Improving SSD-based Caches Lifetime with Write-Once Memory Codes

Ran Koretzki
Improving SSD-based Caches Lifetime with Write-Once Memory Codes

Final Paper

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Ran Koretzki

Submitted to the Senate
of the Technion — Israel Institute of Technology
Sivan 5777 Haifa June 2017
This Final Paper was done under the supervision of Prof. Yaakboi and Prof. Schuster in the Faculty of Computer Science.

Acknowledgements

I sincerely thank my advisers, Prof. Eitan Yaakobi and Prof. Assaf Shuster, for the guidance, the patience and the time spent in my research and for the great help in writing this work. I had the privilege and the honor to work directly under these two professional scholars and dedicated teachers. Without their help, I could not develop a storage theory and build a cost model for good use of SSD-based cache. In addition, I would like to thank Dr. Gala Yadgar for the help she provided me which allowed me to graduate successfully.

I would like to thank my parents and my fiancee for their great support throughout the research period, and for giving me mental push along the way to continue studying, not to despair even when the research did not progress at the desired speed. My great desire to advance in research and to meet my schedule was due to their great encouragement and support. I would also like to thank my mentor at the faculty and my partner in instruction of Elad Kravi, for making my time teaching enjoyable. His professionalism and creative ideas helped me solve difficult problems and gave me momentum for further studies and professional career.

The generous financial help of Technion is gratefully acknowledged
# Contents

List of Figures

Abstract 1

Abbreviations and Notations 3

1 Introduction 5

2 Background and Related Work 9
  2.1 NAND Flash Memory and SSDs 9
  2.2 Write-Once Memory Codes 10
  2.3 Flash Based Cache 12
  2.4 Cache Algorithms 12
  2.5 Problem Statement 14

3 Cache Design 15
  3.1 Design Overview 15
  3.2 Read Cache Design 16
  3.3 Write Cache Design 18

4 Storage Cost Model 23
  4.1 Read Cache Cost Model 23
  4.2 Simple Write Cache Cost Model 25
  4.3 Combined Read and Write Cache Cost Model 27

5 Implementation Challenges 29
  5.1 DiskSim 29
  5.2 Microsoft SSD Extension 29
  5.3 The Attempt to Implement WOMAC 30

6 Discussion 33
  6.1 Open Questions 33
  6.2 Evaluation Methodology 34
List of Figures

1.1 A depiction of the life cycle of a 3 bits of flash page containing a 2 bit word. At first, bits are clean, then they are written in first write. Later, they are rewritten to express a different value. Bits are changed irreversibly from a value of 0 to a value of 1 until an erase operation is performed on them. ................................................................. 6

3.1 WOMAC cache areas, divided into a read cache and a write cache. Each one is divided further into first and second writes areas. Each cell depicted the data it stores in the area, and the data structures that manages its entries. Further description of data flow is given in the following chapters and in figure 3.4. ................................................................. 16

3.2 A 7-segmented ghost buffer on T2_R and B2_R, T2_R contains data, and B2_R contains only meta data. B2_R for simulating a larger LRU size. Each segment has an equal amount of records. When there is a hit on one of the ghost buffer segments, the cumulative hit rate for that segment and the following segments is increased. e.g. if a hit has accord on S6, the hit is marked for S6 and S7. ................................................................. 17

3.3 Read cache architecture employs two LRU lists of logical blocks and two ghost queues. The first time that page 11 is read, there is a cache miss and it is written to T1_R, as a second write. When page 11 is read again, the page is written to T2_R as a first write and the copy in T1_R is invalidated. ................................................................. 18

3.4 Data flow in WOMAC ................................................................. 20

3.5 Write cache architecture employs two LRU lists of logical blocks and two ghost queues. The first time that page 1436 is written there is a cache miss, and it is written to T1_W, as a first write. When page 1436 is getting a write request again, the copy in T1_W is invalidated and is written to T2_W as a second write. Additionally, when there is a write request on page 949 and there is a hit in ghost queue B1_W, the page is written to T2_W, and the ghost record is invalidated. ................................................................. 21
Abstract

Solid State Disks (SSDs) have the potential to revolutionize the storage system landscape. They have gained popularity as cache devices in data centers because, they are faster in read and write operations and have lower power consumption, compared to the traditional magnetic hard disks (HDD). However, SSDs have a limited number of times it can write to a single physical location, and there is a second limitation. The SSD must perform an erase operation before it can write to the same location again. Write-once memory (WOM) codes were studied as a way to increase the number of writes on a write-once storage medium to more than once. In [YYS15], the usage of WOM-codes in SSD was studied to increase its lifetime by 50%; theoretically, reusing invalid pages for additional writes, and thus reducing the number of erase operations. The additional writes are inflated when using WOM-codes and come with the cost of utilizing the overprovisioning space in SSDs to maintain constant external capacity. In cache devices, there is no such constraint on external capacity, so we could use a more significant percentage for second writes in order to extend a bit further the SSD lifetime. In this work, we propose a caching algorithm for SSD-based caches that will leverage WOM-codes to reuse invalid pages for gaining an additional number of writes in SSD. We study the use of these additional writes to improve in both cache hit ratio and SSD lifetime, without the constraint of maintaining constant external capacity.
Abbreviations and Notations

T1_R - Ordered list of recent read cache entries the are written in second write, and were referenced once.

T2_R - Ordered list of recent read cache entries the are written in second write, and were referenced at least twice.

T1_W - Ordered list of recent write cache entries the are written in first write, and were referenced one.

T2_W - Ordered list of recent write cache entries the are written in second write, and were referenced at least twice.

B1_R - Ghost entries recently evicted from the T1_R, but are still tracked.

B2_R - Ghost entries recently evicted from the T2_R, but are still tracked.

B1_W - Ghost entries recently evicted from the T1_W, but are still tracked.

B2_W - Ghost entries recently evicted from the T2_W, but are still tracked.

C_{D,RPOS} - Sum of the average seek time and the average rotational delay for HDD reads.

C_{D,WPOS} - Sum of the average seek time and the average rotational delay for HDD writes.

P - Data size in bytes.

B - The bandwidth of the HDD.

C_{DR} - The data read cost of an HDD.

C_{DW} - The data write cost of an HDD.

C_{PROG} - The cost of writing a single page to SSD.

C_{READ} - The cost of reading a single page from SSD.

N_P - Pages count in each physical block.

C_{E} - The erase operation cost of a flash memory block.

C_{1ST,PW} - The cost of a page write to SSD cache in a first write with the overhead costs of cleaning that page.

C_{2ND,PW} - The cost of a page write to SSD cache in a second write.

r - The read cache ratio within the full cache space.

s - The second writes space ratio within the read cache space.

x - The second writes space ratio within the write cache space.
$HR_{1ST}(s, r)$ - The hit rate given cache size of 1st write area in the read cache.

$HR_{2ND}(s, r)$ - The hit rate given cache size of 2nd write area in the read cache.

$HW_{1ST}(x, r)$ - The hit rate given cache size of 1st write area in the write cache.

$HW_{2ND}(x, r)$ - The hit rate given cache size of 2nd write area in the write cache.

$H_{1st,r}$ - The read request portion from hit rate given cache size of 1st write area in the write cache.

$H_{2nd,r}$ - The read request portion from hit rate given cache size of 2nd write area in the write cache.

$C_{HR}(s, r)$ - The read cost of the hybrid storage system.

$C_{HW}(x, r)$ - The write cost of the hybrid storage system.

$C_{HY}(x, s, r)$ - The overall access cost of the hybrid storage system.

$S_f$ - The full cache size.

$S_r$ - The read cache size.

$S_w$ - The write cache size.

$IO_R$ - The rate of read requests.

$IO_W$ - The rate of write requests.
Chapter 1

Introduction

In recent years, NAND flash-based Solid State Drives (SSD) have increased in popularity as storage devices, as their prices continue to drop and capacity continues to grow [GDS12]. This popularity, over prior storage technology, is due to their short latency in read and write operations, increased throughput and reduced power consumption.

However, flash technology has its limitations. Before a flash page can be programmed, it must be erased. An erase operation is done in bulk of flash pages, called blocks. Those blocks can endure an erase operation a limited number of times before they can no longer be read reliably [APW+08]. Counterintuitively, as flash technology improves with the goal of increasing storage capacity, it has made flash less resistant to failures, causing shorter lifetime, which is measured in program / erase cycles (P/E) [KC06, GDS12, Koh09].

Flash-based SSDs have become an active area of research in the systems community [Des13], and much work has been done to reduce the number of erase operations to prolong the SSD lifetime. Suggested methods include reducing the number of erase operations needed by reducing the write traffic to flash [KA08, DSDL09, KLL+12, Des12, PD11, SSKM10]. Others suggested several leveling techniques to distribute the wear cost evenly between the drive’s blocks [PDZ11, MD11] or pages [JNI14]. Another way to approach the issue is to add a third step in the program / erase cycle of a page [YYS15, MB15, MYY+16].

A promising technique for reducing block erasures is to use write-once memory (WOM) codes, first introduced in [RS82]. WOM codes alter the logical data before it is physically written, thus allowing the reuse of cells for multiple writes. They ensure that, on every consecutive write, ones may be overwritten with zeros, but not vice versa. Reusing flash cells with this technique might make it possible to increase the amount of data written to the block before it must be erased. WOM codes require additional storage for code redundancy. Figure 1.1 depicts the life cycle of a 3bit flash page containing a 2 bit word on using WOM code.

A study [YYS15] demonstrates how to integrate WOM codes in an SSD drive, in which invalid pages are reused for additional writes, without modifying the drive’s
Figure 1.1: A depiction of the life cycle of a 3 bits of flash page containing a 2 bit word. At first, bits are clean, then they are written in first write. Later, they are rewritten to express a different value. Bits are changed irreversibly from a value of 0 to a value of 1 until an erase operation is performed on them.

exported storage capacity or page size. The pages can be written in the first writes as unmodified data, with no additional overhead. They can be written on invalid pages from the first programming step as inflated second writes and be used again.

Flash-based disk cache is another active area of research in the systems community, and much work has been done to integrate SSDs as caching tiers for primary storage. Several caching algorithms were developed to use the SSD as a caching device [HWF+16, SYK14, LYLCL13, DSDL09, LW14]. Other work investigated caching on SSD drive composed of various types of flash chips with different endurance limits [HS10, OLC+13]. Another approach focused on limiting the cache space to optimize caching performance [OCLN12].

We introduce a new cache type called WOMAC, which stands for Write-Once Memory Aware Cache. WOMAC constitutes a new point in the design of SSD cache systems. It continues the work done in [YYS15], and removes the constraint of the drive’s exported capacity to remain constant. Caching devices don’t have the requirement of a fixed size and can change their size to achieve additional goals [OCLN12, YPFV02, LYLCL13], since caches are transparent to applications and end users. The goal of this work is to extend the lifetime of SSDs used as caching tiers even more than [YYS15]. The lifetime of SSD acting as caching device, can be as little as 8 days [LCQX14], this is due to high write traffic, common in write caches in data centers. In WOMAC, an SSD drive controller could decide wisely when it is appropriate to store data in first writes or to perform inflated second writes and when to evict it.

WOMAC divides the SSD into a read cache and a write cache, with each one further divided into a first write area and a second write area. The algorithm places the cached data in each of the areas based on the request type, read or write, and based on the popularity of the data, with the goal of using the additional write to address the weakness in cache based SSD. Each area has LRU eviction policy and a ghost queue, to maintain history on the evicted pages’ popularity. The algorithm can recycle a block evicted from the first write area. If it chooses to clean a block, the block is erased and used again in the first write area. If it chooses to recycle a block, the block is used for second writes and assigned to the second write area. In addition, the algorithm decides the percentage in the SSD dedicated to first writes, second writes and free space depending on hit ratio and adjusts it based on the workload. A block, evicted from the
second write area, is cleaned and moved to the first write area.
Chapter 2

Background and Related Work

2.1 NAND Flash Memory and SSDs

NAND Flash is a non-volatile memory (NVM) which doesn’t require power to retain the data it contains. NAND flash is widely used due to its short latency in read and write operations, increase throughput and reduced power consumption compared to the traditional hard disk drives (HDD). NAND flash is usually packaged as Solid State Drives (SSD). SSDs have become widely used in a variety of appliances ranging from personal laptop drives, external drives, hybrid drives, server storage and data centers caching tiers [GF08].

The smallest unit of SSD is the NAND flash cell, which is a transistor with a floating gate that can retain an electric charge in the form of electrons. The cell’s state is its charge level; namely, the number of electrons in the floating gate. Because this affects the behavior of the transistor, it is necessary to increase the threshold voltage (Vth) required to activate it. Therefore, a cell’s state can be determined by measuring the voltage needed to activate the transistor.

In SSDs, read and write operations are performed in bulk of cells, called pages, ranging in size from 2KB to 8KB [Sam14]. Usually, in-place page updates are prohibited. When a logical page is updated, the update is written to the new clean physical page, and the old copy location is marked as an invalid page. These invalid pages must be erased and then programmed again. Hence the traditional Program/Erase cycles (P/E). The erase operation is performed in bulks of pages, called blocks, ranging in size 1MB to 3 MB [Sam14].

Over time, the SSD drive runs out of clean pages. To provide extra clean pages, the SSD employs a mechanism called garbage collection. It does so by taking blocks of invalid pages and cleaning them by an erase operation. If such blocks are available, it will choose blocks with the minimum number of valid pages to copy them to a new block and then erase the block. Every time a block is erased, its electrons are drained and stress is exerted on the cells. Cells constantly lose their electrons due to voltage leaks to neighboring cells [CHMM12]. Eventually the cells wear out from erase operations, once
this happens, they can no longer retain their desired charge level to be read reliably.

The Flash Translation Layer (FTL) is a software layer built on raw flash memory. It hides special characteristics of flash memory from upper file systems by emulating a block device like HDD. It also contains mapping from a logical page address to a physical location. Moreover, it carries out garbage collection and selects which physical block to use each time data is written. This selection process uses a technique, called wear leveling, which is designed to extend the life of solid state storage devices by arranging data so that erasures and writes are distributed evenly across the medium. There are two strategies of SSD wear leveling: dynamic and static. Dynamic wear leveling pools erased blocks and selects the block with the lowest erase count for the next write. Static wear leveling, on the other hand, selects the target block with the lowest overall erase count, erases the block if necessary, writes new data to the block, and ensures that blocks of cold data are moved when their block erase count is below a certain threshold.

To help manage the garbage collection process, manufacturers decided to add additional storage to SSDs as an additional buffer to allocate new blocks for write operations. This additional storage, called overprovisioning space, is not visible to the host. This can decrease the number of writes to the SSD blocks as well as improving the wear-out rate to blocks, since the write load is distributed on a larger number of blocks.

NAND flash technology has advanced in 2 methods with the principal goal to increase SSDs capacity. The first method is by shrinking NAND cells in size, with the latest iteration being 16nm in size [HPG+14]. The other is by increasing the number of states a single cell can hold. At first, there were only single level cells (SLC), each cell able to store only one bit of information, with the states of charged and uncharged. Later cells that could hold more than one bit were created, called multi-level cell (MLC) devices, and triple-level cell (TLC) devices. This breakthrough was made possible by storing and probing different voltage levels, making it possible to represent more than two states, which meant an increase in storage density and lower cost per bit. However, using high order level cells comes with a price. Additionally, this makes SSDs more vulnerable to voltage leaks [PDWZ12]. Thus, the SSD lifetime is cut shorter. Also, more complex read/write algorithms are now required to handle such cells. There are also costs in performance [GDS12]. As Flash technology evolved, the lifetime of SSDs was cut short from 100,000 (P/E) cycles in 50nm SLC, to merely 3,000 (P/E) cycles in MLC [YPL+11] and as little as 500 (P/E) cycles in TLC SSD devices [GPNG11]. This issue has sparked an active area of research in SSDs, to prolong the SSD lifetime as the technology continues to evolve.

2.2 Write-Once Memory Codes

Once, write once media, e.g. punch cards and optical disks, were common. They represented a value by changing a value at specific positions from a default of ”0” to an
irreversible value of "1". To use such media more than once, a coding method, called write-once memory (WOM) coding, was first proposed by Rivest and Shamir [RS82]. This code can support 2 writes, a first write and a second rewrite. This example had an expansion factor of 1.5, meaning it inflates the required storage size to store a word by 50%. They give a simple WOM code example, which maps a 2 bit logical word to a physical data word of 3 bit. This mapping is presented in Table 2.1.

Table 2.1: WOM Code Example

<table>
<thead>
<tr>
<th></th>
<th>1st write</th>
<th>2nd write</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
<td>111</td>
</tr>
<tr>
<td>01</td>
<td>001</td>
<td>110</td>
</tr>
<tr>
<td>10</td>
<td>010</td>
<td>101</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>100</td>
</tr>
</tbody>
</table>

For example, if the first word to be stored is 10, then 010 is written, by programming only one cell. If the second message is 01, then 110 is written, programming the first cell as well. Note that bits can only be changed irreversibly from a value of "0" to a value of "1" without erasing, which is why a special encoding is required. If both first and second words are identical, then the cells do not change their value between the first and second writes.

WOM codes are an active area of research, and many WOM code constructions were proposed [LKY12, JFL10, JCS12]. A construction specifies a given number of cells: the number of writes achievable, the amount of information that can be added in each write and how each successive write is encoded. The goal in the design of WOM codes is, for a given number of physical bits to write on to maximize the number of writes and number of values that could be written.

The limitation asymmetry between write and erase operations in SSDs has sparked the study of WOM codes as a potential tool for improving the SSD lifetime and access efficiency. WOM codes allow in-place updates with no need for prior erasing of the cells. A way to specify a WOM code is the integer number of user writes \( t \) it guarantees before an erase operation is required. The larger \( t \) gets, the more in-place writes the device supports. Thus, the added in-place writing capabilities can improve the lifetime of a standard SSD. Use of WOM-codes must not affect the SSD interface and transparency to the user, to remain compatible with existing systems.

The use of WOM codes in SSD inflates data written in second writes by 29%, a theoretical lower bound [RS82]. This means a page size information written in second write requires more than one physical page to be written on. Prior work [YYS15] was done to integrate WOM codes without modifying the drive’s exported storage capacity or page size. Second writes were written on two used physical pages in different physical blocks, so that they are read and written in parallel, avoiding additional latency. Invalid page data was pre-fetched in advance to avoid additional delays. This invalid data is used in the encoding of the information written in second writes over those invalid pages.
Moreover, the additional storage required for inflated second writes is provided by the SSD’s overprovisioning space to keep the exported storage capacity the same.

This work will focus on utilizing WOM codes in SSDs serving as caching devices for primary storage. In addition, we remove the constraint of keeping the exported capacity to remain the same, with the goal to extend even further the lifetime of SSDs.

2.3 Flash Based Cache

To alleviate the limitations of HDD slow speed and SSD high cost, hybrid storage solutions were developed that bridge the gap between them. For example, hybrid disks, composed from a conventional magnetic disk with NAND flash memory cache, are commercially available [Del12, Sea17]. We consider hybrid storage that uses NAND flash memory based SSDs as a non-volatile cache and traditional HDDs as primary storage. The goal of a hybrid storage solution is providing SSD-like performance at the HDD-like price. To achieve this, it requires near-optimal management of the flash memory cache. SSD-based caches were integrated to caching tier of data centers and were widely studied on various types of workloads [KM06, LML+13, CMB+10, LLM+12, MCLA11].

However, SSDs exhibit worse performance for random writes than sequential writes, due to the unique physical characteristics of NAND flash memory [CKZ09]. A random write causes excessive internal fragmentation. Increase in fragmentation results in frequent garbage collection inside the SSD, which incurs more block erases and thus drastically increases the average write latency while reducing the lifetime of the SSD. Cache access patterns are often random, and to better improve the usage of an SSD-based cache, many studies were performed to use and mitigate the limitations of flash memory.

FlashTier [SSZ12] extends the general block interface for flash-based cache devices to eliminate the cost of address mapping in the cache manager. It silently evicts blocks from a cache during the garbage collection process. [CMB+10] proposed a temperature-aware replacement policy, TAC, to manage the SSD-based buffer pool for DBMS. BPLRU [KA08], and LBCLOCK [DSDL09] are examples for using DRAM buffers to improve the random write performance of Flash SSDs. [KRM08] suggested that by splitting Flash based disk caches into read and write regions, overall performance and SSD lifetime can be improved. Further works also suggested splitting the data to hot and cold regions [JNI14, KRM08, MKC+12].

2.4 Cache Algorithms

In this chapter, we describe related works that are behind the basic premise of WOMAC. [KRM08] proposes using flash memory as a disk cache in web servers. Web servers serve requests by clients and can be improved by using flash memory since it has fast read and write operations and high throughput. However, for write-intensive
applications, since flash memory is unable to perform in-place updates, flash memory becomes the system performance bottleneck. To solve this performance problem, the traditional approach is to write data to free pages and invalidate current pages. In this case, garbage collection (GC) is always applied to reclaim these invalid pages. However, GC is time and power consuming. To improve the SSD efficiency of GC, [KRM08] has proposed to split the flash cache into read and write caches. A read cache is designed for serving read requests and a write cache for serving write requests. OP-FCL [OCLN12] was studied as a way to apply flash memory as a disk cache. Like the disk cache of [KRM08], OP-FCL considers GC efficiency and cache performance to dynamically partition disk cache into a read cache, a write cache, and an overprovisioned space (OPS). OP-FCL achieves the trade-offs between GC efficiency and cache performance to overcome the write performance bottleneck. OP-FCL uses a cost model to dynamically adjust the sizes of the three areas, that is, the read cache, write cache, and the OPS according to the workload for optimal hybrid storage performance.

In Adaptive Replacement Cache (ARC) [MM03], which was designed for RAM memory, the cache space is divided into two lists, denoted as T1 and T2. T1 stores metadata of one-time accessed blocks, whereas T2 stores metadata of blocks that have been accessed at least twice. The division between them is dynamically adjusted to balance between recency and frequency. ARC keep track on recently evicted pages using ghost queues. A ghost queue is a list of recently evicted pages metadata. If the ghost queue is at maximum size, the item in the tail of the queue is removed upon an insert to the head of the queue. ARC maintains two ghost queues B1 and B2, to keep track of blocks evicted from T1 and T2, respectively. The space division between T1 and T2 is tuned based on hit rates in B1 and B2. Roughly speaking, frequently accessed blocks can stay longer than LRU.

Lazy Adaptive Replacement Cache (LARC) [HWF+16] is a novel algorithm to improve the performance and endurance for SSD-based disk caches. LARC tries to keep seldom accessed blocks out of the cache. It uses a ghost cache to keep a record of block access history and to identify potentially popular blocks. Cache replacement is triggered when a popular block is identified. As a result, blocks in a cache tend to be more popular and can be preserved longer to improve the hit rate. Meanwhile, LARC avoids unnecessary cache replacements and hence reduces the amount of data written to the SSD.

WOMAC uses a segmented ghost queue like in ARC, called ghost buffer, to simultaneously measure hit rates of various cache sizes. Keeping information of evicted data in the ghost buffer makes it possible to measure the hit rate of a cache larger than the actual cache size. In addition, WOMAC uses the additional writes gained from using WOM to store newly read page data as testing ground. Like in LARC, when a popular page is identified, it triggers a cache replacement in the regular writes area.
2.5 Problem Statement

Now that we have reviewed the SSD devices, lifetime problem and the Write Once Memory coding, we can formally state the problem of combining the two. Given that an SSD device supports in-place physical writes that only increment the cell levels, implement a dynamic caching scheme for a hybrid system that finds the workload dependent optimal balance between cache size and the SSD lifetime. Specifically, we propose a cost model that is used to determine the optimal caching space between first writes and second writes cache sizes for a given workload. In our solution, caching space is divided into a read cache and a write cache. The algorithm decides dynamically the sizes of these four areas based on cost model and recent statistics.
Chapter 3

Cache Design

The principal idea underlying this work is when using WOM-codes for write requests can be fulfilled by using invalid pages to write valid data. This reduces the rate of erase operations, and the SSD (P/E) life cycle is extended. Keeping in mind WOM codes overheads, we need to make sure that their benefits are not canceled out by the reduction in the cache space. That means more cache misses and more costs on read and write requests from the hybrid system, in which the SSD is a caching device and the HDD is the primary storage device. Our hypothesis is that finding the balance between them will achieve improvement in the SSD lifetime at a small cost to the cache hit rate.

3.1 Design Overview

The main idea of our proposed architecture is that inflated second writes will be mainly used for rarely read pages. Inserting such pages to caches, will not increase the hit rate, and would incur a cost in SSD write operations. A read request miss of such a page will lead to the eviction of a page more likely to be read and thus decrease the hit rate. More, frequent write requests to the same logical page, results in invalid pages and will shorten the SSD lifetime. To alleviate this issue, frequent updates were performed to buffer in memory [KA08, DSDL09], before written to flash, but SSD memory is limited in size.

In our proposed scheme, we suggest using the second write area in the write cache as an additional buffer. This buffer will consume some of those frequent writes and rarely requested reads, to reduce the number of erase operations. For other write requests, we do not use WOM codes, thus saving the storage overhead that they incur. Figure 3.1 shows the cache division of reading/writing blocks using WOMAC, which are further described in the following chapters. If we correctly select write requests to be written in second write, we should enjoy a longer SSD lifetime and reduced costs of WOM-codes overheads.

We devised cost models for SSDs and HDDs cost like those proposed in [OCLN12]. The cost model is used to determine the best division between read cache and write
In this chapter we described the read cache structure and then, the write cache structure. In the next chapter, we derive the read/write cost model used to determine the optimal read cache and write cache sizes and the sizes of their sub-areas.

![WOMAC cache areas, divided into a read cache and a write cache. Each one is divided further into first and second writes areas. Each cell depicted the data it stores in the area, and the data structures that manages its entries. Further description of data flow is given in the following chapters and in figure 3.4.]

**3.2 Read Cache Design**

The first issue, when designing such a system, is efficiency when reading and writing data as inflated second writes because they take more than one physical page of storage. In [YYS15], two blocks from two different planes were paired. Two methods to perform read and write operations on them were performed, parallel and sequentially. The results show that page operations performed in parallel are much more efficient then performed sequentially. That is why we have decided to statically bind two physical different blocks from different planes to one logical block. The idea of logical blocks was introduced in [KM06].

We use logical blocks in two modes. In the first, for regular first writes, the data is written sequentially on the first block and then on the second block. Once a logical block is filled it changes state to sealed from any additional first writes until an erase operation. The second mode is when the logical block is recycled for second writes which are done in parallel like in [YYS15]. When a logical block is erased, both physical blocks perform an erase operation, remain under the same logical block and are used again for first writes. This is meant to keep both blocks at the same wear level and to reduce the computational complexity of matching blocks with the same wear level.

The second issue was what we should put in the second writes area of the read cache. Ideally, data that is not likely to be read again should not be written to the cache. Such
data pollutes the cache, and incurs cost in SSD writes with no benefits. However, since it is not possible to predict future workload, we could use the second writes area to estimate pages that are likely to be requested again. We look to Adaptive Replacement Cache (ARC), as a solution for this issue. We further discuss ARC in chapter 2.

In our design, we use logical block LRUs, and ghost queues like ARC, but we will use our cost models to tune the amount of space each one of the lists gets. In addition, we mark first writes as the list T1\textsubscript{R}, which stores blocks of pages that were accessed at least twice and would likely be accessed again soon. Its matching ghost queue is B1\textsubscript{R}, that contains evicted records out of T1\textsubscript{R}. The list T2\textsubscript{R} stores pages in second write and were accessed only once. T2\textsubscript{R} has the ghost queue B2\textsubscript{R}, to record evicted pages. To simulate various cache sizes simultaneously, on each list and matching ghost queue, we use N-segmented ghost buffer. In other words, we divide the ghost buffer into N-segments corresponding to N cache sizes and thus, hit rates of N cache sizes can be obtained by combining the hit rates of the segments. From the hit rates of N cache sizes, we obtain the read/write hit rate curves by interpolating the missing cache sizes. These curves will be used in our cost model, as described in chapter 4. Figure 3.2 depicts an 8-segmented ghost buffer on T2\textsubscript{R} and B2\textsubscript{R}. The eviction policy from T1\textsubscript{R} and T2\textsubscript{R} is a small modification of LRU, but is optimized to take care of wear leveling and blocks full of invalid data. First, we evict blocks that have no valid pages, which could be the case, if their pages were invalidated in a write operation and written to the write cache, or moved from T1\textsubscript{R} to T2\textsubscript{R}. Second, we evict blocks that are younger by more than 10% of the total lifetime than the average lifetime of all blocks. We evict such a block to perform wear leveling. It also helps to evict pages from the cache that were not accessed recently, but remained due to a popular page in their block. Finally, if none of the above holds, the cache will evict the logical block end of the LRU list. Upon eviction, either the block is recycled for second writes, or it is erased for the use of first writes. Moreover, the record of the evicted block is added to the corresponding ghost queue. The ghost queues size and segment sizes of ghost buffers are open questions to be investigated. When a ghost queue gets full, the last block is evicted. Ghost segments are not affected directly by changes to T1\textsubscript{R} to T2\textsubscript{R},
since they are used to mark hit locations within the list.

Since all new writes are done to T1_R, which contains only blocks in second write mode, it stands to reason that we will run out of blocks from the used block pool. In this case, we resupply the used block pool, either by triggering an early eviction from T2_R or from T1_W, which is the first writes area of the write cache. When a cache hit occurs in T1_R, the block is moved to the MRU position of the list, and the page is written to a regular first write in T2_R, and invalidated in T1_R. This is done under the assumption, that the page contains hot data, since it was read twice recently, and could be read again soon. This page should be in the same location with other hot data for better performance [JNI14, KRM08, MKC+12]. When there is a hit in T2_R, the block is placed at the MRU position of the list and the hit is marked in the segmented ghost buffer. Also, when there is a hit at one of the ghost queues, instead of writing the page again as a second write in T1_R, it will be written to T2_R as a regular first write, since it is a popular page. The read cache logic is described in figure 3.4a and its detailed data structure is described in figure 3.3. Also, the read cache algorithm is described in Algorithm 3.1.

![Read Cache Diagram](image)

Figure 3.3: Read cache architecture employs two LRU lists of logical blocks and two ghost queues. The first time that page 11 is read, there is a cache miss and it is written to T1_R, as a second write. When page 11 is read again, the page is written to T2_R as a first write and the copy in T1_R is invalidated.

### 3.3 Write Cache Design

Previous studies have shown that separating read and write requests in flash memory storage have a significant effect on performance [KRM08]. We have divided the flash cache space into two areas, namely a write cache and a read cache.

In the write cache, the structure is like the read cache, however the semantics are different. We first assume that a write request contains cold data and consider it as hot data in case of a write cache hit or ghost hit. We took an opposite approach in terms of
**Algorithm 3.1** Read Cache Algorithm

1. if read request miss then
2. read page from HDD
3. if page has a ghost queue hit then
4. increase counter for ghost buffer segment
5. write page in T1_R // as first write
6. else
7. write page in T2_R // as second write
8. end if
9. else //a read hit
10. read page from SSD
11. increase counter for ghost buffer segment
12. if hit was in read cache second write area then
13. write page in T1_R. // as first write
14. end if
15. if hit was in write cache then
16. increase counter for hit rate in write cache
17. end if
18. end if
19. if T1_R or T2_R are full then
20. allocate new logical block to the full list
21. run Choose Victim Block Algorithm on the full list //Algorithm 3.2
22. evict the chosen victim block
23. move victim block meta data to matching ghost queue
24. if matching ghost queue is full then
25. evict last ghost block from ghost queue //ghost segments are not affected
26. end if
27. end if

**Algorithm 3.2** Choose Victim Block

1. if there is a block with no valid pages then
2. return a block with no valid pages
3. else
4. get an active block with minimum erase count
5. if this block erase count ≤ 0.9 × average block erase count then
6. return the block with minimum erase count
7. else
8. return the last block in the LRU
9. end if
10. end if
(a) read: on a read miss, the page is read from HDD, and will check the ghost cache if it has a record. If it has, the page will be store the block in first write area, or else in second write area. If there is a hit in the second write area, the page is read and is promoted to first write area. On a hit to first write area, the page is read.

(b) Write: on a write hit, the page is written to the second write area. If it has a ghost record, the page is written to second write area. If no records of the page exist, the page is written to first write area.

Figure 3.4: Data flow in WOMAC

storing hot and cold data. If there is write cache miss, it will be written in first writes area, marked T1_W, otherwise, it is written in area T2_W, the second writes area. The main idea is to use the additional writes, gained by using WOM-codes, to pay the costs of storing hot data. Those write requests usually wear out the SSD cache faster than cold data write requests. The eviction policy is the same as in the read cache, with the addition that evicted blocks with valid pages are de-staged to HDD and their meta-data is also recorded in the ghost queue.

In addition, there are segmented ghost buffers in the write cache that act the same as in the read cache to estimate the hit rates on the variable cache size. Also, when serving a write request any previous copy or ghost record, is discarded and marked invalid. The write cache logic is described in figure 3.4b and its detailed structure is described in figure 3.5. and the write cache algorithm is described in algorithm 3.3.
Figure 3.5: Write cache architecture employs two LRU lists of logical blocks and two ghost queues. The first time that page 1436 is written there is a cache miss, and it is written to $T_1^W$, as a first write. When page 1436 is getting a write request again, the copy in $T_1^W$ is invalidated and is written to $T_2^W$ as a second write. Additionally, when there is a write request on page 949 and there is a hit in ghost queue $B_1^W$, the page is written to $T_2^W$, and the ghost record is invalidated.

**Algorithm 3.3 Write Cache Algorithm**

1. if write request miss then
2.   if page has a ghost queue hit then
3.     increase counter for ghost buffer segment
4.     write page in $T_2^W$ // as second write
5.     invalidate ghost queue entry of the page
6.   else
7.     write page in $T_1^W$ // as first write
8.   end if
9. else// write request hit
10. increase counter for ghost buffer segment
11. write page in $T_2^W$ // as second write
12. invalidate prior version of the page
13. end if
14. if $T_1^W$ or $T_2^W$ are full then
15. allocate new logical block to the full list
16. run Choose Victim Block Algorithm on the full list //Algorithm 3.2
17. de-stage valid pages to HDD from victim block
18. move victim block meta data to matching ghost queue
19. if matching ghost queue is full then
20.   evict last block from ghost queue //ghost segments are not affected
21. end if
22. end if
Chapter 4  

Storage Cost Model

In this chapter, we review the cost models for SSDs and HDDs as derived from [OCLN12, WZB04], then we present our addition to costs when separating first and second writes for a read cache and then a write cache. Finally, we derive our combined storage cost model for SSD caches.

4.1 Read Cache Cost Model

<table>
<thead>
<tr>
<th>Request type</th>
<th>Cost</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read hit in 1st write area</td>
<td>$C_{\text{READ}}$</td>
<td>$HR_{\text{1ST}}(s,r)$</td>
</tr>
<tr>
<td>Read hit in 2nd write area</td>
<td>$C_{\text{READ}} + C_{\text{1ST}_PW}$</td>
<td>$HR_{\text{2ND}}(s,r)$</td>
</tr>
<tr>
<td>Read request miss</td>
<td>$C_{\text{DR}} + C_{\text{2ND}_PW}$</td>
<td>$1 - HR_{\text{1ST}}(s,r) - HR_{\text{2ND}}(x,r)$</td>
</tr>
</tbody>
</table>

Table 4.1: Request types in the read cache.

Read and write operation on HDD are characterized by seek time and rotational delay. Assume that $C_{D,RPOS} + C_{D,WPOS}$ are sums of the average seek time and the average rotational delay for HDD reads and writes, respectively. Also, assume that $P$ is the data size in bytes and $B$ is the bandwidth of the disk. The data read and write costs of an HDD are derived as $C_{\text{DR}} = C_{D,RPOS} + \frac{P}{B}$ and $C_{\text{DW}} = C_{D,WPOS} + \frac{P}{B}$, respectively.

Assume $C_{\text{PROG}}$ is the time (cost) of writing a single page to SSD. When cleaning a block, it incurs the cost of cleaning for the page. Since cleaning must be done after a write on a page, the cost of erase operation goes to first writes, but not second writes. We mark the cost of a write to cache in a first write as the page program time with the overhead costs of cleaning that page. Since cleaning is a block size operation, it will be the average cost of cleaning a page from a block with $N_P$ pages. $C_E$ is the erase operation cost of a flash memory block.

$$C_{\text{1ST}_PW} = C_{\text{PROG}} + \frac{C_E}{N_P}$$

In our extension to the cost model, we mark the cost of serving a write request...
to cache in the second write as the program time of 2 physical pages in parallel. We do not consider a overhead of reading these pages in advance to build the WOM-code expression on the written page. It is done when the system is idle, and in the worst case, however, it may not complete before any read or write requests arrives. In [Yadgar2015], prefetching never delayed reads or first writes to the point of degrading performance.

\[ C_{2ND,PW} = C_{PROG} \]

On a read request, the storage first checks whether the requested data is in the cache. If it is, the system reads it from the SSD cache and transfers it to the host system. If it is not in the cache, the system reads the data from the HDD, stores it in the SSD cache and transfers it to the host system. If the SSD cache is full with data, then it must evict the least valuable data to make room for the new data. We use a modified LRU (Least Recently Used) replacement policy to select the least valuable data, as described in chapter 3.2. In the case of a read request to the cache, the selected data need only to be invalidated, which can be done essentially for free. We introduce parameter \( s \), which is the second write space ratio within the read cache space, where \( 0 \leq s \leq 0.95 \). A note: we limit the maximum portion of a second write in the read cache, preserve space for frequent read.

In the original cost model, it was assumed that the SSD read cache was a single space with a hit rate curve. In our extension, each read cache area has its own hit rate curve based on its size. Let us assume that \( HR_{1ST}(s) \) and \( HR_{2ND}(s) \) are the cache read hit rates for the read cache, for a given cache sizes of 1st write and 2nd writes areas, respectively. With rate \( HR_{1ST}(s) \) the system reads the requested data from the cache with cost \( C_{READ} \), the page read operation cost. When reading a page in second write, there is a need to read the page from 2 physical pages in which they reside and decode them to the original value the page was written in. We assume a negligible cost in decoding it, so the only cost of reading a page in second write is \( C_{READ} \), with the hit rate of \( HR_{2ND}(s) \). Furthermore, on a hit, we promote the page to the first write area, hot data area, by writing it with a cost of \( C_{1ST,PW} \) and invalidating the copy in the second writes area.

With rate \( 1 - HR_{1ST}(s) - HR_{2ND}(s) \), the system reads data from disk with cost \( C_{DR} \) and stores it into SSD with cost \( C_{2ND,PW} \) to the second writes area. We also assume a negligible cost in encoding data to WOM code for second write. Then, \( C_{HR}(s) \), the read cost of the hybrid storage system with a read cache with second writes, is as follows.

\[ C_{HR}(s) = HR_{1ST}(s) \times C_{READ} + HR_{2ND}(s) \times (C_{READ} + C_{1ST,PW}) + (1 - HR_{1ST}(s) - HR_{2ND}(s)) \times (C_{DR} + C_{2ND,PW}) \] (4.1)

Let us now take the SSD read cache size into consideration. For a given full cache size, \( S_f \), the read cache size \( S_r \), and the write cache size \( S_w \). We introduce a new
parameter \( r \), which is the read cache ratio within the full cache space, where \( 0 \leq r \leq 1 \).

We can approximate from \( r \), the read cache size \( S_r = r \times S_f \) and the write cache size \( S_w = (1 - r \times S_f) \). These sizes are approximated values as they do not consider a small pool of cleaned blocks and a small pool of used blocks, and are available for read and write cache, for immediate write operations. If \( r \) is 1, then all caching space is used as a read cache and, if it is 0, all caching space is used as a write cache. With the hit ratio counters, we can measure and draw a curve of the read hit rates of the read cache first writes and second writes, which now has two parameters \( s \) and \( r \).

Let us mark \( HR_{1ST}(s, r) \) and \( HR_{2ND}(s, r) \) as the cache read hit rates for the read cache first and second write areas, respectively. Then, the read cost of the hybrid storage system is now modified as follows.

\[
C_{HR}(s, r) = HR_{1ST}(s, r) \times C_{READ} + HR_{2ND}(s, r) \times (C_{READ} + C_{1ST, PW}) + (1 - HR_{1ST}(s, r) - HR_{2ND}(s, r)) \times (C_{DR} + C_{2ND, PW})
\]

### 4.2 Simple Write Cache Cost Model

<table>
<thead>
<tr>
<th>Request type</th>
<th>Cost</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write hit in 1st write area</td>
<td>( C_{2ND, PW} )</td>
<td>( HW_{1ST}(x, r) )</td>
</tr>
<tr>
<td>Write hit in 2nd write area</td>
<td>( C_{2ND, PW} )</td>
<td>( HW_{2ND}(x, r) )</td>
</tr>
<tr>
<td>Write miss</td>
<td>( C_{1ST, PW} + C_{READ} + C_{DW} )</td>
<td>( 1 - HW_{1ST}(x, r) - HW_{2ND}(x, r) )</td>
</tr>
<tr>
<td>Read hit in 1st write area</td>
<td>( C_{READ} )</td>
<td>( HW_{1ST}(x, r) \times H_{1ST, R} )</td>
</tr>
<tr>
<td>Read hit in 2nd write area</td>
<td>( C_{READ} )</td>
<td>( HW_{2ND}(x, r) \times H_{2ND, R} )</td>
</tr>
</tbody>
</table>

Table 4.2: Request types in the write cache.

Unlike the read cache, which contains data that has recently been read but never written back, the write cache keeps data that has recently been written, but not yet de-staged. Therefore, data in the write cache is dirty and must be written to the HDD when evicted from the cache. If there is a write to a page that exists in the read cache, it is regarded as a write miss because clean data is only invalidated in the read cache. We also discuss the case of reading data in the write cache later. In the following cost model derivation, we assume only write-back policy for the write cache. For the write-through policy, our model needs to be modified to reflect the additional writes to HDD, that happens for each write to the SSD cache. This will result in a far less efficient caching system.

There can be five types of requests to our write cache design, presented in table 3 with their cost and rate. The first two are write hit requests with existing data in the cache. The hit can be in either the first writes area or in the second writes area. If there is a hit in the first write area, the old data becomes invalidated and the new data is written to the second writes area in the write cache with cost \( C_{2ND, PW} \). If the hit was in the second writes area, it is written again there with a cost of \( C_{2ND, PW} \).
The third type, is a write miss request, which is a write request to data that does not exist in the write cache. In this case, the cache replacement policy selects victim data that should be read from the first writes cache area and de-staged to the HDD with cost $C_{\text{READ}} + C_{\text{DW}}$ to make room for the newly requested data. Note we are assuming the system is in a steady state, meaning when the cache is full, the LRU data is evicted from the cache to accommodate newly arriving data. After evicting the data, the system writes the new data to the write cache in first writes area with cost $C_{1\text{ST.PW}}$. We note that de-staging data from a second writes costs $C_{\text{READ}} + C_{\text{DW}}$, since the cache reads two physical pages in parallel.

The last type of request is a read request, which is a read request to existing data in the write cache. This happens when a read request is for data that is already in the write cache. In this case, the request can be satisfied with cost $C_{\text{READ}}$. Note that there is no read miss request to the write cache because read requests to data not in the cache are handled by the read cache.

We introduce parameter $x$, which is the second writes space ratio within the write cache space, where $0 \leq x \leq 0.95$. The write cache size is determined by one additional parameter $r$, the read cache ratio within the full cache space. Therefore, $1 - r$ is the write cache ratio within the full cache space. We also limit the number of pages in second write in this case.

Let us assume that $HW_{1\text{ST}}(x, r)$ and $HW_{2\text{ND}}(x, r)$ are the cache write hit rates for the write cache for given cache sizes of 1st write and 2nd writes areas, respectively. At the rate $HW_{2\text{ND}}(x, r)$, there is a hit in the first write area, and data is written to second write with a cost $C_{2\text{ND.PW}}$. At the rate $HW_{1\text{ST}}(x, r)$, the page is written again to the second writes area with a cost of $C_{2\text{ND.PW}}$. Otherwise, with a rate of misses, $1 - HW_{1\text{ST}}(x, r) - HW_{2\text{ND}}(x, r)$ the write request does not have a hit in the write cache. Servicing this request requires reading and evicting existing data and writing new data to the write cache. Hence, the cost is $C_{1\text{ST.PW}} + C_{\text{READ}} + C_{\text{DW}}$. In summary, the write cost of the hybrid storage system can be given as follows.

$$C_{\text{HW}}(x, r) = (HW_{1\text{ST}}(x, r) + HW_{2\text{ND}}(x, r)) \times (C_{2\text{ND.PW}}) + (1 - HW_{1\text{ST}}(x, r) - HR_{2\text{ND}}(x, r)) \times (C_{\text{READ}} + C_{\text{DW}} + C_{1\text{ST.PW}})$$

(4.3)

So far, we have described a purely write cache with no read request hits. Now let us consider the case of a read hit in the write cache. For simplicity of our model, we only approximate the read hit rate in the write cache. Assume that $H_{1\text{ST}, R}, H_{2\text{ND}, R}$ are the portion of hits, which are read requests, in the write cache for first and second writes areas respectively. Then $1 - H_{1\text{ST}, R}, 1 - H_{2\text{ND}, R}$ are the portion of hits, which are read requests, in the write cache for first and second write areas respectively. Then, the read hits are satisfied with the cost of reading a page. Now we can calculate the
cost for both read and write requests in the write cache.

\[
C_{HW}(x, r) = \left( H_{2ND}(x, r) \right) \times \left( H_{2ND,R} \times C_{READ} + (1 - H_{2ND,R}) \times C_{2ND,PW} \right) \\
\times \left( H_{1ST}(x, r) \right) \times \left( H_{1ST,R} \times C_{READ} + (1 - H_{1ST,R}) \times C_{2ND,PW} \right) \\
\times \left( 1 - H_{1ST}(x, r) - H_{2ND}(x, r) \right) \times \left( C_{READ} + C_{DW} + C_{1ST,PW} \right)
\]

(4.4)

### 4.3 Combined Read and Write Cache Cost Model

Now, let \( IO_R \) and \( IO_W \), be the rate served in the read and write caches among all requests respectively. For example, of a total of 1000 requests, if 750 requests are served in the read cache and 250 requests are served in the write cache, then \( IO_R \) is 0.75 and \( IO_W \), is 0.25. Then we can derive that \( C_{HY}(x, s, r) \) is the overall access cost of the hybrid storage system, that has separate read and write caches in their respected parts of first and second write areas.

\[
C_{HY}(x, s, r) = IO_R \times C_{HR}(s, r) + IO_W \times C_{HW}(x, r)
\]

(4.5)

With this cost model defined, we can find the value of the three parameters \((x, s, r)\) that, can bring the minimum overall access cost of the hybrid storage system based on current workload. It would be difficult to formulate a drawing of such a function because it has 4 dimensions to display. To find the values of \((x, s, r)\) that bring the overall access cost function to a minimum, we will use an iterative approach.

On each epoch, a period of thousands of request a partition algorithm runs to evaluate the optimal partitioning between the four areas. To obtain the optimal \( x, s \) and \( r \), we extended the algorithm used in [OCLN12] to optimize all three parameters. The computational overhead for executing this algorithm should be small because they run once every epoch and the calculations are just simple arithmetic operations. The algorithm is a simple iterative approach to scan all 3 parameters for the minimum cost function value. Each parameter is scanned from its minimum value up to 1.0, and each time is increased by a \textit{step} value. The size of an epoch is an open question. After the run of this algorithm, the cache should evict data gradually from each area to fit their new partition size.
Chapter 5

Implementation Challenges

We attempted to evaluate the performance of the WOMAC cache scheme, using DiskSim [BSSG08]. DiskSim has a cache module that models the cache used in modern storage systems. This attempt was not completed. In this chapter, we will review DiskSim and Microsoft’s SSD Extension and explain the challenges that prevented us from completing this work.

5.1 DiskSim

This work relied on DiskSim v4.0 to provide a disk system simulation. DiskSim was first developed at the University of Michigan, but was enhanced at Carnegie Mellon University. DiskSim is written in C and requires only a few POSIX interfaces. DiskSim can be configured to use any supported device as the cache. It does so, using the DiskSim cachedev structure, which allows additional cache configurations such as size, write scheme and flush policy. When idle, the cache triggers an eviction of dirty pages from caching device to the primary storage, to reduce the latency costs of the write operations upon eviction. This allows the developer to declare an SSD as a cache for an HDD.

Version 4.0 of DiskSim was released in late 2008 but has been updated by third parties since then via extensions and patches.

5.2 Microsoft SSD Extension

Microsoft researchers wrote an SSD model extension for DiskSim in 2009 [APW+08], so that they may better understand SSD performance. Their work explores the internal organization of SSDs, and their expansion reflects upon the various configurations they tested. DiskSim v4.0 does not natively support SSDs, but this extension introduces a model for testing SSDs.

The model is modularized and implements the components of FTL, such as garbage collection, wear-leveling policies, indirect mapping and others. It also contains SSD
parts common to most SSDs like: elements, dies, planes, blocks, and pages. Since the original model contained only a regular program / erase cycle for block lifetime, we have augmented the model’s current implementation to include second writes in [YYS15].

In that work, a block life cycle was extended to include phases for second write. First writes are unmodified and the information is written exactly like in the original SSD model. When all pages in the block are invalidated, the algorithm can either choose to clean the block and use it again for first writes, or it can recycle it to be used for second writes. In second writes the logical page is written on two physical pages with the same offset in two recycled active blocks. For a page written in a first write, the page map points to the physical location of this page. For a page written in a second write, the map points to the physical location of the first half of this page, in plane 0. Thus, if a page mapping points to a page in plane 0 and the corresponding block map entry is non-null, the page is stored on two physical pages, whose addresses are now known.

Additional modifications for the SSD model included: the probability of failure in case of encoding error for second writes, an upper limit number of blocks in second writes and an age limit when a block can no longer support second writes.

5.3 The Attempt to Implement WOMAC

Implementing WOMAC is especially challenging because the cachedev structure, does not support a needed capability of a cache, the ability to store in a cache a wider range of addresses than what the caching device can store. It had only a one to one bitmap for marking cached data and another one to one map for dirty data. e.g. if the primary storage device is size 4GB and a cache device is 1GB, then the cache would not handle requests beyond the 1GB addressing space. When cachedev would receive a request within the 1GB space, it passes the relevant requests to the caching device and, made a mark in mappings.

To solve this issue, our approach was to create a structure to allocate and hold the mapping between the primary storage address and the logical block address (LBA), within the acceptable range of the cache device. When the SSD is the cache device, there is an additional mapping between this LBA and the SSD’s physical address. Additionally, a list of cached addresses was added to track the use of the cached data, and to evict data, when the number of mappings have reached a certain limit. Two eviction policies : LRU and RANDOM, were set in the simulator parameters file.

During implementation, there were many challenges to solve. E.g. the use of SSD as a caching device which was not fully compatible with cachedev data structure and required additional workaround and fixes. One challenge, for example, was a synchronization issue when handling two read requests that are close in intervals. In this case, cachedev first allocates LBA mapping to the first request, then it requests a read from the primary storage, then passes a write request to the SSD as the caching device and marks the
entry in the LRU list. The issue is that the second read request entered the cachedev structure before the read request from the primary storage has finished thus resulting in an invalid read request from SSD.

The main challenge was to implement the WOMAC caching scheme in the cachedev structure. Since WOMAC needs to handle directly data placement of data in first or second writes in the SSD model, it requires full knowledge of the SSD model state. It would also need to take over all the FTL functions including wear leveling and garbage collection. Since cachedev and SSD model are two separate entities that pass from one to the other messages in the form of IO request objects, we attempted to extend the request object to include additional fields for cachedev to suggest placement of a write request in first or second write to the SSD model. We added fields with a reply from SSD model on where the pages were written and in which type of write. The problem with those fields was pollution in the following requests, that contain incorrect values. It was challenging trying to track down this bug, since an IO request object passes many states within the simulator, gets replicated throughout its life cycle and passes many various parts of the simulator.
Chapter 6

Discussion

We remain with questions about the potential incorporating of WOM codes into an effective caching tier and their potential to extend the SSD lifetime. In this chapter, we will conclude the work done so far and review the open questions and evaluation methodology that is needed to complete this work.

6.1 Open Questions

The work done so far about designing WOMAC has opened a few open questions. In this section, we will review the open questions remaining in this work. In the following section, we will describe what experiments should be done to evaluate the potential of WOMAC and what are the metrics to be compared between other SSD caching schemes. In addition, some experiments should be done to validate some of the design choices and parameters of WOMAC.

The first design choice was the use of logical blocks. WOMAC can operate without the use of logical blocks, but it would require matching two blocks with invalid data of roughly the same age, to perform a clean operation on similar wear off blocks. In addition, is a logical block size of 2 physical blocks appropriate? We could use 4, 8 or more physical blocks in a logical block. In the paper that proposed logical block, the size was 4 physical blocks [KM06]. Would the use of larger logical blocks improve WOMAC?

Another question is, what is the number of segments should be in ghost buffer? Too large and the computational complexity of the iterative algorithm to search for minimum costs could be time-consuming. For example, if there are 100 segments, each one is 1% disk space, each loop will run 100 times and a total of 1 million calculations of $C_{HY}(x, s, r)$ will be executed. If the number of ghost queue segments are too small, we might miss a close to the optimal point of $C_{HY}(x, s, r)$, using extrapolated hit rate curves. A similar question is what should be the period length the partition algorithm should run to adjust the different cache areas. Too short, and partition resizes could incur a lot of unnecessary evictions from areas that shrunk. Too long, and the partitioning
algorithm would not fit the current access pattern of the workload. Also, could we have utilized more RAM in WOMAC? E.g. putting in RAM pages from ghost queue hits and only after another hit, write them back to Flash to improve performance.

The most significant points in WOMAC design are addressed in the following questions: Were the pages we have selected to be written to second write area the right ones for both read and write caches? When should the algorithm promote a page from second to first write in the read cache? Since the current scheme incurs an overhead of write amplification, should we have prompted the page upon its block eviction or would the promotion have been too late at that stage?

6.2 Evaluation Methodology

In this section, we describe how WOMAC should be evaluated. First, we describe the evaluation for WOMAC design choices. Then, we describe how WOMAC should be evaluated against other SSD caching schemes.

The first question we should answer is, what kind of pages should be placed in a second write. Second writes do provide additional writes to the SSD lifetime, but at the cost of less available storage. The choice should be to utilize as much as possible second writes with little impact on the cache hit rate. We should evaluate those choices for the read and write cache separately.

For the read cache, we should evaluate the same SSD configuration for several read traces under two options. The first option is to put new read requests pages in the first write area and put repeated request pages in the second write area. The second option is vice versa. A decision should favor the option that has longer lifetime, with a small degradation of the hit rate compared to the other. An additional experiment should test the impact on when should we promote a page from the new read page area to the popular page area? Upon block eviction or on read cache hit? We should favor the option that has a higher hit rate, since the page is written again in both options, and should result in a similar number of writes. In addition, we should check if using a memory cache for promotions can reduce false identifications of popular pages, since a page could be read twice and get promoted to the popular page area and never be read again. For write cache, we should evaluate the same SSD configuration for several write only traces in a similar fashion on what kind of pages should be placed in the second write.

Now, we will try to answer the questions about parameters of epoch length and segment sizes. The calculations of Algorithm 1 can be done in the background by the SSD controller while serving IO requests. So even if the calculation is done over a period of a few seconds, in which millions of calculations could be done in the background, a segment size of 0.5% should be acceptable, since each iteration contain less than 25 arrhythmical operations. This should run a maximum of 8,000,000 \( (200^3) \) calculations of \( C_{HY}(x, s, r) \). Furthermore, it should provide a sufficiently close values of the parameters
to their optimal point values on the cost model graph. The period length should be
tested on a variable number of requests with the scales of \((2^{12}, 2^{14}, \ldots, 2^{20})\) for a
number of traces with mixed read and write requests. We expect that we will receive
similar results to [OCLN12], of optimal length for the epoch of between \(2^{14}\) to \(2^{16}\), since
those period sizes capture the current access patterns of a workload that worked for the
algorithm we extended.

So far, we have covered the experiments for the design choices of WOMAC. Now,
we describe how it could be evaluated. It should be compared against two other SSD
caching schemes. The first is OP-FCL [OCLN12] on baseline simple SSD drive that
does not support second writes. This caching scheme is the one we based our design
on, because of the partitioning between read and write cache space and the use of cost
models. The other scheme should be the Reusable SSD [YYS15]. This scheme uses
second writes but acts as the storage device, keeping the external capacity the same
and keeping the number of blocks in second write below a certain limit. Since it doesn’t
have a cache eviction policy, we suggest using the block level LRU victim selection
policy. We also suggest partitioning the cache to a read cache and write cache. For
example, if 70\% of the cache writes come from read requests and the rest comes from
write requests, the division should be 70\% of drive capacity to read cache and 30\% to
write cache. This partitioning is adjusted based on current workload. So, new clean or
recycled blocks are allocated to relevant cache based on the type of requests.

The experiments should be conducted using two types of traces based on their scale.
The first type is ‘Small Scale’ traces which should be up to 25x external capacity of
the SSD. The second type of traces is ‘Large Scale’ that are more than 25x of the SSD
external capacity. Both experiments should test the schemes response time and hit rate
on known traces. In addition, the experiments should check the use of second writes
and are measured in erase counts and how they differ between the different experiment
scales. We suggest using 4 types of SSD specifications: 2 specifications that differ in
external capacity and 2 specifications that differ in overprovisioning space. We would
like to know if those parameters effect the results.

To determine whether WOMAC provides a sufficient caching scheme, we first suggest
examining the results between WOMAC and Reusable SSD, checking if WOMAC has a
higher hit rate and lower latency. If the hit rate is improved and so is the reduction in
latency, this would indicate if WOMAC is a better caching scheme. Then we should
compare the hit rate and latency to OP-FCL and check if the WOMAC hit rate and
latency don’t fall by much. This will indicate if WOMAC slightly degrades the cache
performance. On the other hand, we must check if WOMAC fulfilled its purpose, to
provide SSD with a longer lifetime. We should compare all 3 schemes with the ‘Large
Scale’ traces, to see which has the least amount of erase operations. If WOMAC will
have equal or less erase operations than Reusable SSD and less erase operations than
OP-FCL, then it is successful in increasing in SSD lifetime.

An additional thing to examine when running the traces, is the partitioning between
the four areas in the algorithm. We could build a graph like in [OCLN12], in which we
could see if the partitions stabilize in size over time when access patterns remain the
same.
Chapter 7

Conclusion

We presented WOMAC, a practical design for applying second writes to SSDs that could act as efficient caching devices with a longer lifetime. This design is general and is applicable to current flash architectures that support rewrites to a page, requiring only minor adjustments within the cache controller and FTL, without additional hardware or interface modifications.

The design of WOMAC innovates in the attempt to balance between cache size and SSD lifetime by using a cost model to dynamically adjust between them, based on current workload. More, it identifies good usages of the extra writes gained from using second writes to improve SSD-based caches. This design can be extended for flash architectures that allows for higher degrees of parallelism and could support third and maybe fourth writes, by combining 4 physical blocks to a logical block.

Finally, most previously suggested schemes for extending SSD lifetime are orthogonal to the design of WOMAC and can be combined with WOM-codes. The SSD endurance improvement that could be achieved by WOMAC could mask some of its drawbacks in cache hit rates, making it more viable for the use of SSD-based caches.
Bibliography


ניקיםلاحובעליים.כ.Adaptermasוואלהמאוזןאתכןוהג’SDD,ונמדקתאתאדר
רוחשושל.גلاعبпередמלאתמד닱מתושעתוכילדשםบทקשתמד닱הלידה
יפורלאיר,ובכןלתושמ pwmymןתחקה泪水Garbage Collection
רוחשושל.

עבורהמדאםמויתליוםאריכטייקוטורהההיבריםמטמוןלאכוןאパーテים,עילשק
מתמציםמנכיתאןוהשלכלאפרשדמטומות واضحהת.הםسقطפריפודותרומתאורכית
שלנהבורה:

1. האריךאריכטייקוטורההיבריםמטמוןמעטוליאתמעishmentsבדהינו.אמדירפמקירות.

2. האריךמעלעםהליקתחברניםאתמע ENUMיםمشاركةעלמטאמרימרורמל_TIMES
ומToolStripןכןאורלאمعنىשלמדיעהקרבאתמדיםלאחריםשלמדיעהראשה
ופיתנתנוצרה.

3. האריךפונקציתבניסטוי לתחרותאריכטייקוטורההיבריםמטמוןמטפסהלק
,אמדירפמקירות,WOM

4. אריכטייקוטורההצירפתיאתמעטוליﻵנהחברחקפירות,מעלמט.Timerוסטרתלאדיארי
אתאריךהיהיששלכומיאמרםטרמייקוטורההствоватьспектראIA שבכובד,
ומובלהוא.
invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.

invalid SSD or invalid SSD error, please ensure correct SSD type and configuration.
SSD) Solid State Drive: a non-volatile memory technology that uses flash memory chips to store data. Unlike hard disk drives (HDDs), SSDs have no mechanical parts and are therefore faster and more durable. They are used in personal computers, servers, and data centers.

Write technology: this is a technology that allows SSDs to write data directly to the memory array, unlike traditional HDDs which require multiple steps. This results in faster write speeds and improved performance.

Once Memory (WOM): a technology that allows SSDs to retain data even when power is lost. This is achieved by using a combination of battery backup and other hardware components.
הודות

אני מודע�能 מסר את של פורפ', אח''ט יעקב וינשנס, על华尔מה. התכנית
ורוממי ב-Sharkey עם ליברמן של uname רחת ב-2014 עם הבכורה שחבר
יפוי היהות של החקירות במאגרים משנים מאוזנים. כל אלה, אלי והשביח
'autograph' בכתובת האָס📸, כולל במועד עלולות ש洧ים טווח ביב uy אחרים
בוןקט.

ארכיון החרוזים של ראטס, כולל דרך על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארץ החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העצה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העزة האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העזה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העזה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העזה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העזה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העזה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העזה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העזה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העזה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העזה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העזה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העזה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העזה האפשרותיות של שונים
ולשימים הממשלבה.

ארכיון החרוזים על הגיה על העזה האפשרותיות של שונים
ולשימים הממשלbah.
שיפורי תרשים התויה של דיבורı מטמון מבוסס על שימור בקודי כתיבה רב-_prices

חוקר על עבדות נמר

רטנור הלטסלט

וך ידיעת ממציא

רו קורץ

ורטנור הלטסלט — מומח venev יישואל

וניע 0777777777

חיפה יוני 2017

شيفرת תורלה ההימיות של ידך ב SSD בימוס שימור בקוד בכתב
רבים-עשרות

רות קררצקי