University of Cincinnati

Date: 12/7/2015

I, Mingyang Wang, hereby submit this original work as part of the requirements for the degree of Doctor of Philosophy in Computer Science & Engineering.

It is entitled:
Improving Performance And Reliability Of Flash Memory Based Solid State Storage Systems

Student’s name: Mingyang Wang

This work and its defense approved by:

Committee chair: Yiming Hu, Ph.D.
Committee member: Kenneth Berman, Ph.D.
Committee member: Karen Davis, Ph.D.
Committee member: Wen-Ben Jone, Ph.D.
Committee member: Carla Purdy, Ph.D.
Improving Performance And Reliability Of Flash Memory 
Based Solid State Storage Systems

A dissertation submitted to the 
Graduate School 
of the University of Cincinnati 
in partial fulfillment of the 
requirements for the degree of 

Doctor of Philosophy 
in 
Computer Science and Engineering 

in the Department of Electrical Engineering and Computing Systems 
of the College of Engineering and Applied Science 

by 

Mingyang Wang 
2015 

B.E. University of Science and Technology Beijing, China 

Committee: 
Professor Yiming Hu, Chair 
Professor Kenneth Berman 
Professor Karen Davis 
Professor Wen-Ben Jone 
Professor Carla Purdy
Abstract

Flash memory based Solid State Disk systems (SSDs) are becoming increasingly popular in enterprise applications where high performance and high reliability are paramount. While SSDs outperform traditional Hard Disk Drives (HDDs) in read and write operations, they pose some unique and serious challenges to I/O and file system designers.

The performance of an SSD has been found to be sensitive to access patterns. Specifically read operations perform much faster than write ones, and sequential accesses deliver much higher performance than random accesses. The unique properties of SSDs, together with the asymmetric overheads of different operations, imply that many traditional solutions tailored for HDDs may not work well for SSDs. The close relation between performance overhead and access patterns motivates us to design a series of novel algorithms for I/O scheduler and buffer cache management. By exploiting refined access patterns such as sequential, page clustering, block clustering in a per-process per-file manner, a series of innovative algorithms on I/O scheduler and buffer cache can deliver higher performance of the file system and SSD devices.

Other than the performance issues, SSDs also face some unique reliability challenges due to the natural properties of flash memory. Along with the well-known write-endurance, flash memory also suffers from read-disturb and write-disturb. Even repeatedly reading from an SSD may cause data corruption because the read voltage may stress neighboring memory cells. As the density of flash memory keeps increasing, the
disturbing problems are becoming even more severe for memory cells to store data reliably.

One of the structural merits of an SSD is its internal parallelism. Such parallelism of flash memory chips could be exploited to support data redundancy in a similar fashion to traditional HDD RAID. Recently an emerging non-volatile memory (NVM) such as PCM is receiving increasing research interest, as it outperforms flash memory by providing in-place update and better performance and reliability. Hybrid solutions, which combine both flash memory and NVM to balance performance and cost, are under special investigation to address the reliability and performance issues of flash memory based storage systems.

To address the reliability concerns, we present a novel storage architecture called i-RAID (internal RAID) that introduces RAID-like parity-based redundancy while avoiding many of its problems. What make i-RAID so unique like no other are its deferred parity maintenance, selective RAID protection and dynamic RAID organization. It solves traditional RAID’s small update problem and avoids SSD RAID pitfalls. Unlike traditional disk drives, SSDs cannot perform in-place updates. We view this unique characteristic as an opportunity instead of a hurdle. The out-of-place update feature means that old data will not be over-written by the new data, which enables us to design some fundamentally new algorithms that defer the computing and updating of parity blocks until the garbage collection time, thereby significantly reducing the overhead and possibly increasing the life-time of SSDs. Our algorithms also dynamically and selectively construct parity stripes only on aged, error-prone blocks, and utilize the internal parallelism of SSDs to further improve performance.
To my parents, my wife, my aunt and uncle
for their unconditional and endless support, love and encouragement.

And to my Buddy...
ACKNOWLEDGEMENTS

The Ph.D. study is a bumpy and winding journey which is not always pleasing and relaxing. It is like an endurance test rivaling an ultramarathon, with full of headwinds, ups and downs. However, throughout this adventure, there are many brilliant, vibrant and supportive people with me.

First and foremost, I would like to express my deepest gratitude to my advisor, Dr. Yiming Hu for his constant encouragement, support and patient guidance. He has provided insightful advice to improve my works, and always backed me up when I faced setbacks. I would never, ever have made this possible without him.

I would also like to extend my gratitude to Dr. Kenneth Berman, Dr. Karen Davis, Dr. Wen-Ben Jone and Dr. Carla Purdy for their precious time to serve as my dissertation committee, and their comments to strengthen my works. I am especially thankful to Dr. Carla Purdy who has tutored me when I was assigned to teach an undergrad course. Dr. Wen-Ben Jone has shared many great ideas, personal experience on research, teaching and career. Many thanks go to Dr. Karen Davis who has carefully and meticulously reviewed and commented my dissertation. Her attention to the finest details always impresses me. Dr. Kenneth Berman also shared great knowledge and materials when I taught Data Structure class.

In particular, I am very grateful to Ms. Julie Muenchen and Ms. Barbara Carter, the Program Director of Graduate Studies, and Prof. Frank Gerner, the Associate Dean of the College of Engineering and Applied Science. Their persistent support and encouragement have made me through tough times.
I would also like to thank Ryan Anderson. With Ryan, I have worked on many projects, from which I kept enriching my skill set, and sharpening my coding skills. I have learned a lot from him.

My gratefulness also goes to my friends and colleagues, Huaping Wu, Xiaoyu Yang, Jianghao Guo, Jiayang Guo, Jiapeng Wang and his wife Nan Yang, Vadambacheri Manian Karthik and his wife Balasubramanian Sanchayeni, and many others who have helped and supported me in any aspect.

Finally and most importantly, to my parents and to my family whom I owe a debt of gratitude, my wholehearted gratefulness is beyond any words.
# Table of Contents

**Chapter 1**  
**Introduction** ................................................................. 1

1.1 Revisit System Designs ........................................................ 2
1.2 Opportunities and Challenges .............................................. 4
1.3 Research Contributions ...................................................... 7
1.4 Organization of Dissertation .............................................. 9

**Chapter 2**  
**Background** ........................................................................ 12

2.1 Flash Memory ................................................................. 12
2.2 Flash Operations .............................................................. 13
2.3 Structure of SSD ............................................................... 16
2.4 Flash Translation Layer (FTL) ........................................... 18
2.5 Non-volatile Memory (NVM) .............................................. 22

**Chapter 3**  
**Fine-grained Access Patterns Analysis and Its Application to I/O Scheduler**  24

3.1 Access Patterns Analysis .................................................. 24
3.2 Performance Impact of Fine-grained Access Patterns ............ 27
3.3 Pattern Based I/O scheduler .............................................. 30
  3.3.1 The Design of Pattern Detector .................................... 31
  3.3.2 Improving Sequential Write Performance .................... 34
  3.3.3 Improving Non-Sequential Write Performance ............... 36
  3.3.4 Merging also Benefits Sequential Writes ....................... 37
3.4 Experiment and Performance Evaluation ............................ 38
  3.4.1 Benchmark Programs and Trace Collector ..................... 39
  3.4.2 Simulation Results .................................................... 40
    3.4.2.1 Wearing Reduction ............................................ 40
    3.4.2.2 I/O Time Improvement ....................................... 41
    3.4.2.3 Results of Pattern Detections and Scheduling Decisions .... 43
3.5 Related Works ............................................................... 46
3.6 Summary ............................................................................ 46
List of Figures

Figure 2.1: Control Logic block diagram, adapted from [41]  ........................................... 13
Figure 2.2: SSD components and flash memory schematic, adapted from [22] ............... 17
Figure 2.3: Cases when the log block and the data block are garbage collected .............. 20
Figure 3.1: Gcc shows clear write access patterns and patterns’ transitions .................... 26
Figure 3.2: Cases when a sequential write may still cause a costly block full merge ...... 28
Figure 3.3: Mapped logical addresses (in sectors and in pages) written by a series of sequential write requests ................................................................................................... 30
Figure 3.4: A series of sequential write requests in small footsteps are mapped to the same page unit .................................................................................................................................... 37
Figure 3.5: Relative Erase Cycle Improvement ................................................................ 41
Figure 3.6: Overall I/O Time Improvement ..................................................................... 42
Figure 3.7: Write Performance Improvement .................................................................. 43
Figure 4.1: A series of sequential write requests in sectors are mapped to the same page unit .................................................................................................................................... 51
Figure 4.2: A series of sequential writes may still cause a costly log block merge. Note that page #0 was logged twice ........................................................................................................ 52
Figure 4.3: An example of pattern based dirty pages partition ......................................... 53
Figure 4.4: Write time and garbage collection overhead ............................................... 58
Figure 4.5: Write performance improvement ratio ......................................................... 59
Figure 4.6: Total erase cycles invoked on the device .................................................... 60
Figure 4.7: Erase cycles reduced by Pattern-Based-Buffer .............................................. 61
Figure 4.8: Erase cycles invoked by full merge and block switch respectively ............. 63
Figure 5.1: i-RAID stripe and reconstruction during garbage collection before garbage
collection ........................................................................................................................... 73
Figure 5.2: i-RAID stripe and reconstruction during garbage collection after garbage
collection ........................................................................................................................... 74
Figure 5.3: Timing diagram comparison ........................................................................ 78
Figure 5.4: The flow diagram of i-RAID running with GC ............................................ 79
Figure 5.5: i-RAID mapping table and its hash map ....................................................... 81
Figure 5.6: Average response time of i-RAID with varied thresholds compared to the
baseline ............................................................................................................................. 86
Figure 5.7: Additional Erase Cycles of i-RAID with varied thresholds compared to the
baseline ............................................................................................................................. 87
Figure 5.8: Uncorrectable error rate comparison between 4-bit ECC and i-RAID with
varied thresholds ........................................................................................................... 90
List of Tables

Table 1.1: Comparison of current memory/storage technologies [19] ............................... 3
Table 3.1: SSDSim Parameters.......................................................................................... 38
Table 3.2: Number of Requests Seen By SSD.................................................................. 43
Table 3.3: Patterns Transition Detected............................................................................ 44
Table 3.4: Number of Scheduling Decisions .................................................................... 45
Table 5.1: iRaidSim simulator parameters ........................................................................ 82
Table 6.1: Projected 2020 characteristics of PCRAM, adapted from [103] ..................... 95
Chapter 1

Introduction

Solid State Disks (SSDs), flash memory based persistent storage systems, have been receiving tremendous interests from research communities and industries. SSDs are different from traditional Hard Disk Drives (HDDs) by removing the moving parts, such as the spinning disc and mechanically moving disk arm. This significantly reduces the access latency, enabling true processing throughput and user productivity. The well-known advantages offered by SSDs over HDDs include shock-resistant, lower read/write latency, better random access performance, and lower power consumption. Therefore, SSDs are becoming increasingly popular in consumer products and are also gaining momentum in enterprise applications [1-3] where high performance and high reliability are paramount.
1.1 Revisit System Designs

Before the emergence of SSDs, hard disk drives (HDDs) were dominant in computer storage systems. However, due to the mechanical nature, hard disk drives have been acting as major bottlenecks for decades, throttling the entire system performance. It is well known that HDDs have long latencies, especially when handling random data accesses [4-6]. HDDs also have serious reliability issues [7, 8]. It is not uncommon that the whole disk is out of service all of sudden, likely causing data loss. Moreover, HDDs consume high power, increasing data center power and cooling expenses [9-11]. Therefore hard disks have been under extensive investigations to address these issues. For example, RAID techniques [7, 12] have been widely adopted to enhance data reliability by incorporating data redundancy. As another instance, to alleviate the long latency of disk accesses, data are normally buffered, since most of data accesses show temporal or spatial localities [13]. Following that, an effective replacement policy for the buffer cache is necessary, from LRU [14, 15] to other state-of-the-art schemes, such as [16]. Moreover, read-ahead [4, 6], a.k.a., prefetch, is adopted to reduce sequential read latency. To overcome random write performance degradation and improve data reliability, written data might be logged by the file system [17, 18].

Unfortunately, all the performance and reliability issues arise from the physical structure of HDDs. Essentially hard disk drives are mechanical devices. Data are persisted on the mechanical components — spinning disk platters. Other mechanical components — disk heads which are attached to disk arms are floating on the disk platters and swinging back and forth to read and write data. The performance
improvement for a storage system had been pushed to the limit, until Solid State Disks
emerged.

Solid State Disks (SSDs) — flash memory based secondary storage devices,
fundamentally address the technical issues of HDDs, in that without any mechanically
moving components, SSDs are electronic computer storage systems which are completely
built on semiconductor flash memory chips. Due to the much more superior performance
and much less power consumption delivered by flash memory based SSDs as depicted in
Table 1.1, SSDs are quickly being incorporated into the existing computer storage
systems.

<table>
<thead>
<tr>
<th></th>
<th>SRAM</th>
<th>DRAM</th>
<th>Hard Disk</th>
<th>NAND Flash</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cell Size</strong></td>
<td>&gt; 100 F²</td>
<td>6 - 8 F²</td>
<td>(2 / 3) F²</td>
<td>4 - 5 F²</td>
</tr>
<tr>
<td><strong>Read Latency</strong></td>
<td>&lt; 10 ns</td>
<td>10 - 60 ns</td>
<td>8.5 ms</td>
<td>25 µs</td>
</tr>
<tr>
<td><strong>Write Latency</strong></td>
<td>&lt; 10 ns</td>
<td>10 - 60 ns</td>
<td>9.5 ms</td>
<td>200 µs</td>
</tr>
<tr>
<td><strong>Energy per bit access</strong></td>
<td>&gt; 1 pJ</td>
<td>2 pJ</td>
<td>100 - 1000 mJ</td>
<td>10 nJ</td>
</tr>
<tr>
<td><strong>Static Power</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Endurance</strong></td>
<td>&gt; 10¹⁵</td>
<td>&gt; 10¹⁵</td>
<td>&gt; 10¹⁵</td>
<td>10⁵</td>
</tr>
<tr>
<td><strong>Non-volatility</strong></td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

However, most system designs which are proposed for HDDs may not be appropriate
for SSDs and sometime even hurt the system performance. For such an instance,
anticipatory I/O scheduler [20] is tailored to the mechanical properties of HDDs. When
scheduling a series of I/O requests, particularly read requests, anticipatory scheduler
defers dispatching the subsequent read accesses which are located on different disk
cylinders, anticipating that an upcoming request will still be located on the current disk
track. By doing so, seek latency which is caused by repositioning disk arms to a different cylinder can be significantly reduced. However, an SSD which is absent of any mechanical moving parts, has much more consistent and better random read access performance. Anticipatory scheduler cannot deliver any improvement to an SSD based storage system and may even impair the system responsiveness. For another instance, a disk blocks’ arrangement technique tailored for HDDs which attempts to arrange data to aligned disc tracks [21] based upon data access patterns is not applicable for SSDs. To this end, the tailored system designs for SSDs’ own properties motivate us to revisit the components of operating systems.

1.2 Opportunities and Challenges

Flash memory based SSDs beat conventional magnetic hard disks in many aspects such as latency, power consumption, as depicted in Table 1.1. The semiconductor nature of flash memory makes SSDs fundamentally distinguished from HDDs, offering the opportunities for storage systems to achieve substantial improvement. However, the proprietary properties of flash memory introduce some brand new unique challenges and difficulties to system designers. These deficiencies are also originated from flash memory’s electronic properties, so it is difficult to overcome by the physical hardware itself. To this end, software optimization plays an important role in overcoming the hardware difficulties. The following unique characteristics not only bring challenges, but also provide opportunities for us to achieve higher storage system performance and reliability.
First, the operations in SSDs are highly asymmetric. A write operation is several orders costly than a read operation, in terms of latency and energy consumption, as a write operation consumes longer time to program a flash memory unit due to this error-prone process. The discrepancy in operation cost is not only the result of asymmetric latency between read and write operations, but also the erase-before-program restriction of flash memory [22-24]. What’s more important, the internal structures of SSDs, along with the out-place-update nature, make SSDs more sensitive to access patterns, in other words, the asymmetric operation cost of different access patterns. More specifically, sequential read accesses deliver higher performance than random read accesses, and sequential write accesses perform much faster than random write accesses. The SSD operations also suffer from amplification issue. A read or write operation is in a unit of a page which is several sectors in size. However a unique erase operation for SSDs is in an even larger unit, which normally consists of a few pages. This motivates us to fine-tune system designs to tailor to such asymmetric operation performance.

Secondly, flash memory based SSDs suffer from reliability issues. The most well-known problem is write-endurance. Flash memory cannot execute in-place update. A flash memory unit has to be erased before being reprogrammed. The erase operation not only invokes much longer latency, but also reduces the life-span of SSD devices, as flash memory can only sustain for a limited program/erase cycles (P/E cycles). Even worse, previous studies found that some flash memory could show significant error rates from the very beginning, and Bit Error Rates (BERs) start to increase sharply long before the memory cells reach their rated lifetime [25, 26]. Another less well-known but equally serious problems are called read-disturb and program-disturb (or write-disturb) [27-30].
Whenever a cell is read or written, a voltage is applied to the block, which might stress neighboring cells in the block, causing them to gradually lose data. This is a foundation limitation of flash memory that cannot be avoided. While the disturbs cause relatively few errors for a fresh new flash block, recent studies showed that read-disturb error rates increase by several orders of magnitudes when a block is erased by as few as 2000 times [27]. This implies that there will be data corruption when an SSD is repeatedly read. As flash memory continues to scale to accommodate larger capacity, read-disturb and program-disturb are becoming even bigger concerns [31-33]. Because the flash memory with different P/E cycles exhibits different reliability properties, this motivates us to treat blocks with different age or health in different ways. While young fresh blocks can be protected by ECC alone, aged blocks which are subject to higher risk of errors should be protected additionally. Our proposed novel internal RAID architecture is such a technique, providing selective, deferrable, flexible and dynamic RAID protection solution.

A very unique characteristic of SSDs is internal parallelism structure. Exploiting the parallelization of device operations enables us to design some fundamentally different architecture. For example, our proposed internal RAID architecture while enhancing data reliability through redundancy, exploits such internal parallelism to reduce the overhead caused by traditional RAID operations.

In our research works of this dissertation, we conducted extensive studies on current system designs to address SSDs’ performance and reliability issues. The major contributions are as follows.
1.3 Research Contributions

We examine performance impacts from different refined access patterns on an SSD storage system. We identify some unique patterns which could hurt system performance and reduce device life-span. For example, although SSDs favor sequential write accesses, under some circumstance, even sequential write accesses could still result in suboptimal performance. We reveal that coarse classification of access patterns, such as only to classify into sequential and random, is subpar for an SSD system. We further identify some refined access patterns such as block clustering and page clustering could lead to different performance overhead in SSD systems. Therefore we treat them differently. We design a set of I/O scheduling algorithms, including pre-alignment, inner-padding, write merging, merging-and-splitting which exploit these fine-grained access patterns. Our algorithm monitors runtime patterns in a per-process per-io-stream granularity as well as captures any transitions of patterns. Our I/O scheduler adopts different schemes for different patterns. For example, for imperfect sequential write accesses, our scheduler will amend these imperfect disk requests to achieve anticipated performance of sequential accesses. For block clustering patterns, our scheduler defers and reorders the requests, so it could potentially convert scattered write accesses to sequential write accesses.

We further extend our refined pattern analysis to buffer cache management. When dirty pages in the buffer cache of operating systems are committed to the backend devices, current scheme only considers temporal locality. We reveal that for sequential write requests which are not likely to be rewritten again, only considering temporal locality may likely evict other non-sequential write accesses prematurely. This premature commitment of non-sequential write accesses over sequential write accesses not only
adds overhead to SSDs’ garbage collection but also reduces the device life-span. Our 
access pattern based buffer cache management not only considers the temporal locality 
but also exploits spatial locality of write accesses. We develop a set of different dirty 
page commitment schemes tailored to the access patterns. Essentially, the buffer cache is 
virtually partitioned according to the access patterns.

To overcome the unique reliability challenges of flash memory based SSDs, we 
propose a novel internal RAID architecture — named as \textbf{i-RAID}. What make i-RAID so 
unique like no other are its \textit{deferred parity maintenance}, \textit{selective RAID protection} and 
\textit{dynamic RAID organization}. It solves traditional RAID’s small update problem and 
avoids SSD RAID pitfalls. To our best knowledge, our work is the first study which 
constructs RAID inside an SSD \textit{selectively} on flash memory blocks based on their age 
and health status. This selective RAID perfectly fits in with the flash memory’s reliability 
behavior, i.e., an aged flash memory block with high erase cycles is subject to much 
higher risk of errors than a fresh young block. Moreover, our internal RAID architecture 
offers flexibility of RAID striping, since it constructs RAID on physical flash memory 
blocks which is another major innovation of our RAID architecture. Furthermore, our 
internal RAID architecture views the out-of-place characteristic as an opportunity instead 
of a hurdle. It completely defers the computation and update of parity blocks until the 
later stage of garbage collection time to avoid small update problem in traditional RAID. 
Even more promising, our internal RAID architecture much better benefits from the 
internal parallelism structures of SSDs so that parity maintenance can run with garbage 
collection operation in parallel.
1.4 Organization of Dissertation

So far we have given a brief introduction on flash memory. We also present the advantages and challenges of flash memory based SSDs which motivate our works in this dissertation. In the rest chapters, we will address in detail our system designs, proposed storage system architecture and their key features. For each of the proposed design or architecture we will also present relative backgrounds as well as our experiments and their results. The rest of this dissertation is organized as follows:

In Chapter 2, we present overall background on flash memory, SSDs and emerging storage technologies. We introduce SSD structures including the software layer inside SSDs, and the internal parallelism structure formed by multiple flash memory chips.

In Chapter 3, we revisit the I/O scheduler in operating systems. We reveal that an I/O scheduler which does not take into account the performance impact of different access patterns can result in degraded I/O performance, decreasing life-span of SSD devices. We further observe that even sequential write accesses can still result in poor performance of SSDs if the sequential write accesses are not aligned with flash memory blocks. These observations motivate us to design an innovative I/O scheduler which is based on fine-grained access patterns in a per-process per-stream manner. We propose a set of novel scheduling algorithms — including pre-alignment, inner-padding, write merging, merging-and-splitting, which exploit these refined access patterns to improve the system performance of SSDs. Simulation results show that these scheduling algorithms not only improve the write performance, system responsiveness, but also largely reduce SSDs’
erase cycles, which is directly translated to a major improvement on the life-span of SSD devices.

In Chapter 4, we extend our fine-grained access pattern analysis to investigate buffer cache which is another component playing an important role in operating system performance. We revisit the buffer cache eviction schemes, especially for dirty pages since write performance is the key factor of the overall performance for an SSD based system. We examine the performance impacts on the write buffer cache under different access patterns. We propose to partition the write buffer based on the sophisticated, refined write access pattern analysis. In our design, dirty pages which are from sequential write requests are given higher priority to be committed to the device. On the other hand, dirty pages from clustered or random write streams will stay longer in the cache, therefore they will have better chances to coalesce or reorder within a flash block which is the unit for garbage collection. This potentially converts scattered write accesses into sequential ones which offer much higher performance in SSD devices. Experimental results show that pattern based write cache can not only improve system responsiveness but also improve SSDs’ life-span.

In Chapter 5, we focus on the reliability issue of SSDs. As flash memory continues to scale, write-endurance, write-disturb and read-disturb are becoming even more severe concerns. While a strong ECC (Error Correction Code) scheme can be very effective to protect young flash blocks, it may not be sufficient for aged blocks. Previous studies [34] have shown that using traditional RAID-5 algorithms directly on SSDs may lead to many pitfalls. For example, small update problems of traditional RAID technique, coupled with SSDs’ out-of-place update and write amplification may introduce some huge
performance penalty and other reliability issues. To address these issues while still enhancing data reliability, we propose a novel solution called i-RAID (internal RAID) that introduces RAID-like parity-based redundancy while avoiding many of its problems. Unlike traditional disk drives, SSDs cannot perform in-place updates. We view this unique characteristic as an opportunity instead of a hurdle. The out-of-place update feature means that old data will not be over-written by the new data, which allows us to design some fundamentally new algorithms that defer the computing and updating of parity blocks until the garbage collection time, thereby significantly reducing the overhead and possibly increasing the life-time of SSDs. More important, our algorithms also dynamically and selectively construct parity stripes only on aged, error-prone blocks, and utilize the internal parallelism of SSDs to further improve performance. What’s more promising, our dynamic RAID striping design can better benefit from inner parallelism of flash memory chips to reduce the latency caused by the redundancy updates. Simulation studies show that compared to non-redundant SSDs, under a pragmatic and moderate i-RAID threshold, the time and erase overhead added by i-RAID are surprisingly small, while the reliability is still significantly enhanced over ECC-only scheme.

Finally, we conclude our studies in this dissertation in Chapter 6.

The works in this dissertation have been published in conferences. The study in Chapter 3 was published on the proceedings of the 29th Symposium on Applied Computing (SAC) in 2014 [35]. The content in Chapter 4 is based on the paper published in the 32nd International Performance Computing and Communications Conference (IPCCC) in 2013 [36]. The work in Chapter 5 has been accepted as paper publication in the 31st Symposium on Applied Computing (SAC) in 2016 [37].
Chapter 2

Background

2.1 Flash Memory

In SSDs, data are hosted by flash memory which is a type of Electrically Erasable Programmable Read-Only Memory, EEPROM in short. There are two technologies of flash memory — NOR and NAND [38]. NOR flash supports random accesses and byte-program capability while suffering from slower program and erase operations. NAND, on the other hand, offers faster program and erase operations but slower random accesses, is absent of byte accesses, and also needs to be accessed in a larger unit, i.e., a flash page. Most flash memory based devices such as SSDs, thumb drives adopt NAND flash for its higher density and lower cost. Flash memory in this chapter refers to NAND flash memory.
In terms of NAND flash memory, there are two types of flash memory cells — SLC (Single-Level Cells) and MLC (Multi-Level Cells) [39]. Each MLC stores two bits in a unit of memory cell, while a SLC stores a single bit in it. The MLC flash technology is mostly employed in consumer products for its lower cost per unit. However it has longer operation latency and shorter life-span than its SLC counterpart. SLC flash memory can sustain more cycles (100k) of erasures than MLC memory (10k). Additionally, a page in SLC memory can be partially programmed (e.g., 4 times between consecutive erase operations) before an erase operation is required so that a typical page (4KB) can be programmed sector by sector (e.g., 512 bytes) [40].

2.2 Flash Operations

![Control Logic block diagram](adapted from [41])

In most NAND devices, the basic unit for a read or write operation is a page, which is composed of a group of memory cells sharing a common I/O bus. The page size is usually a multiple of the sector size, say a 2KB page contains four 512B sectors [39].
Multiple pages form a block which is the atomic erasure unit. The operations exported to upper file system layer through device driver are mostly same with HDD for compatibility, with some variation such as TRIM [42]. Inside an SSD, flash memory operations mainly include read, program, erase, copy-back and reset, as documented in technical report [40]. Flash memory manufactured by other suppliers has similar flash operations. General speaking, all operations involve a control logic which has been programmed with appropriate state machine in charge of assert or de-assert corresponding control signals and row/column address pins at prompt clock cycles, as do other types of semi-conductor memory. An exemplar schematic adapted from [41] is shown in Figure 2.1. To read a page from flash memory, a read command is issued into command register followed by supplying address to address register. Control logic will select/deselect specific enable signal at appropriate clock cycles. A read operation need around 25 microseconds (us) to read into data register from flash memory. Since then, data could be shifted out from on-chip data register via buses. The cache mode is supported through cache registers by many vendors of flash memory. In this case, while cache register is supplying data for output, data register could be fetched more data from flash memory synchronously. This double-buffering technique is beneficial for sequential read in that data accesses can be pipelined.

A program operation can flip bits in memory cells in only one way, say from logic 1 to 0. A program operation is issued by latching program command code into command register followed by the clock cycles of issuing address into address register with appropriately asserting/de-asserting pins. After command and address cycles, data is input into data register. Another command cycle is initiated to issue a confirm command
to start programming. Similar to the cache mode in read operations, double-buffer technique could be applied to program operation via redundant registers. While data in data register are programmed into flash array, next sequential data are fetched into cache register from controller. By doing this, reloading to cache register and programming oriented from data register could be carried on in parallel.

Flash memory is well known for its erase-before-program restriction. Flash memory cells need to be erased before they can be re-programmed. Flash memory performs erase operation in a unit of a block which consists of a few flash pages (e.g., 64 pages, 128 pages). A flash memory block can only sustain a limited number of program/erase cycles. This is due to the wear-out of the isolation tier which encapsulates the flash memory. Along with the process of increasing erasures on a block, the block may suffer from aggravated reliability issues. Erase operation begins after command and block address are latched into registers. A confirm command will confirm the erase action. The erase operation is the most costly operation which takes up to several milliseconds and is orders of magnitude slower than read and write operations.

Additionally, flash memory also supports internal copy-back operation which moves data across flash pages bridged via cache register without occupying channels, processor and RAM buffer so that these resources are free to be used by other operations. This operation is beneficial to block cleaning since block cleaning can be running in background while the flash device is servicing other requests.
2.3 Structure of SSD

Most SSD devices are based on NAND flash memory technology. The major components inside an SSD include flash controller, flash memory packages, RAM and processor, as depicted in Figure 2.2 [22]. Some processor may have native on-chip flash controller. ECC is also a necessary component. It may be implemented with hardware or software. Depending on the encoding algorithm, different number of bit-error can be fixed or found. To accommodate larger capacity, SSDs normally are packed with more than one flash package, each of which is composed from one or more dies. Dies normally share serial I/O bus and have common or separate control pins. Each die is organized as multiple planes. A plane is made of a group of blocks and may have its own execution resources such as register set. A flash memory block is the basic unit for an erase operation and in turn consists of multiple flash pages which are the basic unit for read or write operation. A block also contains spare area intended for metadata such as erase counter, logical address, ECC and etc. which can be stored together somewhere in a block or dispersed with each page.
Although most SSD suppliers regard internal architecture as intellectual properties, SSD internals were still explored by means of public technical documents such as [41, 43] or by means of experimental techniques such as [23, 44]. F. Chen et al. in [44] found out that with the aid of the SATA-2 interface which could issue multiple requests to the attached devices at the same time, SSD internal structures, such as the number of packages and mapping policy, could be uncovered. Based upon those, a rich collection of literature has been published regarding the internal of an SSD and the architectural
impact on SSD performance, as well as proposed potential improvement leveraged by architectural benefits [22, 24]. One of the major differences between SSDs and HDDs is the highly internal parallelism of SSDs. Unless a HDD has multiple actuators (which is extremely rare), it is hard for a HDD to serve multiple requests from/to disk media simultaneously.

There are multiple parallelism levels available inside an SSD [22, 44]. At the package level, each package may have its dedicated serial pins interconnecting with the controller. Under an extreme fully interconnecting configuration, each package may have its own control signals, leaving each package as an independent autonomous domain, although more hardware resources such as pins and more complex logic will be needed. Within the package, multiple dies form the secondary parallelism level, each being able to execute command independently. At this level, since multiple dies in the same package share serial pins, execution will be interleaved. Within a die, multiple planes form the lowest level parallelism. For instance, a command may span pages across two planes following that accesses to different planes can be executed concurrently.

### 2.4 Flash Translation Layer (FTL)

Because of flash memory’s program-erase-program nature, SSDs cannot perform in-place update directly. Instead, when programming a page, most SSDs actually write the new data into a new empty location and mark the old data as invalid. A mapping table is then needed so that applications or the operating systems can find the latest copies of the data. The SSD controller maintains a software layer, known as the Flash Translation Layer (FTL), which maps the logical address space, which is used by the applications and
the operating systems, to the physical address space, which indicates the actual locations of valid data blocks.

Essentially, there are two main types of FTL — page mapping and block mapping [45, 46]. Page mapping maps the logical address space to the physical address space in a flash page, while block mapping uses the much larger flash block as the mapping unit.

Intuitively, the fine-grained page mapping provides more flexibility, better performance, and less wearing than block-based mapping, because only the written page needs to be remapped and relocated. On the other hand, in the coarse-grained block mapping, a single page update may incur other pages in the same block be relocated to another block due to the larger mapping granularity. However page mapping demands much more RAM space than block mapping to hold the mapping table, resulting in a higher hardware overhead. It is very common for an SSD to have a capacity of hundreds of or even thousands of GBs, so the mapping table size of page-based mapping can be huge. A state-of-the-art page mapping was proposed by Gupta et al. in [47]. It is inspired by virtual memory systems, and will load a fragment of the page mapping table into RAM on demand.

Most systems employ log-structured block-based hybrid mapping schemes [48-50], which are inspired by log-based file systems [17]. In these systems, regular data is still mapped in a flash block to reduce the space overhead, while any updated pages will be temporarily appended into log blocks which are mapped in the unit of a page. There are some variations as to how a log block is associated with a data block, for example, the one presented by Jesung et al. [48] (a.k.a. BAST) and the one proposed in [49] (a.k.a. FAST). In BAST, each log block can only be associated with one data block. While in FAST, log blocks are classified into random blocks and sequential blocks.
random log blocks can be associated with any data blocks. On the other hand, the sequential log block can only be associated with one data block which is similar to BAST. When a data block updated from its first page (page offset is zero within the block), the sequential log block is allocated to it. If consecutive writes to the pages are sequential, only block switch is needed in the garbage collection which will be introduced shortly.

When the random log blocks need to be garbage collected, each valid page in it will be merged with that page’s associated data block. In BAST, a small random update to a data block may occupy a single log block. On the other hand, in FAST, as long as there are free page slots in the random log blocks, they can be used to append updated pages. So log blocks can receive much higher utilization, prolonging device’s life-span. In any hybrid mappings, when the system runs out of log block space, a cleaning process — Garbage Collection (GC), will run to clean stale data pages and free up log blocks.

(a) A switch operation    (b) A partial switch operation    (c) A full merge operation

Figure 2.3: Cases when the log block and the data block are garbage collected
In a log-structured block-based FTL, there are three cases when blocks are garbage collected — a full merge, a partial switch and a full switch. The latter two are less expensive than the first one, since both require fewer erase, read and write operations. In the case of a block switch, the log block has been written from the first page of data block through the last one, all pages are valid in ascending order of the logical number. The original data block is then erased, because all of its pages have been invalidated. The system marks the log block as the new data block, simply by updating the mapping table to reflect the new mapping which points to the log block, as shown in Figure 2.3(a). The overhead is the lowest, since only one block erasure operation is involved.

In terms of a partial switch, only some pages from the beginning of the data block are logged, and the rest pages of the data block are still valid in the original data block as depicted in Figure 2.3(b). As a result, these pages have to be copied from the data block to the log block before a switch can be performed, resulting in a slightly higher overhead.

In a full merge operation, the pages are updated and appended to the log block in random order. Any valid pages need to be read out from either the data block or the log block and written into a new data block in the sequential order, as shown in Figure 2.3(c). Assuming that the log block is fully logged, the extra cost includes copying (read and write) an entire block worth of data to the new data block, and erasing both the old data block and the log block. The overhead is significantly higher than a switch operation, since write operations are very expensive in an SSD, and erasure operations are even worse — they are often two orders of magnitude slower than read operations. In addition, the extra erasure operations will significantly reduce the life-span and the reliability of SSD devices. Generally speaking, sequential writes to the device are likely
to lead to block switches, while random writes result in block merges. The differences of performance and wearing overheads partially explain why SSDs prefer sequential writes to random ones.

### 2.5 Non-volatile Memory (NVM)

Recently, as an emerging storage technology, non-volatile memory (NVM) has been receiving much interest from academia and industry. Compared to flash memory, NVM can perform in-place update, supply better read/write performance and higher write endurance, not to mention the elimination of the most expensive erase operation of its flash memory counterpart. A state-of-the-art NVM is phase-change memory (also known as PCM, PRAM or PCRAM) [51]. In PCM, a memory cell unit is made of a glass type material, of which the states of crystalline and amorphous are switched with heating or pulse applied. The crystalline and amorphous states of materials have a wide range of resistivity, forming the states for a storage unit. The amorphous, high-resistance state is translated to a bit ‘0’, while the crystalline, low-resistance state stands for a bit ‘1’. PCM is superior in write performance compared to flash memory as changing the state of memory cell is faster (in the order of nanosecond), also without the need of erasing the memory before rewrite (in-place update).

As a promising technology, research has been conducted to incorporate PCM into existing systems [52]. For example, studies in [53, 54] proposed to replace DRAM with PCM as a system level buffer cache and examined the tailored cache policy for PCM. Due to the cost and manufacturability difficulties, a hybrid system which combines both
flash memory and PCM is especially gaining research interests. Kim et al. in [54] proposed to use PCM to host metadata and FTL table in an SSD. The work by Sun et al. in [55] incorporated PCM into an SSD device as a log block region. There are also many other studies to incorporate PCM into an existing system [56-62]. Other than PCM, there are some other types of NVM, such as STT-RAM [63] and Memristor [64, 65] which explore the properties of different materials or technologies to persist data.

So far, we have introduced some key background knowledge about flash memory and SSDs, as well as emerging storage technologies. We have also learned the challenges and difficulties in performance and reliability faced by the system designers. In the following chapters, we are about to address these challenges in system designs. The unique properties introduced by flash memory and the structures of SSDs motivate and enable us to invent some novel storage architecture and algorithms to improve performance and reliability.
Chapter 3

Fine-grained Access Patterns Analysis

and Its Application to I/O Scheduler

3.1 Access Patterns Analysis

Compared to a HDD, an SSD owns very different internal structure and thus exposes different performance bottlenecks to system designers. One of the unique characteristics of SSDs is the asymmetric operation cost. Write operations are an order of magnitude slower than read operations. Due to the internal parallelism structures, sequential accesses see much higher throughput than random accesses. For most SSDs which employ log-structured block-based FTLs, sequential write accesses are much more favored by SSDs than random write accesses, as sequential write accesses could invoke less costly garbage collection operations, which will be addressed in detail. Because
SSDs’ performances are sensitive to access patterns, especially write accesses [66], we explore the processes’ behaviors in product systems and their exhibiting access patterns.

In a real world system, multiple processes are simultaneously running, and they may exhibit very different access patterns on I/O streams. As a simple example, a process which issues write requests in a random manner (e.g., DBMS) is running simultaneously with a process which writes files sequentially (e.g., an installer). The file blocks written by DBMS may be overwritten again in the near future because of the spatial locality, while the file blocks written by the installer will be unlikely to be modified again.

Moreover, a process may open multiple files and exhibit different access patterns on these I/O streams. For example, Gcc may open and read multiple header files in random fashion and write object files in sequential pattern. A web server may update database in random, while write a log file in sequential. It is even possible that a process exhibits different access patterns on an I/O stream in different stages, in other words, pattern transitions occur.
By inspecting the traces from real world benchmark programs, we can capture the write access patterns. Figure 3.1 shows a sequence of write requests by one process forked by Gcc which was used to compile the Linux kernel source tree. It is obvious that the program demonstrates different I/O patterns during different time when writing this file. Most of the time, the accesses are sequential. Sometimes we see clustered accesses — the program writes the same page or block repeatedly. There are some pure random accesses but they are few and far between. Gcc forked many concurrent processes, each of which may be writing its own files and exhibiting different write patterns at different time.

Although sequential write accesses could normally deliver better performance of throughput and less wearing cost than random ones, do all sequential write accesses result in optimal performance and invoke less expensive garbage collection operations? Is a
coarse classification of access patterns which only considers sequential and random good enough for SSD devices? A careful examination reveals that it may not be as simple like that.

### 3.2 Performance Impact of Fine-grained Access Patterns

Although sequential writes likely lead to block switches in the garbage collection process, a careful examination reveals that many sequential writes still result in full merges, impairing overall performance. With the log-structured block-based FTL which is commonly employed in SSDs, consider a simple but common case shown in Figure 3.2(a), where the starting address of a sequential write is not aligned with the starting address of a flash memory block. The block contains some data, starting from page 0. The application issues a sequential write to the block with the starting address at page 2. On the first write, the FTL allocates a log block for the data block, and writes page 2 into the beginning of the log block. The system keeps redirecting the subsequent writes to the log block, until the log block fills up. At this moment, the FTL has to perform an expensive full merge by copying the first two pages from the original data block to a new data block, and copying the rest data from the log block. Finally the old data block and the log block are erased for future use. Note that a partial switch is impossible here, since pages in the log block do not start from page 0 within the flash memory block. While this is a perfectly sequential request, the FTL cannot perform a cost-effective block switch operation because the starting address is not aligned with the block boundary.
Figure 3.2: Cases when a sequential write may still cause a costly block full merge
(a) Sequential writes but unaligned with a block boundary, (b) Imperfect sequential writes occasionally skip pages, (c) Small sequential writes may result in overlapped logged pages

Figure 3.2(b) shows a different situation that the sequential writes may occasionally skip a few pages, leaving a hole in the logged pages. In this example, page 1 is skipped over by the sequential writes. In this case, a block merge operation is also needed during the garbage collection. Another potential problem is caused by the improper timing when the I/O scheduler fulfills the requests, as shown in Figure 3.2(c). In this example, the updates to page 3 were split into two different requests by the application. If the first write request is scheduled to commit to the device before the next request which updates the rest of page 3 arrives, the log block may append page 3 twice — one for the first write and the other for the second write to page 3. In this case, the perfectly sequential write may still cause a block merge.
Moreover, by further looking into the sequential write requests from the Gcc trace, it is easy to recognize that a lot of sequential writes will not result in efficient block switches in the FTL. For example, Figure 3.3 shows the footprints from one sequence of sequential write requests in logical sector number (a) and logical page number (b). Although they all seem to follow the ascending order of logical addresses, by looking into the shadow area which is the magnification of a small area, it is clear to see that requests are not strictly sequential, as logical page numbers from adjacent requests may overlap (they access the same page number repeatedly, as indicated by the flat line in the shadow area in Figure (b)). In this case, if some of the overlapped requested pages are flushed to the device, it may result in duplicated logged pages in the log block.

Other than the sequential write patterns, is the coarse pattern classification good enough for SSDs? Put another way, should we only classify non-sequential patterns as random? The flash memory’s out-of-place update nature together with write amplification suggests not. Recall flash operations in SSDs. A read or write operation is performed on the unit of a page (e.g., 2KB or 4KB) which is multiple sectors (e.g., 512B) in size. More specifically, should we treat a process which updates in a page size repeatedly equally to a process which writes a file in pure random? Obviously the answer is no. We simply call this kind of pattern a page clustering pattern, that is, a small range of the file in a flash page size is updated repeatedly. Furthermore, the garbage collection process executes an erase operation in SSDs which is in an even larger unit — a flash block which consists of a number of pages (e.g., 64 2KB or 4KB pages). For a commonly adopted log-structured block-based FTL, in the process of garbage collection, a block switch operation is much less costly than a block merge, as a block switch only needs one
erase and could invoke no additional read and write operations. Following this, should we treat a process which is exhibiting a clustering write pattern in a larger unit of a flash block size equally to a clustering write pattern in a much smaller page size? Definitely not. Identifying a block clustering pattern enables us adopt some different strategy which could potentially convert a block clustering pattern to a sequential one.

(a) The mapped logical address in sectors    (b) The mapped logical address in pages

Figure 3.3: Mapped logical addresses (in sectors and in pages) written by a series of sequential write requests

3.3 Pattern Based I/O scheduler

Fine-grained access patterns are a set of patterns which are tailored to SSDs’ unique characteristics. The patterns are captured in a granularity of per-process per-io-stream. These patterns combine both temporal locality and spatial locality, in addition to other information such as request size, inter-arrival time.
3.3.1 The Design of Pattern Detector

When access patterns are captured at file system layer, files and their semantics, as well as the context where they are opened to operate are accessible. The access patterns on a file stream from a process context can be utilized to predict future patterns for incoming requests on that file due to the temporal and spatial locality. For example, if a process writes a file in a strong sequential order, the future requests on the same file can be predicted as sequential so that scheduling policy can be fine-tuned on this file. Also if the access pattern is switched to non-sequential (e.g., clustering), then scheduling policy can be adjusted accordingly. Since a process may open multiple files and exhibit different patterns on each I/O stream, we developed a real-time, per-process per-stream based pattern detection scheme which classifies fine-grained write patterns into four types — (a) sequential, (b) page clustering, (c) block clustering and (d) random. Each of the above patterns is defined as follows:

1. A sequential pattern is detected when a series of consecutive requests issue logical addresses in an ascending order;

2. A page clustering is captured when a process repeatedly writes in a specific page unit;

3. A block clustering is captured when a process repeatedly writes in a specific block unit;

4. A pattern which is none of above is classified as random.

In order to filter out transit “noise”, that is, to avoid falsely or excessively switching pattern types and to capture a pattern which exhibits strong and steady locality, each pattern is allocated with a bit map. The number of bits $n$ will determine how many times
in a row a pattern has to be detected, before the algorithm considers the I/O stream has entered into the new pattern. In another words, an I/O stream will switch to a new pattern when all $n$ bits are set. For example, consider a bit map with 3 bits. Initially, the process/file combination is classified as running in the random pattern, and all bit maps are clear. After a sequential access is captured, its bit map will be shifted to left by one bit and the least significant bit will be set to one. If next two accesses are still in sequential pattern, the bit map will be set to 111. Only in such a case, the pattern type of this process/file combination will be changed to the new sequential pattern. By doing this, we try to capture a pattern which exhibits strong and steady locality.

The pattern detector algorithm is described in Algorithm 3.1. The pattern transition is depicted in Algorithm 3.2.
Algorithm 3.1: Pattern Detector

Procedure PatternDetector
Input: pId, fId, reqSectorStart, reqSectorEnd
1. pageStart <- sectorStart / SectorPerPage
2. pageEnd <- sectorEnd / SectorPerPage
3. blockStart <- pageStart / PagePerBlock
4. blockEnd <- pageEnd / PagePerBlock
5. FileTable <- ProcessTable[pId];
6. FileEntry <- FileTable[fId];
7. currPtn <- FileEntry->PatternStruct
8. If(reqSectorStart == currPtn->lastSectorEnd + 1)
9. If (currPtn->PatternType != SEQUENTIAL)
10. currPtn->ptnSeq <- currPtn->ptnSeq << 1
11. else
12. currPtn->ptnSeq < 1
13. currPtn->lastDetectPtn < SEQUENTIAL
14. else if(reqSectorStart < currPtn->lastSectorEnd + 1)
15. if(currPtn->lastPageEnd == pageStart)
16. if (currPtn->PatternType != PAGE_CLUSTER)
17. currPtn->ptnPgCls <- currPtn->ptnPgCls << 1
18. else
19. currPtn->ptnPgCls < 1
20. currPtn->lastDetectPtn < PAGE_CLUSTER
21. else if (currPtn->lastBlkEnd == blockStart)
22. if (currPtn->PatternType != BLK_CLUSTER)
23. currPtn->ptnBlkCls <- currPtn->ptnBlkCls << 1
24. else
25. currPtn->ptnBlkCls < 1
26. currPtn->lastDetectPtn < BLK_CLUSTER
27. else
28. if (currPtn->PatternType != RANDOM)
29. currPtn->ptnRandom <- currPtn->ptnRandom << 1
30. else
31. currPtn->ptnRandom < 1
32. else if(reqSectorStart > currPtn->lastSectorEnd + 1)
33. if (pageStart == currPtn->lastPageEnd + 1 || pageStart == currPtn->lastPageEnd)
34. if (currPtn->PatternType != SEQUENTIAL)
35. currPtn->ptnSeq <- currPtn->ptnSeq << 1
36. else
37. currPtn->ptnSeq < 1
38. else if (pageStart > currPtn->lastPageEnd + 1)
39. if (currPtn->lastBlkEnd == blkStart)
40. if (currPtn->PatternType != BLK_CLUSTER)
41. currPtn->ptnBlkCls <- currPtn->ptnBlkCls << 1
42. else
43. currPtn->ptnBlkCls < 1
44. currPtn->lastDetectPtn < BLK_CLUSTER
45. else
46. if (currPtn->PatternType != RANDOM)
47. currPtn->ptnRandom <- currPtn->ptnRandom << 1
48. else
49. currPtn->ptnRandom < 1
50. currPtn->lastDetectPtn < SEQUENTIAL
51. else if (pageStart > currPtn->lastPageEnd + 1)
52. if (currPtn->lastBlkEnd == blkStart)
53. if (currPtn->PatternType != BLK_CLUSTER)
54. currPtn->ptnBlkCls <- currPtn->ptnBlkCls << 1
55. else
56. currPtn->ptnBlkCls < 1
57. currPtn->lastDetectPtn < BLK_CLUSTER
58. else
59. if (currPtn->PatternType != RANDOM)
60. currPtn->ptnRandom <- currPtn->ptnRandom << 1
61. else
62. currPtn->ptnRandom < 1
3.3.2 Improving Sequential Write Performance

To solve the potential hazards of sequential writes mentioned in the earlier section, one technique we proposed is called *pre-alignment*. The idea is that once the pattern detector figures out that a sequential write touches a block in the first time and does not start from the block boundary as shown in Figure 3.2(a), the scheduler will *prepend* a few extra write requests to the sequence to bring it to alignment. The extra write requests will copy the first few data pages, which are to be skipped by the original write request, from the data block to the log block. Using the above example in Figure 3.2(a), the scheduler will read page 0 and 1 from the SSD and prepend the writes to the two pages to the original unaligned requests. While this seems to result in two extra page-reads and two extra page-writes, overall we will achieve a major performance gain. The reason is that by using a few extra reads and writes, the FTL may perform a simple block switch in GC rather than a full merge. It thereby avoids one extra block erasure, which is extremely expensive, and avoids many costly copy operations associated with a full merge. To put it another way, these prepended read and write operations will be compensated later.
because in the case of a full merge, these pages which are prepended here will still need to be read out from the old data block and written to the new data block. Moreover SSD operations are highly asymmetric. Read performance is superior in SSDs and read is not evidently subject to different patterns (i.e., random or sequential). Even more, if the pages from the prepended read request are hit in the buffer cache, there will be no need to read out from the device. Finally, all these scheduling operations are performed at the OS level and are transparent to SSDs. There is no need to modify the OS/disk interface in order to pass the semantics down to devices.

To fix the problem caused by a less-than-perfect sequential write shown in Figure 3.2(b), where a sequential write skips some pages in the middle, we proposed a technique called *inner-padding*, which manually inserts write requests of skipped pages. This is similar to pre-alignment except that the inserted reads/writes are in the middle of a block. Because a block contains a number of pages, for either pre-alignment or inner-padding, we try to avoid the risk of issuing too many such read and write requests for a series of write requests with small footsteps. For example, assume there are 64 pages per block, and a series of sequential writes updates page number 30, 31 and 32. In this case, the scheduler will disable pre-alignment, since if it prepends the first 30 pages and the request pattern switches to non-sequential, it may invoke too much additional cost to the SSD without achieving a light-cost block switch. To this end, the scheduler will evaluate the average request size, and inner-padding or pre-alignment is only executed for large sequential writes. When the average write request size is above a threshold in pages (15% of a block size in our current setting), paddings and pre-alignment will be activated.
3.3.3 Improving Non-Sequential Write Performance

In addition to sequential write performance, non-sequential and random write performance is also very important. Previous studies have shown that in real systems, most write requests demonstrate very good temporal and spatial localities. Pure random writes do not happen very frequently. For example, Ruemmler and Wilkes [13] found out that an NVRAM cache of only a few hundred KBs can absorb most metadata write requests. A high percentage of written data will be quickly overwritten within a very short period of time, and this is particularly true at the page or block granularity.

While such strong localities are a major plus to regular disk systems since they can be exploited to improve performance, they pose a very unique challenge to SSDs and call for careful consideration, because data in flash memory cannot be overwritten. Repeatedly updating a page in an SSD will only cause many pages being appended to the log block, and eventually cause an expensive merge operation, reducing both the performance and reliability. We define such repeated access patterns to a page or a block as page clustering or block clustering, which can be recognized by our pattern detector.

When a clustering pattern is detected, the scheduler tries to apply a technique called write merging to improve the performance. Instead of sending the write request to the SSD immediately upon receiving it from the process, the scheduler will hold the request for a short period of time as evaluated by the current stream’s inter-arrival time, say a few tens of milliseconds, anticipating that the next request overwriting the same page will arrive so that the two can be merged. For a block-clustered write, not only the requested page but also the pages in the same block which are in the OS write buffer receive a delay too, anticipating the block will get overwritten shortly.
3.3.4 Merging also Benefits Sequential Writes

A simple extension to the above write merging technique, called *merging-splitting*, can be used to effectively solve the third sequential write hazard shown in Figure 3.2(c). As a further illustration, consider a series of sequential write requests as shown in Figure 3.4. The process first writes to sector #0, then sector #1 and #2, followed by sector #3 through #8. The writes are strictly sequential. However, the SSD sees a very different picture. Assuming that each page accommodates four sectors, the SSD will receive a write sequence of page #0, #0 and #0 through #2. Again such repeated writes to the same pages add major overheads to the FTL, shorten the life-span of the device, and reduce overall performance.

Once such patterns are detected, merging-splitting works by waiting for a short period before issuing a write request to the SSD, hoping to merge the request with the next upcoming sequential write request to the same page. It may split a single request to two sets of pages — one includes the sequence of pages before the last partially written page and the other is the partially requested page. As an example, in the write request #3...
in Figure 3.4, a write request with page #0 and #1 will be dispatched to the device while the partially written page #2 will be deferred. The time to be deferred will be determined according to the inter-arrival time of the current I/O stream as it is captured by the pattern detector.

### 3.4 Experiment and Performance Evaluation

Table 3.1: *SSDSim* Parameters

<table>
<thead>
<tr>
<th><em>SSDSim</em> parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Size</td>
<td>2 KB</td>
</tr>
<tr>
<td>Sector Size</td>
<td>512 B</td>
</tr>
<tr>
<td>Block Size</td>
<td>128 KB</td>
</tr>
<tr>
<td>Capacity</td>
<td>8GB</td>
</tr>
<tr>
<td>Page Read</td>
<td>33 us</td>
</tr>
<tr>
<td>Page Write</td>
<td>102 us</td>
</tr>
<tr>
<td>Erase</td>
<td>1.5 ms</td>
</tr>
<tr>
<td>Log block ratio</td>
<td>3%</td>
</tr>
</tbody>
</table>

Our proposed pattern-guided scheduler was evaluated through a set of simulators including an SSD device simulator and a system level simulator. The flash memory based SSD simulator named as *SSDSim*, was revised based on Disksim [67] which is originally developed by the Parallel Data Lab at Carnegie Mellon University. Disksim emulates the components within a HDD based storage system, such as rotational hard disks, buses, controllers, and etc. We replaced the HDD module with an SSD module and retained the storage hierarchy infrastructure of Disksim. In our SSD module, two log-structured block-based FTL mapping schemes — BAST and FAST were implemented in the *SSDSim*. In the FAST scheme, as its originally proposed in the work [49], a single log block is used as the dedicate sequential writes’ log block. The rest of log blocks are used for random data page updates. To evaluate our proposed pattern-based I/O scheduler, a
system level simulator, which includes functionalities such as LRU buffer cache and baseline *Noop* scheduling as well as the proposed pattern detector and pattern-guided scheduler, is incorporated on top of the SSDSim. The parameters of SSDSim are shown as in Table 3.1.

### 3.4.1 Benchmark Programs and Trace Collector

We tested a set of commonly used benchmark programs, including GCC, glimpse (a source code scanner tool), TPCH/TPCR and Firefox. File system traces were captured with a modified Linux *strace* utility [68] which can intercept system calls in the context of a process.

GCC (ver. 4.3.3) is evaluated by letting it compile the Linux source tree (ver. 2.6.34). In the GCC workload, write requests dominate. It features a large volume of writes which are either sequential or block/page clustering. Glimpse [69] is a text scanner tool often used to search source code files. It creates an index database to facilitate searching. We used the tool to scan Linux source code under */usr/src/* directory, and recorded the traces during the database creation stage. The benchmark features a large volume of write requests, dominated by large sequential writes. It also issues a large number of read requests, most of which are sequential too.

TPCH and TPCR [70, 71] are two popular database benchmarks. Both are dominated by significantly large numbers of read requests rather than writes. As a result, we do not expect to see noticeable performance improvement under our scheme. In terms of writes, we found that they feature a mix of different write patterns. None of the patterns dominates. This is due to the randomness of database search and update transactions.
Recently Firefox web browser received more attention to be used to evaluate file system performance. Firefox trace was recorded via browsing multiple web sites such as news, interactive web forms, online videos and etc.

3.4.2 Simulation Results

The metrics for the performance evaluation includes the I/O time, including read, write and the time invoked by garbage collection. We also evaluated the wearing overhead in terms of the total erase cycles on the SSD device.

3.4.2.1 Wearing Reduction

For all benchmarks, our algorithms significantly reduce the number of erasures, which are the most expensive operation in an SSD, as shown in Figure 3.5. The reduction on erase cycles is the result that the pattern based scheduler can detect and eliminate most of sequential write hazards. Moreover by deferring and grouping clustering patterns’ write requests, a large number of clustered writes (page or block) can be coalesced. For block clustering, it may potentially dispatch a series of block clustering writes in a sequential order, so that a light-cost block switch is invoked by FTL in the device.
3.4.2.2 I/O Time Improvement

Our results show that the proposed schemes can indeed significantly improve I/O performance under either BAST or FAST log-structured block-based FTL, as shown in Figure 3.6. Using patterns, our algorithms reduce the I/O time of glimpse by almost 46% and that of Firefox by over 37% under BAST FTL. Under FAST FTL, I/O time is reduced by 56% in GCC and by 51% in glimpse. Only slight performance improvements are delivered on TPCH and TPCR, which are expected, as these traces are read dominated. Although pattern based scheduler injects additional read requests, overall it achieves performance gain by improving write performance and reducing erase cycles.
Figure 3.6: Overall I/O Time Improvement

Figure 3.7 pays a close examination of the write performance which also includes the garbage collection overhead. Clearly, the pattern-guided scheduling did an impressive job of reducing write I/O time. In fact, the largest reduction (over 60%) comes from TPCR. However this trace contains few writes, so the overall performance improvement in Figure 3.6 is not as impressive. TPCH shows a similar result for the same reason. For GCC, the improvement is more significant under FAST than BAST. This is, we believe, related to the single dedicated sequential write log block as GCC is dominated by multiple concurrent sequential write I/O streams. For other benchmarks, the improvement is similar for either FAST or BAST FTL.
3.4.2.3 Results of Pattern Detections and Scheduling Decisions

Table 3.2: Number of Requests Seen By SSD

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>READ</th>
<th>WRITE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Baseline</td>
<td>Pattern Based</td>
</tr>
<tr>
<td>gcc</td>
<td>32,373</td>
<td>32,504</td>
</tr>
<tr>
<td>glimpse</td>
<td>2,383,935</td>
<td>2,383,935</td>
</tr>
<tr>
<td>tpch</td>
<td>9,025,385</td>
<td>9,025,370</td>
</tr>
<tr>
<td>tpcr</td>
<td>7,330,189</td>
<td>7,330,152</td>
</tr>
<tr>
<td>firefox</td>
<td>21,902</td>
<td>21,923</td>
</tr>
</tbody>
</table>
We also looked into the details that how the pattern based scheduling affects the behaviors of the device by capturing write access patterns and applying different scheduling policies. Table 3.2 shows the distribution of read and write requests in terms of the number of pages committed to the device. It is obvious to see that pattern-based scheduling reduces many write requests by merging them together. It also introduces slightly more read requests because of the alignment and padding operations, which does not pose a major concern since read operations are much faster than writes and erasures.

Patterns are detected “on-the-fly” in real time. Table 3.3 shows how many times various pattern transitions have occurred. As mentioned before, to filter out noise in access patterns, in our experiment, the algorithm must detect the same pattern three times in a row before it declares that the I/O stream has entered a new pattern. While simple, our results suggest that the algorithm may seriously underestimate the number of useful patterns. In the future study, a more accurate detection algorithm should be under investigation. Here Gcc shows a mix of different patterns, although sequential writes dominate. Glimpse has only sequential writes. This is the result that it creates the index files in sequential after scanning the sources. The TPC benchmarks’ writes are largely non-sequential. However it is clear that a large percentage of these non-sequential writes

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Sequential</th>
<th>Page Cluster</th>
<th>Block Cluster</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>1572</td>
<td>855</td>
<td>956</td>
<td>195</td>
</tr>
<tr>
<td>glimpse</td>
<td>585</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tpch</td>
<td>448</td>
<td>1188</td>
<td>346</td>
<td>843</td>
</tr>
<tr>
<td>tpcr</td>
<td>405</td>
<td>1088</td>
<td>330</td>
<td>769</td>
</tr>
<tr>
<td>firefox</td>
<td>723</td>
<td>24</td>
<td>288</td>
<td>112</td>
</tr>
</tbody>
</table>
shows good locality and can be classified into page/block clustering. Firefox shows mix of patterns with similar weights.

Table 3.4: Number of Scheduling Decisions

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>block align needed</th>
<th>block already aligned</th>
<th>pre-align</th>
<th>inner-padding</th>
<th>split sequential request</th>
<th>delayed page cluster</th>
<th>delayed block cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>93</td>
<td>5,497</td>
<td>22</td>
<td>1</td>
<td>10,759</td>
<td>2,885</td>
<td>3,252</td>
</tr>
<tr>
<td>glimpse</td>
<td>0</td>
<td>23,575</td>
<td>0</td>
<td>0</td>
<td>1,475,980</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tpch</td>
<td>0</td>
<td>714</td>
<td>0</td>
<td>0</td>
<td>39,906</td>
<td>12,538</td>
<td>1,715</td>
</tr>
<tr>
<td>tpcr</td>
<td>0</td>
<td>710</td>
<td>0</td>
<td>0</td>
<td>39,003</td>
<td>11,434</td>
<td>1,617</td>
</tr>
<tr>
<td>firefox</td>
<td>2</td>
<td>1,744</td>
<td>0</td>
<td>0</td>
<td>18,955</td>
<td>54</td>
<td>2,683</td>
</tr>
</tbody>
</table>

Table 3.4 lists the number of scheduling decisions based on associated write access patterns. For instance, since all write requests from glimpse are block aligned, there is no need to pre-align a sequential write. However since the ending addresses of write requests from glimpse are mostly not aligned with the page end, the scheduler split a large number of sequential write requests, anticipating that the following write would make it aligned. Please note that the numbers of decisions are much larger than the numbers of pattern transitions in Table 3.3. The reason is that the programs can access different data while staying in one pattern. For example, a program may sequentially write one address range, and then sequentially write another different address range. We classify the entire period as a single sequential transition, while the scheduler may have to issue multiple alignment operations at the beginning or end addresses of the two different regions.
Finally, while the scheduler inserts pre-aligning and padding requests, not all the extra reads will go to the device, since there are many read buffer hits at the OS level, so the actual overhead is even lower.

### 3.5 Related Works

Fine-grained access patterns have been exploited in HDDs [16, 72-76]. Current system designs are mostly tailor to HDDs’ characteristics. For example, the buffer cache replacement algorithm was modified so that the buffered pages in the cache which exhibit sequential access patterns will be evicted at a higher priority than the pages which demonstrate loop patterns, despite that the LRU policy may evict the second case. Since the pages which show loop patterns may be more likely to be referred again than the sequential ones, loop reads can catch more hits in the cache. Most previous solutions mainly differed in the granularity that the patterns are classified (per file, per process or per program counter).

### 3.6 Summary

While SSDs have demonstrated many notable advantages over HDDs, they also pose some unique challenges to I/O and file system designers, since many traditional solutions optimized for HDDs may not work well for SSDs. For example, I/O workloads that demonstrate strong temporal locality on writes, which benefit almost all traditional systems, may actually cause significant performance and reliability degradation in SSDs, because the latter cannot perform in-place updates. Moreover, sequential writes that are
not perfectly aligned with the block boundary, may also increase overhead and reduce reliability.

In this study, we proposed to capture file access patterns in a per-process per-stream manner. These patterns are then used to guide a set of novel scheduling policies, including \textit{pre-alignment, inner-padding, write merging, merging-and-splitting}, to improve the write performance of SSDs which employ log-structured block-based FTLs. The pattern based I/O scheduler takes special considerations of many unique requirements of SSDs, and is highly optimized for SSD-based storage architectures. Moreover, our solutions do not require any modifications to the existing disk I/O interface.

Simulation results show that pattern based scheduling schemes can improve write performance by up to 60\%. Moreover, the schemes reduce SSD’s erase cycles by up to 64\%, which is directly translated to a major improvement on the lifespan of SSDs.
Chapter 4

A Flash-Aware Buffer Cache

Management

4.1 Introduction

In the era of HDDs, OS buffer cache plays a critical role in bridging the longer-latency storage device with faster processing unit. For reads, it helps maintain the disk blocks for a period of time. A future request to the same block will be directly serviced by the buffer cache. Moreover, the buffer could host read-ahead blocks to serve future sequential reads. For writes, dirty file blocks are temporarily hosted in the buffer cache, so that any overwrites to the not-yet-flushed dirty file blocks will be coalesced. Also, grouping the dirty file blocks together could improve responsiveness since in HDDs sequential accesses outperform random ones. Since buffer cache is much smaller than the
storage of device, the pollution control [77] and eviction policy are under investigation. In line with the temporal locality, LRU [78] is a much simple and effective scheme.

SSDs exhibit different characteristics in read and write operations, therefore traditional buffer cache design may not be appropriate for SSDs. For example, the prefetching is effective in HDDs considering HDDs’ performance difference in sequential and random read accesses. However, SSDs expose much less performance difference in sequential read accesses and random read accesses, since there are no long-latency moving parts. Due to this, many researchers suggest to disable prefetching when data are hosted on SSDs.

4.2 Background

Buffer cache has been under extensive investigations to tailor for SSDs’ unique properties. For example, Kim et al. in [79] proposed to use the whole on-device cache as the write buffer due to the asymmetrically higher overhead of write operations than read. For another instance, in the work of [80] from Park et al., the authors presented CFLRU which evicts OS buffer cache by considering a page’s dirty or clean status. More specifically, a clean page is given higher preference to be reclaimed.

In a product system, buffer cache’s replacement is not always tied to the moments when dirty file blocks are flushed to the backend device. For example, in a typical Linux system [81], a dirty file block in the OS buffer could be flushed to the backend device under three possible scenarios:
1) When the amount of dirty pages in the buffer cache is beyond a pre-configured ratio, the dirty page flush will be executed *regardless* of whether the buffer cache is full or not. Actually in this case, the dirty pages will still be in the cache without being evicted, after they are committed to the device. Any future read to the same page will be hit in the cache before it is evicted;

2) When a page stays dirty in the buffer cache for a pre-configured time and then expires, that dirty page will be flushed still with no regards to whether the buffer cache is full or not;

3) At the last moment, the buffer cache runs out of space, thus some pages have to be evicted from it. In this case, a heuristic scheme is adopted in Linux which gives higher reclamation preference to clean pages than dirty ones. If there are no more clean pages to be evicted, a dirty page will be flushed and evicted.

### 4.3 Observation and Motivation

Current buffer cache design which only takes consideration of temporal locality is suboptimal for an SSD storage system. Multiple processes running at the same time could lead to many dirty pages in the OS buffer, exhibiting different write patterns. As a simple example, a process which issues write requests in a random manner is running simultaneously with a process which writes files sequentially. The file blocks written by the former process may be overwritten again in the near future because of the spatial locality, while the file blocks written by the installer will be unlikely to be modified again. The traditional LRU buffer cache may not work well on this. It will be a good idea to
keep dirty pages of the former in the cache for a longer period of time to capture both temporal and spatial localities. Keeping the dirty pages from the installer program in the cache, on the contrary, will be a waste of space and will not benefit write performance, since these blocks are not likely to be accessed again.

![Diagram of write requests](image)

Figure 4.1: A series of sequential write requests in sectors are mapped to the same page unit

Consider another scenario which may invoke additional overhead. We refer this as write-hazard. Consider a process which issues small write requests in sectors in a sequential order. In an SSD device, a write operation is performed in the unit of a page, which contains multiple sectors in size. Adjacent write requests from this process may be mapped to the same logical page unit. As shown in Figure 4.1 which assumes a page containing four sectors, the request #1 writes sector #0 which falls in the page #0, following that request #2 writes sector #1 and #2 which are still in the page #0. Then the
third request writes sector #3 through #8 which also starts from page #0. Invoking write buffer flushing between request #1 and request #2 or between request #2 and request #3 may append duplicate pages into the log block in the backend device, as shown in Figure 4.2 where the page #0 is logged twice. This will result in an expensive block full merge during the garbage collection.

Figure 4.2: A series of sequential writes may still cause a costly log block merge. Note that page #0 was logged twice

4.4 Design of Flash-Aware Buffer Cache Management

In this work, we propose a flash-aware OS buffer cache which exploits both temporal locality and spatial locality. We extend our research in fine-grained access pattern analysis to the buffer cache management. With these patterns, we design a set of dirty
page flushing policies for the OS buffer cache. We virtually partition the dirty pages according to their associated write patterns.

![Pattern Based Dirty Pages Partition](image)

**Figure 4.3: An example of pattern based dirty pages partition**

For each pattern, we devise an adaptive dirty flush policy. Each dirty page in the buffer cache is associated with a pattern type. The dirty pages with the same pattern will be linked together in a linked list, so that dirty pages in the buffer cache are virtually partitioned according to their associated patterns. When a page is dirtied by a process, it will be moved to the head of the pattern list. As a result, in each list, the head will be the most recently written page while the tail will be the least recently written one.

When the system needs to flush dirty pages, it will scan the lists according to the following priorities. The sequential pattern will be given the highest priority to be flushed since its pages are the most likely to be written only once, therefore it is pointless to keep them in the cache. The random will be given the next priority. The page clustered and block clustered dirty pages will have the lowest priority, since they may be overwritten in the near future hence we want to keep them in the cache.
As an example, consider the buffer cache in Figure 4.3. The referenced pages including dirty \((D)\) and clean \((C)\) pages are linked together in LRU order. When the buffer cache runs out of free space, a clean page (page #3 in this example) will be selected to evict, since read requests have much lower overhead than write requests. If there are no clean pages available, dirty pages which exhibit sequential pattern will be evicted first. In this example, the system will evict page #2 instead of page #1.

In addition to the above mentioned on-demand eviction, many systems will start flushing the dirty pages when the number of dirty pages exceeds a ratio of buffer capacity. In this case, the disk blocks belonging to sequential pattern will also be given a higher priority to move to the SSD than those of other patterns. Page #2, #4 and #6 in the virtual list of sequential pattern will be flushed in a sequence in its original requested order.

To prevent a block from staying in the cache for excessive time without getting a chance to be flushed, a scan flag is given to each of such pages. When the policy scans the dirty pages in the lists of page or block clustering type, a second chance will be given to each page. A flag will be set for each such a page when the scanner scans that page for the first time. If the page is overwritten, its flag will be cleared. Otherwise, when the scanner reaches this page and if its flag is still set, it will be flushed. When a dirty page of block clustering pattern is hit in the buffer, the dirty pages in the buffer which belong to the same block unit will have the flag cleared too. In this way, the pages written in a clustering pattern to a certain block may get a chance to coalesce and reorder. The pages will be flushed from each list till a pre-set number of dirty pages or all of the dirty pages are written to the SSD. Moreover, for the block clustering pattern, the dirty pages in the same logic block unit will be flushed in the ascending order of the page number,
regardless of the requested order. By doing this, an expensive block merge may potentially be converted to a less costly block switch in the SSD.

To avoid the write hazard mentioned earlier, when a sequence of sequentially written pages are flushed, the last page of this series of writes which is not aligned with the flash page boundary will be held in the cache, hoping that the next sequential request can be merged with it. As an example, in the write request #3 in Figure 4.1, only the page #0 and #1 will be flushed, while because page #2 is partially written, it will be put on hold in the cache, setting the flag, anticipating the next sequential write will make it aligned. The pseudocode of dirty cache flush algorithm is given in Algorithm 4.1

---

Algorithm 4.1: Dirty Cache Flush

```
Procedure DirtyCacheFlush
enum PatternType {SEQUENTIAL, RANDOM, PAGE_CLUSTER, BLK_CLUSTER}
1. for (currType := SEQUENTIAL; currType <= BLK_CLUSTER; currType++)
2.   dp := Partition[currType].prev
3.   while (dp != &Partition[currType])
4.     switch (currType)
5.       case SEQUENTIAL:
6.           if (dp->unaligned)
7.               if (dp->deferTimes < DEFER_THRESHOLD)
8.                 dp->deferTimes := dp->deferTimes + 1
9.               else
10.                  commit dp
11.             else
12.                commit dp
13.       case PAGE_CLUSTER:
14.           if (dp->deferTimes < DEFER_THRESHOLD)
15.             dp->deferTimes := dp->deferTimes + 1
16.           else
17.             commit dp
18.       case BLK_CLUSTER:
19.           if (dp->deferTimes < DEFER_THRESHOLD)
20.             dp->deferTimes := dp->deferTimes + 1
21.           else
22.             commit all pages in the same logical block with dp in page ascending order
23.       case RANDOM:
24.           commit dp
25.     dp := dp->prev
```

---

55
4.5 Experimental Setup and Performance Evaluation

4.5.1 Simulator

We developed a system level simulator on the top of the SSD simulator (SSDSim) which is based on Disksim [67]. The system level simulator implements the LRU buffer cache including the baseline heuristic write buffer which is employed in a product system, as well as pattern-based flash-aware buffer cache. A log-structured block-based FTL [48] is implemented in the SSD simulator. The system starts to write dirty pages back to the SSD if the total number of dirty pages exceeds an upper bound (20% of total buffer capacity in the experiments). In the baseline system, dirty pages are grouped by their belonged inode and flushed, while in the pattern based system, dirty pages are flushed according to their associated patterns. The system also flushes a dirty page when it stays in the buffer cache beyond a certain time. We set this time to 120 seconds. For both baseline and pattern-based schemes, if the buffer cache has no free space and needs to be freed up, a clean page is given higher priority than a dirty one to evict. In the case that there are no more clean pages to be reclaimed — this happens when dirty pages from a burst of write requests are appended to a small buffer, for the baseline scheme, it flushes and reclaimas pages according to the LRU order while for the pattern-based scheme, it flushes and reclaims dirty pages according to the pattern priority mentioned in the earlier section. We disabled file block prefetching in the experiments, since unlike HDDs, SSDs do not benefit from prefetching due to the absence of seek and rotation latency. We also varied the system buffer cache size throughout the experiments.
4.5.2 Benchmark Programs

We tested a set of write-dominant benchmark programs, including *Gcc*, *Eclipse IDE*, *Firefox and HTML Rendering*, to evaluate the proposed algorithms. Traces were captured at file system layer with a modified Linux *strace* [68] utility which is to intercept disk-related system calls in a process context.

*Gcc* (ver. 4.3.3) was recorded by letting it compile the Linux kernel source code (ver. 2.6.34). In *Gcc* workload, large sequential write requests issued by many concurrent processes dominate. The *Firefox* trace was collected by mimicking regular usage by browsing multiple web sites such as news, interactive web forms and online videos. The *Eclipse* and *HTML* benchmarks are part of *DaCapo* benchmark suite [82] which consists of a set of real-world applications. *Eclipse IDE* is simulated to initialize, build and deploy a Java project. In the *HTML* benchmark, a large set of HTML files are rendered from XML documents.

4.5.3 Performance Evaluation

4.5.3.1 Write Performance
Figure 4.4: Write time and garbage collection overhead

We evaluated the write performance in terms of total elapsed write time which includes the time for flash memory program operations and the overhead of garbage collection, including log blocks’ merge and switch. For all benchmarks, our pattern based
write buffer delivers higher performance than the baseline heuristic scheme, as shown in Figure 4.4, under varied buffer cache size. This can be directly seen in Figure 4.5 which shows the ratio of the time reduced by our pattern based write buffer compared to the baseline. The pattern based write buffer shows significant performance improvement over the baseline system for a relative small buffer cache. For example, in the HTML rendering benchmark, there is more than 38% performance gain for 1 MB cache and 15% for 8 MB cache. Under the Firefox workload, total write overhead is reduced by 18% for 1 MB buffer and 5% for 8 MB one. Even for a larger cache, our pattern based buffer cache still gains noticeable performance improvement. For example, in Gcc benchmark, the pattern based buffer cache improves write performance by around 20% for 16 MB cache and 21% for 32 MB one. In HTML rendering benchmark, write performance is improved by 10% for 16 MB cache and by 6% for 32 MB one.

Figure 4.5: Write performance improvement ratio
We also evaluated wearing overhead invoked on the SSD device, indicated by total erase cycles (E/C). As shown in Figure 4.6 and Figure 4.7, the pattern based write buffer
significantly improves the device life-span by reducing the total block erase cycles. For example, in HTML benchmark, E/C is reduced by around 56% for 1 MB cache and by 30% for 8 MB cache. In Firefox benchmark, the reduction is around 39% for 1 MB cache and 13% for 8 MB one. Even for a relative large cache (32 MB), Gcc benchmark can still see 27% improvement in E/C and HTML benchmark can see 13% improvement. The improvement on wearing overhead is contributed by deferring clustered pattern writes, as well as grouping dirty pages which exhibit block clustering write pattern. Moreover giving higher priority to flush dirty pages from sequential pattern over the pages from other patterns reduces the total overwrites to the device. Even more, by detecting and deferring unaligned write requests, the pattern based write buffer avoids the sequential write hazard which may potentially cause a block full merge rather than a block switch.

Figure 4.7: Erase cycles reduced by Pattern-Based-Buffer
Figure 4.8: Erase cycles invoked by full merge and block switch respectively
Figure 4.8 further breaks down the number of erase cycles by indicating the erase cycles invoked by log block merge operations (shown as black area) and the ones invoked by log block switch operations (shown as shaded area). As mentioned earlier, a block switch is less costly than a block full merge since each block merge invokes two block erase operations — one on the old data block and the other on the log block, while a block switch invokes only one erasure. It is obvious from the figure that the pattern based write buffer could either eliminate some of the block full merges or convert some into block switches, which corresponds to lower wearing overhead and higher write performance.

4.6 Summary

Access patterns, especially write patterns, may significantly affect SSDs’ performance due to SSDs’ unique properties. In this chapter, we present a novel write-buffer management scheme which is based on fine-grained write access patterns. Our solution captures and classifies the patterns into sequential, clustering, and random writes. Dirty pages from sequential write streams will be given a higher priority to be committed to the SSD device than those from other types of patterns. As a result, dirty pages of clustered or random write streams will stay longer in the cache, so they will have better chances to coalesce or reorder within a flash block which is the unit for garbage collection. This potentially converts scattered writes into sequential accesses. Moreover, to avoid an SSD-specific problem called write hazard, where two consecutive sequential write requests are mapped to the same flash page, our scheme may put the last page from
an unaligned sequential write on hold for a short amount of time. This will reduce the risk that duplicated pages are appended to the log block in the device. Our simulation results show that our solutions can improve write performance by up to 38% for a relatively small buffer cache and 21% for a larger one. Moreover, our schemes can reduce SSD erasure cycles by up to 56% for a smaller cache and up to 27% for a larger one, which is directly translated to a major improvement on the life-span of SSDs.
Chapter 5

An Internal Redundant Storage Architecture

5.1 The Many Challenges with SSD Reliability

SSD-based storage poses some very unique reliability challenges, in terms of life-span and error rates, to the system designers. The most well-known problem is write endurance. Flash memory can only sustain for a finite and rather low number of erasures, such as 100k cycles for SLC NAND or only 10k cycles for MLC NAND.

Even worse, previous studies found that MLC memory shows significant error rates from the very beginning, and both SLC and MLC memory’s Bit Error Rates (BERs) start to increase sharply long before the cells reach their rated lifetime [25, 26]. While a robust...
ECC (Error Correction Code) scheme can be extremely effective to protect the data inside young blocks which have low BERs, the risk of uncorrectable data corruption increases significantly for aged data blocks.

While wear-leveling algorithms [83, 84] can mitigate the problem, the fundamental limitation of write endurance still seriously affects the life-span of SSD-based storage systems [85]. In enterprise servers and data centers, the limited write endurance becomes a much bigger hurdle, since these applications see much higher I/O traffic than personal users do.

Another less well-known but equally serious problems are called read-disturb and program-disturb (or write-disturb) [25, 27-29]. Whenever a cell is read or written, a voltage is applied to the block, which might stress neighboring cells in the block, causing them to gradually lose data. This is a foundation limitation of flash memory that cannot be avoided. While the disturbs cause relatively few error rates for a fresh new flash block, recent studies show that read-disturb error rates increase by several orders of magnitudes when the block is erased by as few as 2000 times [27]. This implies that there will be data corruption when an SSD is repeatedly read! In severe cases even a strong ECC scheme cannot correct the errors. To mitigate this problem, many systems use a preventive measure called “scrub” [27, 86], which periodically reads data from the SSD, corrects it with ECC, and writes it back. However, doing so will increase the number of writes and erasures, aggravating the reliability and performance problems. As the density of flash memory keeps increasing and the process geometrics decreasing, read-disturb and program-disturb error rates will be increasing quickly, so the problem will become an even bigger concern in the future [31-33, 87]. For example, as indicated
in [88], the density increased by 400% from 2008 to 2010, while the cell endurance dropped from 10,000 to 3,000 cycles for a 25nm MLC device. Moreover, disturbing is even more severe for TLC devices in which each memory cell stores three bits using eight states [89].

It is important to notice that bit errors due to aging flash cells or disturbs might not be the only source (or even the major source) of errors in SSDs. In a recent paper [90], Zheng et al. showed that SSDs might experience massive errors under power faults. Specifically, they found that thirteen out of the fifteen tested SSD devices exhibit surprising failure behaviors under power faults! These errors are often not correctable by ECC. In fact the authors went so far to suggest that SSD devices should not be used for storing important information because of the risk of massive data loss, unless the particular SSD model under power failure is carefully studied and understood. Clearly, introducing block-level redundancy is one of the few ways to alleviate such data failures.

In addition to the reliability problem, studies have shown that aged data blocks also introduce a huge performance penalty. When reading these old blocks, the system often has to undergo many costly retries since these blocks are unreliable. Such deterioration of read performance is not acceptable in enterprise environments, which demand for stable, predictable performance [31].
5.2 Background

5.2.1 RAID Technique

The RAID (Redundant Array of Independent Disks) technique [12] has long been exploited since HDDs. Parity updates are known to be expensive because of the severe ‘small update problem’ [91], where a small update results in four disk accesses — reading old data, reading old parity, writing new data, and writing new parity. Savage et al. [92] proposed to delay parity updates till the disk array is idle. In such systems data is not full-time protected, but the performance can be close to a non-redundant disk array.

Parity logging proposed by Stodolsky et al. [93] writes the parity update out-of-place to a log disk. The mechanism involves several stages. It first buffers the partial parity which only includes the updated data, and then appends all the partial parity images in the buffer to a dedicated log disk in sequential writes. Later on, the partial images as well as their outdated parity will be read out in sequential and XORed and then written back in sequential to the original parity locations. By doing these, scattered random parity accesses are converted to sequential disk accesses which deliver much higher performance in traditional hard disks.

5.2.2 RAID on SSDs

For SSDs, a RAID system can be directly built on multiple SSD devices [28, 34, 94]. However, doing so may lead to many pitfalls [34]. Also as mentioned in [94], since SSDs face write-endurance problem and to avoid all devices wearing out at the same pace (thus more likely that more than one device corrupts at the same time), the author showed that RAID-4 is a better scheme than RAID-5 for an SSD RAID system.
There are also some other works [95, 96] which proposed to construct internal RAID inside an SSD such as RAID-5. Likewise, each flash write needs to read the old data and the old parity, and then write the new data and new parity in out-place manner. To reduce the impact of the high overhead of parity writes, some works [95, 96] proposed to add another level of non-volatile RAM (such as PCM) buffer cache to temporarily buffer parity blocks. For example, in [96], the author proposed to buffer the parity and track the old data to reduce — but not eliminate — read operations on the old data in a RAID stripe before the parity is written to the flash. Later on, this buffered parity is still needed to recalculate the parity of the whole stripe, and read operations on the old data and old parity are invoked too.

In the previous studies which adopted traditional RAID technique, a RAID stripe is statically constructed on logical addresses. Specifically, a RAID stripe is constructed in a way regardless of what physical flash memory blocks the logical addresses of data and parity blocks are mapped to, also regardless of what region the physical blocks are located. Those physical blocks could have different health information (e.g., erase cycles, error counter) or could be located at different regions (e.g., data region or log region). And the logical to physical mappings are dynamically changed along with data and parity updates. Therefore, it is not possible to selectively construct a RAID on specific physical blocks. Moreover, for a traditional RAID, any single update to the data needs a parity calculation and update. Although using buffering can alleviate this problem, it cannot completely eliminate it.

However our proposed i-RAID scheme fundamentally addresses these issues and it is distinguished from other previous works with the following key features.
• i-RAID constructs a RAID stripe on physical flash blocks, and a parity block’s calculation and update are completely deferred to the later stage of garbage collection time, since the out-place update nature will not overwrite the original physical blocks before garbage collection.

• Since a RAID stripe is dynamically organized in garbage collection time, it could better leverage flash chip parallelism. For example, if multiple flash memory blocks are garbage collected at different flash memory chips at the same time, these blocks could form a RAID stripe by i-RAID. There are no extra read operations overhead to the data blocks that are not under cleaning, that is, only the read operations which are executed by garbage collection process are needed. At the same time the write operations to the new parity block can be run in parallel with those operations executed by the garbage collection process.

• Constructing RAID parity for young fresh memory blocks is overkill for most SSD devices, as flash memory cells are quite reliable before any wear and tear have accumulated on the oxide insulation layers. In i-RAID, by leveraging the metadata health information of each individual flash memory block, selective RAID is possible, in other words, constructing RAID stripes ONLY on aged error-prone blocks with high erase cycles or error count.

• i-RAID also has the flexibility to form a RAID stripe with different number of data blocks, for example, forming a four data block RAID stripe on the flash memory blocks which are relative young and at the same time forming a three data block RAID stripe on the flash memory blocks which are relative aged,
offering the flexibility of different performance and reliability trade-offs. We leave this investigation in future studies.

- Constructing traditional RAID directly on SSD devices or flash memory chips may introduce many pitfalls [34, 94]. However, i-RAID can fundamentally solve these. For example, to avoid correlated failure, i-RAID can be configured to build RAID stripes on flash memory blocks with asymmetric aging progress, more specifically, to build RAID stripes on relative fresh blocks coupled with relative aged blocks.

5.3 i-RAID Design

5.3.1 Architecture

*i-RAID* is a set of novel parity-maintenance algorithms running *inside* the SSD controller, which controls multiple flash memory chips. The i-RAID algorithms are closely integrated with the FTL operations in order to minimize overhead.

One of the most unique features of i-RAID is *deferred parity maintenance*. Traditional RAIDs suffer from the small write penalty problem, where a small update results in four disk accesses — reading old data, reading old parity, writing new data, and writing new parity. The out-of-place update feature of SSDs means that new data will be written to the log region first, without overwriting the old data. Old data will co-exist with the new data until the garbage collection time. Before this time, the parity block and the old block are still valid to recover other data blocks in the stripe as shown in Figure 5.1. In this figure, even block #1 has been updated, the RAID stripe is still valid to
recover any data block in it (e.g., block #0), because the updated pages of block #1 are appended to the separate log region. Therefore there is no need to update the parity on every single write. In the current i-RAID architecture, the newly written data are written to the log blocks which are hosted in NVM. Since NVM has much better reliability and performance than flash memory, we didn’t adopt any RAID scheme on log blocks. We also leave NVM-based log blocks’ investigation in future studies. The new parity block needs to be reconstructed only during the garbage collection time, as shown in Figure 5.2.

Figure 5.1: i-RAID stripe and reconstruction during garbage collection before garbage collection
A flash memory block, which is usually composed of multiple pages (e.g., 64 2KB pages), is the atomic unit of cleaning in garbage collection. It is intuitive to execute i-RAID in the unit of a flash block also to avoid lots of small random updates. As shown in Figure 5.2 where data block #1 in package #1 will be merged with its log block to a new block, a new parity needs to be calculated over this new data block. During the merge process, a peer data block (assuming the stripe consists of two data blocks and one parity block) is selected to form a RAID stripe and a parallelized read operation is executed on this selected paired block and a parallelized write operation is executed on the parity block. Also from the figure, it is easy to see that because the peer block is also under
cleaning, these two blocks could form an i-RAID stripe dynamically. The read operations on the two blocks and the write operations to the new data blocks and parity block can be parallelized. Also under such a case, to maintain a RAID stripe, there are no additional read operations on any blocks which are not under garbage collection, which otherwise will be invoked by traditional RAID which is built on logical address space.

Moreover, unlike traditional RAIDs, in i-RAID, the construction of a stripe is not static. Stripes are dynamically constructed and re-constructed during the garbage collection process, and the blocks in a stripe can come from anywhere in the system. Such flexibility allows us to treat blocks differently. For example, the failure model of SSDs and HDDs are completely different. HDDs may crash at any moment so they have to be parity-protected all the time. SSDs, on the other hand, seldom suffer from sudden failures. Most devices, especially SLC devices that are likely to be used in enterprise systems, maintain very low bit error rates until the erase cycle reaches some threshold. They also have robust ECC schemes to handle the low bit error rates very effectively. This implies that for most applications, we can construct parity-protected stripes for only those **aged blocks** where ECC alone may not be sufficient, thereby avoiding many unnecessary overhead.

Finally, since blocks in the stripes are “float”, we avoid the correlated failures problem. In addition, our algorithm spreads the blocks in a stripe on different flash memory packages of an SSD in order to utilize the internal parallelism. For example, the parity write operations can take place concurrently with the garbage collection process most of the time. Also, our designs are integrated with the existing log-based schemes
used by modern SSDs, so they do not introduce additional complexity and overheads of yet another translation layer.

5.3.2 i-RAID Operation

To reduce the cost of parity maintenance, i-RAID uses flash blocks as the minimal striping unit. When a flash block’s erase count reaches above a threshold (the indicator of aging status), it will be labeled as a candidate to form an i-RAID stripe. Previous studies [25, 39, 97] suggested that as blocks wear out, their programming time will take longer while the erase time will be shorter. This can be used in future designs to adjust the threshold dynamically. In our current experimental simulation, however, we used only static thresholds. Note that if the threshold is 0, the data is always protected from the very beginning.

A stripe in i-RAID is made of several data blocks and one parity block. As mentioned earlier, for normal data blocks, i-RAID is triggered only when garbage collection starts and when the new data block to be assembled is an aged block which has been marked as a RAID candidate. Other data blocks in the stripe will be dynamically selected from distinct domains to exploit potential chip level parallelism. Additionally, a candidate free block will be used as the parity block. The system will select the parity block from another distinct chip.

The operations invoked by i-RAID include reading from the peer blocks, computing the parity, and writing to the parity block. In a normal data block merging process, reading from the old data/log block and writing to a new block are all needed, therefore
the operations which are to assemble the new data block and to assemble the parity block can be executed in parallel.

When a data block in a stripe is garbage collected and invalidated by the FTL, its belonged i-RAID stripe will be invalidated too, except in the case of a shadow stripe which will be addressed in detail shortly. The data block to be cleaned will be erased by the regular garbage collector. Also the parity block in this invalidated stripe will be reclaimed.

The rest blocks in the stripe, which are still valid, will be used to form other stripes so they will be continuously protected. At this moment, if there are no available stripes for these blocks to join, we use a technique called shadow stripe. In this situation, invalidating the stripe will be delayed by deferring the erases to the invalidated data block and the parity block, so that the shadow stripe can be used to recover any valid data blocks if they are failed before joining new stripes. Once these valid blocks are selected to form other stripes, or these blocks are also cleaned by the garbage collector at a later time, then this shadow stripe can be cleaned and the parity block will be erased.

(a) Timing diagram of normal cleaning operation
A schematic view of the parallelized operations is depicted in Figure 5.3. As indicated in the timing diagram which assumes a log-structured based FTL is adopted, the first scenario in Figure 5.3(a) is the normal cleaning operations which can be parallelized by means of copy-back operations or by distributing the data and log blocks into distinct parallel domains. Note the fact that old data block and log block could be in the same domain.

The second scenario in Figure 5.3(b) depicts the case when i-RAID is running concurrently with normal cleaning operations. As long as the domain where the peer block is located is different from that of old/log data block, reading from the peer block could be executed along with normal reading operation on the data block. When the data is merged into a new data block, the parity page is written into a parity block simultaneously provided that the parity block is located at a domain distinct from the new data block.
The flowchart in Figure 5.4 further illustrates how i-RAID is integrated with FTL, and cooperating with garbage collection. From the chart, it is obvious to see that in an extreme case that an aged block cannot find any peer blocks to form RAID, i-RAID can even offer a RAID-1-like scheme to protect that block. This although very rarely happens in that an aged block can find peer blocks most of the time during GC which is likely to clean multiple blocks on different chips in parallel, i-RAID is so elastic in terms of RAID organization to provide such a RAID-1 option. This once again demonstrates that i-RAID
is a very versatile scheme in selecting blocks to form RAID organizations. Moreover from the chart, it is easy to see when shadow stripes will kick in for continuous protection if any valid blocks could not join a new RAID stripe immediately after a block in that stripe is cleaned.

5.3.3 Data Structure

When a parity block is constructed, its protected data blocks’ physical addresses are written into the spare area of the flash memory block with other metadata such as erase cycle counter, associated logical block number, ECC and etc. i-RAID maintains a table in RAM. Each entry in the table records the physical addresses of data blocks and their associated parity block in a stripe. The erase counter in the metadata section of a flash block is used to identify aging status. Additionally, assistant data structures are allocated in RAM for i-RAID housekeeping. Once a flash block is erased and its erase count is above the threshold, it will be linked to a list which contains all the free aged blocks. Once an aged block is used to host data pages, it will be removed from the list and form an i-RAID stripe.
To maintain stripes which are constructed dynamically, we use a data structure which is similar to FTL mapping tables. Due to the fact that i-RAID may only cover a subset of flash blocks which are aged, the i-RAID mapping table is relatively compact, since it does not necessarily contain all the blocks as most FTL mapping does. Each entry in the i-RAID mapping table records both data blocks and parity block. An assistant hash map is also built on this table, so that searching for an i-RAID mapping for a given data block could be more efficient. The i-RAID mapping table and its hash map are illustrated in Figure 5.5. If any data block which is protected by i-RAID needs to be recovered in case of data corruption, the i-RAID mapping table will be used to retrieve peer and parity blocks. When one of the data blocks is invalidated by the FTL, the entry will be invalidated except in the case of a shadow stripe which is introduced in 5.3.2. The invalidated block will be erased and linked back to the i-RAID free list, or its erase will be deferred and the block is marked as shadow in the case of a shadow stripe. Following that, the remaining valid data blocks in the former stripe will be linked to an i-RAID candidate list until they are used to form new stripes.
Similar to FTL, the in-memory data structures can be re-constructed after a power failure by looking up the parity blocks’ metadata areas where the stripe organization information is recorded. The data structures can also be check-pointed to a reserved area on the SSD to speed up recovery during normal shutdown/startup.

5.4 Experiment and Results

5.4.1 Simulator — iRaidSim

Table 5.1: iRaidSim simulator parameters

<table>
<thead>
<tr>
<th>Flash Memory parameters</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Size</td>
<td>2 KB</td>
<td>Read Latency</td>
</tr>
<tr>
<td>Block Size</td>
<td>128 KB</td>
<td>Write Latency</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Erase Latency</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PCM parameters</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>80 ns</td>
<td>Write</td>
</tr>
</tbody>
</table>

The proposed i-RAID was evaluated with an event-driven SSD simulator named as iRaidSim. In iRaidSim, the pages in a log block which is hosted on NVM can associate with any flash memory data block pages. In terms of garbage collection, if NVM is full, an LRU algorithm will recycle the least recently updated data blocks. The parameters of iRaidSim simulator are shown as in Table 5.1. The NVM size is 3% of total capacity. Internal parallel structure was implemented to service disk requests in parallel when they can be mapped to different flash packages. Each flash package is emulated with a closed
queue. The queue which simulates the logic of SSD controller will serialize the sub-requests of flash packages’ queues. As for i-RAID striping, we assumed each stripe consists of two data blocks and one parity block. Other stripe sizes will be evaluated in future works.

In this simulation study, i-RAID is compared with the baseline system which has the same parameters and configuration of i-RAID, however it does not have any redundancy mechanism. As a result, it should have the better performance but cannot tolerate any fault.

5.4.2 Workloads

iRaidSim was tested with several write-dominated benchmarks including Gcc, Installer, Firefox, Financial, Eclipse and Untar. Disk traffic was recorded at the block device layer of the storage hierarchy using Blktrace [98] and Blkparse. A post-processing utility was developed to further process the trace file outputted from Blkparse to screen disk I/O operations. Workloads were scaled up so as to effectively evaluate block aging. The i-RAID threshold was also varied to show its impact on block protection and performance overhead.

The Gcc trace was collected by running Gcc (ver. 4.3.3) to compile the Linux kernel (ver. 2.6.35) source code. Untar was collected by decompressing the tarball of the Linux source tree. Installer was recorded by installing open source LibreOffice suite [99]. Firefox was recorded by browsing common websites such as CNN news, Youtube, Twitter and etc. The disk traffic was recorded at the partition where the Firefox cache and temporary folders reside. The Eclipse benchmark is part of DaCapo benchmark suite [82]
which consists of a set of real-world applications. Eclipse IDE is simulated to initialize, build and deploy a Java project. Financial trace was collected by running OLTP applications at two large financial institutions and made available through [100]. An experimental tool — Open Storage Toolkit [101] from Intel, which was also used by F. Chen et al. in their works [23, 44] to explore internal structure in an SSD, provides a way to validate our simulator.

5.4.3 Results

Compared to non-redundant SSD systems, i-RAID offers parity protection, resulting in some unavoidable extra overhead. As a result, we expect that the latency and erasure count of i-RAID will increase. The main goal of our design is to minimize such extra overheads. i-RAID defers parity maintenance to reduce the number of extra read and write operations. In addition, it exploits the parallelization offered by the multiple flash memory chips and runs concurrently with garbage cleaning operations. An important factor is the value of threshold under which i-RAID is triggered per flash memory block protection. An SSD block can withstand a limited number of erasure cycles before it becomes unusable for writing data. However, as discussed before, the bit error rate may increase significantly as the block becomes aged, long before the block reaches its end of life. In our simulation, we scale up the workload by repeatedly running them until each block wears beyond its maximum rated erasure cycles. However, i-RAID protection will kick in long before this. For example, if the threshold is 10%, then in a static setting, the i-RAID protection will be applied to each block that reaches 10% of its maximum rated erasure cycles, say 1k E/C with rated life time of 10k E/C. With a
smaller threshold, a block will be protected by i-RAID from its earlier life stage, enhancing data reliability but increasing the overhead.

If the threshold is 0, then blocks are protected from the very beginning. While this configuration provides the best protection, we believe that such strong protection is probably overkill for most applications. As discussed before, SSDs seldom suffer from sudden failures. Most devices will not see a noticeable increase of error rate until the erase cycle reaches some threshold. When a block is young, the bit error rate is very low. As a result, the build-in ECC will offer enough protection for most applications. Protections through redundancy are only needed when blocks become aged. Of course, for a very small number of mission-critical applications, full-time protection may still be warranted, so this option is provided by i-RAID.
Figure 5.6: Average response time of i-RAID with varied thresholds compared to the baseline

Figure 5.6 denotes i-RAID’s performance effect in terms of average response time in comparison with the baseline which has no redundancy, under varied i-RAID thresholds. It is clear that time overhead tends to increase with smaller threshold values. Under a moderate and pragmatic value of threshold, such as 90% of rated life erase cycles, the additional latency introduced by i-RAID is surprisingly small. For example, Installer sees 2% at threshold of 90. Financial sees 3% time overhead. With a more conservative threshold, say at 50, Gcc invokes 10% time overhead and Untar sees 11%.
Figure 5.7: Additional Erase Cycles of i-RAID with varied thresholds compared to the baseline

Although latencies can be reduced by parallelizing operations and by the efficient designs of algorithms, the main cost of i-RAID is that additional erases are invoked due to the data redundancy. Figure 5.7 denotes the additional number of erase cycles invoked by i-RAID under different thresholds, in comparison with the baseline which has no redundancy. It is obvious that wearing overhead tends to be larger under a smaller threshold, and vice versa. For example, i-RAID running Financial benchmark introduces 5% additional erase cycles under a threshold value of 90. This ratio goes up to 25% when the i-RAID threshold is 50. The additional erase cycles will be 49% under threshold of 0.
i-RAID running Gcc introduces around 14% additional erase cycles under a threshold value of 90. This ratio goes up to around 30% when the i-RAID threshold is 50. The additional erase cycles will be 48% when the threshold is set to 0. Other benchmarks show similar behaviors. Due to this increasing wearing overhead, i-RAID’s threshold should be selected with care to balance data reliability and additional wearing costs. Actually this is one of the benefits offered by i-RAID — selective i-RAID protection.

Although i-RAID may introduce additional E/C to the device, probably shortening the overall service life compared with non-redundant devices, however it can improve data reliability with this additional data redundancy. Especially for a higher risk block, say for blocks which have been through 90% of rated life span, the block which has no such redundancy will be under higher data lost rate than the block protected by i-RAID. In short, for the aged blocks, by trading a little bit overhead of erasures, data reliability can be enhanced.

5.4.4 Reliability Analysis

In this section, we use a mathematic model to analyze data reliability when i-RAID is present. With more program/erase cycles, most flash memory sees an increase in error rate — raw bit error rate (RBER). With the presence of ECC, uncorrectable page error rate (UPER) can be reduced by several orders of magnitude compared with RBER. During the earlier stage of life, ECC can guarantee the reliability for most applications. However, as an SSD becomes aged, ECC alone may not satisfy the reliability requirement of many applications, since UPER with ECC may start to soar up more sharply than RBER and finally even close to RBER [27]. To this end, i-RAID is a supplementary technique to ECC to enhance data reliability especially for aged blocks.
In this section, the RBER values are cited from [25]. We assume a 4-bit ECC per sector (512 bytes). The correctable sector error rate (CSER) is calculated using the equation (1) [29, 39] where E is the ECC bit and N is the total bits per sector. Note that the bit error rates from various published resources are different, since flash memory of different types or from different vendors may exhibit very different RBER. Also as the density increases, flash memory will see even higher RBER. In our mathematic analysis, we ignore other types of error such as retention errors and program disturb.

\[
CSER(N, E) = \sum_{n=0}^{E} \binom{N}{n} \cdot RBER^n \cdot (1 - RBER)^{N-n}
\]  

(1)

\[
UPER(P) = 1 - CPER = 1 - CSER^P
\]  

(2)

In Figure 5.8, we plot the UPER with 4-bit ECC along with the UPER of i-RAID with 3 different thresholds indicating 3 different aging stages. Also RBER is plotted as reference. From the figure, it is easy to see that during the earlier life of the device, UPER with ECC and UPERs of i-RAID with different thresholds are all at very low level even though RBER starts climbing soon after few thousand P/E cycles. When i-RAID is set to activate at a threshold larger than zero and before that threshold is reached, ECC will be

Using the equation (2), we can calculate the correctable page error rate (CPER) and uncorrectable page error rate (UPER) from CSER where P is the number of sectors per page. In the case of i-RAID, we simply use the equation (3) [102] to calculate the UPER where S is the number of striping elements in each RAID. Here we assume each stripe has 3 blocks.

\[
UPER_{i-RAID}(S) = (1 - \sum_{n=0}^{1} \binom{S}{n} \cdot UPER^n \cdot (1 - UPER)^{S-n})/S
\]  

(3)

In Figure 5.8, we plot the UPER with 4-bit ECC along with the UPER of i-RAID with 3 different thresholds indicating 3 different aging stages. Also RBER is plotted as reference. From the figure, it is easy to see that during the earlier life of the device, UPER with ECC and UPERs of i-RAID with different thresholds are all at very low level even though RBER starts climbing soon after few thousand P/E cycles. When i-RAID is set to activate at a threshold larger than zero and before that threshold is reached, ECC will be
used as the only safeguard to protect data integrity given that no redundancy is applied. Later when flash memory experiences more P/E cycles, UPER with ECC starts to see noticeable and even sharp increase in error rates and finally even approaches and exceeds RBER. However since i-RAID is already activated before this, UPER with i-RAID is still at very low levels as shown in the figure. This again demonstrates that i-RAID with a larger threshold is good enough to ensure data reliability for most applications, which is also accompanied with lower time and erase overheads.

Figure 5.8: Uncorrectable error rate comparison between 4-bit ECC and i-RAID with varied thresholds
We need to emphasize again that the model captures only errors due to aging flash cells. As discussed before, data corruptions of SSDs under power failures might be a much bigger and serious issue in real world applications [90]. It is very hard to establish a model for such type of errors because they depend on implementation details of particular SSD devices. Nevertheless we believe that the extra block-level redundancy provided by i-RAID, especially the non-overwrite feature of i-RAID redundancy, will significantly alleviate the problem. We will continue to study error models of i-RAID under power failures in our future works.

5.5 Summary

SSDs face some unique reliability challenges, including write-endurance, read-disturb and write-disturb. Flash memory cells become unreliable and even unusable after a number of erasure operations. Moreover, their bit error rates increase significantly after as few as several thousand erasures. While a strong ECC scheme can be very effective to protect young flash blocks from random bit errors, it may not be sufficient for aged blocks or data corruptions caused by other reasons such as power faults. For enterprise applications that demand high performance, multiple SSD devices are needed to run in parallel, further increasing the probability of failures.

In spite of the many challenges which arise from the nature electronic properties of flash memory and are hard to overcome by physical device itself, such as reliability issues and difficulties of in-place update, these challenges turn out to be opportunities for us to design a novel and unique solution (named as i-RAID) which utilizes redundancy to address reliability and performance issues. In i-RAID, parity maintenance is completely
deferred to the garbage collection time, because of the out-place update nature. Also in i-RAID, parity protection can be applied ONLY to selective flash memory blocks which have higher erase cycles or error count, well-tailored to the unique reliability behavior of flash memory. In i-RAID, parity writes can hide their latency for most of the time, because i-RAID utilizes internal parallel structures of SSDs and can form a RAID stripe dynamically on blocks from anywhere and of any number. Even more, i-RAID addresses the pitfalls of SSD RAID, such as correlated failures, because it has the flexibility of choosing which blocks to form a RAID on the go, e.g., forming a RAID stripe on the blocks with asymmetric aging states.

Simulation studies show that compared to non-redundant SSDs, under a pragmatic and moderate i-RAID threshold, the time and erasure overheads added by i-RAID are surprisingly small, while the reliability is still significantly enhanced over ECC-only scheme.
Chapter 6

Conclusions and Future Work

6.1 Conclusion

Storage I/O has been acting as system bottleneck for decades, throttling the whole system processing throughput. The introduction of flash memory based semiconductor storage systems and the emerging NVM storage technologies bridge the gap of much powerful, multicore CPUs and the storage systems, and unleash the true application performance. During the transition of adopting new storage architectures, traditional system designs which are tailored for mechanical spinning hard disks necessitate a revisit.

In this dissertation, we first address the SSD based storage performance. We identify that a traditional coarse pattern classification is not enough for SSDs’ asymmetrical performance. We propose the much refined access patterns which are tailored to an SSD based storage system. We further reveal that even a sequential access pattern may not always result in optimal performance. Hence this is also taken into account by our refined access patterns. In our proposed pattern based I/O scheduler and
flash-aware write cache management, we exploit such pattern information, and devise different schemes for different patterns.

SSDs face very serious and unique challenge of reliability, including write-endurance, write-disturb and read-disturb. These reliability issues are becoming more severe when flash memory blocks experience increasing program/erase cycles. By exploiting the internal parallelism exported by multiple flash memory chips inside an SSD, we propose a novel internal RAID architecture — named as i-RAID. Our i-RAID architecture is so different that it is possible to construct RAID only on selective aged error-prone blocks. This also suits to flash memory reliability behaviors. The internal parallelism of multiple flash memory chips enables i-RAID to hide much of the latency invoked by redundancy update. The unique out-place-update nature of flash memory also enables i-RAID to completely defer RAID maintenance to garbage collection time, further improving system performance. Moreover the RAID stripe is dynamically formed during garbage collection time, leading to much more flexibility of RAID. In future studies, i-RAID can be investigated in many aspects.

6.2 Future Work

As flash memory continues to scale down to smaller processing technique, and SSD devices continue to nibble into HDDs’ market share, reliability is becoming much more concern now and in the future. Internal RAID system of an SSD is promising since it exploits the flash memory health information and internal parallelism structure of flash memory chips. Our i-RAID is one such effort which exploits flash memory’s unique
properties and unique reliability model. Because i-RAID constructs RAID on selective aged memory blocks, and also forms RAID stripes dynamically in that it only selects blocks during garbage collection time, further studies could be conducted on i-RAID to explore different RAID schemes. For example, RAID stripes with different number of blocks are possible to be constructed on blocks with distinct aging status. To solve the correlated failure of SSD RAID, i-RAID can be constructed on flash memory blocks with asymmetric aging status, e.g., to construct i-RAID stripes on relative young blocks together with relative aged blocks. Furthermore, the dynamic RAID striping adopted by i-RAID demands extensive studies on the RAID block selection schemes when multiple flash memory blocks are garbage collected at the same time. The interplay between i-RAID and wear leveling is another topic needed to be fully studied.

Table 6.1: Projected 2020 characteristics of PCRAM, adapted from [103]

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity</td>
<td>1 TB</td>
</tr>
<tr>
<td>Read or write access time</td>
<td>100 ns</td>
</tr>
<tr>
<td>Data rate</td>
<td>&gt; 1 GB/s</td>
</tr>
<tr>
<td>Sustained I/O rate</td>
<td>238,000 SIO/s</td>
</tr>
<tr>
<td>Sustained bandwidth</td>
<td>975 MB/s</td>
</tr>
<tr>
<td>Write endurance</td>
<td>10^12 writes</td>
</tr>
</tbody>
</table>

Although the current high cost of NVM technologies (such as PCRAM), the projected cost and characteristics of PCRAM are very encouraging as shown in Table 6.1 [103]. A transition from flash memory based solid state disks to PCRAM based storage systems is foreseeable. During this transition, a hybrid system solution which
incorporates both flash memory and PCRAM is a way to balance cost and performance. In such a hybrid system, since PCRAM has much higher endurance than flash memory, a data reliability protection scheme is necessary for flash memory. Not surprisingly, our i-RAID architecture already introduces one such solution.
Bibliography


[31] STEC Inc, Engineering MLC Flash-Based SSDs to Reduce Total Cost of Ownership in Enterprise SSD Deployments, 2011.


[89] E. Yaakobi, L. Grupp, P. H. Siegel, S. Swanson, and J. K. Wolf, “Characterization and error-correcting codes for TLC flash memories,” in


