sector size [90]. This will be very problematic for the flash memory as once data is written in a flash sector, no further data can be written without erasing the entire block containing that sector. Thus, update-intensive applications suffer from performance degradation as erase operations are much slower than write operations. Moreover, frequent erasure of the blocks will decrease lifetime of flash memory.

The flash translation layer (FTL) is an intermediate layer inside a flash-based storage (as explained in Chapter 2) that hides the internal details of flash memory (i.e., *in-place* update problem) and allows existing disk-based application to use flash memory without any significant modifications [69]. Thus, update-intensive applications can be greatly benefited by using a flash-based storage equipped with FTL. However, not all flash-based storage devices use FTL [37, 106]. For these devices, flash driver can provide the internal flash information. In some cases, operating system provides FTL

functionality. For example, Windows Mobile emulates FTL in software [106]. In this chapter, we are focusing on this type of flash devices. In addition, we are focusing on flash-based storage with reconfigurable FTL. The intuition of using a reconfigurable FTL is as follows. For the most of flash-based storage products available in the market, internal hardware architectures and FTL designs are not well known [31]. As a result, applications are designed (or modified) based on generic FTL behavior. Although this is adequate for the applications designed for the general computing environment, but for the environments (i.e., high end servers or supercomputers) running a fixed set of applications (i.e., database management systems), huge performance gain could be obtained by using customized flash-based storages designed with application specific FTL. This benefit is demonstrated by a recent project [122], which implements FTL in a reconfigurable hardware (i.e., field-programmable gate array). Each running application has access to FTL and it can reconfigure FTL design based on its own requirements.

In this paper, we propose a novel hierarchical update processing strategy, named *deferred update methodology*, which significantly improves the *in-place* update processing overhead and longevity of the flash-based storage used for a database management system (DBMS). Our goal is to reduce the number of expensive erase operations due to the processing of *in-place* update operations through two intermediate flash storage layers. The main idea is that we always write the changes due to newly incoming updates as logs to the first intermediate layer. Once first layer is full, to make free space we populate the logs from the first layer to the second layer. The first layer acts as a scratch area for the second layer. Finally, when the second intermediate layer is also full, we populate its contents into their actual locations in the flash erase units. These two layers help to batch a set of update logs for the same erase unit together. Finally, we can apply them at once. This results in a huge saving of erase operations where a block is erased only once for a set of bulk updates. Since erase is the most expensive operation, therefore reduction of the number of erase operations helps to improve write performance. On the other hand, this will also help to increase the lifetime of the flash memory due to the limited number of erase operations allowed per block. This work has been published in the MSST 2010 conference [54].

The *deferred update methodology* raises the challenge of data retrieval from the flash memory as a certain record may exist in three different places, i.e., the original record

is stored in its original erase unit, then an update of this record is stored as log in the first intermediate layer. Finally, another update of that record may be stored as log in the second intermediate layer. It is nontrivial to retrieve data in an efficient way that achieves a trade-off between the faster data retrieval and the complication of processing data updates. We use a flash-friendly index to speed up the data retrieval process.

Our experimental results show that *deferred update methodology* performs significantly improve the write processing time and incurs fewer number of erase operations compared to the state-of-the-art in-page logging (IPL) technique [90].

Our key contributions can be summarized as follows.

- A hierarchical update processing methodology named as *deferred update methodology* to improve slow write performance and lifetime of the NAND flash memory. The cost of achieving such improvements is only few flash memory blocks.
- A thorough theoretical analysis of the trade-offs in terms of erase operations, space overhead, and data retrieval overhead for different alternative designs compared to the *deferred update methodology*.

The remainder of the chapter is organized as follows. Section 5.2 describes the related work. Section 5.3 describes the *deferred update methodology* in detail. Section 5.4 gives analytical models of the different alternative designs. Section 5.5 explains our experimental results. Finally, Section 5.6 concludes the discussion.

5.2 Related Work

There is substantial recent interest in utilizing flash memory for non-volatile storage in applications including databases [90, 91, 88, 121, 120, 96, 66, 48, 126, 37, 30] and sensor networks [107, 106, 132]. Here, we are discussing the works that are related to the writing issues of the flash memory. The existing works can be classified into two main categories. **(1)** designing flash-friendly data structures. For example, MicroHash [132], FlashDB [107], random sampling data structure [106], FD-tree [96], and Lazy-Adaptive Tree [30] propose new or modified index structures for flash based storage. However, these works cannot be directly extended to improve flash memory's update processing problem as they mainly target specific index structures. In contrast, our goal is to

design a generic solution which helps database table spaces as well as index structures. **(2)** Improving update processing performance for the flash based database servers. This includes in-page-logging (IPL) technique [90]. Our work also falls into this category. In the rest of this section, we discuss IPL in detail and distinguish our work from it.

The state-of-the-art technique of handling updates in flash memory is the in-page logging (IPL) approach [90]. The main idea of IPL is to reserve one of data pages in a flash erase unit as log page for storing the update logs. Each page consists of multiple flash sectors. When a data page becomes dirty, the changes are recorded as update logs in an in-memory update log sector. Once the log sector becomes full or dirty data page is evicted from the buffer, then the in-memory update log sector is written to the corresponding log page. Whenever a log page becomes full, the update logs in the log page are combined with the original data records in the data pages in that erase unit. The first problem of IPL is that if the data pages have very little update locality, then in-memory log sectors will contain small amount of data, as a result *log page* space will be under-utilized. This space under-utilization accelerates frequent erase operations, which will slow down performance and affects the lifetime of the flash memory. Second problem with IPL is that if power goes off, the update logs stored in the memory will be lost, thus data inconsistency problem will arise. Our work in this thesis targets to develop an efficient update processing methodology that (1) reduces the number of erase operations, (2) increases the space utilization, and (3) avoids data consistency problem.

5.3 Deferred Update Methodology

In this section, we present the *deferred update methodology* that aims to improve write performance and increase lifetime of the flash-based database systems, by minimizing the number of erase operations. The outline of this section is as follows. Section 5.3.1 gives an overview of our methodology. Section 5.3.2 gives a brief description of the update log, timestamp, and index structure. Section 5.3.3 explains in detail how update operations (i.e., INSERT, UPDATE, and DELETE) are processed. Section 5.3.4 explains how queries are processed.

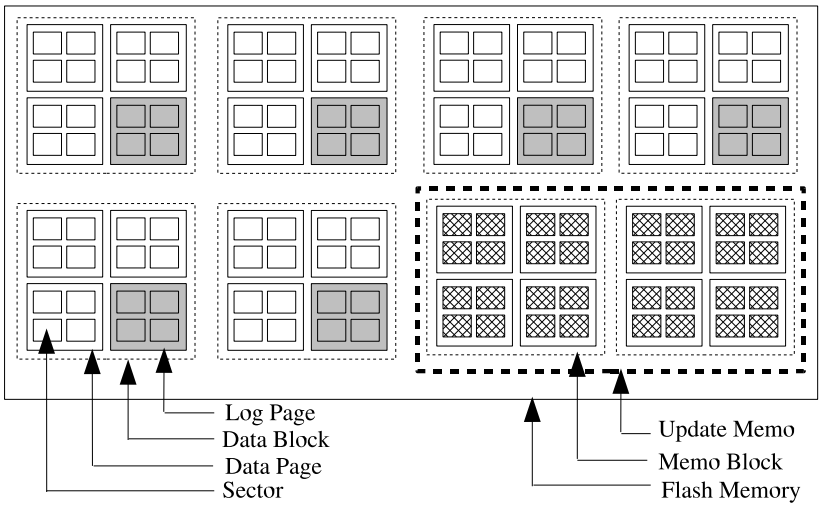


Figure 5.1: Logical Flash Memory view for a DBMS in the *deferred update methodology*

5.3.1 System Overview

The main idea of *deferred update methodology* is to process updates through an intermediate two-level storage hierarchy consisting of an *update memo* and *log page(s)*. This intermediate layer helps to reduce the number of expensive erase operations. Conceptually, we group the database pages by the erase unit containing them and named as *data blocks*, while *update memo* is a set of erase units which is used as a scratch space. In each *data block*, some data pages are also reserved for storing *update logs*. We name these reserved pages as *log pages*. Figure 5.1 represents a logical view of an NAND flash memory that employs our proposed *deferred update methodology*. The boundary of flash memory is depicted by the solid rectangle. In this example, flash memory has eight erase blocks, which are depicted by the fine dotted rectangles. Out of the eight erase blocks, two blocks are reserved as *update memo* blocks, which are bounded by the solid dotted rectangles; while the remaining six blocks are used as *data blocks*. In each *data block*, there are four data pages. One of the data pages will be reserved as *log page*, which is marked in shaded in gray. Each data page consists of four flash sectors as depicted by the smallest solid rectangles.

The left side of Figure 5.2 gives an overview of the update processing methodology through *update memo* and *log pages*. The update processing has three steps. **Step 1:**

when an update transaction (i.e., `INSERT`, `UPDATE`, or `DELETE`) occurs, the changes made by the transactions are stored as *update logs* in the available sectors of the *update memo* blocks. **Step 2:** when *update memo* is full, the latest *update logs* are flushed to the *log pages* of the corresponding *data blocks*. A *timestamp* counter is used to identify the latest *update logs*. In addition, an index is used to speed up the flushing process. **Step 3:** when the *log pages* of a *data block* are also full, the *update logs* are stored in-place with the old data records. Without *update memo* and *log pages*, for every *in-place* update operation, we would need to erase a flash block. However, with the help of *update memo* and *log pages*, processing of the *in-place* update operations are deferred. The *update memo* acts as a buffer and supplies multiple *update logs* at once to the *log pages*. These logs are stored compactly in the *log pages*. Thus, *update memo* provides the opportunity to compact more *update logs* and to bulk updates once for each block, which internal *log pages* cannot do. Overall, *update memo* and *log pages* helps to reduce total number of block erase operations due to *in-place* updates by amortizing cost of single *data block* erase operation among multiple update operations.

The right side of Figure 5.2 gives an overview of the query processing steps in *deferred update methodology*. Queries are processed in the reverse order of the update processing method. It also consists of three steps. **Step 1:** a raw query result set is generated from the data records stored in the data pages using the traditional query processing techniques. **Step 2:** initial raw query result set is modified through the *update logs* stored in the relevant *log pages*. **Step 3:** the new result set is modified further by processing the latest *update logs* stored in the *update memo*. A flash-friendly hash index is used to expedite this step. The second and third steps ensure that query result is correct.

There is a trade-off between the gain in the update processing performance and query processing overhead. Keeping lot of *update logs* in the *update memo* and *log pages* improves update processing performance. However, this strategy increases the overhead of query processing as we have to scan larger *update memo* and more *log pages* to generate correct results. The *number of blocks* in the *update memo* and the number *log pages* in a *data block*, are the two tuning parameters to control the performance gain of the *deferred update methodology*.

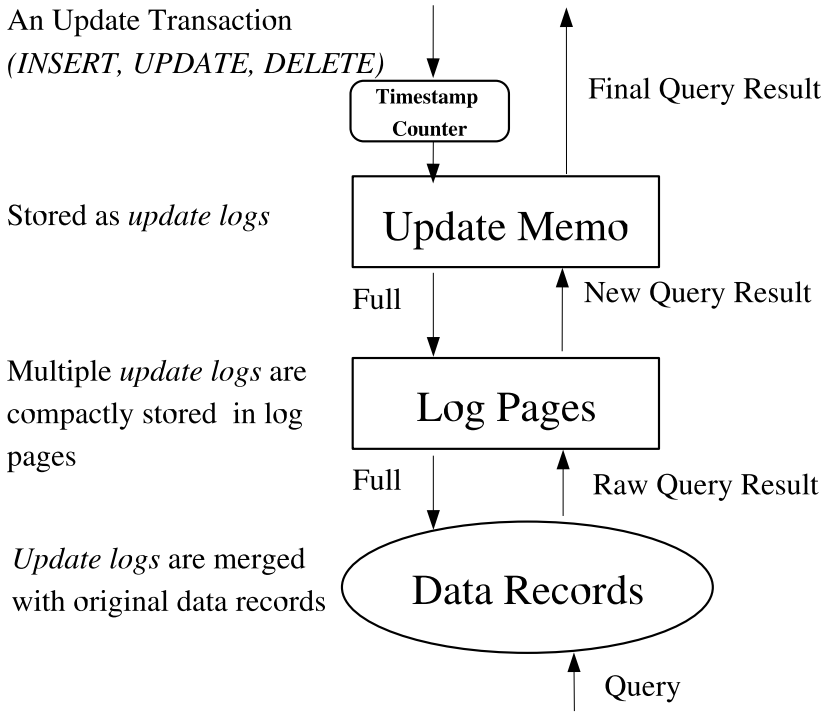


Figure 5.2: Overview of the *deferred update methodology*

5.3.2 Update Log, Timestamps, and Indexing

In this subsection, we describe the structure of the *update log*, the various timestamp values, and the indexing data structure.

Update Log. An *update transaction* represents any `INSERT`, `UPDATE`, or `DELETE` operation. The changes made by the update transaction are stored as *update logs*. Every *update log* has five fields: *rid, opcode, timestamp*, *updateLogContents*, and *previousLogEntryLocation*. The *rid* field indicates the `ID` of original record for which the *update log* is intended for. The *opcode* field indicates the operation code, which denotes `INSERT`, `UPDATE`, and `DELETE` operations, respectively. The *timestamp* field indicates when this *update log* is inserted in the *update memo*. The *updateLogContents* field contains the summary of the changes that are supposed to be applied to the original record. In case of `DELETE` operation, the *updateLogContents* field contains `NULL` value. The *previousLogEntryLocation* field helps to maintain an index over the *update log*. In particular, this field helps to build *MemoIndex*, which will be described later.

Timestamps. There can be multiple *update logs* for the same data record. Therefore, to generate correct result for a query, a mechanism is needed to identify the latest *update log* of a data record. To this end, each *update log* is *timestamped*, when it is stored in the *update memo*. The *timestamp* is assigned by a global counter that is monotonically increasing. Once assigned, the *timestamp* is never changed. The *timestamp* implicitly places a temporal relationship among the *update logs*. In addition to timstamp in the *update log*, one timestamp value, named as *logTimestamp*, is maintained per data block in the master catalog table. The *logTimestamp* keeps track of the global timestamp counter value at the time *update logs* from the *update memo* are flushed to *log pages* of that block. Thus, this timestamp helps to identify the non-flushed *update logs*.

MemoIndex. To quickly find the relevant *update logs* for a specific *data block* from the *update memo*, a hash index is maintained. In the rest of this paper, this hash index is referred as *MemoIndex*. This index speeds up the query processing and flushing of the *update memo* contents. For each hash entry in the *MemoIndex*, the *key* is a *data block* ID and *value* is the location of the very recent *update log* for the corresponding *data block* in the *update memo*. Whenever an *update log* is added in the *update memo*, from the *rid* of the log, the corresponding destination *data block* ID is determined, and *MemoIndex* is checked to find if there is any hash entry for that *data block*. If an entry is found, the current value of that entry is copied in the *previousLogEntryLocation* field of the *update log* and the in-memory hash entry will be updated with the current location of the *update log* in the memo. If no entry is found, that means this log is the only entry for the corresponding *data block*. In this case, `NULL` is stored in the *previousLogEntryLocation* field of the *update log*; and finally, a new hash entry will be added in *MemoIndex* with *data block* ID as key and the current location of *update log* in the memo as value.

The *previousLogEntryLocation* field acts as a backward pointer and helps to avoid updating in the previous *update logs*, which makes *MemoIndex* a flash-friendly data structure. The main idea of *MemoIndex* is easily seen as inspired by the intuition behind the B-File design [106]. The *MemoIndex* requires very small amount of memory. For example, for 100MB data, which requires 400 flash blocks (assuming a 256KB block size), needs 400 hash entries. Assuming, each hash entry taking eight bytes, in total we need only 3200 bytes (3.12 KB) of additional memory. In case of power failure, the

MemoIndex can be rebuilt by scanning *update logs* stored in the *update memo* and *log pages*.

5.3.3 Update Transaction Processing

An *update transaction* including `INSERT`, `UPDATE`, or `DELETE` operation is processed in the same fashion. When update transactions are executed, instead of performing *in-place* updates, the corresponding changes due the update transactions are first stored as *update logs* in the *update memo*. When memo is full, *update logs* are flushed to the corresponding *log pages*. Once *update memo* is full, we need to flush it so that future incoming *update logs* can be temporarily stored there. The simple way of flushing the *update memo* would be to flush the entire memo at once by moving all the *update logs* to the corresponding *log pages*. However, it will be a very time consuming task. To improve performance, instead of the flushing the entire *update memo*, we flush one victim block from the memo. There can be various policies to select a victim block. In this paper, we are using a simple round-robin policy. After selecting a victim block, its contents (i.e., *update logs*) are flushed to the corresponding *log pages*. Finally, the *log pages* are also full, the *update logs* are merged in bulk with the original records in the *data blocks*. During update processing, within a flash block, we always perform write operations sequentially, which complies with the flash memory's physical restriction of writing (i.e., in a block sectors need to be written sequentially [31]).

When flushing the *update logs* from the victim block of the *update memo* to *log pages* of *data block*, the related logs for the same *data block* from the entire memo are flushed simultaneously. The *MemoIndex* described in Section 5.3.2 helps to quickly find the related *update logs*. Flushing all relevant logs simultaneously helps to reduce the number of erase operations, as this strategy provides the opportunity to compactly store more *update logs* in lesser space in the *log pages*. However, this complicates the flushing scheme of a victim block, as there may be some non-flushed *update logs* and some flushed logs that are already stored in the *log pages* or merged with the *data blocks* contents. Clearly, only the non-flushed logs need to be stored in the *log pages*. With the help of *logTimestamp* value, the non-flushed logs can be easily identified. If an *update log*'s timestamp value is less than the corresponding *data block*'s *logTimestamp* value, then *update log* has already been flushed to the *log pages* in an earlier block eviction

phase. Now, we describe the detail of the update processing algorithm.

Algorithm. Algorithm 2 gives the pseudocode to process an update transaction. When an *update log* reaches the *update memo*, a timestamp value is assigned to the *update log* and timestamp counter is incremented (Lines 1-2 in Algorithm 2). We use *MemoIndex* to find the *previousLogEntryLocation* field of the update log (*newUpdateLog*) by using block ID as key (Line 3 in Algorithm 2). The *rid* field of an *update log* provides the block ID information. If an entry is found in *MemoIndex*, the *previousLogEntryLocation* will be value of that entry; otherwise, we set it to `NULL`. The *newUpdateLog* contains the *rid*, *opcode* of operation (i.e., `INSERT`, `UPDATE`, and `DELETE`), and the summary of changes by the update transaction, and *previousLogEntryLocation* (Line 4 in Algorithm 2). Now, we check if there are enough free space left in *update memo* to store the *newUpdateLog*. If this is the case, we store *newUpdateLog* in the next available free space of the *update memo* (Lines 5-6 in Algorithm 2). If there are not enough free space in the *update memo*, which means *update memo* is full. To make free space, we select a victim block in the *round robin* fashion for flushing (Line 9 in Algorithm 2). For each *update log* in the victim block, we determine whether this log has been already flushed or not (Line 12 in Algorithm 2). If an *update log* is not flushed before, with the help of *MemoIndex*, we read all other non-flushed logs from the entire *update memo*, which also correspond to the same *data block* as the current log, and flush them in bulk to the corresponding *log pages* (Lines 14-24 in Algorithm 2). The logTimestamp value of a block helps to identify the non-flushed logs for that block. We again check, if there are enough free space left in the *log pages* of the respective *data block* to fit the flushed logs. If *log pages* have enough free space, we compactly store the flushed logs in the free space of the *log pages* and update the logTimestamp (Lines 25-27 in Algorithm 2). If *log pages* are full, we merge the flushed logs from *update memo* and current logs in the *log pages* with the old records stored in the corresponding *data block* (Line 30 in Algorithm 2). The procedure for combining the old data records and *update logs* is similar to the algorithm described in the IPL approach [90] and it incurs only one block erase operation. After flushing is done, we remove the hash entry corresponding to current processing *update log* from *MemoIndex* to remember that for its destination block all logs have been flushed (Line 32 in Algorithm 2). This helps to speed up look-up of the non-flushed logs in the later eviction phases. When all *update logs* from the victim block and other relevant logs

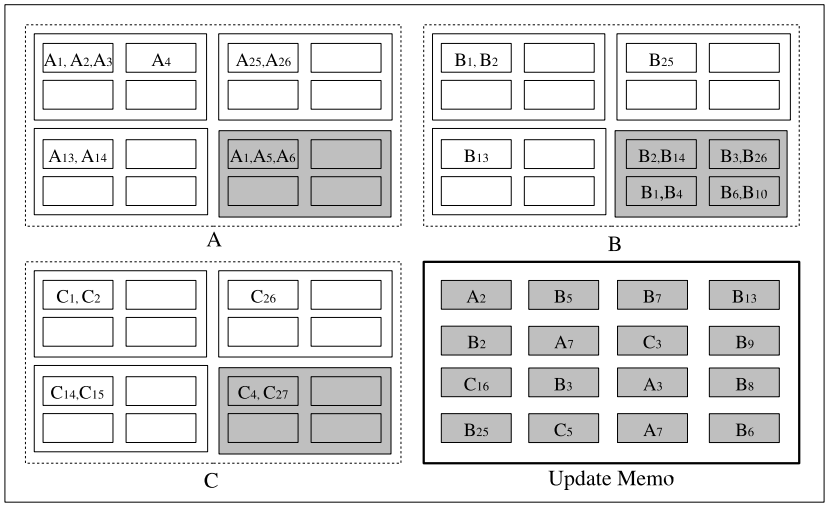


Figure 5.3: Update Processing Example (initial state)

are written to the corresponding *log pages*, we erase the victim *update memo* block to make it ready to store new *update logs* (Line 37 in Algorithm 2). Now, we store the *newUpdateLog* in the victim *update memo* block (Line 38 in Algorithm 2). Finally, we update (or insert a new entry, if there is no previous entry found) hash entry in the *MemoIndex* to remember the location of *newUpdateLog* (Line 40 in Algorithm 2).

Example. Figure 5.3 gives an example of processing update transactions. For simplicity, we ignore indexing and the timestamp values. Furthermore, we use full records instead of *update logs*. In Figure 5.3, there are three data blocks A, B, and C, depicted by fine dotted rectangles. Each data block has one *log page* (shaded in gray) and three data pages. Each page consists of four flash sectors. Sectors are depicted by smallest rectangles. Each sector can contain at most three records. This means that a data page contains up to 12 records, and thus a data block contains up to 36 records. Records 1-12 correspond to first data page, similarly records 13-24 correspond to second data page, and so on. In Figure 5.3, the number x next to a block denotes the x-th record in that block. For example, A_1 means the first record of block A. In block A, there are four records, A_1, A_2, A_3, and A_4 in the first data page, two records A_{13} and A_{14} in the second data page, and two records A_{25} and A_{26} in the third data page; the fourth page is the *log page* which has one used sector and three free log sectors. In block B, there are four records and all four log sectors are used. Similarly, in block C, there are five

records and one log sector is used. In Figure 5.3, *update memo* consists of one block and is depicted by solid lines. It has 16 flash sectors. All the *update memo* sectors are used i.e., *update memo* is full. As flash memory is write once and write operations are performed in sector unit, that is why each memo sector contains only one record. However, in the data pages, there are multiple records in some sectors. This happen as with the help *update memo* and *log pages*, we can compact multiple records in a sector and write them at once. In Figure 5.3, the first sector of the *update memo* contains updated record A_2. Similarly, in the second sector contains B_5, and so on.

Suppose, in Figure 5.3, we want to update one more record, A_{27}. However, as *update memo* is already full, we need to clean it to store new records. We partition the *update memo* records into three groups based on the blocks: $\{\{A_2, A_3, A_7\}$, $\{B_2, B_3, B_5, B_6, B_7, B_8, B_9, B_{13}, B_{25}\}$, $\{C_3, C_5, C_{16}\}\}$. The partitioned records are stored in the corresponding blocks' *log pages*. If we do not have *log pages*, we have to perform three immediate erase operations. However, *log pages* helps to save these erase operations. We store $\{A_2, A_3, A_7\}$ and $\{C_3, C_5, C_{16}\}$ compactly in the *log pages* of data block A and C, respectively. However, as all the log sectors of the block B are full, we combine current *log page* records $\{B_1, B_2, B_3, B_4, B_6, B_{10}, B_{14}, B_{26}\}$ and memo records $\{B_2, B_3, B_5, B_6, B_7, B_8, B_9, B_{13}, B_{25}\}$ and with the old records $\{B_1, B_2, B_{13}, B_{25}\}$, and write them compactly into three data pages. Now, Block B contains $\{\{B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8, B_9, B_{10}\}$, $\{B_{13}, B_{14}\}$, $\{B_{25}, B_{26}\}\}$ in order in three data pages. Figure 5.4 gives the new state of the data blocks and *update memo*. We erase the memo block to make the flash sectors ready for writing. Finally, we store A_{27} in the first sector of *update memo*.

Discussion. At a first glance, it seems that *deferred update methodology* generates skews of the number of erase operations for flash blocks in the different areas in the flash memory, which would cause performance bottlenecks. For example, *update memo* blocks are erased more frequently than the *data* blocks. However, current flash-based storage devices use various *wear leveling* techniques to even out the erase count of all flash block [46, 64]. Most of the existing algorithms use hot and cold block classification [71]. In *deferred update methodology*, *update memo* blocks are relatively hot compared to *data blocks*. Since, existing wear leveling algorithms can efficiently handle hot and cold blocks, the skewness introduces by *deferred update methodology* will not pose any performance

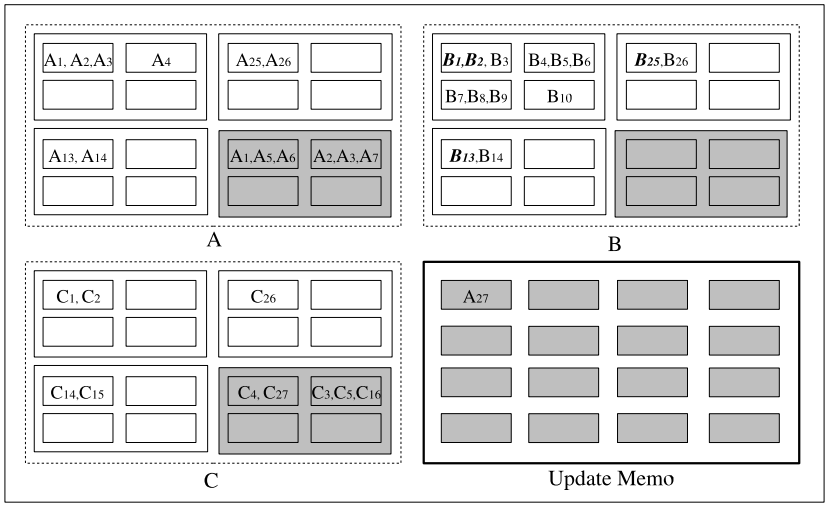


Figure 5.4: Update Processing Example (post processing state)

bottlenecks in the existing flash-based devices. We will address this issue in the future.

5.3.4 Query Processing

Since the update transactions are not immediately reflected in data records, the query processing needs special care for the updated records whose most recent values still reside on either the *update memo* or *log pages*, i.e., not updated yet to the actual data pages. A query is processed in the reverse order to the update processing order. At first, we generate raw query result from the the data pages using the traditional query processing techniques. Next, we modified this raw result by *update logs* stored in the relevant *log pages* and *update memo*, in order. These additional modifications ensure that the generated query results are correct, i.e., include the most recent record values.

Algorithm. Algorithm 3 gives the pseudocode to process a query, q. We generate a raw result set R_{data} from the records stored in the data pages using the traditional query processing techniques (Line 1 in Algorithm 3). Now, we modify the R_{data} using *update logs* in the *log pages*. We need to scan the *update logs* stored in *log pages* of the *data blocks* which contain any log related to query q (Lines 4-10 in Algorithm 3). The *data blocks* that we need to scan depends on the query selectivity . For every *update log* (u), stored in the *log pages* of related *data blocks*, we check whether u satisfies query

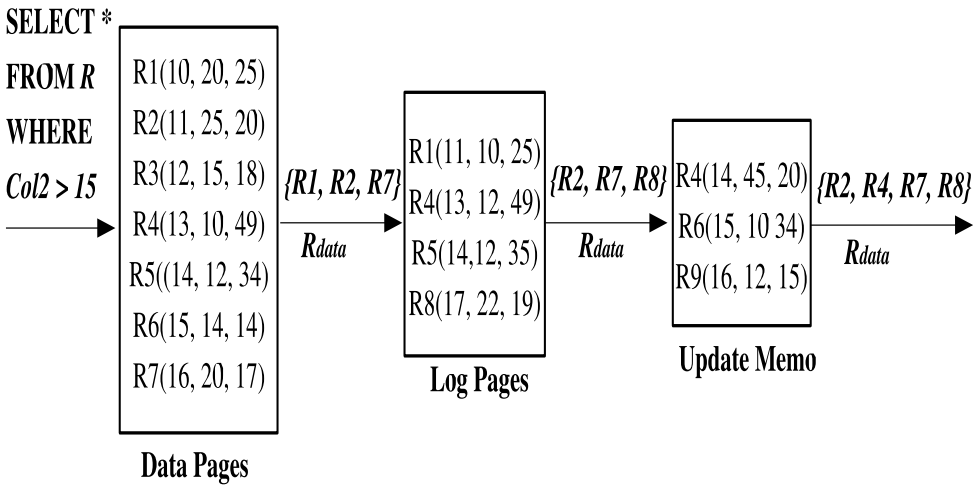


Figure 5.5: Query Processing Example

q. If u qualifies the query criteria, we add the record related to u with updated value in the result set R_{data}. If u does not satisfy query criteria, we further check whether the record related to u is included in the raw result set R_{data}. If the related record is in R_{data}, we remove that record from R_{data} as updated value of that stored in the *log pages* does not satisfy q. After processing the *update logs* in the *log pages*, we again modify the result set R_{data} by the *update logs* stored in the *update memo* (Lines 12-28 in Algorithm 3). The *MemoIndex* and logTimestamp value help to speed up finding of the latest log entries. For every *update log*, (u'), related a block stored in the *update memo*, we check whether u' satisfies query q. If u' satisfies the query criteria, we add the record related to u' with the updated value in the result set R_{data}. If u' does not satisfy the query criteria, we further check whether the record to related to u' is included in the result set R_{data}. If that record is in R_{new}, we remove it from R_{data}, as current value of that record does not satisfy q. Finally, after processing all relevant logs both in the *log pages* and *update memo*, we return R_{data} as query result.

Example. Figure 5.5 gives an example of processing query, *SELECT * FROM R WHERE Col2 > 15*. Similar to previous example, for simplicity, we ignore the timestamp values and use full records instead of *update logs*. Table R has three fields: *Col1*, *Col2*, and *Col3* and contains nine records $R1$ to $R9$. Some of the updated record values are stored in the *log pages* and memo blocks. For example, in Figure 5.5, log pages

contain the updated value of records $R1, R4, R5$ and $R8$. While *update memo* contains updated values of records $R4, R6$, and $R9$. The values of a record stored in the *update memo* is more recent than the values stored in the *log pages*, while the record values stored in the *log pages* are most recent than the values stored in the data pages. When we execute the query over the records stored in the data pages, we get the raw query result set R_{data}, which contains $\{R1, R2, R7\}$. To generate correct output, we modified R_{data} by updated records stored in the *log pages*. From the updated records stored in *log pages*, $R8$ satisfies the query criteria *Col2 > 15*. However, the updated value of record $R1$ does not satisfy this criteria. Therefore, the new intermediate result set R_{data} contains $\{R2, R7, R8\}$. To generate accurate output, we further modify R_{data} by the updated records stored in the *update memo*. From the *update memo*, $R4$ satisfies the query criteria. Therefore, the final query result set R_{data}, contains $\{R2, R4, R7, R8\}$.

5.4 Analysis of Log Pages and Memo

In this section, we analyze *deferred update methodology* that uses both *update memo* and *log pages* compared to the other three alternative approaches including *no log and no memo*, *log only*, and *update memo only* to process N_u update transactions. Our measure of analysis are (a) space overhead ($s_{overhead}$), (b) query processing overhead ($q_{overhead}$), and (c) total number of erase operations (N_E). During this analysis, without loss of generality, we assume that the update transactions are uniformly distributed to all database pages and average *update log* size (u_s) is less than a flash sector size (S). To simplify the analysis, we further assume that there is no index over the *update logs* stored in the *update memo* and each overwrite operation in the flash memory incurs an erase operation. Table 5.1 introduces various symbols used in this analysis. The next four subsections discuss the four alternative approaches using *log pages* and *update memo*. Finally, in the following subsection, we compare all four approaches.

5.4.1 No Log and No Memo Approach

This is existing design with no change for flash memory i.e., this model has no *log pages* and no *update memo*. **Space Overhead.** There is no space overhead in this scheme as there are no *log pages* and no *update memo* blocks to hold the *update logs*. The number

Symbol	Description
DB_{size}	Database size in bytes
B	Flash block size in bytes
P	Database page size in bytes
S	Flash sector size in bytes
n_l	Number of *log pages* per data block
u_s	Average *update log* size in bytes
n_{ul}	Average number of *update logs* each data block receiving during memo cleaning
N_u	Total number of update transactions
N_D	Total number of data blocks
N_q	Total number of data blocks containing data records satisfying query q
N_M	Total number of *update memo* blocks
N_E	Total number of block erase operations to process N_u update transactions
$q_{overhead}$	Query processing overhead
$S_{overhead}$	Space overhead

Table 5.1: Symbols description

of *data blocks* (N_D) to fit all DB_{size} data is $\frac{DB_{size}}{B}$ blocks. As this is the least possible number of blocks to hold the data, we consider this scheme as *no space overhead model* and use it as a baseline for comparison with other alternative designs.

Query Processing Overhead. As there is no space overhead, there is no query processing overhead. This is the best possible possible scheme in terms of query processing overhead. So, we consider this scheme as *no query processing overhead model* and use it as a baseline for comparison with other alternatives.

Erase Operations. In this model, to process every update transaction, in the worst case, we would need to perform one erase operation of the *data block* containing the corresponding data record per update transaction. Therefore, to process N_u update transactions, the total number of erase operations (N_E) would be N_u.

5.4.2 Log Page Only Approach

In this model, each *data block* contains n_l number of *log pages*. However, no *update memo* is maintained. This is similar to the model adopted by the IPL technique [90],

when setting $n_l = 1$.

Space Overhead. In this scheme, out of $\frac{B}{P}$ pages of a *data block*, n_l number of *log pages* are used to temporarily hold the *update logs*. Therefore, we need additional *data blocks* to fit all data. The space for holding data in every block is $B - n_l * P$ bytes. Therefore, to fit DB_{size} data, the required number of *data blocks* (N_D) would be $\frac{DB_{size}}{B-n_l*P}$ blocks. Now, the space overhead ($s_{overhead}$) compared to the baseline *no space overhead model* would be $(\frac{DB_{size}}{B-n_l*P} - \frac{DB_{size}}{B}) = \frac{DB_{size}*n_l*P}{B*(B-n_l*P)}$ blocks.

Query Processing Overhead. To process a query, in addition to the traditional query processing techniques, we have to scan the *update logs* stored in the *log pages* of the *data blocks* to generate correct query output. This extra scanning contributes to the query processing overhead ($q_{overhead}$). We assume that, out of all data blocks, the N_q number of *data blocks* contain the data records that satisfy the query criteria. In these *data blocks*, *log pages* can be full or partially full, or empty. Thus, we can assume that on average the *log pages* are half full. The N_q *data blocks* have in total $N_q * n_l$ *log pages*. On average, we have to scan half of these pages during the query processing. Therefore, $q_{overhead}$ compared to the base line *no query processing overhead model*, would be the time to scan $N_q * \frac{n_l}{2}$ pages of data.

Erase Operations. In this model, each *data block* contains n_l number of *log pages* and each page has $\frac{P}{S}$ sectors. Therefore, in total a *data block* has $n_l * \frac{P}{S}$ log sectors. Every *update log* consumes one sector of the log pages of *data blocks*. Therefore a *data block* hold $n_l * \frac{P}{S}$ *update logs* before it is full. Once full, we need to erase *data blocks* to make free space in the *log pages* to hold future *update logs*. Now, to process N_u update transactions, the total number of erase operations (N_E) would be $\frac{N_u}{n_l * \frac{P}{S}} = \frac{N_u * S}{n_l * P}$.

5.4.3 Update Memo Only Approach

This model has only *update memo*. However, no *log pages* are maintained. There are N_M blocks in the *update memo*. At first, *update logs* are stored in the memo. When memo is full, *update logs* are merged with the old data records. The functionality of this approach is quite similar to space efficient FTL design [85].

Space Overhead. In this model, to fit the DB_{size} data, the required number of *data blocks* (N_D) would be $\frac{DB_{size}}{B}$. In addition, we need space to fit N_M number of *update memo* blocks. Therefore, the space overhead ($s_{overhead}$) compared to the baseline *no*

space overhead model is N_M blocks.

Query Processing Overhead. To generate correct query result, we have to scan the logs stored in the *update memo* in addition to the traditional query processing techniques. This additional scanning of the *update memo* contributes to the query processing overhead ($q_{overhead}$). The *update memo* can be full or partially full or empty. We assume that on average it is half full. Therefore, on average we have to scan half of the *update memo* during the query processing. Now, the $q_{overhead}$ would be the time to scan $\frac{1}{2} * N_M$ *data blocks*, which is equivalent to $\frac{1}{2} * N_M * \frac{B}{P}$ pages of data.

Erase Operations. In this model, erase operations occur due to cleaning of the *update memo* blocks and erasure of the *data blocks*.

Update Memo Block Erases. As we assume that *update log* (u_s) size is less than the flash sector size (S), each *update log* consumes one sector from the *update memo*. Each memo block contains $\frac{B}{S}$ sectors. The N_M memo blocks has in total $N_M * \frac{B}{S}$ sectors. Therefore, before full, *update memo* can hold $N_M * \frac{B}{S}$ *update logs*. Once *update memo* is full, we need to clean it to store future incoming logs. To process N_u update transactions, in total, we need to clean the *update memo* blocks $\frac{N_u}{N_M * \frac{B}{S}}$ number of times. During cleaning, we have to erase N_M memo blocks. Therefore, the total number of erase operations in the *update memo* would be $\frac{N_u}{N_M * \frac{B}{S}} * N_M = \frac{N_u * S}{B}$.

Data Block Erases. Each time the *update memo* blocks need to be cleaned, we have to merge the *update logs* with the old *data blocks* records. During, cleaning each *data block* receives n_{ul} number of *update logs*. To merge these logs, we need to erase the *data blocks*. To process, N_u update transactions, the total number of *data block* erase operations would be, $\frac{N_u}{n_{ul}}$

By summing the *update memo* block erase operations and *data block* erase operations, the total number of erase operations (N_E) would be, $\frac{N_u * S}{B} + \frac{N_u}{n_{ul}} = N_u(\frac{S}{B} + \frac{1}{n_{ul}})$.

5.4.4 Log page and Update Memo Approach

This is the model adopted by the *deferred update methodology*. In this model, we have N_M number of *update memo* blocks and each *data block* has n_l number of *log pages*. In fact, this model is a combination of *Log Page Only* (as described in Section 5.4.2) and *Update Memo Only* (as described in Section 5.4.3) approaches.

Space Overhead. In this model, we need additional space to accommodate *log pages*

	Total Erase Operations (N_E)	**Space Requirements *in blocks***	**Query Processing Overhead ($q_{overhead}$) *in time to scan pages***
No Log and No Memo	N_u	$\frac{DB_{size}}{B}$	0
Log Page Only	$N_u * \frac{S}{n_l * P}$	$\frac{DB_{size}}{B - n_l * P}$	$\frac{1}{2} * N_q * n_l$
Update Memo Only	$N_u * (\frac{S}{B} + \frac{1}{n_{ul}})$	$\frac{DB_{size}}{B} + N_M$	$\frac{1}{2} * N_M * \frac{B}{P}$
Log Page and Update Memo (Deferred Update)	$N_u * (\frac{S}{B} + \frac{1}{\left\lfloor \frac{n_l * \frac{P}{S}}{\lceil \frac{n_{ul} * u_s}{S} \rceil} \right\rfloor * n_{ul}})$	$\frac{DB_{size}}{B - n_l * P} + N_M$	$\frac{1}{2} * (N_q * n_l + N_M * \frac{B}{P})$

Table 5.2: Analytical comparison of different alternative designs using *log pages* and *update memo*

and *update memo*. In every *data block*, n_l pages are used to temporarily store *update logs*. Therefore, we need to increase the number of *data blocks* to fit all DB_{size} of data. The required number of *data blocks* (N_D) would be $\frac{DB_{size}}{B - n_l * P}$. In addition, for *update memo*, we need N_M blocks. The total number of blocks required is $N_D + N_M$. Therefore, the space overhead ($s_{overhead}$) of this model compared to the baseline *no space overhead model* would be $N_D + N_M - \frac{DB_{size}}{B} = \frac{DB_{size}}{B - n_l * P} + N_M - \frac{DB_{size}}{B} = \frac{DB_{size} * n_l * P}{B * (B - n_l * P)} + N_M$ blocks.

Query Processing Overhead. To process a query, in addition to the normal query processing techniques, we have to scan the logs stored in the both *update memo* and *log pages* to generate correct query result. These additional scannings contribute to the query processing overhead ($q_{overhead}$). Similar to the *Update Memo Only* approach (as described in Section 5.4.3), in this scheme, on average the *update memo* is also half full. Thus during the query processing we have to scan $\frac{1}{2} * N_M$ blocks, which is equivalent to $\frac{1}{2} * N_M * \frac{B}{P}$ pages of update data. In addition, similar to the *Log Page Only* approach (as described in Section 5.4.2), we need to scan $N_q * \frac{n_l}{2}$ pages of data. Therefore, the query processing overhead $q_{overhead}$ is the time to scan $\frac{1}{2} * (N_M * \frac{B}{P} + N_q * n_l)$.

Erase Operations. In this model, erase operations occur due to *update memo* cleaning and *data block* erases during the *log pages* cleaning.

Update Memo Block Erases. Similar to the *Update Memo Only* approach (as described in Section 5.4.3), the total block erase operations in the *update memo* in this scheme would be, $\frac{N_u * N_M}{N_M * \frac{B}{S}} = \frac{N_u * S}{B}$.

Data Block Erases. Each time when the *update memo* blocks are cleaned, n_{ul}

number of *update logs* are stored in the the *log pages* of each *data block*. The total size these *update logs* is $n_{ul} * u_s$. To fit these *update logs*, we need $\lceil \frac{n_{ul} * u_s}{S} \rceil$ sectors of a *log page*. We need to take the ceiling, as once some data is written in a sector, without erasing we cannot rewrite any data in that sector. A *data block* contains n_l *log pages*, which has in total $n_l * \frac{P}{S}$ sectors. Therefore, before full, the *log pages* of a *data block* can contain $\left\lfloor \frac{n_l * \frac{P}{S}}{\lceil \frac{n_{ul} * u_s}{S} \rceil} \right\rfloor * n_{ul}$ *update logs*. Once *log pages* are full, the *data block* needs to be erased to accommodate new logs. Now, to process N_u update transactions, the total number of *data block* erase operations performed would be $\frac{N_u}{\left\lfloor \frac{n_l * \frac{P}{S}}{\lceil \frac{n_{ul} * u_s}{S} \rceil} \right\rfloor * n_{ul}}$.

Summing up the *update memo* block erases and *data block* erases, the total number of block erase operations (N_E) would be $\frac{N_u * S}{B} + \frac{N_u}{\left\lfloor \frac{n_l * \frac{P}{S}}{\lceil \frac{n_{ul} * u_s}{S} \rceil} \right\rfloor * n_{ul}} = N_u(\frac{S}{B} + \frac{1}{\left\lfloor \frac{n_l * \frac{P}{S}}{\lceil \frac{n_{ul} * u_s}{S} \rceil} \right\rfloor * n_{ul}})$.

5.4.5 Comparison

Table 5.2 summarizes the total number of erase operations performed, space requirements, and query processing overhead for different alternative designs using *log pages* and *update memo*.

In terms of space overhead and query processing overhead, *No Log and No Memo* approach is the best. For the same number of *log pages* (n_l), *Log Page and Update Memo* approach i.e., *deferred update methodology* has more space overhead and query processing overhead compared to the *Log Page Only* approach because of the additional *update memo*. In this analysis, we have not considered indexing of the *update logs* stored in the *update memo*. This index improves the query processing overhead of the *deferred update methodology*. Compared to the *Update Memo Only* approach, for the same number of memo blocks (N_M), *deferred update methodology* has more space overhead and query processing overhead due to the additional *log pages*. Our experimental results show that for same space overhead, *Update Memo Only* approach and *deferred update methodology* incur comparable query processing overhead, while the *Log Page Only* approach incurs less query processing overhead compared to both of these approaches.

In terms of erase operations, *No Log and No Memo* approach incurs the largest number of erase operations. As erase is the most expensive operation and it incurs lot of flash sector read and write operations, therefore update operations will be slow

compared to other three approaches. In addition, due to excessive erase operations, flash memory will be worn-out in a relatively faster rate. For the other three approaches, the number of erase operations depend on the values of the different parameters. To get a general idea, we put the following parameter values to the total number of erase operations (N_E) calculation formula: B = 256 KB, P= 8 KB, S = 4 KB, n_{ul} = 2, and u_s = 64 bytes. The parameters value of the flash memory are taken from the SSD design project [31]. Since, *deferred update methodology* uses both *log pages* and *update memo*, while *Log Page Only* approach uses only *log pages*, we use $n_l = 1$ in case of *deferred update methodology*, while we use $n_l = 2$ in case of *Log Page Only* approach. As *deferred update methodology* uses half number of *log pages* with respect to the *Log Page Only* approach, the extra space taken by the *update memo* will be offset by the smaller value of n_l. Now, by substituting the parameter values, we get $N_E = N_u * \frac{1}{2}$ for the *Log Page Only* approach, $N_E = N_u * \frac{33}{64}$ for the *Update Memo Only* approach, and $N_E = N_u * \frac{5}{64}$ for the *deferred update methodology*. In this hypothetical example, the number of erase operations performed by the *Log Page Only* and *Update Memo Only* approaches are almost same, while *deferred updates* performs 6.2 times fewer number of erase operations compared to the *Log Page only* approach. Our experimental results shows that, *deferred update methodology* significantly reduces the total number of erase operations. As erase is most expensive operation and a block can only be erased for limited number of times, therefore reduction of the block erase operations helps to improve update processing performance as well as prevents early worn-out of the flash memory.

5.5 Experimental Results

We compare *deferred update methodology* with in-page logging (IPL) technique [90] and *Update Memo Only* (described in Section 5.4.3) approach to handle update transactions. IPL is the state-of-the-art technique for handling update transactions for the flash based storage. It is a special case of *Log Page Only* approach (described in Section 5.4.2) with one *log page* per erase unit. On the other hand, the working principle of *Update Memo Only* approach is very similar to space-efficient log-based FTL design work [85]. We do not consider *No Memo and No Log* approach (described in Section 5.4.1) in the

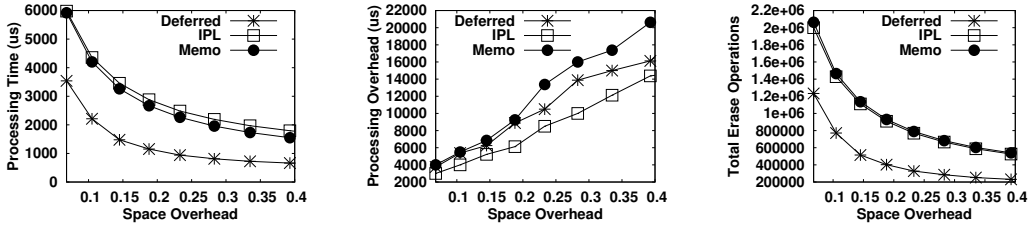
Parameter	Value
Block size	256 KB
Sector size	4 KB
Data Register Size	4 KB
4KB-Sector Read to Register Time	25 μs
4KB-Sector Write Time from Register	200 μs
Serial Access time to Register (Data bus)	100 μs
Block Erase Time	1500 μs

Table 5.3: Parameters values for NAND flash memory

comparison, as it is not suitable for processing update transactions [90]. In the rest of section, at first, we describe the simulator, traces, and performance calculation formulas. Next, we estimate the internal parameters values: *number of log pages per block* (n_l) and *number of update memo blocks* (N_M). Finally, we demonstrate the scalability of the *deferred update methodology* compared to other two approaches by varying the number of update transactions and database size.

Simulator and Traces. To evaluate *deferred update methodology*, similar to IPL technique [90], we have implemented a standalone event-driven simulator in the C language on the Linux platform. This simulator mimics the behavior of both *update memo* and *log pages* as described in Sections 5.3 and 5.4. We use synthetic traces to evaluate *deferred update methodology*. Synthetic traces are generated by a standalone trace generation program written in C language. This program takes database size, page size, and number of update transactions as input and generates a trace file as output. The update transactions in the output trace file are uniformly distributed over all database pages and there is almost no temproal locality (less than 1%). This trace emulates one of worst writes access patterns for the flash memory. The online transaction processing (OLTP) type applications exhibit quite close behavior to this trace. The size of an *update log* in each transaction lies in between 20-100 bytes. We vary the number of update transactions from one million to 100 millions, while we vary database size from 100 MB to 10000 MB. We assume that each database page size is 8 KB.

Performance Calculation Formulas. To calculate the update processing time, we use the following formula: *total number of erase operations * erase time + total sector read operations * (sector read time + page register access time) + total sector*



(a) Average update processing time

(b) Processing overhead for a query having 3% selectivity

(c) Erase count

Figure 5.6: Performance trends with varying space overhead in a 100 MB database. Here, 'Memo' stands for *Update Memo Only* approach.

Space Overhead	**IPL**	**Memo Only**	**Deferred Update**
6.8%	$n_l = 2$	$N_M = 27$	$n_l = 1, N_M = 14$
10.5%	$n_l = 3$	$N_M = 42$	$n_l = 1, N_M = 29$
14.5%	$n_l = 4$	$N_M = 58$	$n_l = 2, N_M = 31$
18.8%	$n_l = 5$	$N_M = 75$	$n_l = 2, N_M = 48$
23.3%	$n_l = 6$	$N_M = 93$	$n_l = 3, N_M = 51$
28.3%	$n_l = 7$	$N_M = 113$	$n_l = 3, N_M = 71$
33.5%	$n_l = 8$	$N_M = 134$	$n_l = 4, N_M = 76$
39.3%	$n_l = 9$	$N_M = 157$	$n_l = 4, N_M = 99$

Table 5.4: Parameters values for Log Pages and Memo Blocks for a 100 MB database processing one million update transactions

*write operations * (sector write time + page register access time).* During the update processing, the simulator takes a trace file as input and as output it provides the total number of erase operations performed, total number of sector read operations, and total number of sector write operations. To calculate the query processing overhead, we use the following formula: *the numbers of extra flash sectors read due to query processing* (sector read time + page register access time).* During query processing, the simulator takes query selectivity and returns *the numbers of extra flash sectors* to process that query. Table 5.3 gives the various flash parameters values used to in the experiments. These values are taken from the SSD design project [31].

5.5.1 Parameter Value Selection

In-page logging (IPL) technique [90] uses *number of log pages per block* (n_l) and *Memo Only* approach uses *number of update memo blocks* (N_M), while *deferred update methodology* uses both n_l and N_M. The space overhead and query processing overhead depend on the value of n_l and N_M. We vary the n_l from 1 to 31, and N_M from 1 to 400, for a database of size 100 MB processing one million update transactions. Since, a block size is 256 KB and a page size is 8 KB, therefore a block can contain 32 pages. Now, maximum number of log pages in a block (n_l) is 31 as we need at least one data page per block. On the other hand, to fit 100 MB database, we need 400 blocks. We restrict maximum number of *update memo blocks* (N_M) to the number of blocks need to fit the original data, that is why the maximum value of N_M is 400. Since, three different approaches uses different number of parameters, to make a fair evaluation, we estimate the average update processing time and query processing time for the same space overhead. Table 5.4 gives different parameters values in the different approaches, for which average update processing time is minimum among all configurations in that approach for the same space overhead. Since, IPL technique uses only *log pages* and *deferred update methodology* uses both *log pages* and *update memo*, for IPL, we set the minimum value of n_l to 2.

Figure 5.6(a) gives the average update processing time for the same space overhead. As expected, with the increase of space overhead, the average update processing time decreases in all three approaches. Compared to IPL, *deferred update methodology* improves average update processing time by 50%-63%, while *Update Memo Only* approach reduces update procesing time by 1%-17%. The average update processing time decreases with the increase of the space, as we can buffer more *update logs* and apply them in bulk. This helps to decrease the total number of erase operations. This trend is shown by Figure 5.6(c). Since erase operations are performed in the block-level and before erasing a block, we need to move data and write them back. Thus, erase operations incurs huge overhead in terms of latency. With the decrease in the number of erase operations, this latency overhead decreases, which helps to improve the update processing performance. In IPl, incerasing On the other hand, Figure 5.6(b) shows that with the increase of space overhead, the query processing overhead also increases. This happens as with increase of space, more *update logs* are buffered and we need to

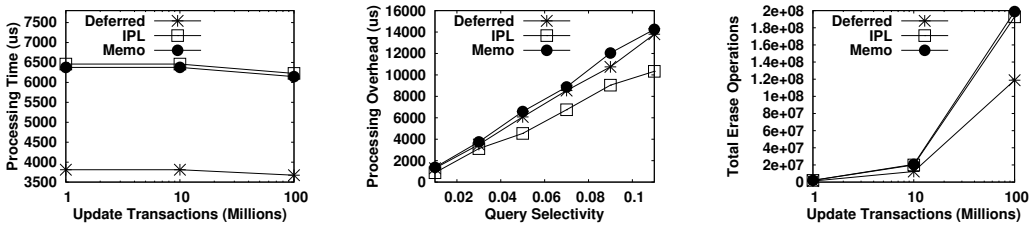
process more logs to generate correct query result. Compared to IPL, *deferred update methodology* increases query processing overhead up to 44%, while *Update Memo Only* approach increases query processing by 1%-17%.

Consistency Check of the Analytical Model. The performance trend in Figure 5.6 is consistent with the analytical model developed in Section 5.4. Table 5.2 shows that with increase of number of log pages (n_l) in IPL (which is a *Log Page only* approach), space overhead increases, erase counts decreases (which helps to improve update processing performance), and query processing overhead increases due to additional processing of the larger number of *update logs* stored in the log pages. Similarly, Table 5.2 shows that with the increase in number of *update memo* block (N_M), for the *Update Memo Only* approach, space overhead increases. However, this additional space helps to hold more *update logs* (n_{ul}), which reduces total number of erase operations and consequently improves the update processing performance. In contrast, increasing N_M introduces overhead in query processing due to additional processing of the larger number of *update logs* stored in the larger *update memo*. According to Table 5.2, in *deferred update methodology*, increasing both (n_l) and (N_M) contribute to the increased space overhead. However, larger n_l and N_M help to hold larger number of *update logs* (n_{ul}) and process them in bulk, which helps in reducing the total number of erase operations. Thus, helps to improve update processing time. On the other hand, query processing overhead increases with the increased space overhead due to processing of large number of *update logs*.

For the rest of this chapter, we keep the space overhead same for all three approaches. Since with the increase of space overhead, query processing overhead also increases, we keep the space overhead as minimum as possible. With this goal, we set the value of n_l to two in IPL and calculate the space overhead. In this case, space overhead is 6.8%. Now, we select a value of N_M in the *Update Memo Only* approach such that space overhead becomes 6.8%. Similarly, for the *deferred update methodology*, we set the value of n_l to one and select a value for N_M such that overall space overhead becomes 6.8%. The parameter values for different database size are listed in Table 5.5. These values are used in the scalabilty demonstration experiments.

DB Size	IPL	Memo only	Deferred Update
100 MB	$n_l = 2$	$N_M = 27$	$n_l = 1, N_M = 14$
1000 MB	$n_l = 2$	$N_M = 267$	$n_l = 1, N_M = 137$
10000 MB	$n_l = 2$	$N_M = 2667$	$n_l = 1, N_M = 1376$

Table 5.5: Parameters values for Log Pages and Memo Blocks for the scalability experiments



(a) Average update processing time

(b) Query processing overhead

(c) Erase count

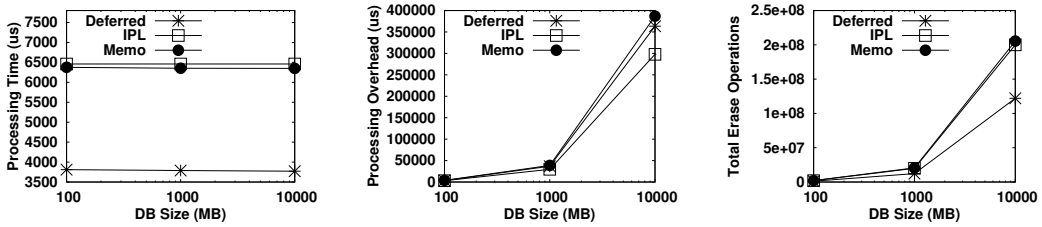
Figure 5.7: Scalability with varying the number of update transactions in a 100 MB database. Here, 'Memo' stands for *Update Memo Only* approach.

5.5.2 Scalability

Now, we demonstrate the scalability of the *deferred update methodology* with the increase in the number of update transactions and size of a database.

Varying Update Transactions

To evaluate the scalability, we vary the number of update transactions from one million to 100 millions in a 100 MB database. Figure 5.7 gives the average update processing time, query processing overhead, and total number of erases operations. From the figure, it is clear that for the same space overhead, *deferred update methodology* scales very well with the increased number of update transactions. Compared to IPL, *deferred update methodology* improves average processing time by 41%, while *Update Memo Only* approach performs almost same. To understand the query processing overhead, we vary the query selectivity form 1%-11%. With the increase of query selectvity, the query overhead increases in all three approaches. This is expected as with the increase of query



(a) Average update processing time

(b) Processing overhead for a query having 3% selectivity

(c) Erase count

Figure 5.8: Scalability with varying database size. Here, 'Memo' stands for *Update Memo Only* approach.

selectivity, we need to process more *update logs* to generate query results. Compared to IPL, *deferred update methodology* incurs up to 30% more query processing overhead, while *Update Memo Only* approach incurs up to 41% more overhead. Usually, update-intensive workloads, for example, OLTP type applications, exhibit significant number of (i.e., 20%-40%) write operations [62], therefore query processing overhead will not be a bottleneck for the update-intensive worklaods. In terms of erase operations, to process the same number of update transactions *deferred update methodology* always outperforms IPL and *Update Memo Only* approach. Compared to IPL, *deferred update methodology* incurs 39% fewer erase operations, while *Update Memo Only* approach incurs almost same number of erase opertions. Since, with the decrease of erase operations, the lifetime of the flash memory also increases, therefore compared to other two approaches, *deferred update methodlogy* will significantly help to improve lifetime of the flash based storage.

Varying Database Size

We vary database size and in proportion we vary the number of update transactions. We process one million update transactions in 100 MB database, ten millions update transactions in 1000 MB database, and 100 millions update transactions in 10000 MB database. Figure 5.8 gives the average update processing time, query processing overhead, and total number of erase operations. It shows that the *deferred update methodology* scales very well with the increase in database size. Compared to IPL, *deferred update*

methodology improves average update processing time by 42%, while *Update Memo Only* approach performs almost same. Figure 5.8(b) shows that the query processing overhead increases with the increase in database size in *deferred update methodology* and *Update Memo Only* approach. This reason behind this trend is that with the increase in database size, the number of buffered *update logs* also increases, and we need to process increasing number of logs to generate query result. Compared to IPL, *deferred update methodology* incurs up to 24% query procesisng overhead, while *Update Memo Only* approach incurs up to 30% overhead. In terms of total erase operations, compared to IPL, *deferred update methodology* performs 41% fewer erase operations, while *Update Memo Only* approach performs 3% fewer erase operations.

5.6 Conclusion

In this thesis, we have proposed a flash-friendly hierarchical update processing technique, named as *deferred update methodology*, for the NAND flash-based storages used in the database servers. Our main goal is to improve update processing overhead and increase lifetime of flash memory by reducing the total number of erase operations preformed in order process update transactions. Experimental results shows that *deferred update methodology* improves average update processing time by approximately 40% compared to the state-of-the-art in-page logging (IPL) technique [90]. In addition, compared to IPL, *deferred update methodology* incurs approximately 40% fewer number of erase operations. It scales very well with the increase of data size and total number of update transactions.

Algorithm 2 Algorithm for Processing Update Transactions

Procedure UpdateProcessor(*rid, opcode, updateLogContents*)

```
1: timeStamp = timeStampCounter value
2: Increment timeStampCounter value
3: Find previousLogEntryLocation from the MemoIndex
4: newUpdateLog = (rid, opcode, timeStamp, updateLogContents, previousLogEntryLocation)
5: if (Free space in the update memo) then
6:    Insert newUpdateLog in the update memo
7: else
8:    // update memo is full
9:    Select a victim update memo block using round robin policy
10:   for (Each update log (u) in the victim block) do
11:      blockId = ID of update log's destination block (B_dest)
12:      if ( timestamp of u > logTimestamp of B_dest) then
13:         // This update log needs to be flushed
14:         if (MemoIndex entry found for the key blockId) then
15:            // There can be more non-flushed entries for the same block
16:            location = MemoIndex entry's value
17:            while (location != NULL) do
18:               Read update log (u') from location
19:               if (timestamp of u' > logTimestamp of B_dest) then
20:                  Add u' to the flushList
21:               end if
22:               location = previousLogEntryLocation of u'
23:            end while
24:            Flush flushList to the log pages in B_dest
25:            if (Enough free space in the log pages of B_dest ) then
26:               Store the flushList compactly in the log pages
27:               Set the logTimestamp of B_dest with the timeStampCounter value
28:            else
29:               // log pages are full
30:               Apply all the latest update logs in log pages and flushList in bulk to the data
                  pages in B_dest
31:            end if
32:            Remove the hash entry with the key blockId from the MemoIndex
33:            Increment timeStampCounter value
34:         end if
35:      end if
36:   end for
37:   Erase victim memo block
38:   Insert newUpdateLog in the free space in update memo
39: end if
40: Update MemoIndex with the location of the newUpdateLog
```

Algorithm 3 Algorithm for Query Processing

1: Generate result set R_{data} for query q from data records stored in the data pages using the traditional query processing techniques
2: **for** (Each block which contains *update logs* related to query q) **do**
3: // **Process the *update logs* in the log pages**
4: **for** (Each latest *update log* (u) stored in the *log pages*) **do**
5: **switch (**case**)**
6: **case** (u satisfies q) & ($u \notin R_{data}$) : add u to R_{data}
7: **case** (u satisfies q) & ($u \in R_{data}$) : add u to R_{data}
8: **case** (u does not satisfy q) & ($u \in R_{data}$) : remove u from R_{data}
9: **end switch**
10: **end for**
11: // **Process the *update logs* in the Update Memo**
12: *blockId* = ID of the current processing block (B)
13: **if** (*MemoIndex* entry found for the key *blockId*) **then**
14: // **There can be more non-flushed entries in memo related** B
15: *location* = *MemoIndex* entry's value
16: **while** (*location* != NULL) **do**
17: Read *update log* ($u\prime$) from *location*
18: **if** (timestamp of $u\prime$ > logTimestamp of B) **then**
19: // **This is a new log**
20: **switch (**case**)**
21: **case** (u' satisfies q) & ($u' \notin R_{data}$) : add $u\prime$ to R_{data}
22: **case** (u' satisfies q) & ($u' \in R_{data}$) : add $u\prime$ to R_{data}
23: **case** (u' does not satisfy q) & ($u' \in R_{data}$) : remove u' from R_{data}
24: **end switch**
25: **end if**
26: *location* = *previousLogEntryLocation* of u'
27: **end while**
28: **end if**
29: **end for**
30: return R_{data}

Chapter 6

BloomFlash: A Flash Memory Based Bloom Filter

6.1 Introduction

The bloom filter is a bit-vector data structure that provides a compact representation of a set of elements (keys). It supports insertion of elements and membership queries. A membership answer is probabilistically correct in the sense that it allows a small probability of a false positive (i.e., an incorrect answer for a non-member element). The bloom filter allows tradeoffs between small size (compactness) and low false positives (accuracy). To keep false positives low, the size of the bloom filter must be dimensioned *a priori* to be linear in the maximum number of keys inserted, with the linearity constant typically ranging from one to few bytes. A bloom filter is most commonly used as an *in-memory* data structure, hence its size is limited by the availability of RAM space on the machine.

An element lookup in a bloom filter involves hashing it to some number, say k, of different positions in the vector and check that these bits are all 1 – if so, it is concluded that element is in the set that the bloom filter represents, otherwise not. An element insertion involves setting the corresponding k bit positions to 1. It can now be easily seen how a false positive can happen during a lookup – an element may not have been inserted in the bloom filter but may have had all its k bit positions set to 1 during insertions of other elements. Further notation on bloom filters and discussion of false

positive probabilities is provided at the beginning of Section 6.2.

An excellent survey of bloom filter applications is provided in [38]. A common use of bloom filters is to identify non-existent elements so as to avoid high query latencies involved with accesses down the storage hierarchy or across the network. Many data center applications, for example, Venti [117], a data deduplication system, and Google Big Table [42], use bloom filters to detect unique data, which helps to avoid slow secondary storage accesses. However, the size of the bloom filter in some applications is too big to fit in main memory. As datasets have grown over time to Internet scale, so have the RAM space requirements of bloom filters, even after partitioned processing of data across multiple servers. In current data center applications, it is not uncommon to allocate upwards of few GB of RAM for storing bloom filters.

When sufficient RAM space is not available, it may be necessary to store the bloom filter in magnetic disk-based storage. However, as bloom filter operations are randomly spread over its length, the slow random access (seek) performance of disks, of the order of 10 msec, becomes a huge bottleneck. The use of independent hash functions destroys any locality that may be present in the element space, hence, even using a RAM based cache does not help to improve bloom filter performance. Since flash memory has faster read performance, of the order of 10-100 μsec in currently available Solid State Drives (SSDs), it is a viable storage option for implementing bloom filters and striking tradeoffs between high cost of RAM and fast bloom filter access times. Flash memory, however, provides relatively slow performance for random write operations (vs. reads and sequential writes). The aim of this paper is to design a flash-based bloom filter which is aware of flash memory characteristics and will provide superior performance compared to the simple design of a single flat bloom filter on flash.

There are two types of popular flash devices, NOR and NAND flash. NAND flash architecture allows a denser layout and greater storage capacity per chip. As a result, NAND flash memory has been significantly cheaper than DRAM, with cost decreasing at faster speeds. NAND flash characteristics have lead to an explosion in its usage in consumer electronic devices, such as MP3 players, phones, caches and Solid State Disks (SSDs). In the rest of this chapter, we use NAND flash based SSDs as the architectural choice and simply refer to it as flash memory.

In this thesis, we present the design and evaluation of BloomFlash, a bloom filter

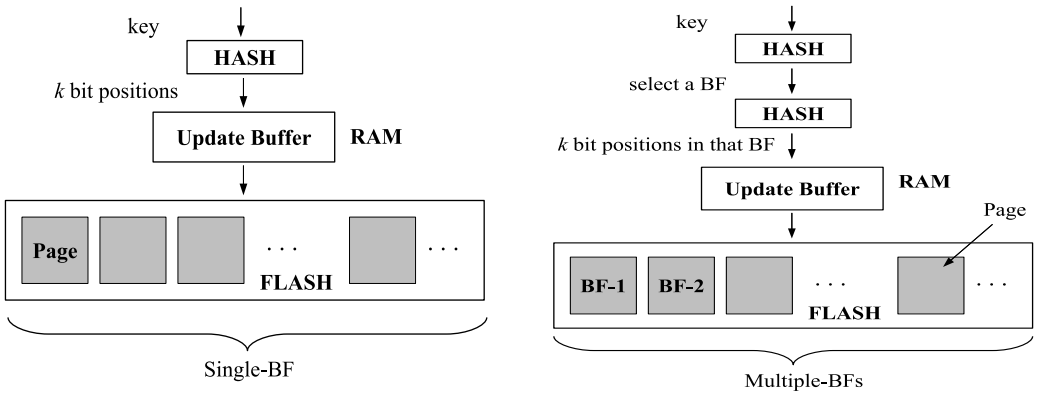
system on flash, that provides a new dimension of tradeoff with bloom filter access times to reduce RAM space usage (and hence system cost). The simple design of a single flat bloom filter on flash suffers from many performance bottlenecks, including in-place bit updates that are inefficient on flash and multiple reads and random writes spread out across many flash pages for a single lookup or insert operation. To mitigate these performance bottlenecks, we exploit two key design decisions: (i) buffering bit updates in RAM and applying them in bulk to flash (using two different flush-to-flash policies) that helps to reduce random writes to flash, and (ii) a *hierarchical* bloom filter design (adapted from [99, 116]) consisting of component bloom filters, stored one per flash page, that helps to localize reads and writes on flash. We use two real-world data traces taken from representative bloom filter applications to drive and evaluate our design. We use two different flash SSDs with different price/performance points for our evaluation. Currently, this work is under review for the publication [57].

The contributions of this chapter are summarized as follows:

- **Bloom filter design on flash:** To the best of our knowledge, this is the first work to propose a bloom filter design which is optimized for secondary storage, and in particular, flash memory. Our design leverages two key ideas: (i) buffering bit updates in RAM and applying them in bulk to flash (using two different flush-to-flash policies) that helps to reduce random writes to flash, and (ii) a *hierarchical* bloom filter design consisting of component bloom filters, stored one per flash page, that helps to localize reads and writes on flash.

- **Evaluation on real-world data center applications:** We use two real-world data traces taken from representative bloom filter applications to drive and evaluate our design. We use two different flash SSDs with different price/performance points for our evaluation. BloomFlash achieves bloom filter access times in the range of few tens of μsec, thus allowing up to about 28,500 ops/sec.

The rest of the chapter is organized as follows. In Section 6.2, we develop the design of BloomFlash. In Section 6.3, we describe two real-world data center applications and used them to evaluate and justify the design choices in BloomFlash. We review related work in Section 6.4. Finally, we conclude in Section 6.5.



(a) Single-Flat-BF (b) Hierarchical-BF

Figure 6.1: Bloom filter designs on flash: Single flat bloom filter vs. hierarchical bloom filter.

6.2 Bloom Filter on Flash

BloomFlash stores the bloom filter bits on flash and uses RAM to buffer bit updates resulting from element insertions so as to mitigate flash writing and garbage collection overheads that could result from excessive in-place bit updates to flash. The bloom filter organization on flash is divided into pages corresponding to flash page sizes that are typically 2KB or 4KB. All read and write operations on flash are performed at the page level. We begin with a single flat bloom filter design and then refine it with the objective of localizing read and write operations, corresponding to element lookups and insertions, to within a single flash page. We present and investigate two different policies for flushing buffered page updates to RAM that work with either design.

We begin with some notation. Let m be the total number of bits in the bloom filter. Let n be the maximum number of elements (keys) inserted. The insertion of a key into the bloom filter involves selecting k bit positions, using k different hash functions, and setting these k bits to 1. A lookup for a key in the bloom filter needs to compute the k bit positions using the k hash functions and checking whether all these bit positions are 1 – if so, the key is inferred to be in the bloom filter. Clearly, there is a chance of a *false positive* event here, since these k positions might have their bits to 1 by other inserted keys, while the current key may not have been inserted at all. After insertion

of n keys, the probability, $FP(n, m, k)$ of a false positive during lookup can be shown to be

$$FP(n, m, k) = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

Given the maximum number of elements inserted n and any one of two quantities m and k, we can minimize the false positive probability by choosing the other quantity as per the equation

$$k = \frac{m}{n} \ln 2 \tag{6.1}$$

6.2.1 Single Flat Bloom Filter on Flash

In our first design, the bloom filter is laid out in sequential logical address space on flash. With a flash physical page size of P bytes (typically equal to 2KB or 4KB), each page of flash holds $8P$ bits, so that the number of flash pages required is $p = \lceil \frac{m}{8P} \rceil$. Henceforth, we will assume that m is an integral multiple of $8P$. During an element lookup, the k different bit positions are translated to (logical) flash page ids and offsets. Each flash page is read to check whether the corresponding bit(s) within it is 0 or 1. A lookup can terminate with a negative answer when a bit position is read as 0 (in which case the remaining bit positions, if any, need not be read). An element insertion operation needs to set (at most) k bit positions to 1 (if they are not already set to 1).

Because flash page writes are inefficient, these bit update operations are deferred using a buffer in RAM. The idea is collect multiple updates for the same page and apply them at once. Since RAM buffer space is limited, we need to periodically flush some of the buffered updates to flash so as to accommodate new updates. It appears that this flushing policy is critical for designing an efficient flash-based bloom filter. We discuss different buffer flushing policies separately after we describe two designs for bloom filter layout on flash.

Due to the random nature of the k bit positions selected for each element, an element lookup will almost always involve k different flash page reads. This makes the element lookup time about k times that of a single flash page read. Also, an element write involves updates to about k flash pages (could be less if some of the k bits are already set) – thus each element write leads to bit updates on about k different flash pages to be buffered in RAM. Both of these are shortcomings of this first simple design,

which we address in the next design that is the choice for BloomFlash.

6.2.2 Using Hierarchical Bloom Filter Design to Localize Reads and Writes

The first design suffers from the drawback that a bloom filter lookup (almost always) involves k flash page reads, since the random nature of the k bit positions will place them in different flash pages with high probability. Although flash media is good for random reads (vs. hard disk), we aim to improve performance further by targeting just a single flash page read for a lookup operation on the bloom filter.

Our solution approach is to partition the single flat bloom filter into many smaller *component* bloom filters, each of the size of one flash page (e.g., 4KB). We call the resulting data structure a *hierarchical* bloom filter. With a flash page size of P bytes, this gives $p = \lceil \frac{m}{8P} \rceil$ component bloom filters, each fitting on one flash page. To insert or lookup an element, a first hash function is used to identify the bloom filter it should belong into. Then, k bit positions are identified *within* this bloom filter for setting or checking the bits. Thus, this design requires only one flash page read per element lookup. Moreover, this design also *localizes bit position writes* associated with an element insertion to within a single flash page – in contrast, in the first design, an element insertion involves setting bits in (almost always) k flash pages. Hence, insertion of n' elements can make at most n' flash pages dirty in the worst case, while that number is kn' in the first design. This aspect of the second design helps to reduce flash page writes. Figure 6.1 gives an overview of the first and second designs.

We next analyze the false positive probability of this composite bloom filter data structure.

Analysis of False Positive Probability

In BloomFlash, the overall bloom filter data structure is realized by multiple component bloom filters and each element is assigned to one component bloom filter (using a hash function). We analyze the false positive probability of this composite bloom filter data structure. Suppose that n keys are inserted into the hierarchical bloom filter and this leads to n_i element insertions in the i-th component bloom filter for $i = 1, 2, \ldots, p$.

Let the size of each small bloom filter be $m' = 8P$ bits. As before, k is the number of bit positions that is checked in each small bloom filter for an element lookup. The probability of an element hashing to the i-th component bloom filter is $1/p$ and the false positive probability of that element in that component bloom filter is $FP(n_i, m', k)$. Thus, the false positive probability of the hierarchical bloom filter is given by

$$\frac{1}{p}\sum_{i=1}^{p} FP(n_i, m', k)$$

Using Jensen's equality for convex functions, it can be shown that the false positive probability of the hierarchical bloom filter is at least that of that of the (equivalent) single flat bloom filter $FP(n, m, k)$ (in the first design), that is

$$\frac{1}{p}\sum_{i=1}^{p} FP(n_i, m', k) \geq FP(n, m, k)$$

with equality if and only if all the n_i's are equal, i.e., the hashing of elements to component bloom filters achieves exactly equal distribution of elements across the component bloom filters. In fact, as the distribution of elements across component bloom filters gets more skewed (or, unbalanced), the gap widens between the false positive probabilities of the hierarchical bloom filter and single flat bloom filter designs. A simple way to understand this effect is as follows: as the number of elements inserted into a component bloom filter increases beyond the average of n/p, its false positive rate increases *non-linearly* above $FP(n, m, k)$; hence, the false positive probabilities of above-average occupancy component bloom filters is *not* averaged out by that of other below-average occupancy small bloom filters.

Thus, it might be necessary to enforce fairly equal load balancing of elements across component bloom filters in this design in order to keep the false positive probability comparable to that of the single flat bloom filter design. One simple way to achieve this is to use the *power of two choice idea* from [34] that has been used to balance a distribution of balls thrown into bins. With a load balanced design for hierarchical bloom filter, each element would be hashed to two candidate component bloom filters and actually inserted into the one that has currently fewer inserted elements. During lookup, each element is looked up in both of its candidate component bloom filters (hence bloom filter lookup times would double).

Our evaluations show that the single-hash based assignment of an element to a component bloom filter achieves fairly equal load balancing of elements across component bloom filters. Hence, a more intricate design for load balancing is not required. We observe in our evaluations (on the data traces used) that the ratio of elements inserted in the maximum occupancy bloom filter to that in the minimum occupancy bloom filter is about 1.1, hence the distribution of each bin is within 10% from the average. Because of this, the false positive probability of the composite bloom filter is about the same as that of the (equivalent) single flat bloom filter. An intuitive explanation for why this fairly equal load balancing is achieved is that the number of elements inserted is orders of magnitude more than the number of component bloom filters – for example, with a flash page size of $P = 4096$ bytes and $m/n = 8$, the number of component bloom filters is $n/4096$. Hence, by deriving analogy from the well studied balls-and-bins problem, we expect the random assignment of balls to bins to achieve a fairly well balanced distribution. (As a side note, using the idea of two choices for inserting an element into a component bloom filter brings down the ratio of elements inserted in the maximum occupancy bloom filter to that in the minimum occupancy bloom filter to about 1.001.)

6.2.3 Buffering Updates in RAM to Reduce Flash Writes

We can buffer updates in RAM for each of the bloom filter designs above so as to reduce the number of flash writes. Since in-place updates to (logical) flash pages actually lead to new (physical) pages being allocated and written and increases garbage collection activity, this strategy can help to reduce write amplification and garbage collection overheads. Additionally, the latency of bloom filter reads could also decrease somewhat since a bit position (write) that is buffered in RAM can be read with reading the containing page on flash. Insertion Group the buffered updates by the page_id.

To realize this strategy, we group and store in RAM bit position updates (changes from 0 to 1 as a result of element insertion) for the same flash page – each update records a bit position offset within the page and all updates within a page share a common page id information. These updates are flushed to flash when the turn arrives to flush the corresponding page. When multiple contiguous flash pages in logical address

page are flushed together (vs. each page separately when their turn arrives), the write performance improves due to the large (sequential) nature of the writes. Hence, we apply the scheduling of pages flushed to flash in the granularity of *contiguous (logical) page groups* rather than individual pages. We evaluate the performance of different page group sizes in Section 6.3 and fix a page group size of 16 in our system implementation.

We investigate two page group flushing policies as follows:

- *Dirtiest group first:* Under this policy, the page group that contains the largest number of bit position updates (i.e., contains the maximum dirty bits) is given priority for flushing to flash. This greedy policy minimizes the total number of write operations. Since write operations more expensive than read operations on flash media, this is the conventional buffering policy. However, this policy ignores the performance degradation that can result from writing page groups in arbitrary order (vs. their ordering in the logical address space) since this results in random write operations to flash.

- *Groups in sequential order:* Under this policy, page groups are flushed out in sequential order in logical address space (with wraparound when the last page group is reached). This treats the bloom filter storage area on flash like an append log (with wraparound) and aims to reap the write performance benefits of using flash in a log-structured manner (the latter has been reported in earlier work [48, 79, 107, 132]).

Flushing of a page group is triggered when a bit position update, resulting from an element insertion process, needs to be buffered in RAM and a pre-configured space threshold in RAM for storing buffered updates is exceeded. Only one page group is flushed at a time and is chosen in accordance with either of the above flushing policies. We investigate the impact of varying buffer space thresholds on bloom filter performance.

6.3 Evaluation

In this section, we evaluate the design of BloomFlash in Fusion-IO [7] and Samsung [15] Solid State Drives (SSDs) with the help of two real-world applications.

Trace	Total bloom filter ops	lookup:insert ops ratio	Average key size (bytes)
Xbox	21.6 millions	165:1	92
Dedupe	27.7 millions	1.3:1	20

Table 6.1: Bloom filter operation statistics in the two traces used in the performance evaluation. The Xbox trace is a lookup intensive trace while the Dedupe trace is an insert (i.e., update) intensive trace.

6.3.1 Applications Used

We describe two real-world data center applications that use bloom filters to reduce data lookup latencies in the system. Data traces obtained from real-world instances of these applications is used to drive and evaluate the design of BloomFlash. The properties of the two traces in terms of associated bloom filter operations are summarized in Table 6.1.

Online Multi-player Gaming

Online multi-player gaming technology allows people from geographically diverse regions around the globe to participate in the same game. The number of concurrent players in such a game could range from tens to hundreds of thousands and the number of concurrent game instances offered by a single online service could range from tens to hundreds. An important challenge in online multi-player gaming is the requirement to scale the number of users per game and the number of simultaneous game instances. At the core of this is the need to maintain server-side state so as to track player actions on each client machine and update global game states to make them visible to other players as quickly as possible. These functionalities map to `set` and `get` key operations performed by clients on server-side state. The real-time responsiveness of the game is, thus, critically dependent on the response time and throughput of these operations.

In such online multi-player gaming applications, front-end caching servers are used to scale a back-end of partitioned database servers implementing a key-value store. Specifically, we consider the Microsoft Xbox LIVE Primetime online multiplayer game [25]. This application frequently uses lookups on non-existing keys to implement the game logic. Key writes arriving at a front-end server are broadcast to all other front-ends so that they can insert that key into their local bloom filters. For a key read operation, if the key is not in the local bloom filter of the front-end, that frontend can return `null`

to the client; otherwise, the read is sent to the appropriate back-end database server.

We evaluate the performance of FlashStore on a large trace of `get-set` key operations obtained from real-world instances of Xbox LIVE Primetime [25] in which the key is a dot-separated sequence of strings with total length averaging 94 characters. The ratio of bloom filter lookup to insert operations in the trace is about 165:1.

Storage Deduplication

Deduplication is a recent trend in storage backup systems that eliminates redundancy of data across full and incremental backup data sets [134]. It works by splitting files into multiple chunks using a content-aware chunking algorithm like Rabin fingerprinting and using SHA-1 hash signatures for each chunk to determine whether two chunks contain identical data [134]. In *inline* storage deduplication systems, the chunks (or their hashes) arrive one-at-a-time at the deduplication server from client systems. The server needs to lookup each chunk hash in an index it maintains for all chunk hashes seen so far for that backup location instance – if there is a match, the incoming chunk contains redundant data and can be deduplicated; if not, the (new) chunk hash needs to be inserted into the index.

Because storage systems currently need to scale to tens of terabytes to petabytes of data volume, the chunk hash index is too big to fit in RAM, hence it is stored on hard disk. Index operations are thus throughput limited by expensive disk seek operations. Since backups need to be completed over windows of half-a-day or so (e.g., nights and weekends), it is desirable to obtain high throughput in inline storage deduplication systems. Typical in such systems (e.g., [97, 134]) is the use of a bloom filter to determine if a chunk hash has been seen earlier so that disk access latencies can be avoided for chunks that are seen for the first time (which is the majority of chunks in the first full backup of a dataset).

To obtain a chunk hash trace from real-world datasets, we have built a storage deduplication analysis tool that can crawl a root directory, generate the sequence of chunk hashes for a given average chunk hash size, and compute the number of deduplicated chunks and storage bytes. We use traces generated from this tool on an enterprise backup dataset to evaluate bloom filter performance. In this application, the key is a 20-byte SHA-1 hash of the corresponding chunk. The trace contains 27,748,824 total

chunks and 12,082,492 unique chunks. Using this, we obtain the ratio of bloom filter lookup to insert operations in the trace as 1.3:1.

6.3.2 C++ Implementation

We have prototyped BloomFlash in approximately 3000 lines C++ code. To implement the k hash functions for bloom filter access, we use MurmurHash [13], which throughly mixes the bits of a value with multiple sequences of multiply, shift, and xor operations to make it pseudo-randomized. We store the bloom filter data in the file system and turn off operating system buffering of file blocks so as to avoid any favorable effects due to it. The value of k is chosen to be 6. Since the number of keys in each trace is known (i.e., the value of n), we use equation(1) to determine the size m of the bloom filter (single or hierarchical). In the rest of this section, by $CacheSizeFactor = x$, we refer that $\frac{SingleFlatBloomFilterSize*x}{FlashPageSize}$ number of update entries are buffered in the RAM cache. We set $FlashPageSize$ to 4KB.

6.3.3 Evaluation Platforms

We use two different Solid State Drives (SSDs) to evaluate the performance of BloomFlash. The first one is a Fusion-IO flash SSD [7] of capacity 80GB. The second one is a Samsung flash SSD [15] of capacity 100GB. The CPU is an Intel Core 2 Duo E6850 with a clock speed of 3GHz per core. We log the average number of ops/sec at a period of every 10,000 `get-set` operations during each run and then take the overall average over a run to obtain throughput numbers. To calculate latency, we divide total execution time by the total number of `get-set` operations.

6.3.4 Result Analysis

In this section, first we evaluate the effect of buffering in the *single flat BF* design. Next, we evaluate the improvement of the *hierarchical BF* design. Finally, we evaluate the impact of group size and cache size on the *hierarchical BF* design.

Single Flat BF Design

Buffering helps to improve the performance of this design by 34%-42%. Figure 6.2

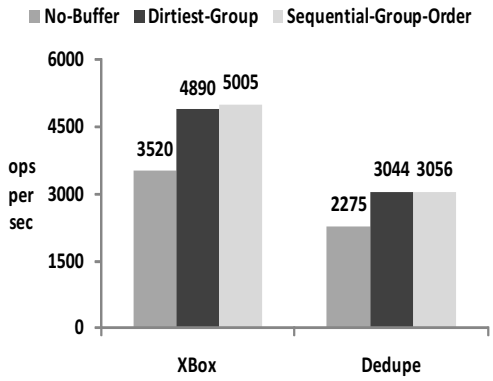
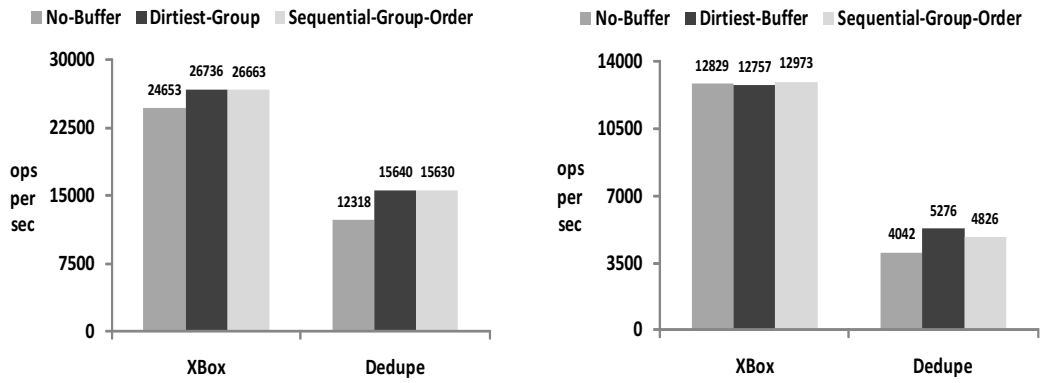


Figure 6.2: Single Flat BF Design on Fusion-IO ($CacheSizeFactor = 4$ and $GroupSize = 16$)

shows the performance trends of *single flat BF* design for Xbox and dedupe traces running in Fusion-IO drive.

Xbox Trace. Buffering helps to improve throughput by 39%-42%. Without any buffering, we achieve only 3,520 ops/sec. However, with buffering we achieve higher throughput. For the *dirtiest-group* flushing policy, we achieve 4,890 ops/sec, while for the *sequential-group-ordering* we achieve 5,005 ops/sec. The average key lookup time is 282.07 μsec, 202.78 μsec, and 198.29 μsec for no-buffering, the *dirtiest-group* flushing, and the *sequential-group-ordering* flushing policies, respectively. As explained in Section 6.2.3, buffering helps to reduce expensive flash write operations by applying the changes due to update operations in bulk to the underlying flash device, which consequently helps to improve throughput. In particular, buffering helps to reduce the total number of flash write operations approximately by 77% for the *dirtiest-group* policy and 75% the *sequential-group-ordering* policy.

Dedupe Trace. Buffering improves throughput approximately by 34%. It helps to improve throughput by reducing the total number of expensive flash write operations. It reduces write operations approximately by 87% for the both flushing policies. The *dirtiest-group* flushing policy achieves 3,044 ops/sec and the *sequential-group-ordering* achieves 3,056 ops/sec, while without any buffering we achieve only 2,275 ops/sec. The average key lookup time is 422.63 μsec, 326.55 μsec, and 325.15 μsec for no-buffering,



(a) Fusion-IO

(b) Samsung

Figure 6.3: Hierarchical BF Design ($CacheSizeFactor = 4$ and $GroupSize = 16$)

	Single-Flat-BF		**Hierarchical-BF**		**Improvement**	
	Xbox (ops/sec)	**Dedupe (ops/sec)**	**Xbox (ops/sec)**	**Dedupe (ops/sec)**	**Xbox**	**Dedupe**
No-Buffer	3,520	2,275	24,653	12,318	7.00x	5.41x
Dirtiest-Group	4,890	3,044	26,736	15,640	5.47x	5.13x
Sequential-Group-Order	5,005	3,056	26,663	15,630	5.33x	5.11x

Table 6.2: Throughput: Single Flat BF Design vs. Hierarchical BF Design (for Fusion-IO drive with $CacheSizeFactor = 4$ and $GroupSize = 16$)

the *dirtiest-group* flushing, and the *sequential-group-ordering* flushing policies, respectively.

From Figure 6.2, it is clear that the throughput is higher for Xbox compared to dedupe case. The main reason behind this trend is that Xbox workload is read-intensive, while dedupe workload is write-intensive. Since current flash-based SSDs provide comparatively faster read performance, certainly we achieve higher throughput for read-intensive Xbox trace. On the other hand, the average key lookup time is higher for dedupe case due to frequent garbage collection operations inside SSDs incurred by excessive write operations.

	Single-Flat-BF		Hierarchical-BF		Improvement	
	Xbox (μsec)	**Dedupe (μsec)**	**Xbox (μsec)**	**Dedupe (μsec)**	**Xbox**	**Dedupe**
No-Buffer	282.07	422.63	38.20	58.95	7.38x	7.17x
Dirtiest-Group	202.78	326.55	35.35	56.73	5.74x	5.76x
Sequential-Group-Order	198.29	325.15	35.36	56.75	5.60x	5.73x

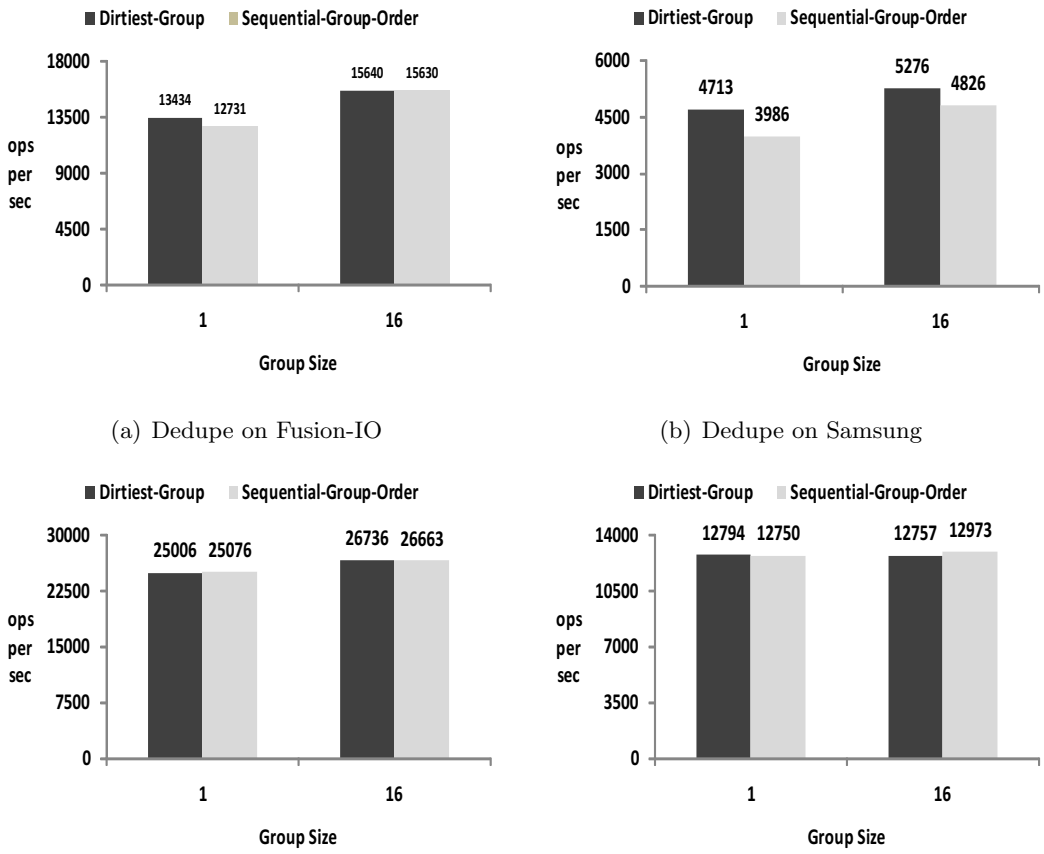
Table 6.3: Latency: Single Flat BF Design vs. Hierarchical BF Design (for Fusion-IO drive with $CacheSizeFactor = 4$ and $GroupSize = 16$)

Hierarchical BF Design

Hierarchical design improves the performance compared to the Single BF design since it localizes read and write operations to a single place for each lookup and insert operation. The improvement gain is linearly related with the value chosen for k (i.e., number of hash functions).

Figure 6.3 shows the performance trends of hierarchical design for Xbox and dedupe traces running in Fusion-IO and Samsung drives. The trends in Figure 6.3(a) show that the *hierarchical BF design* improves throughput 5.11x-7.00x and latency 5.60x-7.38x, compared to the *Single Flat BF design* for the Fusion-IO drive. The main reason for this improvement is the following. The *single flat BF design* uses $k = 6$ different hash functions for the bloom filter operations and these hash functions mostly perform random I/Os in the six different flash locations. Whereas, the *hierarchical BF design* uses one I/O per bloom filter operation, irrespective of the value of k. In addition, this design inherently localizes the updates in the buffer, which consequently helps to further improve throughput by reducing random write operations. This is also evident from the result trend that on average *hierarchical BF design* provides approximately six times more throughput. Table 6.2 and 6.3 give summaries of the improvement in *hierarchical BF design* for the Fusion-IO drive. We also observe similar trends for the Samsung drive.

Xbox Trace. Compared to single flat BF design, throughput improves by 5.33x-7.00x and latency improves by 5.60x-7.38x for the Fusion-IO drive. In particular, Figure 6.3(a) shows that with *hierarchical BF design* for Xbox, *dirtiest-group* flushing policy achieves 26,736 ops/sec and *sequential-group-ordering* achieves 26,663 ops/sec, while without



(a) Dedupe on Fusion-IO

(b) Dedupe on Samsung

(c) Xbox on Fusion-IO

(d) Xbox on Samsung

Figure 6.4: Effect of Group size in the hierarchical BF design ($CacheSizeFactor = 4$)

any buffering we achieve only 24,653 ops/sec. Thus, both *dirtiest-group* and *sequential-group-ordering* flushing policies improve throughput by 8%, compared to having no buffer. Unlike, the *single flat BF* case, here buffering does not significantly help to improve throughput. Buffering helps to reduce total number of flash write operations by 3%. This occurs due to impact of *hierarhical BF design* as it localizes the update operations from six different flash locations to single flash location, consequently reduces the total number of flash I/O operations. Compared to *single flat BF* in Figure 6.2, *dirtiest-group* flushing, *sequential-group-ordering* flushing, and no buffering policies achieve 5.45x, 5.33x, and 7.00x higher throughput, respectively. The average key

look up time is 35.35 μsec, 35.36 μsec, and 38.20 μsec, respectively.

Figure 6.3(b) shows that with Samsung drive, for Xbox, the *dirtiest-group* flushing policy achieves 12,757 ops/sec and *sequential-group-ordering* achieves 12,973 ops/sec, while without any buffering we achieve only 12,829 ops/sec. Thus, in this case buffering does not help much to improve the performance. The main reason is that during flushing, we need to perform read operations in order to fetch old page from flash, need to merge buffered update with it, and finally need to write the updated page back to flash. Clearly, flushing has some overhead in terms of flash read opeations. For Samsung drive, this overhead compensated the gain from buffering due to writing updates in bulk to the underlying flash storage.

Dedupe Trace. Compared to the single flat BF design, throughput improves by 5.11x-5.41x and latency improves by 5.73x-7.17x. Figure 6.3(a) shows with *hierarchical BF design*, the *dirtiest-group* flushing policy achieves 15,640 ops/sec and the *sequential-group-ordering* achieves 15,630 ops/sec, while without any buffering we achieve only 12,318 ops/sec by using Fusion-IO drive. Thus, both *dirtiest-group* and *sequential-group-ordering* flushing policies improve throughput by 27% compared to having no buffer. Compared to the *single flat BF* in Figure 6.2, *dirtiest-group* flushing, *sequential-group-ordering* flushing, and no buffering policies achieve 5.13x, 5.11x, and 5.41x higher throughput, respectively. The average key lookup time is 56.73 μsec, 56.75 μsec, and 58.95 μsec, respectively.

Figure 6.3(a) shows that with Samsung drive, the *dirtiest-group* flushing policy achieves 5,276 ops/sec and the *sequential-group-ordering* achieves 4,826 ops/sec, while without any buffering we achieve only 4,042 ops/sec. Thus, *dirtiest-group* and *sequential-group-ordering* flushing policies improve throughput by 31% and 19%, respectively, compared to having no buffer.

Overall, for *hierarchical BF design*, Xbox achieves higher throughput compared to the dedupe case. This is due to the read intensive nature Xbox and relatively faster read performance of the current generations of flash-based SSDs.

Since hierarchical BF design provides significant improvement in throughput and

latency compared to the single flat BF design, in the rest of this section we focus only on the hierarchical design.

Group Size Effect on Flushing Policies

Now, we study the impact of flushing polices in the *hierarchical BF design* by varying the group size. Figure 6.4 shows the performance trends due to various flushing policies. When group size = 1, every time we flush updates for only one page. Whereas, for group size = 16, every time we flush the updates for 16 consecutive pages. During this set of experiments, we use $CacheSizeFactor = 4$.

For dedupe with Fusion-IO, with $GroupSize = 1$, the *dirtiest-group* and *sequential-group-ordering* flushing policies achieve 9% and 3% higher throughput, respectively, compared to no buffering. While with $GroupSize = 16$, both *dirtiest-group* and *sequential-group-ordering* flushing policies achieve 27% higher throughput compared no buffering case. By increasing group size, we can utilize the better sequential write performance of the flash memory, consequently throughput increases. In Samsung drive, compared to no buffering case, for $GroupSize = 1$, the *dirtiest-group* flushing policy achieves 16% higher throughput. While, for $GroupSize = 16$, both *dirtiest-group* and *sequential-group-ordering* flushing policies achieve 31% and 19% higher throughput, respectively, compared no buffering case. Clearly, for both SSDs larger group size helps to achieve higher throughput.

For dedupe with Fusion-IO, compared to no buffering case, for $GroupSize = 1$, both *dirtiest-group* and *sequential-group-ordering* flushing policies achieve 1% higher throughput. While for $GroupSize = 16$, both *dirtiest-group* and *sequential-group-ordering* flushing policies achieve 8% higher throughput compared no buffering case. In Samsung drive, group size does not have much impact on throughput for the dedupe trace.

Overall, the *dirtiest-group* flushing policy with $GroupSize = 16$ provides higher throughput.

Cache Size Effect

Figure 6.5 shows the effect of increasing cache size on the Xbox trace performance running in Fusion-IO drive. In this set of experiments, we set $GroupSize = 16$.

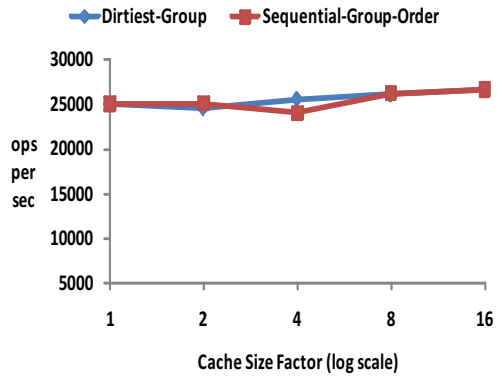


Figure 6.5: Effect of cache size for Xbox in the hierarchical BF design (for Fusion-IO with $GroupSize = 16$)

As expected with the increase of cache size, throughput also increases. This trend occurs as larger cache helps to reduce the total number of expensive write operations. However, the improvement rate is slow. This is due to the read intensive nature of Xbox trace. Since update operations are not frequent, increasing cache size does not help much. For dedupe trace, increasing cache size helps to improve performance due to its write-intensive nature.

6.4 Related Work

Our hierarchical bloom filter design is inspired by the blocked bloom filter design in [116], which uses a set of CPU cache line sized (about 64 bytes) smaller bloom filters. The latter design goes back further in the literature and is adapted from [99]. In our case, each smaller bloom filter is sized to fit into a flash page. In the blocked bloom filter case, lots of bloom filters are needed due to relative smaller size of cache-lines. As a result, the probability of some smaller bloom filters getting overloaded is very high. In contrast, due to relatively larger size of flash pages, we need to use fewer number of bloom filters and the probability of overloading is quite low.

To the best of our knowledge, there is no prior work in the literature that uses hierarchical bloom filters as in BloomFlash to optimize for storage on flash media. Multiple bloom filters have been used to dynamically "grow" a bloom filter when the number of keys to be inserted is not known a priori. The method in [68] starts with

a single standard bloom filter. When insertion of an additional element will increase the false positive probability beyond a threshold, a new bloom filter is instantiated and insertions happen in that one. At any given time, only one bloom filter is active for insertions. A lookup on an element proceeds by searching in *all* bloom filters. In contrast, all component bloom filters in BloomFlash are always active and a hash function is used to assign an element to a component bloom filter. Also, the element lookup process in BloomFlash involves searching in only one component bloom filter.

BufferHash [32], FAWN [33], ChunkStash [55], FlashStore [56], and MicroHash [133] use flash memory to design key-value store using hash table data structure. However, these systems cannot be directly used to design an efficient bloom filter.

6.5 Conclusion

We designed and evaluated BloomFlash, a bloom filter data structure on flash-based storage, that can tradeoff access times for a very slim RAM footprint. For server applications that need to use bloom filters for Internet scale datasets, this frees up constrained RAM space for other computation and lowers system cost. This is the first attempt to adapt the bloom filter, originally designed as an in-memory data structure, for secondary storage. BloomFlash was carefully designed to be flash aware and work with the constraints of flash storage media. BloomFlash exploits two key design decisions: (i) buffering bit updates in RAM and applying them in bulk to flash (using two different flush-to-flash policies) that helps to reduce random writes to flash, and (ii) a *hierarchical* bloom filter design consisting of component bloom filters, stored one per flash page, that helps to localize reads and writes on flash. Evaluations on real-world data traces taken from representative bloom filter applications show that BloomFlash achieves bloom filter access times in the range of few tens of μsec on currently available flash SSDs, thus allowing up to approximately 28,500 operations per sec.

Chapter 7

ChunkStash: A Flash-based Index Storage for Data Deduplication

7.1 Introduction

Deduplication is a recent trend in storage backup systems that eliminates redundancy of data across full and incremental backup data sets [97, 134]. It works by splitting files into multiple chunks using a content-aware chunking algorithm like Rabin fingerprinting and using 20-byte SHA-1 hash signatures [109] for each chunk to determine whether two chunks contain identical data [134]. In *inline* storage deduplication systems, the chunks arrive one-at-a-time at the deduplication server from client systems. The server needs to lookup each chunk hash in an index it maintains for all chunks seen so far for that storage location (dataset) instance. If there is a match, the incoming chunk contains redundant data and can be deduplicated; if not, the (new) chunk needs to be added to the system and its hash and metadata inserted into the index. The metadata contains information like chunk length and location and can be encoded in up to 44 bytes (as in [134, 97]). The 20-byte chunk hash (also referred to as *chunk-id*) is the *key* and the 44-byte metadata is the *value*, for a total *key-value pair* size of 64 bytes.

Because deduplication systems currently need to scale to tens of terabytes to

petabytes of data volume, the chunk hash index is too big to fit in RAM, hence it is stored on hard disk. Index operations are thus throughput limited by expensive disk seek operations which are of the order of 10msec. Since backups need to be completed over tight windows of few hours (over nights and weekends), it is desirable to obtain high throughput in inline storage deduplication systems, hence the need for a fast index for duplicate chunk detection. The index may be used in other portions of the deduplication pipeline also. For example, a recently proposed algorithm for chunking the data stream, called *bimodal chunking* [89], requires access to the chunk index to determine whether an incoming chunk has been seen earlier or not. Thus, multiple functionalities in the deduplication pipeline can benefit from a fast chunk index.

RAM prefetching and bloom-filter based techniques used by Zhu et al. [134] can avoid disk I/Os on close to 99% of the index lookups and have been incorporated in production systems like those built by Data Domain. Even at this reduced rate, an index lookup going to disk contributes about 0.1msec to the *average* lookup time – this is about 10^3 times slower than a lookup hitting in RAM. We propose to reduce the penalty of index lookup misses in RAM by orders of magnitude by serving such lookups from a flash memory based *key-value store*, thereby, increasing inline deduplication throughput. Flash memory is a natural choice for such a store, providing persistency and 100-1000 times lower access times than hard disk. Compared to DRAM, flash access times are about 100 times slower. Flash stands in the middle between DRAM and disk also in terms of cost [95] – it is about 10x cheaper than DRAM, while about 10x more expensive than disk – thus, making it an ideal gap filler between DRAM and disk and a suitable choice for this application.

7.1.1 Estimating Index Lookup Speedups using Flash Memory

We present a back-of-the-envelope calculation for decrease in average chunk index lookup time when flash memory is used as the metadata store for chunk-id lookups. This can be viewed as a necessary sanity check that we undertook before plunging into a detailed design and implementation of a flash-assisted inline storage deduplication system. We use the fundamental equation for average access time in multi-level memory architectures. Let the hit ratio in RAM be h_r. Let the lookup times in RAM, flash, and hard disk be t_r, t_f, and t_d respectively.

In the hard disk index based deduplication system described in Zhu et al. [134], prefetching of chunk index portions into RAM is shown to achieve RAM hit ratios of close to 99% on the evaluated datasets. Lookup times in RAM can be estimated at $t_r = 1\mu$sec, as validated in our system implementation (since it involves several memory accesses, each taking about 50-100nsec). Hard disk lookup times are close to $t_d = 10msec$, composed of head seek and platter rotational latency components. Hence, the average lookup time in this case can be estimated as

$$t_r + (1 - h_r) * t_d = 1\mu\text{sec} + 0.01 * 10\text{msec} = 101\mu\text{sec}$$

Now let us estimate the average lookup time when flash is used to serve index lookups that miss in RAM. Flash access times are around $t_f = 100\mu$sec, as obtained through measurement in our system. Hence, the average lookup time in a flash index based system can be estimated as

$$t_r + (1 - h_r) * t_f = 1\mu\text{sec} + 0.01 * 100\mu\text{sec} = 2\mu\text{sec}$$

This calculation shows a *potential speedup of* $50x$ *using flash* for serving chunk metadata lookups vs. a system that uses hard disk for the same. *That is a speedup of more than one order of magnitude.* At 50x index lookup speedups, other parts of the system could become bottlenecks, e.g., operating system and network/disk bottlenecks for data transfer. So we do not expect the overall system speedup (in terms of backup throughput MB/sec) to be 50x in a real implementation. However, the point we want to drive home here is that flash memory technology can help to get the index lookup portion of inline storage deduplication systems far out on the scaling curve.

7.1.2 Flash Memory and Our Design

There are two types of popular flash devices, NOR and NAND flash. NAND flash architecture allows a denser layout and greater storage capacity per chip. As a result, NAND flash memory has been significantly cheaper than DRAM, with cost decreasing at faster speeds. NAND flash characteristics have led to an explosion in its usage in consumer electronic devices, such as MP3 players, phones, and cameras.

However, it is only recently that flash memory, in the form of Solid State Drives (SSDs), is seeing widespread adoption in desktop and server applications. For example,

MySpace.com recently switched from using hard disk drives in its servers to using PCI Express (PCIe) cards loaded with solid state flash chips as primary storage for its data center operations [22]. Also very recently, Facebook announced the release of Flashcache, a simple write back persistent block cache designed to accelerate reads and writes from slower rotational media (hard disks) by caching data in SSDs [14]. These applications have different storage access patterns than typical consumer devices and pose new challenges to flash media to deliver sustained and high throughput (and low latency).

These challenges arising from new applications of flash are being addressed at different layers of the storage stack by flash device vendors and system builders, with the former focusing on techniques at the device driver software level and inside the device, and the latter driving innovation at the operating system and application layers. The work in this paper falls in the latter category. To get the maximum performance per dollar out of SSDs, it is necessary to use flash aware data structures and algorithms that work around constraints of flash media (e.g., avoid or reduce small random writes that not only have a higher latency but also reduce flash device lifetimes through increased page wearing).

To this end, we present the design and evaluation of ChunkStash, a *flash-assisted* inline storage deduplication system incorporating a high performance chunk metadata store on flash. When a key-value pair (i.e., chunk-id and its metadata) is written, it is sequentially logged in flash. A specialized RAM-space efficient hash table index employing a variant of cuckoo hashing [110] and compact key signatures is used to index the chunk metadata stored in flash memory and serve chunk-id lookups using one flash read per lookup. ChunkStash works in concert with existing RAM prefetching strategies. The flash requirements of ChunkStash are well within the range of currently available SSD capacities – as an example, ChunkStash can index order of *terabytes* of unique (deduplicated) data using order of *tens of gigabytes* of flash. Further, by indexing a small fraction of chunks per container, ChunkStash can reduce RAM usage significantly with negligible loss in deduplication quality. This work has been published in the USENIX 2010 conference [55].

In the rest of the paper, we use NAND flash based SSDs as the architectural choice and simply refer to it as flash memory. We describe the internal architecture of SSDs

in Chapter 2.

7.1.3 Our Contributions

The contributions of this paper are summarized as follows:

- **Chunk metadata store on flash:** ChunkStash organizes key-value pairs (corresponding to chunk-id and metadata) in a log-structured manner on flash to exploit fast sequential write property of flash device. It serves lookups on chunk-ids (20-byte SHA-1 hash) using one flash read per lookup.

- **Specialized space efficient RAM hash table index:** ChunkStash uses an in-memory hash table to index key-value pairs on flash, with hash collisions resolved by a variant of cuckoo hashing. The in-memory hash table stores compact key signatures instead of full keys so as to strike tradeoffs between RAM usage and false flash reads. Further, by indexing a small fraction of chunks per container, ChunkStash can reduce RAM usage significantly with negligible loss in deduplication quality.

- **Evaluation on enterprise datasets:** We compare ChunkStash, our flash index based inline deduplication system, with a hard disk index based system as in Zhu et al. [134]. For the hard disk index based system, we use BerkeleyDB [3], an embedded key-value store application that is widely used as a comparison benchmark for its good performance. For comparison with the latter system, we also include "a hard disk replacement with SSD" for the index storage, so as to bring out the performance gain of ChunkStash in not only using flash for chunk metadata storage but also in its use of flash aware algorithms. We use three enterprise backup datasets (two full backups for each) to drive and evaluate the design of ChunkStash. Our evaluations on the metric of backup throughput (MB/sec) show that ChunkStash outperforms (i) a hard disk index based inline deduplication system by 7x-60x, and (ii) SSD index (hard disk replacement but flash unaware) based inline deduplication system by 2x-4x.

The rest of the paper is organized as follows. We develop the design of ChunkStash in Section 7.2. We evaluate ChunkStash on enterprise datasets and compare it with our

implementation of a hard disk index based inline deduplication system in Section 7.3. We review related work in Section 7.4. Finally, we conclude in Section 7.5.

7.2 Flash-assisted Inline Deduplication System

We follow the overall framework of production storage deduplication systems currently in the industry [134, 97]. Data chunks coming into the system are identified by their SHA-1 hash [109] and looked up in an index of currently existing chunks in the system (for that storage location or stream). If a match is found, the metadata for the file (or, object) containing that chunk is updated to point to the location of the existing chunk. If there is no match, the new chunk is stored in the system and the metadata for the associated file is updated to point to it. (In another variation, the chunk hash is included in the file/object metadata and is translated to chunk location during read access.) Comparing data chunks for duplication by their 20-byte SHA-1 hash instead of their full content is justified by the fact that the probability of SHA-1 hash match for non-identical chunks is less by many orders of magnitude than the probability of hardware error [117]. We allocate 44 bytes for the metadata portion. The 20-byte chunk hash is the key and the 44-byte metadata is the value, for a total key-value pair size of 64 bytes.

Similar to [134] and unlike [97], our system targets *complete deduplication* and ensures that no duplicate chunks exist in the system after deduplication. However, we also provide a technique for RAM usage reduction in our system that comes at the expense of marginal loss in deduplication quality.

We summarize the main components of the system below and then delve into the details of the chunk metadata store on flash which is a new contribution of this paper.

Data chunking: We use *Rabin fingerprinting* based sliding window hash [118] on the data stream to identify chunk boundaries in a content dependent manner. A chunk boundary is declared when the lower order bits of the Rabin fingerprint match a certain pattern. The length of the pattern can be adjusted to vary the average chunk size. The average chunk size in our system is 8KB as in [134]. *Ziv-Lempel compression* [135] on individual chunks can achieve an average compression ratio of 2:1, as reported in [134]

and also verified on our datasets, so that the size of the stored chunks on hard disk averages around 4KB. The SHA-1 hash of a chunk serves as its chunk-id in the system.

On-disk Container Store: The *container store* on hard disk manages the storage of chunks. Each container stores at most 1024 chunks and averages in size around 4MB. (Because of the law of averages for this large number (1024) of chunks, the deviation of container size from this average is relatively small.) As new (unique) chunks come into the system, they are appended to the current container buffered in RAM. When the current container reaches the target size of 1024 chunks, it is sealed and written to hard disk and a new (empty) container is opened for future use.

Chunk Metadata Store on Flash (ChunkStash): To eliminate hard disk accesses for chunk-id lookups, we maintain, in flash, the metadata for all chunks in the system and index them using a specialized RAM index. The chunk metadata store on flash is a new contribution of this paper and is discussed in Section 7.2.1.

Chunk and Container Metadata Caches in RAM: A *cache for chunk metadata* is also maintained in RAM as in [134]. The fetch (prefetch) and eviction policies for this cache are executed at the container level (i.e., metadata for all chunks in a container). To implement this container level prefetch and eviction policy, we maintain a fixed size *container metadata cache* for the containers whose chunk metadata are currently held in RAM – this cache maps a container-id to the chunk-ids it contains. The size of the chunk metadata cache is determined by the size of the container metadata cache, i.e., for a container metadata cache size of C containers, the chunk metadata cache needs to hold $1024*b$ chunks. A distinguishing feature of ChunkStash (compared to the system in [134] is that it does not need to use bloom filters to avoid secondary storage (hard disk or flash) lookups for non-existent chunks.

Prefetching Strategy: We use the basic idea of predictability of sequential chunk-id lookups during second and subsequent full backups exploited in [134]. Since datasets do not change much across consecutive backups, duplicate chunks in the current full backup are very likely to appear in the same order as they did in the previous backup.

Hence, when the metadata for a chunk is fetched from flash (upon a miss in the chunk metadata cache in RAM), we prefetch the metadata for all chunks in that container into the chunk metadata cache in RAM and add the associated container's entry to the container metadata cache in RAM. Because of this prefetching strategy, it is quite likely that the next several hundreds or thousand of chunk lookups will hit in the RAM chunk metadata cache.

RAM Chunk Metadata Cache Eviction Strategy: The container metadata cache in RAM follows a Least Recently Used (LRU) replacement policy. When a container is evicted from this cache, the chunk-ids of all the chunks it contains are removed from the chunk metadata cache in RAM.

7.2.1 ChunkStash: Chunk Metadata Store on Flash

As a new contribution of this paper, we present the architecture of ChunkStash, the on-flash chunk metadata store, and the rationale behind some design choices. The design of ChunkStash is driven by the need to work around two types of operations that are not efficient on flash media, namely:

1. **Random Writes:** Small random writes effectively need to update data portions within pages. Since a (physical) flash page cannot be updated in place, a new (physical) page will need to be allocated and the unmodified portion of the data on the page needs to be relocated to the new page.

2. **Writes less than flash page size:** Since a page is the smallest unit of write on flash, writing an amount less than a page renders the rest of the (physical) page wasted – any subsequent append to that partially written (logical) page will need copying of existing data and writing to a new (physical) page.

Given the above, the most efficient way to write flash is to simply use it as an append log, where an append operation involves one or more flash pages worth of data (current flash page size is typically 2KB or 4KB). This is the main constraint that drives the rest of our key-value store design. Flash has been used in a log-structured manner and its benefits reported in earlier work ([79, 133, 108, 48, 33]. We organize chunk metadata

storage on flash into logical page units of 64KB which corresponds to the metadata for all chunks in a single container. (At 1024 chunks per container and 64 bytes per chunk-id and metadata, a container's worth of chunk metadata is 64KB in size.)

ChunkStash has the following main components, as shown in Figure 7.1:

RAM Chunk Metadata Write Buffer: This is a fixed-size data structure maintained in RAM that buffers chunk metadata information for the currently open container. The buffer is written to flash when the current container is sealed, i.e., the buffer accumulates 1024 chunk entries and reaches a size of 64KB. The RAM write buffer is sized to 2-3 times the flash page size so that chunk metadata writes can still go through when part of the buffer is being written to flash.

RAM Hash Table (HT) Index: The index structure, for chunk metadata entries stored on flash, is maintained in RAM and is organized as a hash table with the design goal of one flash read per lookup. The index maintains pointers to the full (chunk-id, metadata) pairs stored on flash. Key features include resolving collisions using a variant of cuckoo hashing and storing compact key signatures in memory to tradeoff between RAM usage and false flash reads. We explain these aspects shortly.

On-Flash Store: The flash store provides persistent storage for chunk metadata and is organized as an append log. Chunk metadata is written (appended) to flash in units of a *logical page size* of 64KB, corresponding to the chunk metadata of a single container.

7.2.2 Hash Table Design for ChunkStash

We outline the salient aspects of the hash table design for ChunkStash.

Resolving hash collisions using cuckoo hashing: Hash function collisions on keys result in multiple keys mapping to the same hash table index slot – these need to be handled in any hashing scheme. Two common techniques for handling such collisions include *linear probing* and *chaining* [87]. Linear probing can increase lookup time arbitrarily due to long sequences of colliding slots. Chaining hash table entries in RAM, on the other hand, leads to increased memory usage, while chaining buckets of key-value

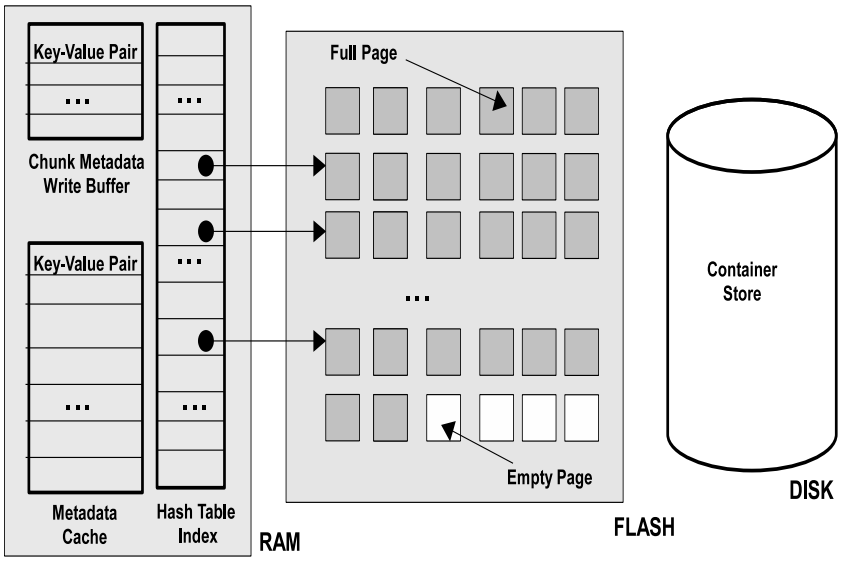


Figure 7.1: ChunkStash architectural overview.

pairs is not efficient for use with flash, since partially filled buckets will map to partially filled flash pages that need to be appended over time, which is not an efficient flash operation. Moreover, the latter will result in multiple flash page reads during key lookups and writes, which will reduce throughput.

ChunkStash structures the HT index as an array of slots and uses a variant of *cuckoo hashing* [110] to resolve collisions. Cuckoo hashing provides flexibility for each key to be in one of $n \geq 2$ candidate positions and for later inserted keys to relocate earlier inserted keys to any of their other candidate positions – this keeps the linear probing chain sequence upper bounded at n. In fact, the value proposition of cuckoo hashing is in increasing hash table load factors while keeping lookup times bounded to a constant. A study [136] has shown that cuckoo hashing is much faster than chained hashing as hash table load factors increase. The name "cuckoo" is derived from the behavior of some species of the cuckoo bird – the cuckoo chick pushes other eggs or young out of the nest when it hatches, much like the hashing scheme kicks previously inserted items out of their location as needed.

In the variant of cuckoo hashing we use, we work with n random hash functions

$h_1, h_2, \ldots, h_n$ that are used to obtain n *candidate positions* for a given key x. These candidate position indices for key x are obtained from the lower-order bit values of $h_1(x), h_2(x), \ldots, h_n(x)$ corresponding to a modulo operation. During insertion, the key is inserted in the first available candidate slot. When all slots for a given key x are occupied during insertion (say, by keys $y_1, y_2, \ldots, y_n$), room can be made for key x by relocating keys y_i in these occupied slots, since each key y_i has a choice of $(n-1)$ other locations to go to.

In the original cuckoo hashing scheme [110], a recursive strategy is used to relocate one of the keys y_i – in the worst case, this strategy could take many key relocations or get into an infinite loop, the probability for which can be shown to be very small and decreasing exponentially in n [110]. In our design, the system attempts a small number of key relocations after which it makes room by picking a key to move to an auxiliary linked list (or, hash table). In practice, by dimensioning the HT index for a certain load factor and by choosing a suitable value of n, such events can be made extremely rare, as we investigate in Section 7.3.4. Hence, the size of this auxiliary data structure is small. The viability of this approach has also been verified in [86], where the authors show, through analysis and simulations, that a very small constant-sized auxiliary space can dramatically reduce the insertion failure probabilities associated with cuckoo hashing. (That said, we also want to add that the design of ChunkStash is amenable to other methods of hash table collision resolution.)

The number of hash function computations during lookups can be reduced from the worst case value of n to 2 using the standard technique of *double hashing* from the hashing literature [87]. The basic idea is that two hash functions g_1 and g_2 can simulate more than two hash functions of the form $h_i(x) = g_1(x) + ig_2(x)$. In our case, i will range from 0 to $n-1$. Hence, the use of higher number of hash functions in cuckoo hashing does not incur additional hash function computation overheads but helps to achieve higher hash table load factors.

Reducing RAM usage per slot by storing compact key signatures: Traditional hash table designs store the respective key in each entry of the hash table index [87]. Depending on the application, the key size could range from few tens of bytes (e.g., 20-byte SHA-1 hash as in storage deduplication) to hundreds of bytes or more. Given

that RAM size is limited (commonly in the order of few to several gigabytes in servers) and is more expensive than flash (per GB), if we store the full key in each entry of the RAM HT index, it may well become the bottleneck for the maximum number of entries on flash that can be indexed from RAM before flash storage capacity bounds kick in. On the other hand, if we do not store the key at all in the HT index, the search operation on the HT index would have to follow HT index pointers to flash to determine whether the key stored in that slot matches the search key – this would lead to many *false flash reads*, which are expensive, since flash access speeds are 2-3 orders of magnitude slower than that of RAM.

To address the goals of maximizing HT index capacity (number of entries) and minimizing false flash reads, we store a *compact key signature* (order of few bytes) in each entry of the HT index. This signature is derived from *both the key and the candidate position number that it is stored at.* In ChunkStash, when a key x is stored in its candidate position number i, the signature in the respective HT index slot is derived from the higher order bits of the hash value $h_i(x)$. During a search operation, when a key y is looked up in its candidate slot number j, the respective signature is computed from $h_j(y)$ and compared with the signature stored in that slot. Only if a match happens is the pointer to flash followed to check whether the full key matches. We investigate the percentage of false reads as a function of the compact signature size in Section 7.3.4.

Storing key-value pairs on flash: Chunk-id and metadata pairs are organized on flash in a log-structure in the order of the respective write operations coming into the system. The HT index contains pointers to (chunk-id, metadata) pairs stored on flash. We use a 4-byte pointer, which is a combination of a logical page pointer and a page offset. With 64-byte key-value pair sizes, this is sufficient to index 256GB of chunk metadata on flash – for an average chunk size of 8KB, this corresponds to a maximum (deduplicated) storage dataset size of about 33TB. (ChunkStash reserves the all-zero pointer to indicate an empty HT index slot.)

7.2.3 Putting It All Together

To understand the hierarchical relationship of the different storage areas in ChunkStash, it is helpful to understand the sequence of accesses during inline deduplication. A

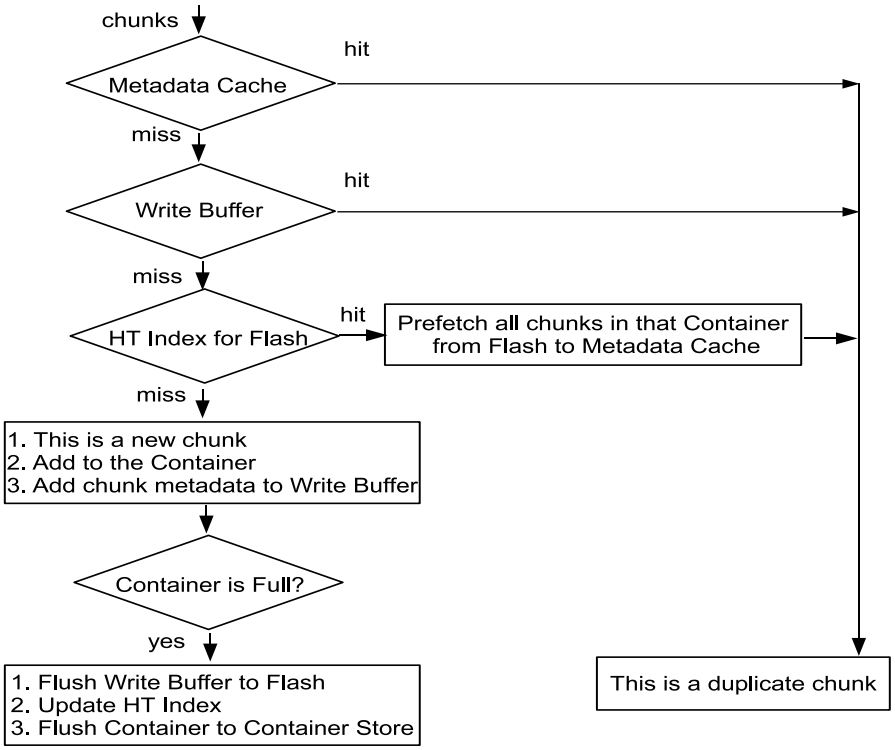


Figure 7.2: Flowchart of deduplication process in ChunkStash.

flowchart for this is provided in Figure 7.2. Recall that when a new chunk comes into the system, its SHA-1 hash is first looked up to determine if the chunk is a duplicate one. If not, the new chunk-id is inserted into the system.

A *chunk-id lookup* operation first looks up the RAM chunk metadata cache. Upon a miss there, it looks up the RAM chunk metadata write buffer. Upon a miss there, it searches the RAM HT Index in order to locate the chunk-id on flash. If the chunk-id is present on flash, its metadata, together with the metadata of all chunks in the respective container, is fetched into the RAM chunk metadata cache.

A *chunk-id insert* operation happens when the chunk coming into the system has not been seen earlier. This operation writes the chunk metadata into the RAM chunk metadata write buffer. The chunk itself is appended to the currently open container buffered in RAM. When the number of chunk entries in the RAM chunk metadata write

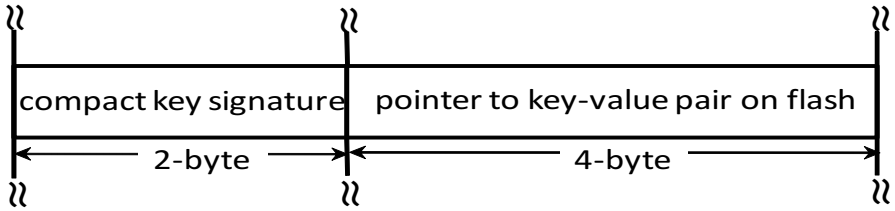


Figure 7.3: RAM HT Index entry and example sizes in ChunkStash. (The all-zero pointer is reserved to indicate an empty HT index slot.)

buffer reaches the target of 1024 for the current container, the container is sealed and written to the container store on hard disk, and its chunk metadata entries are written to flash and inserted into the RAM HT index.

7.2.4 RAM and Flash Capacity Considerations

The indexing scheme in ChunkStash is designed to use a small number of bytes in RAM per key-value pair so as to maximize the amount of indexable storage on flash for a given RAM usage size. The RAM HT index capacity determines the number of chunk-ids stored on flash whose metadata can be accessed with one flash read. The RAM size for the HT index can be determined with application requirements in mind. With a 2-byte compact key signature and 4-byte flash pointer per entry, the RAM usage in ChunkStash is 6 bytes per entry as shown in Figure 7.3. For a given average chunk size, this determines the relationship among the following quantities – RAM and flash usage per storage dataset and associated storage dataset size.

For example, a typical RAM usage of 4GB per machine for the HT index accommodates a maximum of about 716 million chunk-id entries. At an average of 8KB size per data chunk, this corresponds to about 6TB of deduplicated data, for which the chunk metadata occupies about 45GB on flash. This flash usage is well within the capacity range of SSDs shipping in the market today (from 64GB to 640GB). When there are multiple such SSDs attached to the same machine, additional RAM is needed to fully utilize their capacity for holding chunk metadata. Moreover, RAM usage by the HT index in ChunkStash can be further reduced using techniques discussed in Section 7.2.5.

To reap the full performance benefit of ChunkStash for speeding up inline deduplication, it is necessary for the entire chunk metadata for the (current) backup dataset to fit in flash. Otherwise, when space on flash runs out, the append log will need to be *recycled* and written from the beginning. When a page on the flash log is rewritten, the earlier one will need to be evicted and the metadata contained therein written out to a hard disk based index. Moreover, during the chunk-id lookup process, if the chunk is not found in flash, it will need to be looked up in the index on hard disk. Thus, both the chunk-id insert and lookup pathways would suffer from the same bottlenecks of disk index based systems that we sought to eliminate in the first place.

ChunkStash uses flash memory to store chunk metadata and index it from RAM. It provides flexibility for flash to serve, or not to serve, as a permanent abode for chunk metadata for a given storage location. This decision can be driven by cost considerations, for example, because of the large gap in cost between flash memory and hard disk. When flash is not the permanent abode for chunk metadata for a given storage location, the chunk metadata log on flash can be written to hard disk in one large sequential write (single disk I/O) at the end of the backup process. At the beginning of the next full backup for this storage location, the chunk metadata log can be loaded back into flash from hard disk in one large sequential read (single disk I/O) and the containing chunks can be indexed in RAM HT index. This mode of operation amortizes the storage cost of metadata on flash across many backup datasets.

7.2.5 Reducing ChunkStash RAM Usage

The largest portion of RAM usage in ChunkStash comes from the HT index. This usage can be reduced by indexing in RAM only a small fraction of the chunks in each container (instead of the whole container). Flash will continue to hold metadata for *all* chunks in all containers, not just the ones indexed in RAM; hence when a chunk in the incoming data stream matches an indexed chunk, metadata for *all* chunks in that container will be prefetched in RAM. We use an uniform chunk sampling strategy, i.e., we index every i-th chunk in every container which gives a sampling rate of $1/i$.

Because only a subset of chunks stored in the system are indexed in the RAM HT index, detection of duplicate chunks will not be completely accurate, i.e., some incoming chunks that are not found in the RAM HT index may, in fact, have appeared earlier

and are already stored in the system. This will lead to some loss in deduplication quality, and hence, some amount of duplicate data chunks will be stored in the system. In Section 7.3.6, we study the impact of this on deduplication quality (and backup throughput). We find that the loss in deduplication quality is marginal when about 1% of the chunks in each container are indexed and becomes negligibly small when about 10% of the chunks are indexed. The corresponding RAM usage reductions for the HT index are appreciable at 99% and 90% respectively. Hence, indexing chunk subsets in ChunkStash provides a powerful knob for reducing RAM usage with only marginal loss in deduplication quality.

In an earlier approach for reducing RAM usage requirements of inline deduplication systems, the method of sparse indexing [97] chops the incoming data into multiple megabyte segments, samples chunks *at random* within a segment (based on the most significant bits of the SHA-1 hash matching a pattern, e.g., all 0s), and uses these samples to find few segments seen in the recent past that share many chunks. In contrast, our sampling method is deterministic and samples chunks at uniform intervals in each container for indexing. Moreover, we are able to match incoming chunks with sampled chunks in *all* containers stored in the system, not just those seen in the recent past. In our evaluations in Section 7.3.6, we show that our uniform sampling strategy gives better deduplication quality than random sampling (for the same sampling rate).

7.3 Evaluation

We evaluate the backup throughput performance of a ChunkStash based inline deduplication system and compare it with our implementation of a disk index based system as in [134]. We use three enterprise datasets and two full backups for our evaluations.

7.3.1 C# Implementation

We have prototyped ChunkStash in approximately 8000 lines of C# code. MurmurHash [13] is used to realize the the hash functions used in our variant of cuckoo hashing to compute hash table indices and compact signatures for keys; two different seeds are used to generate two different hash functions in this family for use with the double hashing based simulation of n hash functions, as described in Section 7.2.2.

In our implementation, writes to the on-disk container store are performed in a non-blocking manner using a small pool of file writer worker threads. The metadata store on flash is maintained as a log file in the file system and is created/opened in non-buffered mode so that *there are no buffering/caching/prefetching effects in RAM from within the operating system.*

7.3.2 Comparison with Hard Disk Index based Inline Deduplication

We compare ChunkStash, our flash index based inline deduplication system, with a hard disk index based system as in Zhu et al. [134]. The index used in [134] appears to be proprietary and no details are provided in the paper. Hence, for purposes of comparative evaluation, we have built a hard disk index based system incorporating the ideas in [134] with the hard disk index implemented by BerkeleyDB [3], an embedded key-value database that is widely used as a comparison benchmark for its good performance. For comparison with the latter system, we also include a "hard disk replacement with SSD" for the index storage, so as to bring out the performance gain of ChunkStash in not only using flash for chunk metadata storage but also in its use of flash aware algorithms.

BerkeleyDB does not use flash aware algorithms but we used the parameter settings recommended in [5] to improve its performance with flash. We use BerkeleyDB in its non-transactional concurrent data store mode that supports a single writer and multiple readers [131]. This mode does *not* support a transactional data store with the ACID properties, hence provides a fair comparison with ChunkStash . BerkeleyDB provides a choice of B-tree and hash table data structures for building indexes – we use the hash table version which we found to run faster. We use the C++ implementation of BerkeleyDB with C# API wrappers [4].

7.3.3 Evaluation Platform and Datasets

We use a standard server configuration to evaluate the inline deduplication performance of ChunkStash and compare it with the disk index based system that uses BerkeleyDB. The server runs Windows Server 2008 R2 and uses an Intel Core 2 Duo E6850 3GHz CPU, 4GB RAM, and fusionIO 160GB flash drive [7] attached over PCIe interface. Containers are written to a RAID4 system using five 500GB 7200rpm hard disks. A

Trace	Size (GB)	Total Chunks	#Full Backups
Dataset 1	8GB	1.1 million	2
Dataset 2	32GB	4.1 million	2
Dataset 3	126GB	15.4 million	2

Table 7.1: Properties of the three traces used in the performance evaluation of ChunkStash. The average chunk size is 8KB.

separate hard disk is used for storing disk based indexes. For the fusionIO drive, *write buffering inside the device is disabled* and cannot be turned on through operating system settings. The hard drives used support write buffering inside the device by default and this setting was left on. This clearly gives some advantage to the hard disks for the evaluations but makes our comparisons of flash against hard disk more conservative.

To obtain traces from backup datasets, we have built a storage deduplication analysis tool that can crawl a root directory, generate the sequence of chunk hashes for a given average chunk size, and compute the number of deduplicated chunks and storage bytes. The enterprise data backup traces we use for evaluations in this paper were obtained by our storage deduplication analysis tool using 8KB (average) chunk sizes (this is also the chunk size used in [97]). We obtained two full backups for three different storage locations, indicated as Datasets 1, 2, and 3 in Table 7.1. The number of chunks in each dataset (for each full backup) are about 1 million, 4 million, and 15 million respectively.

We compare the throughput (MB/sec processed from the input data stream) on the three traces described in Table 7.1 for the following four inline deduplication systems:

- Disk based index (BerkeleyDB) and RAM bloom filter [134] (***Zhu08-BDB-HDD***),
- Zhu08-BDB system with SSD replacement for BerkeleyDB index storage (***Zhu08-BDB-SSD***),
- Flash based index using ChunkStash (***ChunkStash-SSD***),
- ChunkStash with the SSD replaced by hard disk (***ChunkStash-HDD***).

Some of the design decisions in ChunkStash also work well when the underlying storage is hard disk and not flash (e.g., log structured data organization and sequential writes). Hence, we have included Chunkstash running on hard disk as a comparison

point so as to bring out the impact of log structured organization for a store on hard disk for the storage deduplication application. (Note that BerkeleyDB does not use a log structured storage organization.)

All four systems use RAM prefetching techniques for the chunk index as described in [134]. When a chunk hash lookup misses in RAM (but hits the index on hard disk or flash), metadata for all chunks in the associated container are prefetched into RAM. The RAM chunk metadata cache size for holding chunk metadata is fixed at 20 containers, which corresponds to a total of 20,480 chunks. In order to implement the prefetching of container metadata in a single disk I/O for Zhu08-BDB, we maintain, in addition to the BerkeleyDB store, an auxiliary sequential log of chunk metadata that is appended with container metadata whenever a new container is written to disk.

Note that unlike in [134], our evaluation platform is not a production quality storage deduplication system but rather a research prototype. Hence, our throughput numbers should be interpreted in a relative sense among the four systems above, and not used for absolute comparison with other storage deduplication systems.

7.3.4 Tuning Hash Table Parameters

Before we make performance comparisons of ChunkStash with the disk index based system, we need to tune two parameters in our hash table design from Section 7.2.2, namely, (a) number of hash functions used in our variant of cuckoo hashing, and (b) size of compact key signature. These affect the throughput of read key and write key operations in different ways that we discuss shortly. For this set of experiments, we use one of the storage deduplication traces in a modified way so as to study the performance of hash table insert and lookup operations separately. We pre-process the trace to extract a set of 1 million unique keys, then insert all the key-value pairs, and then read all these key-value pairs in random order.

As has been explained earlier, the value proposition of cuckoo hashing is in accommodating higher hash table load factors without increasing lookup time. It has been shown mathematically that with 3 or more hash functions and with load factors up to 91%, insertion operations succeed in expected constant time [110]. With this prior evidence in hand, we target a maximum load factor of 90% for our cuckoo hashing implementation. Hence, for the dedup trace used in this section with 1 million keys,

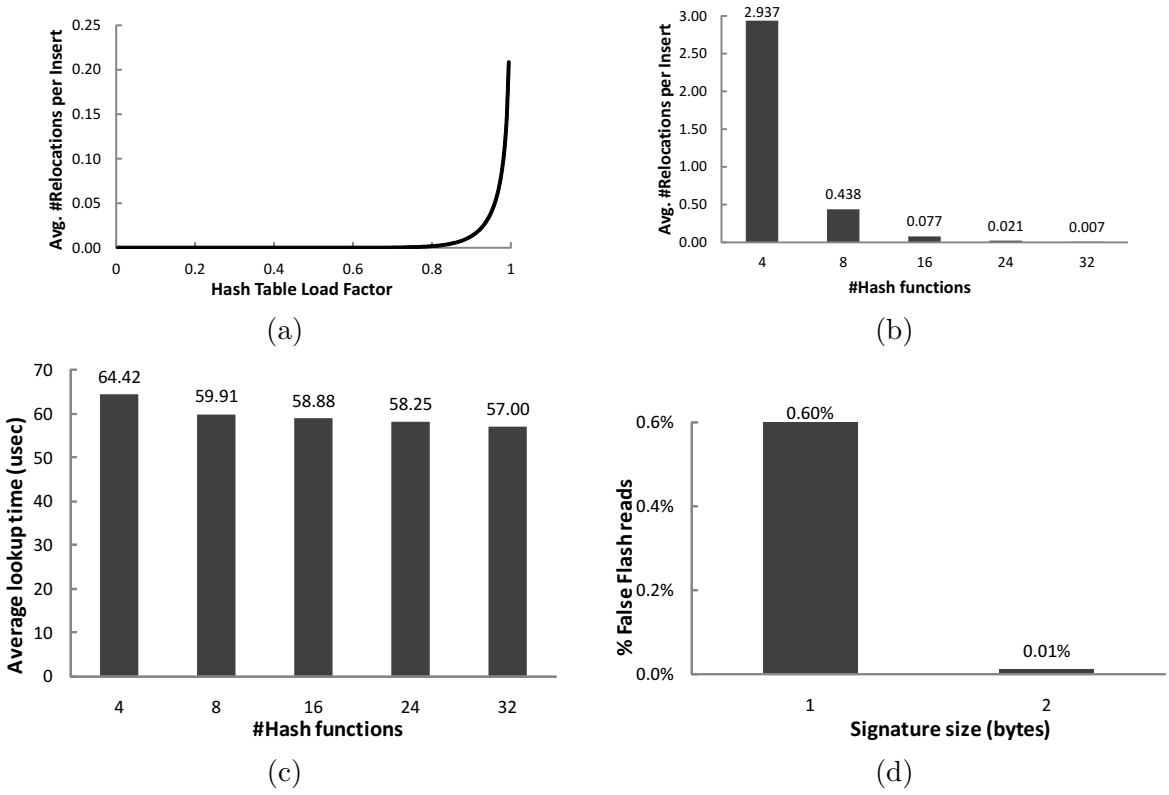


Figure 7.4: Tuning hash table parameters in ChunkStash: (a) Average number of relocations per insert as keys are inserted into hash table (for $n = 16$ hash functions); (b) Average number of relocations per insert vs. number of hash functions (n), averaged between 75%-90% load factors; (c) Average lookup time (μsec) vs. number of hash functions (n); (d) Percentage of false flash reads during lookup vs. signature size (bytes) stored in hash table.

we fix the number of hash table slots to 1.1 million.

Number of Hash Functions. When the number of hash functions n is small, the performance of insert operations can be expected to degrade in two ways as the hash table loads up. First, an insert operation will find all its n slots occupied by other keys and the number of cascaded key relocations required to complete this insertion will be high. Since each key relocation involves a flash read (to read the full key from flash and compute its candidate positions), the insert operation will take more time to complete. Second, with an upper bound on the number of allowed key relocations (which we set

to 50 for this set of experiments), the insert operation could lead to a key being moved to the auxiliary linked list – this increases the RAM space usage of the linked list as well as its average search time. On the other hand, as the number of hash functions increase, lookup times will increase because of increasing number of hash table positions searched. However, the latter undesirable effect is not expected to be as degrading as those for inserts, since a lookup in memory takes orders of magnitude less time than a flash read. We study these effects to determine a suitable number of hash functions for our design. (Note that because of our use of double hashing, the number of hash function computations per lookup does not increase with n.)

In Figure 7.4(a), we plot the average number of key relocations (hence, flash reads) per insert operation as keys are inserted into the hash table (for $n = 16$ hash functions). We see that the performance of insert operations degrades as the hash table loads up, as expected because of the impact of the above effect. Accordingly, for the following plots in this section, we present average numbers between 75% and 90% load factors as the insert operations are performed.

In Figure 7.4(b), we plot the average number of key relocations per insert operation as the number of hash functions n is varied, $n = 4, 8, 16, 24, 32$. At and beyond $n = 16$ hash functions, the hash table incurs less than 0.1 key relocations (hence, flash reads) per insert operation. Figure 7.4(c) shows that there is no appreciable increase in average lookup time as the number of hash functions increase. (The slight decrease in average lookup time with increasing number of hash functions can be attributed to faster search times in the auxiliary linked list, whose size decreases as number of hash functions increases.)

Based on these effects, we choose $n = 16$ hash functions in the RAM HT Index for our ChunkStash implementation. Note that during a key lookup, all n hash values on the key need not be computed, since the lookup stops at the candidate position number the key is found in. Moreover, because of the use of double hashing, at most two hash function computations are incurred even when all candidate positions are searched for a key. We want to add that using fewer hash functions may be acceptable depending on overall performance requirements, but we do not recommend a number below $n = 8$. Also, with $n = 16$ hash functions, we observed that it is sufficient to set the maximum number of allowed key relocations (during inserts) to 5-10 to keep the

number of inserts that go to the linked list very small.

Compact Key Signature Size. As explained in Section 7.2.1, we store compact key signatures in the HT index (instead of full keys) to reduce RAM usage. However, shorter signatures lead to more *false flash reads* during lookups (i.e., when a pointer into flash is followed because the signature in HT index slot matches, but the full key on flash does not match with the key being looked up). We study this effect in Figure 7.4(d) where we fix the number of hash functions to $n = 16$. (We did not find much variation in the fraction of false flash reads when number of hash functions is increased so long as $n \geq 8$.) We observe that the fraction of false reads drops sharply to 0.01% when the number of signature bytes increases from 1 to 2.

Since flash reads are expensive compared to RAM reads, the design should strike a balance between reducing false flash reads and RAM usage. Based on the above numbers, we fix the signature size to 2 bytes in our implementation. Even a 1-byte signature size may be acceptable given that only 0.6% of the flash reads are false in that case.

7.3.5 Backup Throughput

We ran the chunk traces from the the three datasets outlined in Table 7.1 on ChunkStash and our implementation of the system in [134] using BerkeleyDB as the chunk index on either hard disk or flash. We log the backup throughput (MB of data backed up per second) at a period of every 10,000 input chunks during each run and then take the overall average over a run to obtain throughput numbers shown in Figures 7.5, 7.6, and 7.7.

ChunkStash achieves average throughputs of about 190 MB/sec to 265 MB/sec on the first full backups of the three datasets. The throughputs are about 60%-140% more for the second full backup compared to the first full backup for the datasets – this reflects the effect of prefetching chunk metadata to exploit sequential predictability of chunk lookups during second full backup. The speedup of ChunkStash over Zhu08-BDB-HDD is about 30x-60x for the first full backup and 7x-40x for the second full backup. Compared to the Zhu08-BDB-SSD in which the hard disk is replaced by SSD for index storage, the speedup of ChunkStash is about 3x-4x for the first full backup and about

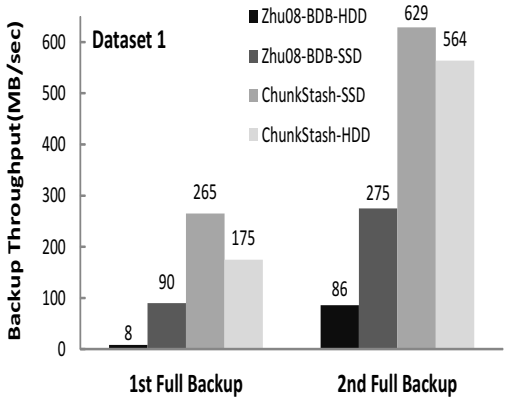


Figure 7.5: Dataset 1: Comparative throughput (backup MB/sec) evaluation of ChunkStash and BerkeleyDB based indexes for inline storage deduplication.

Trace	RAM Hit Rate	
	1st Full Backup	2nd Full Backup
Dataset 1	20%	97%
Dataset 2	2%	88%
Dataset 3	23%	80%

Table 7.2: RAM hit rates for chunk hash lookups on the three traces for each full backup.

2x-4x for the second full backup. The latter reflects the relative speedup of ChunkStash due to the use of flash-aware data structures and algorithms over BerkeleyDB which is not optimized for flash device properties.

We also run ChunkStash on hard disk (instead of flash) to bring out the performance impact of a log-structured organization on hard disk. Sequential writes of container metadata to the end of the metadata log is a good design for and benefits hard disks also. On the other hand, lookups in the log from the in-memory index may involve random reads in the log which are expensive on hard disks due to seeks – however, container metadata prefetching helps to reduce the number of such random reads to the log. We observe that the throughput of ChunkStash-SSD is more than that of ChunkStash-HDD by about 50%-80% for the first full backup and by about 10%-20% for the second full backup for the three datasets.

In Table 7.2, we show the RAM hit rates for chunk hash lookups on the three

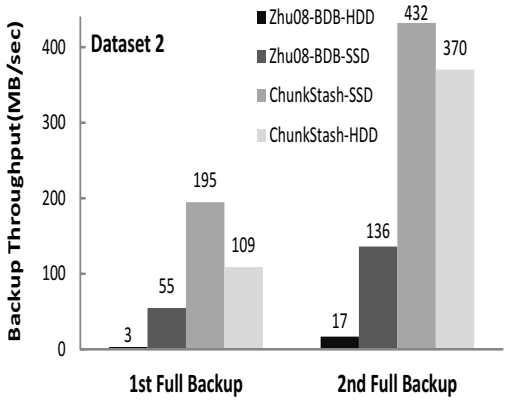


Figure 7.6: Dataset 2: Comparative throughput (backup MB/sec) evaluation of ChunkStash and BerkeleyDB based indexes for inline storage deduplication.

traces for each full backup. The RAM hit rate for the first full backup is indicative of the redundancy (duplicate chunks) *within* the dataset – this is higher for Datasets 1 and 3 and is responsible for their higher backup throughputs. The RAM hit rate for the second full backup is indicative of its similarity with the first full backup – this manifests itself during the second full backup through fewer containers written and through sequential predictability of chunk hash lookups (which is exploited by the RAM prefetching strategy).

When compared to ChunkStash, the relatively worse performance of a BerkeleyDB based chunk metadata index can be attributed to the increased number of disk I/Os (random reads and writes). We measured the number of disk I/Os for Zhu08-BDB and ChunkStash systems using Windows Performance Analysis Tools (`xperf`) [24]. In Figure 7.8, we observe that the number of read I/Os in BerkeleyDB is 3x-7x that of ChunkStash and the number of write I/Os is about 600-1000x that of ChunkStash. Moreover, these writes I/Os in BerkeleyDB are all random I/Os, while ChunkStash is designed to use only sequential writes (appends) to the chunk metadata log. Because there is no locality in the key space in an application like storage deduplication, container metadata writes to a BerkeleyDB based index lead to in-place updates (random writes) to many different pages (on disk or flash) and appears to be one of the main reasons for its worse performance.

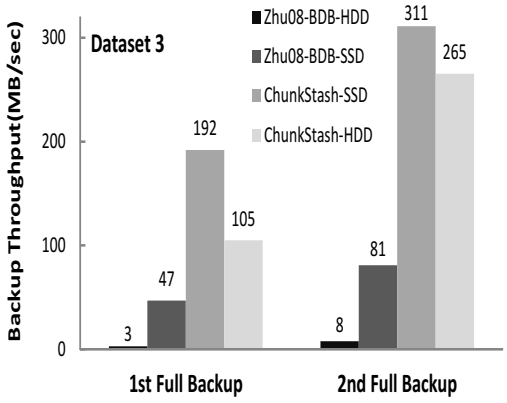


Figure 7.7: Dataset 3: Comparative throughput (backup MB/sec) evaluation of ChunkStash and BerkeleyDB based indexes for inline storage deduplication.

7.3.6 Impact of Indexing Chunk Subsets

In Section 7.2.5, we saw that RAM usage in ChunkStash can be reduced by indexing a small fraction of the chunks in each container. In this section, we study the impact of this on deduplication quality and backup throughput. Because only a subset of the chunks are indexed in the RAM HT index, detection of duplicate chunks will not be completely accurate, i.e., some incoming chunks that are not found in the RAM HT index may have appeared earlier and are already stored in the system. Hence, some amount of duplicate data chunks will be stored in the system. We compare two chunk sampling strategies – uniform chunk sampling strategy (i.e., index every i-th chunk) and random sampling (based on the most significant bits of the chunk SHA-1 hash matching a pattern, e.g., all 0s), the latter being used as part of a sparse indexing scheme in [97].

In Figure 7.9, we plot the fraction of chunks that are declared as new by the system during the second full backup of Dataset 2 as a percentage of the total number of chunks in the second full backup. The lower the value of this fraction, the better is the deduplication quality (the baseline for comparison being the case when all chunks are indexed). The fraction of chunks indexed in each container is varied as $1.563\% = 1/64$, $6.25\% = 1/16$, $12.5\% = 1/8$, and 100% (when all chunks are indexed). We choose sampling rates that are reciprocals of powers of 2 because those are the types of sampling

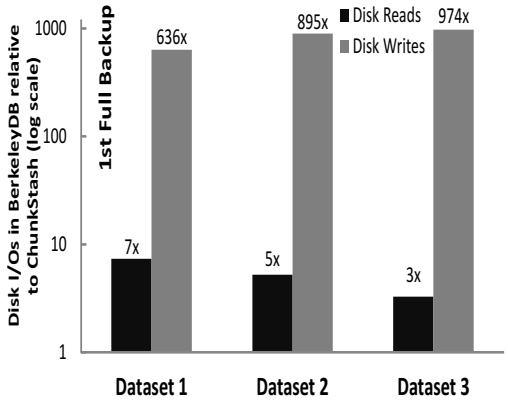


Figure 7.8: Disk I/Os (reads and writes) in BerkeleyDB relative to ChunkStash on the first full backup of the three datasets. (Note that the y-axis is log scale.)

rates possible in the random sampling scheme – when n most significant bits of the SHA-1 hash are matched to be all 0s, the sampling rate is $1/2^n$.

We observe that when 1.563% of the chunks are indexed, the uniform chunk sampling strategy results in only a 0.5% increase in the number of chunks detected as new (as a fraction of the whole dataset). This loss in deduplication quality could be viewed as an acceptable tradeoff in exchange for a 98.437% reduction in RAM usage of ChunkStash HT index. When 12.5% of the chunks are indexed, the loss in deduplication quality is almost negligible at 0.1% (as a fraction of the whole dataset), but the reduction in RAM usage of ChunkStash HT index is still substantial at 87.5%. Hence, indexing chunk subsets provides a powerful knob for reducing RAM usage in ChunkStash with only marginal loss in deduplication quality.

We also note that the loss in deduplication quality with random sampling is worse than that with uniform sampling, especially at lower sampling rates. An intuitive explanation for this is that uniform sampling gives better coverage of the input stream at regular intervals of number of chunks (hence, data size intervals), whereas random sampling (based on some bits in the chunk SHA-1 hash matching a pattern) could lead to large gaps between two successive chunk samples in the input stream.

An interesting side-effect of indexing chunk subsets is the increase in backup throughput – this is shown in Figure 7.10 for the first and second full backups for Dataset 2.

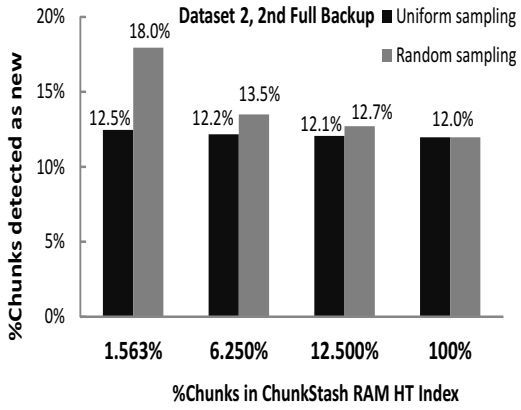


Figure 7.9: Dataset 2: Number of chunks detected as new as a fraction of the total number of chunks (indicating deduplication quality) vs. fraction of chunks indexed in ChunkStash RAM HT index in second full backup. (When 100% of the chunks are indexed, all duplicate chunks are detected accurately.) The x-axis fractions correspond to sampling rates of 1/64, 1/16, and 1/8. For a sampling rate of $1/2^n$, uniform sampling indexes every 2^n-th chunk in a container, whereas random sampling indexes chunks with first n bits of SHA-1 hash are all 0s.

The effects on first and second full backups can be explained separately. When a fraction of the chunks are indexed, chunk hash keys are inserted into the RAM HT index at a lower rate during the first full backup, hence the throughput increases. (Note, however, that writes to the metadata log on flash are still about the same, but possibly slightly higher due to less accurate detection of duplicate chunks within the dataset.) During the second full backup, fewer chunks from the incoming stream are found in the ChunkStash RAM HT index, hence the number of flash reads during the backup process are reduced, leading to higher throughput. Both of these benefits drop gradually as more chunks are indexed but still remain substantial at a sampling rate of 12.5% – with the loss in deduplication quality being negligible at this point, the tradeoff is more of a win-win situation than a compromise involving the three parameters of RAM usage (low), deduplication throughput (high), and loss is deduplication quality (negligible).

7.3.7 Flash Memory Cost Considerations

Because flash memory is more expensive per GB than hard disk, we undertake a performance/dollar analysis in an effort to mitigate cost concerns about a flash-assisted

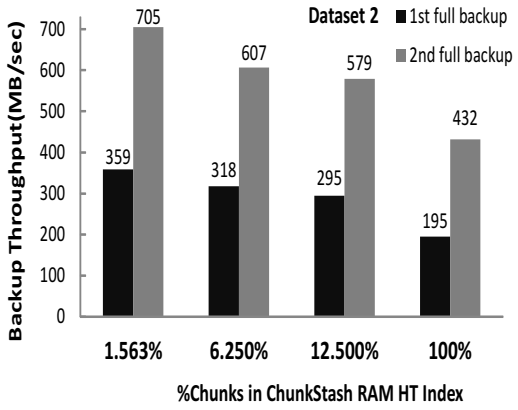


Figure 7.10: Dataset 2: Backup throughput (MB/sec) vs. fraction of chunks indexed in ChunkStash RAM HT index in first and second full backups.

inline deduplication system like ChunkStash. In our system, we use 8KB (average) chunk sizes and store them compressed on hard disk – with an average compression ratio of 2:1 (which we verified for our datasets), the space occupied by a data chunk on hard disk is about 4KB. With chunk metadata size of 64 bytes, the metadata portion is about $64/(4 * 1024) = 1/64$ fraction of the space occupied by chunk data on hard disk. With flash being currently 10 times more expensive per GB than hard disk, the cost of metadata storage on flash is about $10/64 = 16\%$ that of data storage on HDD. Hence, the overall increase in storage cost is about 1.16x.

Using a ballpark improvement of 25x in backup throughput for ChunkStash over disk based indexes for inline deduplication (taken from our evaluations reported in Section 7.3.5), *the gain in performance/dollar for ChunkStash over disk based indexes is about* $25/1.16 = 22x$. We believe that this justifies the additional capital investment in flash for inline deduplication systems that are hard-pressed to meet short backup window deadlines.

Moreover, the metadata storage cost on flash can be amortized across many backup datasets by storing a dataset's chunk metadata on hard disk and loading to flash just before the start of the backup process for the respective dataset – this reduces the flash memory investment in the system and makes the performance-cost economics even more compelling.

7.4 Related Work

We review related work that falls into two categories, namely, storage deduplication and key-value store on flash. The use of flash memory to speed up inline deduplication is a unique contribution of our work – their is no prior research that overlaps both of these areas. We also make new contributions in the design of ChunkStash, the chunk metadata store on flash which can be used as a key-value store for other applications as well.

7.4.1 Storage Deduplication

Zhu et al.'s work [134] is among the earliest research in the inline storage deduplication area and provides a nice description of the innovations in Data Domain's production storage deduplication system. They present two techniques that aim to reduce lookups on the disk-based chunk index. First, a bloom filter [39] is used to track the chunks seem by the system so that disk lookups are not made for non-existing chunks. Second, upon a chunk lookup miss in RAM, portions of the disk-based chunk index (corresponding to all chunks in the associated container) are prefetched to RAM. The first technique is effective for new data (e.g., first full backup) while the second technique is effective for little or moderately changed data (e.g., subsequent full backups). Their system provides perfect deduplication quality. Our work aims to reduce the penalty of index lookup misses in RAM that go to hard disk by orders of magnitude by designing a flash-based index for storing chunk metadata.

Lillibridge et al. [97] use the technique of sparse indexing to reduce the in-memory index size for chunks in the system at the cost of sacrificing deduplication quality. The system chunks the data into multiple megabyte segments, which are then lightly sampled (at random based on the chunk SHA-1 hash matching a pattern), and the samples are used to find a few segments seen in the recent past that share many chunks. Obtaining good deduplication quality depends on the chunk locality property of the dataset – whether duplicate chunks tend to appear again together with the same chunks. When little or no chunk locality is present, the authors recommend an approach based on *file similarity* [36] that achieves significantly better deduplication quality. In our work, the memory usage of ChunkStash can be reduced by indexing only a subset of the chunk

metadata on flash (using an uniform sampling strategy, which we found gives better deduplication quality than random sampling).

DEDE [50] is a decentralized deduplication system designed for SAN clustered file systems that supports a virtualization environment via a shared storage substrate. Each host maintains a write-log that contains the hashes of the blocks it has written. Periodically, each host queries and updates a shared index for the hashes in its own write-log to identify and reclaim storage for duplicate blocks. Unlike inline deduplication systems, the deduplication process is done out-of-band so as to minimize its impact on file system performance. In this paper, we focus on inline storage deduplication systems.

HYDRAstor [59] discusses architecture and implementation of a commercial secondary storage system, which is content addressable and implements a global data deduplication policy. Recently, a new file system, called HydraFS [127], has been designed for HYDRAstor. In order to reduce the disk accesses, HYDRAstor uses bloom filter [39] in RAM. In contrast, we aim to eliminate disk seek/access (and miss) overheads by using a flash-based chunk metadata store.

Deduplication systems differ in the granularity at which they detect duplicate data. EMC's Centera [60] uses file level duplication, LBFS [104] uses variable-sized data chunks obtained using Rabin fingerprinting, and Venti [117] uses individual fixed size disk blocks. Among content-dependent data chunking methods, Two-Threshold Two-Divisor (TTTD) [61] and bimodal chunking algorithm [89] produce variable-sized chunks.

7.4.2 Key-Value Store on Flash

Flash memory has received lots of recent interest as a stable storage media that can overcome the access bottlenecks of hard disks. Researchers have considered modifying existing applications to improve performance on flash as well as providing operating system support for inserting flash as another layer in the storage hierarchy. In this section, we briefly review work that is related to key-value store aspect of ChunkStash and point out its differentiating aspects.

MicroHash [133] designs a memory-constrained index structure for flash-based sensor devices with the goal of optimizing energy usage and minimizing memory footprint. This work does not target low latency operations as a design goal – in fact, a lookup operation

may need to follow chains of index pages on flash to locate a key, hence involving multiple flash reads.

FlashDB [108] is a self-tuning B^+-tree based index that dynamically adapts to the mix of reads and writes in the workload. Like MicroHash, this design also targets memory and energy constrained sensor network devices. Because a B^+-tree needs to maintain partially filled leaf-level buckets on flash, appending of new keys to these buckets involves random writes, which is not an efficient flash operation. Hence, an adaptive mechanism is also provided to switch between disk and log-based modes. The system leverages the fact that key values in sensor applications have a small range and that at any given time, a small number of these leaf-level buckets are active. Minimizing latency is not an explicit design goal.

The benefits of using flash in a log-like manner have been exploited in FlashLogging [48] for synchronous logging. This system uses multiple inexpensive USB drives and achieves performance comparable to flash SSDs but with much lower price. Flashlogging assumes sequential workloads.

Gordon [41] uses low power processors and flash memory to build fast power-efficient clusters for data-intensive applications. It uses a flash translation layer design tailored to data-intensive workloads. In contrast, ChunkStash builds a persistent key-value store using existing flash devices (and their FTLs) with throughput maximization as the main design goal.

FAWN [33] uses an array of embedded processors equipped with small amounts of flash to build a power-efficient cluster architecture for data-intensive computing. Like ChunkStash, FAWN also uses an in-memory hash table to index key-value pairs on flash. The differentiating aspects of ChunkStash include its adaptation for the specific server-class application of inline storage deduplication and in its use of a specialized in-memory hash table structure with cuckoo hashing to achieve high hash table load factors (while keeping lookup times bounded) and reduce RAM usage. Moreover, ChunkStash can reduce RAM usage significantly by indexing a small fraction of chunks per container with negligible loss in deduplication quality – this exploits the specific nature of storage deduplication application.

BufferHash [32] builds a content addressable memory (CAM) system using flash storage for networking applications like WAN optimizers. It buffers key-value pairs in

RAM, organized as a hash table, and flushes the hash table to flash when the buffer is full. Past copies of hash tables on flash are searched using a time series of Bloom filters maintained in RAM and searching keys on a given copy involve multiple flash reads. Moreover, the storage of key-value pairs in hash tables on flash wastes space on flash, since hash table load factors need to be well below 100% to keep lookup times bounded. In contrast, ChunkStash is designed to access any key using one flash read, leveraging cuckoo hashing and compact key signatures to minimize RAM usage of a customized in-memory hash table index.

7.5 Conclusion

We designed ChunkStash to be used as a high throughput persistent key-value storage layer for chunk metadata for inline storage deduplication systems. To this end, we incorporated flash aware data structures and algorithms into ChunkStash to get the maximum performance benefit from using SSDs. We used enterprise backup datasets to drive and evaluate the design of ChunkStash . Our evaluations on the metric of backup throughput (MB/sec) show that ChunkStash outperforms (i) a hard disk index based inline deduplication system by 4x-14x, and (ii) SSD index (hard disk replacement but flash unaware) based inline deduplication system by 2x-3x. Building on the base design, we also show that the RAM usage of ChunkStash can be reduced by 90-99% with only a marginal loss in deduplication quality.

Chapter 8

Conclusion and Future Works

Flash memory is an emerging storage technology which exhibits superior characteristics in terms of power consumption, weight, shock resistance, and read performance compared to the magnetic-disk based storage devices. However, relatively slow random write performance and early wear out are the two main drawbacks of the current generations of flash-based storage devices. This thesis is divided into two parts. In the first part of this thesis, we focus on improving the internal design and architectures of flash-based storage to deal with its physical limitations. On the other hand, in the second part, we focus on innovative uses of the flash-based storage devices.

First, we have proposed a new write cache management policy, named as *Large Block CLOCK algorithm (LB-CLOCK)*, for the non-volatile cache which helps to improve the random write performance. The new caching policy is especially optimized for the physical characteristics of the flash memory. It considers both temporal locality and block space utilization to make cache management decisions. In addition, it can dynamically adapt decisions based on the workload behaviors. Experimental results show that LB-CLOCK algorithm outperforms the existing best known algorithm, BPLRU [81], up to 70% for the update-intensive OLTP workloads.

Second, we have proposed a RAM-space efficient garbage collection metadata management scheme for the flash memory. We have used sampling-based algorithms to emulate existing garbage collection algorithms. The main idea is to store metadata information for only sampled K blocks in the RAM instead of maintaining metadata for all N blocks, where $K << N$. The RAM space requirement is proportional to the value

of K. Our approach is very easy to implement and it incurs less computation overhead. It greatly simplifies the garbage collection management in a flash-based storage controller. Our preliminary experimental results show that sampling-based approximation algorithms scale well for the large-capacity flash-based storage devices.

Third, we have proposed a flash-friendly hierarchical update processing technique, named as *deferred update methodology*, for the flash-based storages used in the database servers. Instead of directly performing the in-place updates, the changes due to update operations are stored as logs in two-level flash-based buffers. The first level buffer acts as a scratch area and helps to improve the locality of the second level buffer. When the second level buffers are full, updates logs are merged with the original data. Experimental results show that *deferred update methodology* improves average update processing time by approximately 40% compared to the state-of-the-art in-page logging (IPL) technique [90]. In addition, compared to IPL, *deferred update methodology* incurs approximately 40% fewer number of erase operations. It scales very well with the increase of data size and total number of update transactions.

Fourth, we have designed a bloom filter [38] data structure on flash-based storage, named as BloomFlash, that can tradeoff access times for a very slim RAM footprint. BloomFlash was carefully designed to be flash aware and work with the constraints of flash storage media. BloomFlash exploits two key design decisions: (i) buffering bit updates in RAM and applying them in bulk to flash (using two different flush-to-flash policies) that helps to reduce random writes to flash, and (ii) a *hierarchical* bloom filter design consisting of component bloom filters, stored one per flash page, that helps to localize reads and writes on flash. Evaluations on real-world data traces taken from representative bloom filter applications show that BloomFlash achieves bloom filter access times in the range of few tens of μsecs on currently available flash-based SSDs, thus allowing up to approximately 28,500 operations per sec.

Finally, we have designed ChunkStash to be used as a high throughput persistent key-value storage layer for chunk metadata for inline storage deduplication systems. ChunkStash uses one flash read per chunk lookup and works in concert with RAM prefetching strategies. It organizes chunk metadata in a log-structure on flash to exploit fast sequential writes. It uses an in-memory hash table to index them, with hash collisions resolved by a variant of cuckoo hashing. The in-memory hash table stores

compact key signatures instead of full chunk hashes so as to strike tradeoffs between RAM usage and false flash reads. Further, by indexing a small fraction of chunks per container, ChunkStash can reduce RAM usage significantly with negligible loss in deduplication quality. ChunkStash can index 6TB of unique (deduplicated) data using 45GB of flash that is well within the range of currently available flash-based storage capacities. Our evaluations on the metric of backup throughput (MB/sec) show that ChunkStash outperforms (i) a hard disk index based inline deduplication system by 4x-14x, and (ii) flash-based index (hard disk replacement but flash unaware) based inline deduplication system by 2x-3x.

8.1 Observations

This thesis makes the following observations.

- A nonvolatile write cache can improve random write performance of the flash memory. Clearly, a write cache will simplify the design of page mapping, wear leveling, and garbage collection algorithms. That is why it is very important to design an efficient write caching algorithm.
- Sampling-based metadata management algorithms can help to solve the capacity scaling problem of the flash memory. It would be very effective for the resource-constraint embedded devices.
- Batch updates are very helpful to improve the update-processing overhead and lifetime of the current generations flash-based storage devices.
- The following two types of operations are not efficient on flash media:
  - **Random Writes:** Small random writes effectively need to update data portions within pages. Since a (physical) flash page cannot be updated in place, a new (physical) page will need to be allocated and the unmodified portion of the data on the page needs to be relocated to the new page.
  - **Writes less than flash page size:** Since a page is the smallest unit of write on flash, writing an amount less than a page renders the rest of the (physical)

page wasted – any subsequent append to that partially written (logical) page will need copying of existing data and writing to a new (physical) page.

- The most efficient way to write flash is to simply use it as an append log (log-structured manner), where an append operation involves a flash page worth of data, typically 2KB or 4KB.

8.2 Future Works

We have listed some areas to extend the current solutions described in this thesis for further improvement in the future.

- We have designed a caching algorithm (as described in Chapter 3), which considers only write operations. However, an flash-based storage can also be equipped with a read cache. This read cache can speed up performance by servicing read operations from the read cache. Moreover, it helps to reduce the load in the flash data channel (i.e., bus) which is shared by both read and write operations. By servicing read operations from a cache, the flash data channel's bandwidth can be saved to destage write operations. Clearly, in this case, a read cache will help to improve the performance of write operations. In the future, we need to investigate the impact of a read cache on the performance of the write caching algorithms.

- We have assumed that the write cache is non-volatile. However, if we consider to apply our write caching algorithm in the client side, which usually uses volatile DRAM-based cache, we need to address the non-volatility issue.

- When designing the write cache algorithm, we assume that FTL uses either block-level or hybrid logical-to-physical page mapping scheme. However, if FTL uses pure page-level mapping, current scheme may not perform well. We need to reevaluate current caching algorithm for this new scenario.

- Our current sampling-based algorithms select samples randomly (as described in Chapter 4). This random selection policy will perform well for the uniformly distributed write workloads. However, for the skewed distributions, sampling-based algorithms will perform better by selecting samples from only the skewed

regions. We need to optimize current sampling-based algorithms for these types of workloads by keeping track of the skewed regions.

- Our current flash-based bloom filter design (as described in Chapter 6) exhibits superior performance for the lookup operations. However, for the insert operations, it suffers from slow random write performance of the flash memory. We need to explore some new optimizations to improve the insert operations performance.

- In the current design of the flash-based index storage for an inline storage deduplication system (as described in Chapter 7), per key-value pair we need six bytes of RAM space to keep track of its location in the flash memory. In this case, RAM space could become a bottleneck to support large capacity flash-based key-value stores. To remedy this problem, we need to design extremely low RAM footprint key-value stores.

References

[1] A Simulator for Various FTL Schemes. http://csl.cse.psu.edu/?q=node/322.

[2] Amazon. http://www.amazon.com.

[3] BerkeleyDB. http://www.oracle.com/technology/products/berkeley-db/index.html.

[4] BerkeleyDB for .NET. http://sourceforge.net/projects/libdb-dotnet/.

[5] BerkeleyDB Memory-only or Flash configurations. http://www.oracle.com/technology/documentation/berkeley-db/db/ref/program/ram.html.

[6] Diskmon for Windows. http://technet.microsoft.com/en-us/sysinternals/bb896646.aspx.

[7] Fusion-IO Drive Datasheet . http://www.fusionio.com/PDFs/Data_Sheet_ioDrive_2.pdf.

[8] GNU Wget. http://www.gnu.org/software/wget/.

[9] Intel Solid-State Drives and Caching. http://www.intel.com/design/flash/nand/index.htm.

[10] Iometer Project. http://www.iometer.org.

[11] MacBook Air: SSD Media Capacity. http://support.apple.com/kb/HT2734.

[12] MacBook Pro Features. http://www.apple.com/macbookpro/features.html.

[13] MurmurHash Fuction. http://en.wikipedia.org/wiki/MurmurHash.

[14] Releasing Flashcache. http://www.facebook.com/note.php?note_id=388112370932.

[15] Samsung SSD. http://www.samsung.com/global/business/semiconductor/productInfo.do?fmly_id=161&partnum=MCCOE64G5MPP.

[16] SATA-IO: Enabling the Future. http://www.serialata.org/3g.asp.

[17] SNIA IOTTA Repository: MSR Cambridge Block I/O Traces. http://iotta.snia.org/traces/list/BlockIO.

[18] SPC: Storage Performance Council. http://www.storageperformance.org/.

[19] The DiskSim Simulation Environment (V4.0). http://www.pdl.cmu.edu/DiskSim/.

[20] Two Technologies Compared: NOR vs. NAND (White Paper). http://www.data-io.com/pdf/NAND/MSystems/MSystems_NOR_vs_NAND.pdf.

[21] University of Massachusetts Amhesrst Storage Traces. http://traces.cs.umass.edu/index.php/Storage/Storage.

[22] White Paper: MySpace Uses Fusion Powered I/O to Drive Greener and Better Data Centers. http://www.fusionio.com/PDFs/myspace-case-study.pdf.

[23] Windows 7 to improve on Solid State Drive (SSD) performance. http://windows7center.com/windows-7-feature/windows-7-to-improve-on-solid-state-drive-ssd-performance/.

[24] Windows Performance Analysis Tools (xperf). http://msdn.microsoft.com/en-us/performance/cc825801.aspx.

[25] Xbox LIVE 1 vs 100 game. http://www.xbox.com/en-US/games/1/1v100/.

[26] Increasing Flash Solid State Disk Reliability. *Technical Report, SiliconSystems* (2005).

[27] Dell Puts SSD Hard Drives in Latitude D420 and ATG D620. http://www.notebookreview.com/default.asp?newsID=3658, April, 2007.

[28] Zeus-IOPS Solid State Drives Surge to 512GB in Standard 3.5“ Form Factor; Offer Unprecedented Performance for Enterprise Computing. `http://www.globenewswire.com/newsroom/news.html?d=117663`, April, 2007.

[29] Samsung Enterprise Solid State Drive. `http://www.samsung.com/global/business/semiconductor/products/flash/ssd/2008/down/Enterprise_SSD_datasheet_10-08.pdf`, October, 2008.

[30] Agrawal, D., Ganesan, D., Sitaraman, R., Diao, Y., and Singh, S. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. In *VLDB* (2009).

[31] Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J., Manasse, M., and Panigrahy, R. Design Tradeoffs for SSD Performance. In *USENIX* (2008).

[32] Anand, A., Muthukrishnan, C., Kappes, S., Akella, A., and Nath, S. Cheap and Large CAMs for High Performance Data-Intensive Networked Systems. In *NSDI* (2010).

[33] Andersen, D., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., and Vasudevan, V. FAWN: A Fast Array of Wimpy Nodes. In *SOSP* (2009).

[34] Azar, Y., Broder, A. Z., Karlin, A. R., and Upfal, E. Balanced Allocations. *SIAM Journal on Computing 29*, 1 (1994).

[35] Ban, A. Wear Leveling of Static Areas in Flash memory. *US Patent, 6732221, M-Systems* (2004).

[36] Bhagwat, D., Eshghi, K., Long, D., and Lillibridge, M. Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup. In *MASCOTS* (2009).

[37] Bouganim, L., r Jnsson, B., and Bonnet, P. uFLIP: Understanding Flash IO Patterns. In *CIDR* (2009).

[38] Broder, A., and Mitzenmacher, M. Network Applications of Bloom Filters: A Survey. In *Internet Mathematics* (2002).

[39] Broder, A., and Mitzenmacher, M. Network Applications of Bloom Filters: A Survey. In *Internet Mathematics* (2002).

[40] Caulfield, A., Grupp, L., and Swanson, S. Gordon: Using Flash Memory to Build Fast, Power-Efficient Clusters for Data-Intensive Applications . In *ASPLOS* (2009).

[41] Caulfield, A. M., Grupp, L. M., and Swanson, S. Gordon: Using Flash Memory to Build Fast, Power-Efficient Clusters for Data-Intensive Applications. In *ASPLOS* (2009).

[42] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A Distributed Storage System for Structured Data. In *OSDI* (2006).

[43] Chang, L.-P. On Efficient Wear Leveling for Large-Scale Flash-Memory Storage Systems. In *SAC* (2007).

[44] Chang, L.-P., and Kuo, T.-W. Efficient Management for Large-Scale Flash-Memory Storage Systems with Resource Conservation. *ACM Transactions of Storage 1*, 4.

[45] Chang, L.-P., and Kuo, T.-W. An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems. In *RTAS* (2002).

[46] Chang, Y., Hsieh, J., and Kuo, T. Endurance Enhancement of Flash-memory Storage Systems: An Efficient Static Wear Leveling Design. In *DAC* (2007).

[47] Chang, Y.-H., Hsieh, J.-W., and Kuo, T.-W. Endurance Enhancement of Flash-Memory Storage Systems: An Efficient Static Wear Leveling Design. In *DAC* (2007).

[48] Chen, S. FlashLogging: Exploiting Flash Devices for Synchronous Logging Performance. In *SIGMOD* (2009).

[49] Chiang, M., Lee, P., and Chang, R. Cleaning Policies in Mobile Computers Using Flash Memory. *Journal Systems and Software 48*, 3 (1999).

[50] Clements, A., Ahmad, I., Vilayannur, M., and Li, J. Decentralized Deduplication in SAN Cluster File Systems. In *USENIX* (2009).

[51] Corbato, F. A Paging Experiment with the Multics System. In *MIT Project MAC Report MAC-M-384* (1968).

[52] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. *Introduction to Algorithms, Second Edition*. McGraw-Hill Higher Education, 2002.

[53] Debnath, B., Krishnan, S., Xiao, W., Lilja, D. J., and Du, D. Sampling-based Metadata Management for Flash Storage. In *Laboratory for Advanced Research in Computing Technology and Compilers Technical Report No. ARCTiC 10-01* (January, 2010). `http://www.arctic.umn.edu/papers/sampling-meta-SSD.pdf`.

[54] Debnath, B., Mokbel, M. F., Lilja, D. J., and Du, D. Deferred Updates for Flash Based Storage. In *MSST* (2010).

[55] Debnath, B., Sengupta, S., and Li, J. ChunkStash: Speeding Up Inline Deduplication Using Flash Memory. In *USENIX* (2010).

[56] Debnath, B., Sengupta, S., and Li, J. FlashStore: High Throughput Persistent Key-Value Store. In *VLDB* (2010).

[57] Debnath, B., Sengupta, S., Li, J., Lilja, D. J., and Du, D. BloomFlash: Bloom Filter on Flash-based Storage. In *Submission for Review* (2010).

[58] Debnath, B., Subramnya, S., Lilja, D. J., and Du, D. LB-CLOCK: Large Block CLOCK Algorithm, An Improved Write-back Caching Scheme for the SSDs. In *MASCOTS* (2009).

[59] Dubnicki, C., Gryz, L., Heldt, L., Kaczmarczyk, M., Kilian, W., Strzelczak, P., Szczepkowski, J., Ungureanu, C., and Welnicki, M. HYDRAstor: a Scalable Secondary Storage. In *FAST* (2009).

[60] EMC Corporation. EMC Centera: Content Addresses Storage System, Data Sheet, April 2002.

[61] Eshghi, K. A framework for analyzing and improving content-based chunking algorithms. *HP Labs Technical Report HPL-2005-30 (R.1)* (2005).

[62] Fronda, F. Flash Solid State Disk Write Endurance in Database Environments . `http://www.bitmicro.com/press_resources_flash_ssd_db4.php`, 2008.

[63] Gal, E., and Toledo, S. Algorithms and Data Structures for Flash Memories. *ACM Computing Surveys 37*, 2 (2005).

[64] Gal, E., and Toledo, S. Algorithms and Data Structures for Flash Memories. In *ACM Computing Surveys* (2005), vol. 37.

[65] Gill, B., and Modha, D. WOW: Wise Ordering for Writes Combining Spatial and Temporal Locality in Non-Volatile Caches. In *FAST* (2005).

[66] Graefe, G. The Five-minute Rule Twenty Years Later, and How Flash Memory Changes the Rules. In *DAMON* (2007).

[67] Graefe, G. The Five-Minute Rule 20 Years Later: and How Flash Memory Changes the Rules. *ACM Queue 6*, 4 (2008).

[68] Guo, D., Wu, J., Chen, H., Yuan, Y., and Luo, X. The Dynamic Bloom Filters. *IEEE Transactions on Knowledge and Data Engineering 22*, 1 (January 2010).

[69] Gupta, A., Kim, Y., and Urgaonkar, B. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *ASPLOS* (2009).

[70] HachMan, M. New Samsung Notebook Replaces Hard Disk Drive with Flash. `http:/www.extremetech.com`, May, 2006.

[71] Hsieh, J.-W., Kuo, T.-W., and Chang, L.-P. Efficient Identification of Hot Data for Flash Memory Storage Systems. *ACM Transaction on Storage 2*, 1 (2006).

[72] Hutsell, W. Solid State Storage for the Enterprise. In *SNIA Tutorial* (2007).

[73] Jiang, S., Chen, F., and Zhang, X. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In *USENIX* (2005).

[74] Jiang, S., Ding, X., Chen, F., Tan, E., and Zhang, X. DULO: an Effective Buffer Cache Management Scheme to Exploit both Temporal and Spatial Locality. In *FAST* (2005).

[75] Jo, H., Kang, J., Park, S., Kim, J., and Lee, J. FAB: Flash-aware Buffer Management Policy for Portable Media Players . *IEEE Transactions on Consumer Electronics 22*, 2 (2006).

[76] Jung, D., Chae, Y., Jo, H., Kim, J., and Lee, J. A Group-based Wear-Leveling Algorithm for Large-Capacity Flash Memory Storage Systems. In *CASES* (2007).

[77] Kang, J.-U., Jo, H., Kim, J.-S., and Lee, J. A Superblock-based Flash Translation Layer for NAND Flash Memory. In *EMSOFT* (2006).

[78] Kawaguchi, A., Nishioka, S., and Motoda, H. A Flash-Memory Based File System. In *USENIX* (1995).

[79] Kawaguchi, A., Nishioka, S., and Motoda, H. A Flash-Memory Based File System. In *USENIX* (1995).

[80] Kim, G., Baek, S., Lee, H., Lee, H., and Joe, M. LGeDBMS: A Small DBMS for Embedded System with Flash Memory. In *VLDB* (2006).

[81] Kim, H., and Ahn, S. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage . In *FAST* (2008).

[82] Kim, H., and Lee, S. An Effective Flash Memory Manager for Reliable Flash Memory Space Management. *IEICE Transactions on Information and Systems E85-D*, 6 (2002).

[83] Kim, J., Kim, J. M., Noh, S. H., Min, S. L., and Cho, Y. A Space-Efficient Flash Translation Layer for Compact Flash Systems. *IEEE Transactions on Consumer Electronics 48*, 2 (2002).

[84] Kim, J., Kim, J. M., Noh, S. H., Min, S. L., and Cho, Y. A Space-efficient Flash Translation Layer for Compact Flash Systems. *IEEE Transactions on Consumer Electronics 48*, 2 (2002).

[85] Kim, J., Kim, J. M., Noh, S. H., Min, S. L., and Cho, Y. A Space-Efficient Flash Translation Layer for CompactFlash Systems. In *IEEE Transactions on Consumer Electronics* (2002), vol. 48.

[86] Kirsch, A., Mitzenmacher, M., and Wieder, U. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing 39*, 4 (2009).

[87] Knuth, D. E. *The Art of Computer Programming: Sorting and Searching (Volume 3)*. Addison-Wesley, Reading, MA, 1998.

[88] Koltsidas, I., and Viglas, S. Flashing Up the Storage Layer. In *VLDB* (2008).

[89] Kruus, E., Ungureanu, C., and Dubnicki, C. Bimodal Content Defined Chunking for Backup Streams. In *FAST* (2010).

[90] Lee, S., and Moon, B. Design of Flash-based DBMS: An In-page Logging Approach. In *SIGMOD* (2007).

[91] Lee, S., Moon, B., Park, C., Kim, J., and Kim, S. A case for Flash memory SSD in Enterprise Database Applications. In *SIGMOD* (2008).

[92] Lee, S., Park, D., Chung, T., Lee, D., Park, S., and Song, H. A Log Buffer-based Flash Translation Layer Using Fully-associative Sector Translation. *ACM Transactions on Embedded Computing Systems 6*, 3 (2007).

[93] Lee, S., Park, D., Chung, T., Lee, D., Park, S., and Song, H. A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation. *ACM Transactions on Embedded Computing Systems 6*, 3 (2007).

[94] Lee, S., Shin, D., Kim, Y.-J., and Kim, J. LAST: Locality-Aware Sector Translation for NAND Flash Memory-based Storage Systems. *SIGOPS Operating Systems Review 42*, 6 (2008).

[95] Leventhal, A. Flash Storage Today. *ACM Queue 6*, 4 (2008).

[96] Li, Y., Hey, B., Luo, Q., and Yi, K. Tree Indexing on Flash Disks. In *ICDE* (2009).

[97] Lillibridge, M., Eshghi, K., Bhagwat, D., Deolalikar, V., Trezise, G., and Camble, P. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *FAST* (2009).

[98] Lofgren, K., Norman, R., Thelin, G., and Gupta, A. Wear Leveling Techniques For flash EEPROM systems. *US Patent 6594183, Sandisk and Western Digital* (2003).

[99] Manber, U., and Wu, S. An algorithm for approximate membership checking with application to password security. *Information Processing Letters 50*, 4 (1994).

[100] Mearian, L. Google Chrome OS Will Not Support Hard-Disk Drives. `http://www.computerworld.com/s/article/9141191/`.

[101] Megiddo, N., and Modha, D. ARC: A Self-tuning, Low Overhead Replacement cache. In *FAST* (2003).

[102] Miller, P. SimpleTech Announces 512GB and 256GB 3.5-inch SSD Drives. `http://www.techmeme.com/070419/p25#a070419p25`, April 2007.

[103] Moshayedi, M., and Wilkison, P. Enterprise SSDs. *ACM Queue 6*, 4 (2008).

[104] Muthitacharoen, A., Chen, B., and Mazières, D. A low-bandwidth network file system. In *SOSP* (2001).

[105] Narayanan, D., Donnelly, A., and Rowstron, A. Write Off-Loading: Practical Power Management for Enterprise Storage. In *FAST* (2008).

[106] Nath, S., and Gibbons, P. Online Maintenance of Very Large Random Samples on Flash Storage. In *VLDB* (2008).

[107] Nath, S., and Kansal, A. FlashDB: Dynamic Self-tuning Database for NAND Flash. In *ISPN* (2007).

[108] Nath, S., and Kansal, A. FlashDB: Dynamic Self-tuning Database for NAND Flash. In *IPSN* (2007).

[109] National Institute of Standards and Technology, FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce, 1995.

[110] Pagh, R., and Rodler, F. F. Cuckoo hashing. *Journal of Algorithms 51*, 2 (May 2004), 122–144.

[111] Park, C., Cheon, W., Kang, J., Roh, K., Cho, W., and Kim, J. A Reconfigurable FTL (Flash Translation Layer) Architecture for NAND Flash-Based Applications. *ACM Transactions on Embedded Computing Systems 7*, 4 (2008).

[112] Park, D., Debnath, B., and Du, D. CFTL: A Convertible Flash Translation Layer with Consideration of Data Access Patterns. *Universit of Minnesota CS Technical Report*, TR 09-023 (2009).

[113] Park, S. K-Leveling: An Efficient Wear-Leveling Scheme for Flash Memory. In *UKC* (2005).

[114] Park, S., Jung, D., Kang, J., Kim, J., and Lee, J. CFLRU: A Replacement Algorithm for Flash Memory. In *CASES* (2006).

[115] Psounis, K., and Prabhakar, B. Efficient Randomized Web-Cache Replacement Schemes Using Samples From Past Eviction Times. *IEEE/ACM Transactions on Networking 10*, 4 (2002).

[116] Putze, F., Sanders, P., and Singler, J. Cache-, Hash-, and Space-Efficient Bloom Filters. *ACM Journal of Experimental Algorithmics 14* (2009).

[117] Quinlan, S., and Dorward, S. Venti: A New Approach to Archival Data Storage. In *FAST* (2002).

[118] Rabin, M. O. Fingerprinting by Random Polynomials. *Harvard University Technical Report TR-15-81* (1981).

[119] Reinsel, D., and Janukowicz, J. Datacenter SSDs : Solid Footing for Growth. `http://www.samsung.com/us/business/semiconductor/news/downloads/210290.pdf`, January, 2008.

[120] Ross, K. Modeling the Performance of Algorithms on Flash Memory Devices. In *DAMON* (2008).

[121] Shah, M., Harizopoulos, S., Wiener, J., and Graefe, G. Fast Scans and Joins using Flash Drives. In *DAMON* (2008).

[122] Shin, J., Xia, Z., Xu, N., Gao, R., Cai, X., Maeng, S., and Hsu, F. FTL Design Exploration in Reconfigurable High-Performance SSD for Server Applications. In *ICS* (2009).

[123] Shmidt, D. Technical Note: TrueFFS Wear Leveling Mechanism. *Technical Report, M-Systems* (2002).

[124] Silberschatz, A., Galvin, P., and Gagne, G. *Operating System Concepts*. John Wiley & Sons, Inc., 2004.

[125] Sykes, J. SSDs to Boost Data Center Performance. `http://download.micron.com/pdf/whitepapers/ssds_to_boost_data_center_performance.pdf`, July, 2008.

[126] Tsirogiannis, D., Harizopoulos, S., Shah, M., Wiener, J., and Graefe, G. Query Processing Techniques for Solid State Drives. In *SIGMOD* (2009).

[127] Ungureanu, C., Atkin, B., Aranya, A., Salil Gokhale, S. R., Calkowski, G., Dubnicki, C., and Bohra, A. HydraFS: A High-Throughput File System for the HYDRAstor Content-Addressable Storage System. In *FAST* (2010).

[128] Wells, S. Method for Wear Leveling in a Flash EEPROM Memory. *United States Patent*, 5341339 (1994).

[129] Woodhouse, D. JFFS: The Journalling Flash File System. *Proceedings of Ottawa Linux Symposium* (2001).

[130] Wu, C.-H., and Kuo, T.-W. An Adaptive Two-Level Management for the Flash Translation Layer in Embedded Systems. In *ICCAD* (2006).

[131] Yadava, H. *The Berkeley DB Book*. Apress, 2007.

[132] Zeinalipour-Yazti, D., Lin, S., Kalogeraki, V., Gunopulos, D., and Najjar, W. A. MicroHash: An Efficient Index Structure for Flash-Based Sensor Devices. In *FAST* (2005).

[133] Zeinalipour-yazti, D., Lin, S., Kalogeraki, V., Gunopulos, D., and Najjar, W. A. Microhash: An Efficient Index Structure for Flash-based Sensor Devices. In *FAST* (2005).

[134] Zhu, B., Li, K., and Patterson, H. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *FAST* (2008).

[135] Ziv, J., and Lempel, A. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory 23* (May 1977).

[136] Zukowski, M., Heman, S., and Boncz, P. Architecture-Conscious Hashing. In *DAMON* (2006).