Algorithms Designs and Implementations for Page Allocation in SSD Firmware and SSD Caching in Storage Systems

Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Shuang Liang, B.E., M.E.

Graduate Program in Computer Science and Engineering

The Ohio State University

2010

Thesis Committee:

Xiaodong Zhang, Adviser

Feng Qin
ABSTRACT

The emerging flash memory based SSD technology poses a new challenge for us to effectively use it in storage systems. Despite of its many advantages that promise a performance breakthrough for the next generation storage systems, the technical limitations of flash memory write operations call for innovative designs that can overcome the long latency of small random write pattern and the limited lifetime of flash memory cells.

Previous studies have attempted to address the write performance limitations by trading capacity at the firmware-level such that the write access pattern can be transformed to avoid small random write patterns. Although it may improve the write performance, it does not utilize the data access locality information available at system level to achieve optimal performance. In this thesis, we study techniques to improve the performance via system-level designs. We consider a major application scenario of flash SSDs – SSD based disk cache. Based on a new disk cache model for SSD based cache, we propose cache management algorithms which utilize the data access locality for write access pattern transformation with minimal cost to cache hit ratio. We implemented the proposed algorithm in an SSD based second-level disk cache system. The results show that the proposed algorithm can significantly improve the performance of existing cache management algorithms.

To maximize the lifetime of flash SSDs, we study techniques that can reduce the number of write operations, which consume the life-cycle of flash memory cells. Unlike previous approaches of flash memory life-cycle management, which balance the wear-level of flash memory
cells through data migration, we focus on reducing the internal data movement overhead introduced by flash memory SSD firmware to mask the long latency of small random write. We found that the internal data movement overhead can be managed effectively by capacity over-provisioning based on the write request working-set size, and we show a theoretical analysis to illustrate the relationship between the minimal over-provisioning capacity and the page allocation policy to achieve zero data movement overhead. Based on these findings, two SSD firmware page allocation algorithms are proposed to improve the capacity efficiency. Our trace-driven simulation using the Microsoft SSD extension of the well-known Disksim simulator shows that the proposed algorithm can significantly reduce the internal data movement overhead.
To My Family
ACKNOWLEDGMENTS

I am indebted to many great people.

First of all, I would like to thank my advisor, Prof. Xiaodong Zhang for his patience and thoughtful advice during my graduate study. It is Prof. Zhang’s help, motivation, support, and encouragement that has made me come to this step.

I would like to thank my research collaborators: Song Jiang, Ke Chen, Weikuan Yu, Ranjit Noronha and Prof. D.K. Panda. They have inspired and helped me in various aspects of my research. I am also very grateful to my lab-mates in The Ohio State University: Feng Chen, Xiaoning Ding, Lei Guo, Rubao Li, Enhua Tan, and Qi Zhang. Their friendship and support have brought me joyful and unforgettable memories in these years.

I would like to thank Prof. Feng Qin for his advice as a committee member so that I can complete this thesis.

I am also grateful to Peg Steele and Nikki Strader. Their support and friendship provided me an environment to be productive in research while I was working as a graduate administrative assistant.

Finally, I am indebted to my family for their unconditional love and support so that I can go through the toughest moments in these years.
VITA

1999 ................................................. B.E. in Computer Science, Huazhong University of Science and Technology
2002 ................................................. M.E. in Computer Science, Huazhong University of Science and Technology
2003-present ......................... Graduate student in Computer Science and Engineering, The Ohio State University

FIELDS OF STUDY

Major Field: Computer Science and Engineering

PUBLICATIONS

Research Publications


Shuang Liang, Weikuan Yu, and D.K. Panda. “High Performance Block I/O for Global File System (GFS) with InfiniBand RDMA”. In Proceedings of 35th International Conference for Parallel Processing (ICPP’06)


# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Table of Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>v</td>
</tr>
<tr>
<td>Vita</td>
<td>vi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>x</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xi</td>
</tr>
</tbody>
</table>

## Chapters:

1. Introduction
   - 1.1 Rationale and Challenge of SSD based Disk Caches
   - 1.2 Challenge of Maximizing the Lifetime of Flash SSD
   - 1.3 Contribution of the Thesis
     - 1.3.1 Flash SSD based Disk Cache
     - 1.3.2 Maximizing the Lifetime of Flash SSD
   - 1.4 Thesis Organization

2. Background and Related Work
   - 2.1 Flash Memory SSDs Basics
   - 2.2 Enhancing Flash SSDs via. FTL
     - 2.2.1 Performance Optimization
     - 2.2.2 Managing the Life-cycles of SSDs
   - 2.3 Related Work
     - 2.3.1 Caching Models and Algorithms
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Product Specification of SSDs</td>
<td>30</td>
</tr>
<tr>
<td>4.1 Workload Description</td>
<td>45</td>
</tr>
<tr>
<td>4.2 Parameters of DiskSim-SSD</td>
<td>46</td>
</tr>
<tr>
<td>4.3 Over-provisioning Capacity vs. Workload Storage Address Space Size</td>
<td>50</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>SSD endurance/ECC vs. progress geometry (source: Micron at Memcon’09)</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>An example SSD Internal Organization (source: Microsoft at Usenix’08)</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>The SSD based two level cache architecture.</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>Histogram of the evicted page address for MIN, LRU, and DULO replacement algorithms.</td>
<td>19</td>
</tr>
<tr>
<td>3.3</td>
<td>Selection of new active placement blocks.</td>
<td>22</td>
</tr>
<tr>
<td>3.4</td>
<td>Detailed Placement Algorithm</td>
<td>24</td>
</tr>
<tr>
<td>3.5</td>
<td>Zone based Admission Control Algorithm</td>
<td>27</td>
</tr>
<tr>
<td>3.6</td>
<td>Overall Performance of SSD based L2 cache</td>
<td>31</td>
</tr>
<tr>
<td>3.7</td>
<td>Detail Performance of Placement and Admission Control</td>
<td>32</td>
</tr>
<tr>
<td>3.8</td>
<td>Effects of Exclusive Cache</td>
<td>33</td>
</tr>
<tr>
<td>3.9</td>
<td>$\alpha$ and Placement Block Size on Miss Ratio</td>
<td>34</td>
</tr>
<tr>
<td>3.10</td>
<td>Comparison of High-end and Low-end SSDs</td>
<td>35</td>
</tr>
<tr>
<td>3.11</td>
<td>The architecture of the system implementation of SSD based L2 cache in Linux kernel.</td>
<td>36</td>
</tr>
<tr>
<td>3.12</td>
<td>Replaying the OLTP Traces on Real System for 3 Runs.</td>
<td>37</td>
</tr>
</tbody>
</table>
3.13 Evaluation Results of TPC-H on MySQL ........................................ 40

4.1 Sensitivity to Write Working-set based OPC ................................... 47

4.2 Sensitivity to Disk Capacity based OPC ......................................... 48

4.3 State Transitions of Physical Pages in SSDs .................................. 52

4.4 An Example for Write Forward Distance based Allocation. .............. 57

4.5 Comparison of Algorithms (a) ....................................................... 60

4.6 Comparison of Algorithms (b) ....................................................... 61
CHAPTER 1

Introduction

The exponential growth of information in modern society has been constantly demanding the advancement of storage technology to provide better performance and larger capacity for various data intensive applications. In the past 50 years, the magnetic disk as the major online system storage technology has been successfully keeping up with the capacity requirements, however, it has not been able to catch up with the performance scaling trend of data processing speed. This fact has lead to the well-known I/O bottleneck in computer systems, which is becoming increasingly severe. In face of the performance challenge, storage technologies other than magnetic disks are being actively investigated [17]. NAND flash memory based solid state devices (SSD) as one of the promising technologies is emerging. Compared with conventional magnetic storage devices, NAND SSDs can improve the storage access latency by one to two orders of magnitude. Although its capacity is still catching up with the magnetic disks, off-the-shelf NAND SSDs have already scaled to hundreds of gigabytes. This large capacity allows flash SSDs to be used in a wide range of applications, hence flash SSDs are becoming a major storage component in various general purpose systems. In the meantime, the trend also makes flash SSDs related system optimizations an interesting research topic for system performance improvement.
In this thesis, we consider flash SSD optimization techniques in two aspects: (a) cache management algorithms to improve the performance of SSD based cache for magnetic disks; (b) methods to extend the lifetime of flash SSDs.

1.1 Rationale and Challenge of SSD based Disk Caches

Although flash SSDs have scaled to hundreds of gigabytes, its capacity scalability is limited due to several technical limitations. At the same time, magnetic disks are still scaling at about 40\% per year [17, 48]. This implies that magnetic disk based storage system can provide a much larger capacity than flash SSD based systems. However, in a typical storage system, active working set in a certain period of time is much smaller than the total storage capacity. Using flash SSDs as disk caches, the storage system’s performance can be improved without compromising the large capacity that is enabled by the magnetic disks.

Using flash SSDs as disk caches can also provide a cost-effective storage solution for a given performance. This is because the cost of SSDs are one order of magnitude higher than magnetic disks. However, the performance advantage of flash SSDs is dependent on the workload access patterns. For some workloads, using magnetic disks results in better performance vs. cost ratio. Since the access pattern of workloads change dynamically, using flash SSDs as disk caches can maximize the cost-effectiveness of a storage system. At the same time, the benefit of using flash SSDs as disk caches cannot be revoked by DRAM based disk cache, although the performance vs. cost ratio of DRAM may be higher than flash memory. This is because the effectiveness of a disk cache is highly dependent on the cache algorithm’s ability to predict the access pattern of workloads. For unpredictable workloads, increasing the capacity is the most effective way to increase cache hit ratio. Given the $/GB of DRAM is much higher than flash memory, SSD based cache is irreplaceable in designing cost-effective storage systems.
A well-known issue of flash memory is the limitations of write operations. Specifically, the programming (write) speed is much slower than the read speed; furthermore, the small random write performance can be orders of magnitude slower than sequential writes. This leads to a caching model based on a non-uniform access latency to the caching device. However, existing cache management algorithms assume that the caching device is a fast device with very small access latencies, such that that caching loading cost is negligible. Since flash SSDs’ access latency is dependent on its access types (read or write) and access pattern (sequential or random), new cache algorithms must be proposed to consider the workload access pattern to the caching device.

1.2 Challenge of Maximizing the Lifetime of Flash SSD

Due to the technical limitations, each memory cell in a flash memory chip has only a limited number of programming cycles (also called endurance), such that each write operation reduces the remaining lifetime of the device. Moreover, as the memory cell density increases to catch up with the capacity scaling requirement, the number of life-cycles of memory cell may decrease and the bits used for error correction (ECC) purpose increase. This trend is shown in figure 1.1. This poses another challenge for flash memory based storage systems, which is to minimize the total number of write operations of the flash chips.

In flash SSDs, the actual number of write operations performed can be much higher than the number of writes requested by the user due to the internal data movement overhead of flash SSDs. Therefore, understanding such internal data movement overhead and designing algorithms to minimize such overhead is critical to extend the lifetime of flash SSDs.
1.3 Contribution of the Thesis

The contribution of this thesis includes two aspects: cache management algorithms for flash SSD based disk cache design and methods to maximize the lifetime of SSDs.

1.3.1 Flash SSD based Disk Cache

We made the following contributions for the design of SSD based disk cache: First, we have made a strong case to effectively optimize SSD cache performance at the buffer cache level, particularly addressing the weakness of SSD write operations. We have proposed a cache placement algorithm based on a special write access pattern, which is able to perform well on various SSDs and independently from existing flash translation layer (FTL) optimizations. We show that the additional overhead of our placement algorithm due to the inherent asymmetric performance issues of SSDs can be controlled to be minimal in comparison with a symmetric caching device. Second, we have proposed a space efficient admission control algorithm, which utilizes both spatial and temporal locality to minimize cache load to improve system performance. Third, we have implemented our proposed algorithms as a module for page cache management in the Linux kernel, and

Figure 1.1: SSD endurance/ECC vs. progress geometry (source: Micron at Memcon’09)
evaluated it with MySQL to demonstrate the performance benefits. To our best knowledge, this is the first research and development work to optimize SSD write performance at the OS level.

### 1.3.2 Maximizing the Lifetime of Flash SSD

In order to maximizing the lifetime of flash SSDs, we made the following contributions: 
(a) we show empirical results on the relationship between over-provisioned capacity of SSDs and the number of internal write operations for a variety of real server workloads. The results show that write-intensive workloads can be supported by SSDs with thin over-provisioning based on write request working-set size; 
(b) we provide a theoretical analysis on the page allocation constraints that the SSD firmware should follow to achieve zero internal data movement overhead for a given over-provisioning capacity; 
(c) we propose page allocation algorithms to reduce the internal system data movement overhead. Our algorithms consider both the online and offline case of workloads; 
(d) we implement our algorithms in the Disksim-SSD simulator and evaluate its performance for real workloads. The results show that the proposed page allocation algorithms can significantly improve the average response time and reduce the internal data movement overhead of existing algorithms.

### 1.4 Thesis Organization

The rest of this thesis is organized as follows. In Chapter 2, we discuss the technological characteristics of flash SSDs and related work for disk cache designs and flash memory based system. In Chapter 3, we discuss a new storage cache model for flash SSD based disk cache, and present the design, implementation and evaluation of a system based on this cache model. In Chapter 4, we discuss the factors that are correlated with lifetime of flash memory SSDs, and present the measurement results based on real workloads trace-driven simulation to understand the impacts of these factors in real systems. Then we provide a theoretical analysis, and discuss the
derived algorithms to maximize the lifetime of flash SSDs and its evaluation results. Finally, in Chapter 5 we conclude the thesis.
CHAPTER 2

Background and Related Work

2.1 Flash Memory SSDs Basics

Flash memory is an electronic storage device based on metal-oxide-semiconductor field-effect transistors (MOSFET), and was invented in the 80’s. There are two types of flash memory chips: NAND gate based or NOR gate based. Due to its better capacity scalability, NAND based flash has been widely used in various portable and embedded devices since late 80’s. As the density of NAND gate scales, currently NAND flash SSD can provide storage capacity of several hundred gigabytes.

NAND flash SSDs interface with the host system in a similar way as magnetic disk drives, by which the storage space is represented as a linear block address space. The Flash Translation Layer (FTL) implemented within the SSD firmware is responsible for translating logical addresses to physical addresses. There are two types of NAND flash SSDs: multi-level cell (MLC) and single-level cell (SLC). MLC stores multiple bits in a single memory cell, while SLC stores a single bit of information per cell. Therefore, SLC SSDs provide better performance and reliability than MLC SSDs, while MLC SSDs provide larger capacity and is more cost-efficient [25].

As flash SSDs are free of mechanical positioning, they have significant advantages for access latencies over magnetic disk drives. However, the access latencies of flash memory are not uniform.
Specifically, the write (programming) latency is much longer than the read latency; furthermore, the small random write performance can be orders of magnitude slower than sequential writes. This is because flash memory requires that a data block must be erased before it can be re-programmed. In other words, flash memory cell cannot be directly overwritten. The basic unit of an erasure operation is called an \textit{Erasure Block (EB)}, which is several tens to hundreds of times larger than the basic unit, \textit{page}, of read and write operations. This adversely affects the write response time in two ways: (i) the latency of erasure operation must be included in the write response time; (ii) to update a page, other pages within the same EB must be read before the erasure, and be written (together with the updated page) back after the erasure, leading to \textit{write amplification} for write operations. For small writes, the impact is severe due to the long erasure latency and high write amplification ratio. For example, the Samsung K9XXG08UXM SLC flash memory chip has 64 pages in a single EB. The erasure latency is about 1.5 millisecond, while the per page write (programming) latency is about 0.2 millisecond and the per page read latency is 0.025 millisecond [1]. In the worse case, a single page update triggers 63 page read, 1 block erasure, and 64 page write, which in total costs around 79 times more than a single page programming. Therefore, in the worse case, the write latency to the flash memory can be comparable or even larger than the typical disk access latency. In order to use flash SSDs as disk cache, the cache loading latency must be carefully managed to achieve the performance benefits.

A NAND flash memory SSD can be made of multiple chips in its internal organizations. As described in [1], an SSD can be made of multiple dies, and each die contains multiple planes as shown in Figure 2.1. Such organization may help increase the total bandwidth and capacity of flash SSDs.
2.2 Enhancing Flash SSDs via. FTL

FTL maps logical pages to physical pages on the flash memory chips. Such a mapping is essential to flash memory for both performance and reliability.

2.2.1 Performance Optimization

In order to improve the performance of small random writes, two techniques are commonly used in FTLs: log block based optimizations and log structured optimization, each with pros and cons [21, 1, 31].

In the log block approach, EBs are divided into two different kinds: data blocks and log blocks. Data blocks are used for data storage, and are mapped at block granularity. Log blocks are a small amount of blocks reserved by the FTL to store updated pages temporarily, and are mapped at page granularity. As the log blocks are filled up, the updates are merged with the data blocks. This approach improves small random write performance if multiple updates in the log block belong to
the same EB, reducing the write amplification rate. It requires a smaller mapping table because the data blocks are mapped at block granularity.

The log structured approach is akin to Log Structure File system (LFS) [45]. It manages the logical to physical address mapping in page granularity. Incoming updates can always be appended to the latest updated EB to form a sequential write access sequence as long as there is space available, while, at the same time, the FTL marks the original page as invalid as it is superseded by the updated page. When there is no EB available for appending, garbage collection is started to reclaim invalid pages. The garbage collection process serves the similar purpose of block merge as in the log block approach. However, since garbage collection has the option of reclaiming the EBs with the least number of valid pages (smallest write amplification rate) among all the blocks, it has a better chance to achieve high performance. Due to its finer granularity mapping, this method requires larger mapping table, which may offset its performance benefits. The mapping table is a random access data structure. Due to the cost constraints of SSDs, only a part of the mapping table can be buffered in the on-board RAM buffers, while others are accesses on-demand from the flash memory cells. Large mapping tables may degrade performance for the table accesses, particularly the mapping table updates for write operations.

Despite of these optimizations, the performance of flash SSDs are still access pattern dependent [6, 10]. several recent detailed measurements on a variety of flash memory devices including both high-end and low-end devices have shown this dependency. And it is suggested that small random writes should be conducted in a focused area (meaning with strong spatial locality) for better performance. This illustrates that optimizing the SSD write patterns to reduce randomness is important for the various technologies used in specific SSD implementations.
2.2.2 Managing the Life-cycles of SSDs

The EBs of flash memory have limited erasure cycles in the range of 100K to 1M. As some blocks may be updated more frequently than other blocks for a workload, worn-out blocks must be managed to avoid failures. This is achieved by remapping out-of-cycle blocks to new locations reserved for such purposes.

An ideal way to manage the life cycle of memory cells is to keep the remaining cycles of memory cells as uniform as possible, such that the lifetime of a SSD can be optimized. Wear-leveling algorithms are designed for this purpose. Most commonly used technique in these algorithms is to migrate hot data from pages with less remaining life-cycle to pages with more remaining cycles, such as the algorithm studied in [1]. However, this may introduce additional write operations due to the data migration. Moreover, in order to implement wear-leveling and the garbage collection algorithms, SSD designs have utilized the over-provisioned capacity, which can be quite significant in some cases to achieve high performance. This makes the capacity, lifetime and performance of flash SSDs closely related with each other. Understanding and management of all these aspects is critical to achieve the application requirements cost-effectively.

2.3 Related Work

Our work focuses on optimizing storage system with NAND flash SSDs. Specifically, we consider a new caching model for flash SSD caches which considers the loading cost of the caching device; and we propose methods to maximize the lifetime of flash SSDs via. capacity management and propose new page allocation algorithms in flash SSD firmware to reduce the internal data movement overhead and improve the capacity efficiency. In this section, we discuss the related work of our studies.
2.3.1 Caching Models and Algorithms

Since previous work on cache designs assume that the caching device is much faster than the device being cached, the cache loading cost is trivial and can be neglected in the design. To our best knowledge, no previous work investigated the loading cost for cache performance. We discuss the existing work on cache algorithms here.

Cache management algorithms provide the basis for virtual memory to work well, and is a fundamental issue for computer systems. A lot of studies have investigated this issue, and the locality principle is developed to guide cache algorithm designs [15]. The caching models studied in these early work follows the uniform block size and uniform fetch cost caching models. In this type of model, cache hit ratio is the single metric that determine the cache performance. It is found out that the optimal algorithm follows a simple rule: always replace the block farthest the future to be accessed in the cache. In online systems instead, algorithms based on temporal locality such as LRU is shown to be the most effective [4]. Various algorithms based on this temporal locality principle are developed to improve LRU by incorporating access frequency, specific access patterns, and application context into future access prediction [43, 28, 55, 12, 27, 39, 20].

In storage-aware caching models, the fetch cost is non-uniform. Although theoretical results provide a solution to this caching model, its complexity is $O(kn^2)$, where $k$ is the cache size and $n$ is the number of requests [13]. Due to its high complexity, both off-line and online heuristics have been proposed [53, 36]. These algorithms trade-off the hit ratio for fetch costs of data blocks by keeping high cost but less recently used blocks in cache while evicting low cost but more recently used blocks. For storage devices, such as disks (or flash memory), sequential access patterns are much more efficient than random accesses. Such trade-off is mainly between hit ratio and access patterns [26, 31, 16].
The general caching model allows both the block size and fetch cost to be variable. Unfortunately, it is known that determining the optimal performance is NP-hard, and several approximation algorithms are proposed [8].

Several studies on multi-level caching are related to our work. Multi-level caches can maintain either the inclusion or the exclusion property. It was shown that exclusive cache can improve the hit ratio of storage caches due to the enlarged effective total cache size [11, 51, 3].

2.3.2 Flash Related Systems

Early work on flash memory considered managing flash memory based on ideas similar to log-structured file systems (LFS)[52, 29] to mask the long write latency due to the erasure operation before a memory cell can be re-programmed. This approach is also applied in flash memory based embedded file systems. In [18], the authors surveyed in detail the the data structures, algorithms and systems for flash memory, and it points out that most flash-specific file system use the same overall principle of LFS to avoid in-place updates. The log structure is not only applicable for flash file systems, it can also be used at both application level or block driver level.

In the past years, several flash memory based file systems are designed to store data in embedded devices [14, 54, 24, 23]. In these work, a log structured file system organization [45] is applied to improve flash memory write performance. Similar optimization techniques are also used in SSD FTLs [1].

In order to optimize the write performance further, methods to optimize the on-board write buffers management are also proposed. These studies showed that choosing the pages whose corresponding EB has the most number of buffered pages or sequential pages as the write buffer replacement victim can improve the NAND flash write performance of the well-known LRU principle. However, these optimizations require that the buffered data demonstrates spatial locality to
fulfill the performance benefits. In summary, these optimizations can be considered as storage-aware cache replacement algorithms discussed above.

In [6] and [10], the authors evaluated various SSDs in order to characterize the performance of existing SSDs on the market, and they found that the SSDs write performance is highly dependent on workload spatial locality even with the FTL optimizations. These results show that further research on optimizing SSDs performance to reduce workload impact is needed.

Recently, several projects lead by industry have investigated the feasibility to integrate flash memory into operating systems. Microsoft ReadyBoost allows the system to use USB flash memory devices as part of the virtual memory [40]. Intel Turbo Memory (ITM) implements a system to use NAND flash as a non-volatile disk cache [38]. L2ARC is a recent Sun micro-systems implementation of an SSD based L2 cache in OpenSolaris [35]. Our research results can be applied in these systems.

Adopting SSDs in database systems has become an actively discussed topic in database community recently. In [33], the authors propose a in-page logging method to reduce the write overhead for DBMS using flash memory as storage. An evaluation of using SSDs in enterprise database applications was presented in [34]. In [32], the authors considered the data migration problem in a hybrid storage systems of both SSDs and hard disk drives. In their work, they assume hard disks outperform SSDs for writes regardless of the access pattern such that writes to SSDs should be avoided. In our design, we consider the performance impact that different access pattern makes on the write latency, which models the SSD characteristic more accurately.

Due to the strong research interest on NAND flash SSDs recently, Microsoft has developed an SSD simulator for flash SSD related research [44]. In addition, they also evaluated the cost-effectiveness of migrating to flash based storage in enterprise environments[42]. The results showed that SSD-disk-cache based server solutions can lower the total storage system cost, while
SSD replacement solution still needs SSD costs to be lowered by a factor of 3-3000 (depending on the workloads) to break even with a disk based solution. The energy saving benefits and reliability trade-off of using flash memory as disk caches is studied in [30]. Building flash memory based storage system for high performance computing system is presented in [9].
CHAPTER 3

Flash SSD based Disk Cache

3.1 An SSD-based Caching Model

In a typical general purpose computer system, an SSD based disk cache is a second level (L2) disk cache. As shown in Figure 3.1, the first level (L1) disk cache is the DRAM based buffer cache, implemented within an operating system. On each L1 disk cache miss, the SSD based L2 disk cache is queried. If the L2 disk cache has the data, the data page is loaded into L1; otherwise, the request is forwarded to the storage system for the data access. On each L1 eviction, the data block is considered for loading into the L2 disk cache. If the L2 cache decides to admit the evicted data page, it is placed at a location chosen by the L2 cache management algorithm. In particular, when the L2 cache is full, an existing data page on L2 cache must be replaced (or evicted) to admit the new data. If the replaced data page is clean, it can be silently dropped; otherwise, it is flushed to the storage before replacement.

In this model, the L1 cache is loaded on each miss. It admits all data pages and place them based on the replacement algorithm. In contrast, the L2 cache is loaded on L1 eviction. It admits data pages selectively based on its admission criteria. This multi-level caching model is consistent with real system architectures, where both the SSD and the storage are connected to the server system through system I/O controllers. Due to this organization, data flows between storage and
SSD must go through system memory, which hosts the L1 buffer cache. In order to well distribute the data among the levels of the cache hierarchy based on the locality strength of data blocks, it is natural to follow such a load-on-eviction design for the L2 cache.

Figure 3.1: The SSD based two level cache architecture.

The overall performance of this multi-level cache depends on multiple factors, including the hit ratios of both L1 and L2 caches, average cache miss penalty at each level, and \textit{L2 cache loading overhead}. The cache loading overhead is conventionally not considered by cache management algorithms, because most caches are load-on-miss, therefore the loading cost is equivalent to the miss penalty. However, in our multi-level caching model, L2 cache is loaded on evictions, where cache loading overhead is different from miss penalty.

We study L2 disk cache algorithms in this model. Although improving the performance of the existing L1 disk cache is part of the overall system performance goal, it has been studied extensively in previous works and is not the focus of this thesis.
3.2 Semi-Random Writes for SSD based Disk Cache

Although SSDs are designed as a storage device, when it is used as a caching device, it follows a different contract from the storage device contract. The storage contract requires two responsibilities from an SSD drive: (i) any write request to the device must be written to the specified (logical) device address; (ii) any read request to the device must fetch back the latest copy of the data written to the requested address. However, the cache contract is different: (i) the cache may serve the write request at any (logic) device address or even not to serve at all; (ii) it may serve the read request by responding “the data do not exist on the drive”. In other words, when the cache needs to load new data, it can choose any location to place the data to minimize the cache loading overhead and it can overwrite any existing data if needed. This difference enables optimization to transform the write accesses to the SSD based disk caches. However, most existing cache management algorithms assume fast access random access caching devices, therefore their replacement policy does not consider the impact of placement locations on performance. In contrast, an SSD based cache must consider this placement cost without compromising cache hit ratio to optimize cache performance.

The principle to maximize the cache hit ratio is to replace the data page that is least likely to be used in the near future, as the optimal off-line replacement algorithm MIN does. However, in practice, the data access sequence is not known in advance. Therefore, various heuristics are used to predict the likelihood of future accesses. For example, LRU (Least Recently Used) algorithm re-orders the pages on each access to the cache. However, these algorithms may generate small random write patterns to the caching devices for the cache loads. Figure 3.2 shows the histogram of victim block locations of an OLTP trace for several representative cache replacement algorithms: MIN is the optimal off-line algorithm; LRU is a widely used replacement heuristics in online systems; DULO is a recently proposed algorithm that considers sequentiality in replacement decisions.
to reduce disk access cost. The histogram shows that all the algorithms exhibit randomness in the victim page addresses.

Figure 3.2: Histogram of the evicted page address for MIN, LRU, and DULO replacement algorithms.

Semi-random write is a write access pattern that allows randomness in the access pattern without introducing write amplification. Semi-random write sequentially updates each page within an EB, while it allows multiple write sequences to be interleaved. For example, the following sequence in the format of (EB-id, offset within EB) represents a semi-random write sequence: (3, 0), (1, 0), (1, 1), (1, 2), (3, 1), (1, 3), . . . . This write access pattern allows randomness at EB-level, which provides a unique opportunity to design SSD-based cache placement algorithms. It can be shown that this access pattern does not amplify the number of writes for both kinds of FTL optimizations, if the firmware can detect the interleaved sequences and put them into different EBs on placement. In fact, such sequence detection algorithms have proposed [37]. The overhead for implementing such algorithm is small, especially when the number of interleaving streams is not huge.
3.3 Our SSD-based L2 Cache Design

In our caching model, two new aspects: placement and admission must be considered in the design. Placement considers where to store a data page in the cache; admission considers whether to admit a data page into the cache. Both aspects work to minimize the cache loading overhead without compromising the cache hit ratio to improve overall system performance.

The EB-aware Admission Controlled (EBAC) framework aims to improve the SSD based cache performance by optimizing cache placement and admission control through utilizing the unique performance characteristics of SSDs. More specifically, our placement algorithm utilizes the the semi-random write access pattern as discussed in Section 3.2, such that no small random writes exist in our cache load pattern, which prevents SSD performance degradation. The admission algorithm uses a zone based selection mechanism to load data blocks that are likely to be accessed in the future. Since it is able to utilize data access locality in a coarse management level, namely zones, it achieves high management space efficiency, such that it is practical to manage the data access locality for the entire storage address space. In Section 3.3.3, we also discuss the exclusive cache design and its effect on SSD based L2 cache.

3.3.1 SSD Aware Placement

The main idea of our placement algorithm is to generate only semi-random write accesses for the cache loads (so that cache loading overhead is small), while still maintaining high cache hit ratio.

In our placement algorithm, the logic address space of an SSD is divided into placement blocks, each consisting of a fixed number of consecutive logic pages. At any time, an evicted page from L1 cache is only allowed to be placed at certain placement blocks, and those blocks are called active
placement blocks (APB). For each active placement block, an offset in this block, called placement point, is maintained.

For simplicity, we first discuss the case where there is only one active placement block (APB). For an admitted page to the L2 cache, we simply place it to the placement point in this APB. If the placement point has reached the end of the APB, we choose a new placement block as the APB. Otherwise, we simply increment the placement point by one in the current APB (consequently, the next page load will be placed right after the current one). For instance, assume that the number of pages per placement block is four. The current placement point in the APB is 3. For a sequence of three placement requests (326, 27, 18), the pages 326 and 27 are placed at the addresses 3 and 4 within the current APB, respectively. For page 18, however, we need to select a new APB and put it to the placement point there.

From the SSD drive’s point of view, an L2 cache loading request (once being decided where to place) is nothing more than a write access. It is clear from the above description that cache loads are written to the SSD sequentially in active placement blocks. Therefore, this method incurs considerably lower loading overhead, since, for SSD drives, sequential writes are significantly more efficient than random writes. The disadvantage is the slightly higher L2 cache miss ratio. This is because when we select a new APB, the existing pages within the new APB need to be evicted, and the eviction may cause additional L2 cache misses, compared to the method of simply replacing the page least likely to be accessed soon. Fortunately, as to be demonstrated by the experiments, the benefit of lower loading overhead far exceeds this disadvantage, see Section 3.4 for details.
Now, we describe how to choose a new active placement block (APB) when the current one is full. New APB selection is similar to traditional cache replacement algorithms, because selecting an APB is essentially selecting several pages together for replacement. Our algorithm assumes that there exists a ranking of all the pages in the L2 cache, such that the higher rank value a page has, the more likely it is accessed in near future. Note that most cache replacement heuristics, such as the well known LRU algorithm and its variants, approximate this property. For example, in LRU, all the pages are ranked by their access time stamps. A special case in our algorithm is the rank of invalid pages in the L2 cache. Invalid pages are those superseded by later updates from L1 cache. We assign 0, the lowest rank value, to invalid pages, since they are not to be accessed ever. Let the rank of a placement block be the total rank values of the pages in this block. Then the placement block with the lowest rank value is our choice of the new APB. Intuitively, this algorithm selects the placement block consisting of pages that (as an aggregate) are the least likely to be accessed in the near future. However, it is still possible that the selected APB contains some pages with high rank values. Replacement of these high rank pages may increase the miss ratio. In order to control this, we introduce a parameter $\alpha$, where $0 \leq \alpha < 1$. When a new APB is selected, we will keep the existing pages (in this APB) that have the top $\alpha$ percentile rank values. For example,
assume we have 5 placement blocks, each consisting of 4 pages, and 17 valid pages are resident in
the L2 cache. As shown in Figure 3.3 (a), the resident pages are marked by their ranks. Suppose
that $\alpha$ is 50%. According to our algorithm, any page having a rank value between 8 and 17 should
be kept, since we must keep the top 50% of all the pages. Therefore, the first placement block is
selected as the new APB; the page with rank 11 is reloaded into this APB (because it is the only
page that has a rank value between 8 and 17 within this placement block), and the placement point
becomes 2, see Figure 3.3 (b). More details are described in Algorithm 1 of Figure 4.5.

**Remark.** As described above, when we choose a new active placement block, we need to
reload some high rank pages already resided on this APB, and evict others. The parameter $\alpha$
controls how many pages may need to be reloaded. Indeed, suppose an APB contains $m$ pages,
then it is easy to show that at most $\alpha m$ high rank pages need to be reloaded. Note that the more
high rank pages are reloaded to the new APB, the less space APB has for future cache loading, and
the sooner another new APB is needed. On the other hand, however, the more high rank pages are
reloaded, the less negative impact is caused to the L2 cache hit ratio. As such, it seems desirable to
seek a good trade-off. Fortunately, the above analysis tends to consider worst cases; in practice, the
overall L2 cache performance is not very sensitive to the choice of $\alpha$. See Section 3.4 for further
discussions.

As the final issue in the algorithm, two active placement blocks (APB) are used – one is for the
pages that tend to be updated later, and another is for the pages that are likely to be read-only. Note
that, when a newly admitted page is an update of an existing page in the L2 cache, the new copy
is loaded to a new location (the placement point), while the original copy is marked as *invalid*.
Since invalid pages waste L2 cache space, we intend to “cluster” invalid pages together, so that,
according to our APB selection algorithm, the space occupied by invalid pages can be reclaimed
sooner. To this end, we use two separate APBs, when an admitted page is dirty, we put it into one
**Algorithm 1** placement algorithm

**Input**: $x$ - the data page to be loaded;  
**Input**: timestamp - the current timestamp;  
/* executed on each data page load */

**Procedure** placement($x$, timestamp)  
if $x$ is clean and an copy exists  
    $x_{existing}.timestamp \leftarrow$ timestamp;  
    return ;  
if $x$ is clean and no copy exists  
    load $x$ to the APB for clean pages;  
if $x$ is dirty and an old copy exists  
    invalidate $x_{existing}$;  
    load $x$ to the APB for dirty pages;  
if $x$ is dirty and no old copy exists  
    load $x$ to the APB for dirty pages;  
$x.timestamp \leftarrow$ timestamp;  
if the current active placement block is not full  
    increment placement point on APB by one;  
else  
    choose the placement block with the lowest rank value as new APB;  
    for all page $p$ on the new APB do  
        if $p$ is dirty and $p.rank$ is NOT among top $\alpha$ percentile rank values  
            flush $p$ to the storage  
        if $p.rank$ is among top $\alpha$ percentile rank values  
            reload $p$ to the new APB;  
    return

Figure 3.4: Detailed Placement Algorithm
APB (in the hope that they may be updated again); otherwise, it is clean, and we put it into another APB (in the hope that they are read-only pages). The rational here is that, as observed in practice, recently updated data (namely, dirty pages) are more likely to be updated again [19]. We should point out that, since there are two APBs and the write accesses are sequential within each APB, the write pattern is semi-random write. Therefore, as discussed in Section 3.2, since semi-random writes are almost as efficient as sequential writes, utilizing two APBs won’t increase the cache loading cost much.

### 3.3.2 Zone based Admission

The performance of SSD L2 cache can be further improved by admission control to allow cache loads only for data pages that are likely to be requested in future, such that the number of unnecessary cache loads can be minimized (recall that SSD cache load is relatively expensive). The idea is to use past access history of a data page to speculate its future access probability. The challenge here is that we have to keep meta-data for all data pages in the storage address space (rather than in the L2 cache address space only). Thus, it requires a highly space-efficient design. We propose a zone based admission control algorithm to achieve this goal.

Note that existing cache replacement algorithms cannot be directly applied for admission control. This is because these algorithms decide the future access probability based on the temporal property of data pages, such as recency. Since the data pages under admission consideration are just evicted from L1 cache, they are among the most recently accessed data. Therefore, by those algorithms, all evicted pages should be admitted, leading to no admission control essentially.

We use both spatial locality and recent access history to set admission criteria. In our algorithm, the storage space is divided into units called *zones*. Each zone is a continuous range of the storage address space, and may contain multiple data pages. Typically, a zone is much larger than a
placement block or a data page. An example of the sizes of these units is as follows: suppose that a data page is 4K and the placement block is 128K, the zone size can be set as 1M. Therefore, a zone contains 256 data blocks or 8 placement blocks. Each zone maintains a counter and two vectors. The counter records the total number of L1 cache evictions of the data pages in this zone. The hit vector is a shift bit vector recording the most recent L2 cache hit/miss history of the data pages in the zone. For example, if the most recent access of a zone is (hit, miss, miss, hit) on L2 cache, then the hit vector stores (1001). The access vector is a bit vector recording data pages accessed recently. For example, if data pages 0 and 4 are accessed in this zone, the access vector stores (10001). The space overhead for these history information is small. For example, for a 1TB storage space, if the hit vector is 64 bit, then the hit vector takes 8M. The access vector uses 1 bit per data page, then the access vector takes 32M. The 8 bit counter needs 1M. In total, it costs only 0.0041% of the total storage space. And this space requirement can be well accommodated by the memory system of a typical 1TB data server.

Based on these information, each zone is either hot or cold. A zone is hot under conditions: (i) if the eviction count from this zone is less than a threshold; or (ii) the 1’s in the hit vector is greater than a lower bound; or (iii) its access vector has more 1’s than a hot zone. Condition (i) allows a small percentage of data pages in each zone to be admitted to test its hotness; (ii) allows data pages to be admitted from zones with high L2 cache hit history; (iii) allows a zone previously marked as cold to be re-tested if it has more accesses than an existing hot zone. Note that, in case (iii), we simply reduce the eviction counter of this zone to be slightly lower than the admission threshold, which gives another opportunity for this zone to be tested. The zones that do not pass the above conditions are marked as cold. Our admission control algorithm only admits data pages from a hot zone. See the Algorithm 2 of Figure 4.6 for details.
**Algorithm 2** Zone based admission

**Input:** x - the data page requested; z - the zone x belongs.
/* executed on each request for data page. */

**Procedure** record_history(x, z)
if x is in the L2 cache
hit ← 1
else
hit ← 0

z.hit_vector ← (z.hit_vector << 1) + hit
i ← block offset of x in z
z.access_vector[i] ← 1
return

**Input:** x - the data block evicted; z - the zone x belongs.
**Output:** 1 - hot zone; 0 - cold zone.
/* executed on each eviction of L1 cache. */

**Procedure** mark_status(x, z)

z.count++
if (z.count < Threshold.count)
return 1

if (ones(z.hit_vector) ≥ Threshold.hits_low)
sample_z ← z;
return 1

if ones(z.access_vector) ≥ ones(sample_z.access_vector)
z.count ← Threshold.count * β, where 0 ≤ β ≤ 1 is a pre-specified parameter;
z.access_vector ← 0
return 1;

return 0;

---

Figure 3.5: Zone based Admission Control Algorithm
The zone based admission control is unique in several aspects. First, given a data page, the algorithm uses the zone meta-data to infer its future access likelihood. This method effectively utilizes the spatial locality to infer temporal locality for access prediction, which enables our algorithm to realize admission control with high management space efficiency. Second, for each zone, the algorithm uses its hit vector as a sliding window to maintain its recent cache hit/miss information. This (small) sliding window effectively captures the intuition that recent history is important for prediction. Finally, the algorithm has a mechanism to re-test the temperature of a zone by considering zone access frequency. This enables the algorithm to detect cases where a cold zone becomes hot.

3.3.3 On Exclusive Cache

For L2 caches, *exclusiveness* refers to the property that L2 cache does not contain any data resident in L1 cache. It has been a commonly used technique to improve performance [51, 11, 3]. This is because a recently requested page by L1 cache is not to be requested again before it is evicted by L1 cache. According to this rule, as data pages in L2 cache are requested by L1 cache, they should be evicted from the L2 cache to save space for other data pages. Such a design is based on the assumption that the reloading overhead is small, so that the performance gain from cache hit ratio improvement is greater than the performance loss from potential reloads for the same data page in the future. In the case of SSD based cache, those assumptions may not hold true for two reasons. First, compared with DRAM based L1 cache, the SSD based L2 cache can be tens of times larger considering their price differences. Therefore, maintaining exclusiveness can only enlarge the effective total cache size of both L1 and L2 by a small percentage. Second, reloading requested pages can significantly increase the number of cache loads to the SSD based cache, especially when the workload hit ratio is high, which can offset the small benefit gain from
exclusive caching. Based on above considerations, in our L2 cache design, we do not follow this exclusiveness rule.

3.4 Experimental Studies

To study the performance of SSD based L2 cache, we conduct extensive experiments using both trace driven simulation and system implementation to evaluate our designs. The simulation approach can provide detailed statistics to analyze the impact of each system components. And the system implementation demonstrates the realistic performance effects in real world. In our evaluation, we studied the performance of existing symmetric cache management algorithms, our proposed placement and admission algorithms. We also compare the system performance between a disk-only system and an SSD-based L2 cache system. Our evaluation uses industry standard benchmarks such as TPC-H benchmarks [49] and OLTP traces collected by SPC from two large financial institutions [47]. The TPC-H benchmark is a decision support workload containing almost only of data reads. The OLTP workloads are on-line transaction processing workloads. The first OLTP trace is dominated by write requests; while the second trace contains both reads and writes. We use two different SSD devices in our evaluations. The Intel X25-E is a relatively high-end SSD device. The Patriot PE32GS25SSDR 2.5” 32GB drive is a relatively low-end SSD device. The public performance specifications are listed in Table 3.1.

Our host system is a Dell PowerEdge SC440 server equipped with 2.8MHZ dual Pentium D processors and 2GB main memory. The disk drives are 80GB and 160GB Western Digital Caviar SATA drives [50], where the 80GB drive is hosting the system image, and the 160GB drive is used for testing.
Table 3.1: Product Specification of SSDs

<table>
<thead>
<tr>
<th></th>
<th>Patriot</th>
<th>Intel X25-E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential Read:</td>
<td>up to 175MB/s</td>
<td>up to 250MB/s</td>
</tr>
<tr>
<td>Sequential Write:</td>
<td>up to 100MB/s</td>
<td>up to 170MB/s</td>
</tr>
<tr>
<td>4K Read</td>
<td>N/A</td>
<td>35000 IOPS</td>
</tr>
<tr>
<td>4K Write</td>
<td>N/A</td>
<td>3300 IOPS</td>
</tr>
<tr>
<td>Interface:</td>
<td>SATA I/II</td>
<td>SATA II</td>
</tr>
</tbody>
</table>

3.4.1 Simulation

We implemented a multi-level cache simulator using our caching model as shown in Figure 3.1. The L1 cache is a DRAM based page cache, which is implemented with existing cache replacement algorithms such as LRU. The L2 cache is an SSD-based page cache, which is implemented with the proposed EBAC algorithm. For the purpose of comparison, our simulator also supports two other schemes. The SSD unaware scheme, noted as “UA”, uses the SSD as a random access caching device as in existing systems. The exclusive caching scheme, noted as “Ex”, follows the exclusive caching rule. We use real SSD and disk drives in our simulator, therefore the performance results include the effects of internal SSD FTL and disk firmware optimizations. On each L1 miss and L2 load, we use the O\_DIRECT flag to access disk and SSD for data transfers.

The OLTP traces used are from Storage Performance Council (SPC). We name the two trace files collected from two large financial institutions as OLTP1 and OLTP2. We also collected the I/O traces from running the TPC-H benchmark with the 21 queries (excluding query 18 due to its long running time) on MySQL/InnoDB using the Linux blktrace tool. In order to involve all the three levels of the memory hierarchy: L1, L2 and disks in our experiments, we set the size of higher level as a percentage of lower level and have the hard disk hold the entire dataset. For all
the workloads, the L1:L2 ratio is 1:16. For all cases, the L1 cache size is 1.6% of the total working set.

Overall Performance

Figure 3.6: Overall Performance of SSD based L2 cache

Figure 3.6 shows the overall trace replay time for the two OLTP traces in three different cases: (1) “disk” is the case where no SSD based L2 cache is used; (2) “UA” uses existing SSD-unaware algorithms; (3) “EBAC” uses our proposed design. The results show that using SSD-based L2 cache in the system can significantly improve the performance of a disk-only system when the L2 cache management algorithm is designed to accommodate the performance characteristics of the SSDs. For example, the existing SSD unaware algorithms can only improve the performance of OLTP1 by less than 4.7%, while the EBAC algorithm can improve the performance by 61.3%. For OLTP2, the existing SSD unaware algorithm improves the performance by 53.6%, while EBAC improves by 76.2%. Finally, for the TPC-H trace, the existing SSD unaware algorithm improves by 44.5%, while EBAC improves the system performance by 55.6%. The results show that EBAC consistently outperforms the existing cache algorithms.
By examining the detailed statistics of the results, it shows that the performance benefit of EBAC comes from its ability to trade cache hit ratio for a more friendly write access pattern. For example, in terms of total number of disk accesses, EBAC is 2.6% higher than “UA” for OLTP1, while the total number of cache loads of EBAC is only 62.7% of “UA”. Overall, the performance improvement using EBAC is 56.6% higher than “UA”. This demonstrates the importance of reducing cache load and using SSD-friendly loading sequence rather than focusing only on reducing disk accesses.

The Effects of Placement and Admission Control

Figure 3.7: Detail Performance of Placement and Admission Control

Figure 3.7 shows the decomposed performance of EBAC by configuring placement and admission control individually. In the figure, “PL” represents placement, and “Zone” represents zone based admission control. The results show that both components contribute to the overall performance advantage over “UA”, and each alone may significantly improve the performance. For example, placement alone improves the performance by 52.6%, 41.9% and 18.9% for the three workloads, respectively; admission control alone improves the performance by 46.2%, 22.3%, and
0.66%, respectively. This again confirms the importance of placement and admission control to SSD based cache performance.

The Effects of Exclusive Cache

![Figure 3.8: Effects of Exclusive Cache](image)

Figure 3.8 shows the effects of using exclusive cache for SSD based L2 cache. The results show that exclusive cache causes additional cache loads and can degrade performance. For example, for “UA”, the exclusive cache scheme degrades the performance by 9.4%, 54.0%, and 39.6% for OLTP1, OLTP2, and TCP-H, respectively. Correspondingly, the additional cache loads are 3.8%, 98% and 145.8%. For EBAC, with the exclusive cache design, it is even harder to maintain data blocks of similar temporal locality within the same placement block due to invalidation of data blocks upon requests. Therefore, not only additional cache loads are involved, the cache miss ratio also increases, leading to performance degradation.

α and Placement Block Size on Miss Ratio

We discussed in Section 3.3.1 that in our placement algorithm, the overhead due to asymmetric read/write performance characteristics of SSDs can be influenced by \( \alpha \). However, it remains to be studied that how \( \alpha \) affects the hit ratio in reality. In addition, although the placement block size
should be set as large as the EB size of the SSD, the actual value that maximizes the performance still needs to be evaluated. This is because due to FTL optimization, the semi-random write performance for smaller placement block may be close to larger placement block, yet smaller placement block may lead to a higher hit ratio. For these reasons, we measure the miss ratio changes as $\alpha$ and the placement block size changes. As shown in Figure 3.9, when $\alpha$ decreases, the miss ratio change is small. For example, for OLTP1, the miss ratio starts from 48.15\% and gradually increases to 51.5\%. Similarly, for the other two traces, the miss ratio changes are within 3.7\%. Comparing with the miss ratios under “UA” schemes: 48.27\%, 29.09\% and 40.08\%, the differences are very small. The results demonstrate that for real workloads, the miss ratio loss due to the placement algorithm is low.

**Comparing High-end and Low-end SSDs**

The previous results are all based on the Patriot SSD drive. Now, we compare the performance of using both the Patriot and Intel X25-E SSD drive. The Intel X25-E drive is a high-end SLC SSD drive. It uses *over provisioning* and on-drive buffers to improve the SSD write performance. Over provisioning reserves a significant portion of flash memory cells to provide space for writing, such that the block merge or the garbage collection process can be delayed for better opportunities.
In the case of log structured FTL, the over provisioned space can also help decrease the average number of valid pages in each EB that needs to be reclaimed, such that the write amplification rate can be reduced. Due to these optimizations as well as its high performance SLC flash memory chip, Intel SSD is much more expensive than the MLC Patriot SSD\(^1\).

In Figure 3.10, we show the results of both “UA” and EBAC for these two devices. It shows using EBAC, the low-end SSD performs closely to the “UA” scheme of the high-end SSD. Although for OLTP1, the Patriot SSD is 16.6\% slower than the Intel SSD, for both OLTP2 and TPC-H, the Patriot SSD is 7.2\% and 2.6\% better than the Intel SSD. Using EBAC can also improve the performance of the high-end SSD. For example, EBAC improves the performance of OLTP2 by 17.8\%. The reason for such improvement is similar to the low-end SSD case. For OLTP2, the total disk accesses of EBAC is 1.8\% more and its cache loads are 9.4\% more than “UA”, however, its cache loads patterns are more SSD performance friendly, resulting in the 17.8\% overall performance improvement.

![Comparison of High-end and Low-end SSDs](image.png)

Figure 3.10: Comparison of High-end and Low-end SSDs

\(^1\)Accordingly to the current market price, a 32GB Intel X25-e drive is around 5x the price of a 32GB Patriot drive.
3.4.2 System Implementation

The EBAC framework implementation for SSD based L2 cache contains two components: the eviction engine and the L2 cache manager, as shown in Figure 3.11.

![Figure 3.11: The architecture of the system implementation of SSD based L2 cache in Linux kernel.](image)

The eviction engine is responsible to forward each page that the kernel page cache manager evicts to the L2 cache manager. The L2 cache manager has multiple functions: (1) decides whether to load an evicted page into the SSD based cache; (2) place each page admitted into the SSD cache; (3) serve each L1 miss if it is resident in the L2 cache or forward it to the disk driver if not; (4) flush dirty blocks in L2 cache to the disk periodically. The L2 cache manager maintains the L2 cache data structures, including the mapping table for cached data blocks; the data structure for the placement and admission algorithms. These data structures are memory resident for operations. And on power-off or system reboot, it is flushed into persistent storage. On startup, it is reconstructed from the persistent storage such that SSD cached data can be put into use immediately when the system is back online.
Our implementation allows the SSD to serve as a L2 cache for specified devices. A management interface is provided to register storage devices that intend to cache its data with the L2 cache manager. Our implementation uses Linux Kernel 2.6.18. The kernel eviction engine is implemented within the kernel page frame reclaim code. The L2 cache manager is implemented as a kernel module. The total lines of code is around 1700. Since it is implemented at operating system level, it enables us to evaluate its performance for both raw device trace replaying and DBMS transaction processing. We evaluate the raw device replay by sending requests to the L2 cached device using Linux kernel supported asynchronous I/O. The transaction processing is evaluated using MySQL/InnoDB. We use raw partitions for the shared table space of the InnoDB storage engine. And by registering the raw partition with the L2 cache manager, it is automatically cached by the SSD cache without any change to the DBMS.

(a) OLTP1  
(b) OLTP2

Figure 3.12: Replaying the OLTP Traces on Real System for 3 Runs.
OLTP Traces

To confirm the benefits of SSD based L2 cache on real systems, we have replayed the OLTP traces on our system. In order to evaluate each design in a comparable system environment, we run each trace multiple times. For each trace, we start from an empty L1 kernel buffer cache and an empty L2 SSD cache. Beginning the second run, each run just follows the last run, such that the following runs can benefit from a warm L1 and L2 cache. Since the working set sizes of these traces are around 4GB and 2GB, respectively, we only set a 500M SSD partition for the L2 cache. In addition, we also reserve a large amount of DRAM to put aside, such that the L1 buffer cache can be limited to be much smaller than the L2 cache. However, since the system interface only allows us to reserve around 1.6GB DRAM and the memory is dynamically allocated for the L1 buffer cache, we expect the L1:L2 cache size ratio to be larger than those in practical systems. This situation is due to the relatively small working set size of the OLTP traces. However, this actually requires a better L2 cache design to see significant performance improvement, because a larger L1 cache leads to less evictions and misses, which may diminish the effects of L2 cache. Therefore, we consider it a meaningful configuration for our evaluation.

As shown in Figure 3.12, when replaying the trace with EBAC supported L2 cache, the performance can be significantly improved in comparison with both “disk” (no L2 cache) and “UA”. For example, for the last run, EBAC improves the “disk” performance of OLTP1 by 64.0%; similarly, OLTP2 is improved by 81.7%. In comparison with “UA”, for the last run, EBAC improves the performance of OLTP1 by 61.6%; and OLTP2 is improved by 70.3%.

There are several results that needs some detail explanation. First, the later runs of “disk” does not seems to benefit from earlier runs due to a warm cache. By investigating the kernel, we find out that this is because the kernel frees all the buffered pages for raw devices when its corresponding device file is closed. Therefore, each run of the “disk” case essentially starts with a cold L1 cache.
However, for the later runs of “UA” and EBAC, they are able to benefit from both a warm L1 cache and L2 cache, so their performance is improved significantly. However, by comparing “UA” and EBAC, it is clear that our design performs much better than the existing design. Second, for “UA” of OLTP1, the L2 cache supported cases perform better with a cold cache rather than a warm cache. This is because when the L1 cache is cold, no eviction exists at the beginning of the run, such that no L2 cache loading overhead for evictions needs to be paid. Since the cache loading overhead is high for “UA”, the overhead is significant. On the other hand, since OLTP1 is a write-intensive workload, it does not really benefit much from a warm cache. Due to these reasons, the first run actually performs better. In fact, “UA” scheme actually performs worse than the “disk” scheme in the later runs. Third, the performance improvement of EBAC over “UA” is not significant for the first runs of OLTP2. The reason for this is because the L2 cache is started empty. Since the working set size of OLTP2 is smaller, therefore, both “UA” and EBAC can place the data blocks sequentially without considering replacement in L2. However, in later runs, the cache is filled up due to the significant amount of write requests in the workload, thus replacements start to work. Since “UA” replaces data pages based only on hit ratio, its disadvantage in SSD based cache starts to show.

**TPC-H Queries**

Finally, we evaluate the TPC-H benchmark performance on MySQL 5.1/InnoDB platform. We use the dbgen and qgen tool from the TPC to generate dataset and queries. MySQL is configured based the 2GB main memory of our system. And the scale factor of the dataset is 1. We set the L2 cache size as 1/8 of the dataset. Similar to other evaluations, to create a comparable test environment, for each configuration we restart the database and run the 22 queries in the generated sequence without interruption. Figure 3.13 shows the results of the test. The results again confirms
that our design consistently improve the performance of the “disk” scheme and the “UA” scheme with an average performance improvement of 47.3% of the 22 queries.

![Graphs of execution times for Disk, UA, and EBAC schemes.](image)

Figure 3.13: Evaluation Results of TPC-H on MySQL

3.5 Summary

In this chapter, we study how to effectively use SSDs as a L2 disk cache, following the DRAM based buffer cache as the L1 disk cache. We show that SSD based cache must address two new aspects: placement and admission, which are not considered by most cache management algorithms. We propose new algorithms to address these issues. Our placement algorithm is able to minimize the additional overhead due to the inherent asymmetric performance of SSDs in comparison with a random access caching device with symmetric performance characteristics. Our admission algorithm is highly space efficient and can utilize both spatial and temporal locality to minimize unnecessary cache loads to improve system performance. The performance of our algorithms are studied in detail by real SSDs based simulation and system implementation. The evaluation using industry standard benchmarks, such as TPC-H and OLTP storage traces from two large financial
institutions shows that our proposed algorithms can significantly improve the system performance. We plan to deploy the SSD-based L2 buffer cache in a data intensive production environment to further test its performance and reliability.
4.1 Overview

As discussed in Chapter 2, the evenness of the remaining cycle of different flash memory cells is managed by FTL. Therefore, the key to maximize the lifetime of flash SSD is to minimize the total number of write operations that are issued to flash memory chips. In Chapter 3, we discussed reducing the number of write requests issued to SSDs by designing new cache management algorithms. Since the total number of write requests includes both user issued requests and SSD internal data movement overhead, it is important to understand the cause of this internal data movement overhead and optimize it. Moreover, such optimization should be accomplish at no cost of SSD performance.

4.2 Capacity, Access Pattern, Performance and Wear

4.2.1 Over-provisioning Capacity and Write Amplification

Log structured management is a major design principle in flash memory based systems to avoid in-place updates [18]. Log structured management always appends updates to the end of the log instead of writing to the previous location of the data. For each update, the previous copy of the data is simply invalidated, and the garbage collector will clean the storage to reclaim the
space occupied by stale (invalid) data pages. Since both current data and stale data co-exist in log structured storage systems, the physical storage space is larger than the logical storage space available to the user. The capacity difference is the Over-Provisioned Capacity (OPC). OPC has a strong correlation with the performance and wear of SSDs, due to its impact on the cleaning efficiency.

Cleaning efficiency is ratio between the number of workload requested writes and the sum of workload requested write plus the number of data pages moved during cleaning (also called garbage collection (See Chapter 2 for details). The reciprocal is the write amplification rate. This metric reflects the system write overhead due to the limitations of flash memory write operation. When the cleaning efficient is 100%, the write overhead is zero, and the write amplification rate is 1 (no amplification). Such cleaning process is usually done in the background to minimize its impact on foreground tasks. Theoretically, in the worse case, the write amplification ratio can be as large as the number data pages within an EB if OPC is less than \(1/P\) of the logical storage space, where \(P\) is the number of pages within an EB (In the worse case, each EB may have only one invalid page, such that the garbage collection always has a write amplification rate of \(P\)). The larger the write amplification rate is, the more number of writes are issued to the flash memory chip, decreasing both the performance and remaining life-span of SSDs. Let us call the number of up-to-date (valid) data pages within an EB the occupancy of an EB. Then, OPC determines the upper bound occupancy of the EB that is least occupied. If the garbage collector uses the greedy cleaning policy, which always chooses the EB least occupied for cleaning, then the worse case write amplification rate for all workloads is determined by OPC. Since most cleaning policies considers the occupancy of EBs when choosing the cleaning target, higher OPC in general leads to better performance and less overall wear.
4.2.2 Access Pattern Impacts

Although the write amplification rate is highly dependent on OPC in general, for a given workload, the actual write amplification is also dependent on the access pattern of the workload. An example is an event logger, which always record new events sequentially towards the end of the log file. Once the file space is filled up, it wraps up to beginning of the file to replace the oldest data. In this example, if the logical file addresses are mapped appropriately onto the flash chip to keep logically sequential data in the same EB, then 100% cleaning efficiency can be achieved for this workload, and a very small OPC to just accommodate a single EB is adequate to achieve the optimal performance and wear. However, most workloads do not follow such sequential update patterns. And many have substantial random patterns, as shown in the evaluated server workloads obtained from multiple sources in the next section.

The access pattern is not only determined by the workload access sequence itself, it is also influenced by the SSD organization. Within a single SSD, multiple flash memory chips are organized as flash arrays to achieve designed performance and storage capacity requirements. Similar to RAID, data can be striped among the chips to utilize the benefit of parallelisms. On the other hand, the striped mapping may allocate logically sequential data pages on to different chips, such that sequential accesses for each flash memory chip actually corresponds to strided workload accesses.

Summary In this section, we have discussed the qualitative relationship among the over-provisioning capacity, access pattern, and write overhead. In the next section, we will investigate the quantitative relationship among over-provisioning capacity, access pattern and SSD performance and wear using experiment studies. Specifically, several questions need to be answered via thorough measurement based investigation: (a) How does the over-provisioning capacity affect the performance and wear of SSDs for realistic workloads? (b) How can SSDs or system designs be
optimized to reduce the wear of SSDs? More specifically, how to significantly reduce the write amplification rate for a given over-provisioning capacity?

4.3 Workload Measurements

4.3.1 Experiment Environment

The experiments are conducted using the Microsoft SSD extension for the CMU DiskSim simulation environment [44, 7]. We measured fifteen workloads. Twelve workloads are traced from enterprise servers by Microsoft Research Cambridge [41]. For multi-volume traces, we use volume 0 for evaluation. Two OLTP workloads are SPC traces from two large financial institutions [47]. And one workload is a one-hour Cello99 trace from the HP [22]. The detailed information on the workloads are listed in Table 4.1.

<table>
<thead>
<tr>
<th>name</th>
<th>address space</th>
<th>write working set</th>
</tr>
</thead>
<tbody>
<tr>
<td>hm</td>
<td>13.94GiB</td>
<td>11.94%</td>
</tr>
<tr>
<td>mds</td>
<td>33.92GiB</td>
<td>1.01%</td>
</tr>
<tr>
<td>prn</td>
<td>66.32GiB</td>
<td>18.70%</td>
</tr>
<tr>
<td>proj</td>
<td>16.24GiB</td>
<td>10.28%</td>
</tr>
<tr>
<td>prxy</td>
<td>20.71GiB</td>
<td>3.48%</td>
</tr>
<tr>
<td>rsrch</td>
<td>16.89GiB</td>
<td>1.76%</td>
</tr>
<tr>
<td>src2</td>
<td>15.63GiB</td>
<td>3.34%</td>
</tr>
<tr>
<td>stg</td>
<td>10.81GiB</td>
<td>3.96%</td>
</tr>
<tr>
<td>ts</td>
<td>21.97GiB</td>
<td>2.50%</td>
</tr>
<tr>
<td>usr</td>
<td>15.90GiB</td>
<td>4.11%</td>
</tr>
<tr>
<td>wdev</td>
<td>16.96GiB</td>
<td>2.09%</td>
</tr>
<tr>
<td>web</td>
<td>33.91GiB</td>
<td>2.16%</td>
</tr>
<tr>
<td>cello</td>
<td>7.36GiB</td>
<td>6.94%</td>
</tr>
<tr>
<td>oltp1</td>
<td>8.25GiB</td>
<td>43.36%</td>
</tr>
<tr>
<td>oltp2</td>
<td>8.29GiB</td>
<td>7.68%</td>
</tr>
</tbody>
</table>

Table 4.1: Workload Description
We used the default settings for most of the Disksim-SSD simulator, which simulates the Samsung K9XXG08UXM SLC flash memory chip [1]. In order to investigate the quantitative relationship between over-provisioning capacity, workload working set size and write amplification, we vary the per-plane block number to adjust the SSD physical capacity, such that the overall chip array organization is kept the same for all configurations as an eight element array which is consistent with most Samsung NAND chip products [46]. The detailed values of other parameters are list in Table 4.2.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page read to Register</td>
<td>25us</td>
</tr>
<tr>
<td>Page program(write) from register</td>
<td>200us</td>
</tr>
<tr>
<td>Block erase</td>
<td>1.5ms</td>
</tr>
<tr>
<td>Pages per block</td>
<td>64</td>
</tr>
<tr>
<td>Block mapping scheme</td>
<td>full stripping</td>
</tr>
<tr>
<td>Cleaning policy</td>
<td>greedy</td>
</tr>
<tr>
<td>Background cleaning</td>
<td>Yes</td>
</tr>
<tr>
<td>Copy-back</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

Table 4.2: Parameters of DiskSim-SSD

### 4.3.2 Evaluation Results

**Response Time Sensitivity to Over-provisioning Capacity**

To measure the sensitivity of Over-Provisioning Capacity (OPC) on SSD performance and wear, we conducted two sets of experiments. In the first set of experiments, we configured an SSD of 64GiB in physical capacity, then we vary the reserved capacity for over-provisioning from 10% to 45% of the physical capacity for all the workloads that can fit onto this disk. In the second set of experiments, we configure the OPC according to the write request working set size as listed in Table 4.1. For each workload, the physical capacity is configured as the workload storage address.
space size plus five times of the write working set size. Then we vary the reserved capacity for over-provisioning from 20% to 300% of the working set size. We report the average response time and the write amplification rate for each experiment. These experiments can demonstrate the performance and wear sensitivity to disk capacity based over-provisioning and workload based over-provisioning strategies, respectively.

Figure 4.1: Sensitivity to Write Working-set based OPC
Figure 4.2: Sensitivity to Disk Capacity based OPC
As shown in Figure 4.2, when the over-provisioning is based on disk capacity, the response time does not show significant changes for all the workloads. This shows disk capacity based over-provisioning provides more than necessary capacity for the desired performance. On the other hand, when the over-provisioning is based on write working set size, the performance changes as the OPC increases. Although the sensitivity may vary among the workloads, for most workloads, OPC at 140% can achieve an average response time similar to larger OPC scenarios. Due to design of the Disksim simulator, some workloads require an adequate OPC to be able to finish the simulation without saturation. This minimal required OPC is also different among the workloads, ranging from 40% to 200%. However, most workloads are able to complete when OPC is at 60%. The response time of some workloads are very sensitive to the OPC especially when the OPC is relatively small. For example, for the src2 workload, the average response time changes from 3.29 ms when the OPC is 40% to 1.04 ms when the OPC is 140%. Some changes slowly, for example, the ts workload changes from 0.77 ms when the OPC is 60% to 0.62 ms when the OPC is 140%.

**Effects on Write Amplification**

In general, the response time impact is consistent with the write amplification rate. For the disk capacity based over-provisioning, the write amplification rates are 100% for most cases. While for write working set based over-provisioning, the write amplification rates distribute in a wide range from 100% to 876.70%.

Overall, the OPC impact on the write amplification rate is larger than on the response time. For example, the write amplification of the same src2 workload in the above example changes from 473.76% to 107.17% when the OPC changes from 40% to 140%, showing a difference of more than 4 times while the the response time differs only by 3.16 times. This can be explained by two causes: (a) the cleaning (garbage collection) can be scheduled in the background to utilize the idle period between request arrival to hide all(or part) of the data movement latency, reducing
the effects on workload response time; (b) workloads contain both read and write requests, such that write amplification only affects part of the workloads. In some extreme cases, the response time does not differ much, while the write amplification rate difference are quite significant. For example, the write amplification rate of oltp2 workload changes from 233.06% to 129.59% when the OPC is increased from 60% to 140%, while the response time only changes from 0.20ms to 0.19ms. This indicates that capacity efficiency not only affects performance, but also has a critical implication on the lifespan of the SSD.

<table>
<thead>
<tr>
<th>name</th>
<th>M-OPC</th>
<th>OPC ratio</th>
<th>response time ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>hm</td>
<td>11.94%</td>
<td>5.78%</td>
<td>1.12</td>
</tr>
<tr>
<td>mds</td>
<td>1.21%</td>
<td>1.19%</td>
<td>1.07</td>
</tr>
<tr>
<td>proj</td>
<td>16.45%</td>
<td>5.80%</td>
<td>1.13</td>
</tr>
<tr>
<td>prxy</td>
<td>10.44%</td>
<td>2.50%</td>
<td>1.08</td>
</tr>
<tr>
<td>rsrch</td>
<td>2.82%</td>
<td>1.03%</td>
<td>1.13</td>
</tr>
<tr>
<td>src2</td>
<td>4.68%</td>
<td>1.81%</td>
<td>1.12</td>
</tr>
<tr>
<td>stg</td>
<td>5.54%</td>
<td>1.49%</td>
<td>1.13</td>
</tr>
<tr>
<td>ts</td>
<td>3.00%</td>
<td>1.91%</td>
<td>1.09</td>
</tr>
<tr>
<td>usr</td>
<td>8.22%</td>
<td>2.27%</td>
<td>1.05</td>
</tr>
<tr>
<td>wdev</td>
<td>2.51%</td>
<td>1.23%</td>
<td>1.10</td>
</tr>
<tr>
<td>cello</td>
<td>11.10%</td>
<td>1.77%</td>
<td>1.17</td>
</tr>
<tr>
<td>oltp1</td>
<td>95.39%</td>
<td>12.43%</td>
<td>1.04</td>
</tr>
<tr>
<td>oltp2</td>
<td>19.97%</td>
<td>2.21%</td>
<td>1.01</td>
</tr>
</tbody>
</table>

Table 4.3: Over-provisioning Capacity vs. Workload Storage Address Space Size

Summary

We make the following observations based on the measurement results.

1. It is more efficient to use write request working set than disk capacity as reference for over-provisioning capacity design. For all the workloads, the OPC can be significantly reduced
for similar write amplification rate and average response time. To explain such effect, in Table 4.3 we show the minimal OPC (M-OPC) as percentage of each workload’s total storage address space size, when the write amplification rate is less than 1.10. To illustrate its capacity efficiency, we compare its the response time with the response time when the over-provisioning capacity is 45% of the disk capacity. The result is listed as response time ratio in Table 4.3. OPC ratio in Table 4.3 is the ratio of M-OPC vs. 45% of the disk capacity. It shows M-OPC uses only 1.03% to 12.43% of disk capacity based over-provisioning capacity with an average of 3.19%, while the response time impact is within 1.47% to 16.50% with average of 9.40%.

2. Although cleaning optimization, such as background cleaning can be used to reduce the impact of cleaning overhead on request response time, the impact of OPC on the average response time is still significant. The key for further optimization is to reduce the write amplification rate, which is beneficial for both the response time and lifetime of SSDs.

4.4 Write Amplification Rate Reduction Algorithms

Based on the measurement results, it shows appropriately over-provisioning workloads is effective to reduce the write amplification rate, and such over-provisioning should be workload dependent. Therefore, it is important to understand what workload dependent factors are most critical in determining how much capacity to provision. In essence, the cause of write amplification is due to the non-uniform update time of pages within an EB, such that when garbage collection is in progress, valid data pages need to be relocated. In general, there are two approaches to this issue: (a) write data in the order as they are requested, however, choosing the target to be reclaimed carefully so that the relocation cost is small; (b) carefully selecting the location of each write, such that the
update time of pages within each EB can be close to each other to reduce the relocation cost. These two methods can be complimentary to each other.

Previous work on LFS garbage collection policies, such as [45, 5], investigated methods to improve the write amplification ratio. These studies focus on choosing the appropriate segments to clean. In contrast, we propose to optimize data allocation policies.

4.4.1 Algorithm Preliminaries

For the clarity of presentation, let us first define the operation model of the SSD cleaning process and a few concepts.

**Operation Model** The physical data pages within an SSD can be in three states: *valid*, *invalid* and *free*. The valid pages are mapped to logical addresses such that it is accessible by the user of the SSD. As soon as a valid page is updated, it becomes an invalid page and the logical data page mapping is relocated to another page allocated for the update. Once the invalid pages are erased, they become free pages. Free pages can be allocated for write requests. Both invalid and free data are not mapped to logical addresses. And the total number of invalid and free pages are equal to the over-provisioning capacity. Figure 4.3 illustrates such state transitions.

![Figure 4.3: State Transitions of Physical Pages in SSDs](image)

Figure 4.3: State Transitions of Physical Pages in SSDs
The system invokes the cleaning process to reclaim the invalid pages based on its policies. A common heuristic is to trigger it when the number of free pages are below a threshold. And when choosing the target for cleaning, two criteria are usually considered: (a) the number of invalid pages within an erase block (EB); (b) the age of the page. The greedy cleaning algorithm considers only the first criterion, while the cost-benefit cleaning algorithm weights criterion (a) against (b). For the simplicity of presentation, in our analysis, we consider the greedy algorithm with threshold based invocation.

**Concepts** Let us consider a request sequence of logical data pages, and each write request is assigned a *write sequence number*, starting from 1 in ascending order. The *forward write sequence number* of a write request is the write sequence number for the data page’s next update. For example, let us assume a request sequence in the format of (type, page number): (write, 524), (read, 783), (write, 24), (write, 24), (read, 83), (write, 524). Then the forward write sequence number for the four write requests are 4, 3, $\infty$, $\infty$.

When the logical data pages are mapped to physical pages, the physical pages also have a forward write sequence number which is the same corresponding to its logical mapping. We also define the forward write sequence number of an invalid physical page as the the forward write sequence numbers of its latest mapping. Free pages do not have a forward write sequence number. For example, assume in an SSD, the physical page 3 is mapped to logical page 8 at write sequence number 12, and the forward write sequence number of logical page 8 is 32. Then, the forward write sequence number of physical page 3 is 32 too. When the 32nd write request comes, which invalidates the mapping of physical page 3 and remaps its to a new physical page, the write forward distance of physical 3 is still 32.
**Lemma 4.4.1** The forward write sequence number of invalid pages are always smaller than the current write sequence number. And if the forward write sequence number of a page is less than the current write sequence number, then it must be an invalid page.

**Theorem 4.4.2** Assume $p$ pages within each EB, and $t$ is the threshold of free pages before cleaning is invoked. If the data pages are allocated such that the forward write sequence number difference of any two data pages within an EB is less than a constant $k \times p$, and the over-provisioning capacity is $k \times p + t$ pages, then the write amplification rate is 1, i.e. the internal data movement overhead is 0.

**Proof** When the cleaning is invoked, let $i$ be the smallest forward write sequence number among all invalid pages. Since there are at least $k \times p$ invalid pages (the cleaning invocation precondition), according to lemma 4.4.1, $i + k \times p$ must be less than or equal to the current write sequence number. Since the forward write sequence number difference between any pages within this EB is less than $k \times p$, then the forward write sequence number of any pages in this EB must be less than $i + k \times p$. Therefore, any page within this EB must be invalid. Invalid pages involve no data movement overhead.

The above theorem shows that for a given over-provisioning capacity ($k \times p + t$), allocating the pages based on the write sequence number can results in optimal cleaning efficiency. As an example, for a ring-buffer-like sequential write sequence loop, if the algorithm allocates the pages simply according to its arrival sequence, then the forward write sequence number of all pages within an EB are consecutive, and any difference between two pages are less than $p$. Then with an over-provisioning capacity as small as one EB plus the threshold size, the write amplification rate can be 1.
4.4.2 WFWD Allocation Algorithm

Theorem 4.4.2 provides the guidance for designing an optimal allocation algorithm. However, there are a few more challenges to design a page allocation algorithm for real workload.

*First*, for a given over-provisioning capacity, it can be impossible to allocate the incoming requests such that pages within the same EB is less than the threshold to achieve zero amplification. For example, assuming an SSD whose EB contains 2 pages, and the over-provisioned capacity is 6 pages (3 EBs) with a cleaning threshold of 2 pages. Then page requests must be placed such that the forward write sequence number of the pages within an EB is less than 4. For a sequence of incoming requests of forward write sequence number as: (10, 45, 20, 35), such criterion cannot be satisfied because there are at most 3 EBs for page allocations. *Second*, to achieve the optimal allocation, it may require the algorithm to have look-ahead capability. If the forward write sequence numbers of incoming requests are (100, 2, 9, 3, 7, 101), then the optimal algorithm should group (100, 101), (2, 3) and (9, 7), respectively. In this example, to group (100, 101), it must be known in advance that the request of forward write sequence number 101 is coming, such that the one with 100 can be set aside to wait for it. Such look-ahead capability requires more detailed information passing from the workloads, otherwise buffering resource is needed to allow allocation re-ordering for the requests.

Due to these difficulties, we propose WFWD – a heuristic to optimize page allocations without the look-ahead capability. We try to group pages of similar forward write sequence number as close as possible, such that if a page is updated, the pages within the same EB are likely to be updated too. We first assume that the forward write sequence number is informed by the system upper layer. Later, in Section 4.4.3, we consider the case when such information is unavailable, and propose a prediction algorithm to estimate the forward write sequence number.
WFWD is based on write forward distance, which is defined as the forward write sequence number minus the write sequence number of a page. A constant number of buckets are defined, each of which corresponds to a range of write forward distance. We allocate the data pages into different buckets according to the ranges. The size of each bucket is the same, and it is in multiples of the EB size. However, as soon as a bucket is filled up, new EBs are assigned to the bucket, therefore, the number of data pages allocated from each bucket can be different.

The ranges corresponding to each bucket are varied in size. We number the buckets in ascending order according to the average write forward distance of its range. The range sizes of buckets monotonically increase as the write forward distance increases. The range increasing step is determined by a step function, which is a configurable system parameter. By default, we use an exponential function in our evaluation. In Figure4.4, we illustrate the designs by a concrete example. In this example, there are four buckets in total. Each bucket consists of one EB, which contains four pages. Bucket 1 corresponds to write forward distance range of (1-128). There are 11 pages allocated from this bucket. Similarly, there are 6, 2 and 4 pages from bucket 2, 3 and 4, respectively.

### 4.4.3 Predicting the Write Forward Distance

When the forward write sequence number is not informed by the system upper layer, we need to estimate such information using the access history of the workload. We use two heuristics in our estimation. The first heuristic is to predict the write forward distance of a certain data page using the past write intervals. This heuristic is used when the past write intervals are regular. The second heuristic is to predict the write forward distance of a certain data page by its friends’ write forward distance estimation. A friend of a data page is a data page that is recently updated, and its logic address is within the same neighborhood of the aforementioned data page. This heuristic is
used when the first heuristic cannot be applied. When multiple friends exist, we choose to use the closest friend as a reference. If a data page does not have a friend and its past update history is not regular, then the data page is allocated from the bucket with the largest number.

To record the access history, we track up to four recent write access intervals of a logical page. If the variation of the intervals is small, it is considered as regular. For the management of friends, we maintain a constant-sized pool of recently updated data pages. Only the meta-data of the data pages are managed, and the LRU algorithm is used to replace old entries in this pool. If a page is not considered as regular, the pool is queried to locate friends.

### 4.4.4 Implementation in DiskSim-SSD

This DiskSim simulator is a storage sub-system simulation environment [7]. The Microsoft SSD extension implements SSD devices for the DiskSim simulator as the storage device. As

![Figure 4.4: An Example for Write Forward Distance based Allocation.](image-url)
introduced in [1], each SSD is internally organized as an array of NAND flash memory chips. Each chip can consist of multiple planes.

The simulator supports multiple allocation scheme based on whether a physical page is from a certain plane, planes sharing a bus, or a chip. In [1], it shows the greedy policy based chip allocation scheme performs best. Since our algorithm and the existing algorithms in DiskSim-SSD is complimentary, we implement our algorithm on top of this best performer. We maintain a set of buckets in our algorithms for each plane of the SSD. This enables our design to take advantage of the parallelism among the planes of the same flash chip. For example, when two requests of the same write forward distance arrive, the data pages can be allocated from different planes, such that the two pages can be written concurrently. The impact can be significant, as shown in our evaluation.

4.4.5 Performance Evaluations

We use the same setup as in section 4.3.1 for our experiments and over-provision the SSDs based on write working-set size. We show the results for our write forward distance based algorithms. In comparision, we implement the MODA algorithm in the DiskSim-SSD simulator. MODA is a recently studied data allocation policy for embedded flash devices [2]. It allocates data pages based on its recent access frequency. Data pages are allocated on the same EBs, when the recent access frequency is more than (or less than) a static threshold. More details on the MODA algorithm can be found in [2]. Our proposed policy is different in that it is based on write forward distance, while the existing one is based on recent access frequency. In addition to Moda, the original Disksim-SSD results are also included. In the original Disksim-SSD design, the pages are allocated using a greedy strategy. It always chooses to allocate a page from a plane which has most number of free pages. It balances the space utilization among the planes and exploits the
possible parallelism among the planes. Such strategy is complimentary to both the proposed write forward distance based algorithm and the MODA algorithm. And we implement our algorithm and MODA algorithm on top this existing design.

Results

Figures 4.5 and 4.6 show the results of average response time and write amplification rate of different workloads using different algorithms. The bars in these graphs correspond to average response time, while the lines correspond to write amplification rate. Overall, there are several trends for these algorithms.

First, for the same over-provisioning capacity, the average response time and write amplification rate of write forward distance based algorithms consistently perform better than MODA and the original algorithm of Disksim-SSD. This is reflected by the lower response time and write amplification rate. In addition, due to the designs of Disksim, the simulator may stop simulation when enough requests are queued for processing. In many cases of the experiments, the write forward distance based algorithm can finish the simulation without saturation while others cannot, as shown by the missing data points for these algorithms in the graphs. For example, for the stg workload, when the over-provisioning capacity is 60% of write working set size, both MODA and the original algorithms saturate, while the proposed algorithms can report results.

Overall, the algorithmic impact is significant when the over-provisioning is relatively small. For example, when the over-provisioning capacity is 40%, the average improvements of the informed write forward distance algorithm over the MODA algorithm and the original algorithm for all the workloads in terms of average response time is 16.6% and 26.1%, respectively. And the average reduction of write amplification rate are 48.1% and 153.7%, respectively. The corresponding improvements for the prediction based algorithms are 11.2%, 22.1%, 39.9%, and 145.8%, respectively.
Figure 4.5: Comparison of Algorithms (a)
Figure 4.6: Comparison of Algorithms (b)
In real system, these improvements have several implications: (a) For the same over-provisioning capacity, both the performance and life-span of the SSDs can be improved using write forward distance based algorithms; (b) When the response time and wear-level is acceptable for the system requirements, write forward distance based algorithm can accommodate the same workload with less capacity, or more workloads can be accommodated by the same SSD.

Second, among the algorithms, the original Disksim-SSD algorithm only considers parallelisms in page allocation, while the other schemes also consider the data page property (write forward distance for the proposed algorithms and recent access frequency for the MODA algorithm). The results show that considering data page property can lead to significantly improvements for both performance and write amplification rate. However, based on our experiments, we note that the consideration of parallelisms in page allocation is also indespensible, though the details of utilizing parallelism in flash arrays are out of the scope of this paper.

Third, For the two proposed algorithms, the prediction based scheme performs closely to the informed scheme for most workloads. This shows that the write forward distance based algorithm can be implemented transparently in existing software stack, as the prediction based algorithm requires no additional information other than those already available in existing block protocols. Of course, in cases where the protocol can be enhanced to inform the write forward distance from the upper layer, the performance of the algorithm can be further improved.

4.5 Summary

Extending the lifetime of flash SSDs is one of the main challenges as the capacity of flash memory scales up to meet the ever-increasing capacity demand. In this chapter, we analyze the relationship between capacity, access pattern and SSD lifetime. Based on real workload measurements, we present the interaction between capacity, workload access pattern and SSD wear. The
results show that it is effective to trade the logical capacity of SSDs to extend their lifetime, and an effective trade-off should be based on the working set size of write requests of the workload. In order to improve the effectiveness of such trade-offs, the key is to reduce the write amplification ratio. We present a theoretical analysis of the relationship between the over-provisioning capacity and the data allocation strategy to achieve optimal write amplification ratio. Based on such analysis, we propose write forward distance based data page allocation algorithms, which, for a given over-provisioned capacity, can improve the write amplification rate of existing designs.
CHAPTER 5

Conclusions

The flash memory based storage is a promising technology that will enable high performance and low power consumption storage systems. It is considered to be the major technology for the next generation storage systems. Despite of its many advantages, two limitations are inherent in flash memory based systems: (a) random write performance is not competitive. (b) the life-cycle of flash memory is limited and each write operation consumes its life-cycle. These call for system designs which can minimize impacts of these limitations.

In this thesis, we discuss techniques to optimize the performance and lifetime of NAND flash memory based SSDs. In particular, we consider cache management algorithms for SSD based disk cache to utilize high performance SSD access patterns and reduce unnecessary cache admissions; we also consider page allocation algorithms that minimize the write amplification rate to maximize the lifetime and performance of SSDs. We have implemented systems to validate our designs. The results show that the proposed techniques successfully achieve the design goals.
BIBLIOGRAPHY


[22] HP Labs. Storage Tools and Traces.


