University of Cincinnati

Date: 4/27/2012

I, Purushotham Pothu Raju Kolla, hereby submit this original work as part of the requirements for the degree of Master of Science in Computer Engineering.

It is entitled:
Parallel Garbage Collection in Solid State Drives

Student's name: Purushotham Pothu Raju Kolla

This work and its defense approved by:

Committee chair: Yiming Hu, PhD

Committee member: Wen Ben Jone, PhD

Committee member: Carla Purdy, PhD
Parallel garbage collection in Solid State Drives

A thesis submitted to the
Division of Research and Advanced Studies
of the University of Cincinnati
in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in the School of Electronic and Computing Systems
of the College of Engineering and Applied Science

by

Purushotham Pothu Raju Kolla
Bachelor of Technology in Electronics & Communications Engineering
National Institute of Technology, Bhopal, India

Thesis Advisor and Committee Chair: Dr. Yiming Hu
Committee Members: Dr. Carla Purdy and Dr. Wen Ben Jon
Abstract

Flash memories are making their way into both desktop and server environments. Over the years, the major limitation to the wide- adoption of flash memories has been their cost. However, with the advancements in the semiconductor industry, the price per gigabyte (GB) gap between the conventional disk drives and flash memories is getting closer. As such, flash memories can replace disks, where disk utilization is less and extra spindles are added just to increase performance. Though they ventured into the storage architecture as cache and as a hybrid counterpart with Hard Disk Drives (HDD), slowly they are expected to replace the disk drives in servers and super computers [1]. The other major drawback with flash memory is its inability to sustain unlimited erase cycles, which directly limits their lifetime [2]. In order to improve reliability, it is proposed to create redundancy [3].

Creating a Redundant Array of Independent Disks (RAID) is a conventional way of providing redundancy in hard disk drives (HDD) [4]. The same idea is adopted in Solid State Drives (SSD). In addition to the conventional RAID techniques that are implemented at the device level (external RAID), redundancy can be created in an SSD at a much lower level (internal RAID) [3]. The scope of this work is limited to internal RAID.

This work uses i-RAID [3]; an architecture and simulator for internal RAID as background and proposes two improvements. The first contribution is to improve the dynamic stripe formation using access patterns. Another enhancement is to utilize the idle domains when i-RAID is not active by invoking parallel instances of garbage collection.
This thesis describes how these methods can affect the performance of the device and explains how the internal parallelization of an SSD can be better exploited. Both the methods are evaluated individually and the findings are presented. Though both the methods have a great potential to improve the performance of the device, the earlier work (on which the current work is based) is done in such a way that exploiting access patterns during stripe formation could not provide much improvement.

The following work is divided into 6 chapters. Chapter 1 provides an introduction to flash memories and describes the internal architecture of Solid State Drives. Chapter 2 details about the various RAID architectures and ends with a note on the parity update problem. Chapter 3 discusses redundancy in Solid State Drives. Chapter 4 is on using access patterns for stripe formations. Chapter 5 focuses on parallel garbage collection; chapter 6 concludes and suggests future work.
Table of Contents

Abstract .................................................................................................................................................. iii

List of Figures ........................................................................................................................................ viii

List of Tables .......................................................................................................................................... ix

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

1.1. Flash Memories overview ............................................................................................................. 1

1.2. Comparison of SLC/MLC flash ..................................................................................................... 3

1.3. Throughput of a single flash chip .................................................................................................... 4

1.4. Flash Terminology ......................................................................................................................... 5

1.5. SSD Architecture ............................................................................................................................ 7

1.5.1. Native Flash File Systems and Block based File Systems ....................................................... 7

1.5.2. Flash Translation Layer ............................................................................................................ 8

1.5.3. Internal Parallelism in Solid State Drives ............................................................................... 11

Chapter 2. RAID architectures and the parity update problem ............................................................ 14

2.1. RAID Terminology ......................................................................................................................... 14

2.2. RAID5 and the parity update problem .......................................................................................... 17

Chapter 3. Background Work ................................................................................................................ 19

3.1. Device level redundancy in SSDs ................................................................................................. 19


3.3. iRaider – Simulator for i-RAID [3] .............................................................................................. 21

3.4. Block selection process for i-RAID ............................................................................................... 23
Chapter 4. Access Patterns for Block Selection ................................................................. 24

4.1. Flow chart representation ....................................................................................... 25

4.2. Related Work for calculating hotness .................................................................... 26

4.3. Performance Analysis ............................................................................................. 29

4.3.1. Performance of i-RAID ..................................................................................... 30

4.3.2. Observations ...................................................................................................... 31

4.3.3. Performance of i-RAID using Hot Cold associations ........................................ 31

4.3.4. Observations ...................................................................................................... 33

Chapter 5. Parallel garbage collection in Solid State Drives ............................................. 33

5.1. Garbage Collection ................................................................................................. 34

5.2. Parallel garbage collection using LRU .................................................................... 35

5.3. Performance Analysis ............................................................................................. 36

5.3.1. Parallel garbage collection using LRU ............................................................. 36

5.3.2. LRU size Vs. Erase count ................................................................................ 38

5.3.3. Observations ...................................................................................................... 39

Chapter 6. Conclusions and future work ......................................................................... 40

6.1. Conclusions ............................................................................................................ 40

6.2. Future work ............................................................................................................ 40

Bibliography .................................................................................................................. 41
List of Figures

Figure 1: A flash transistor cell.................................................................................................................. 2
Figure 2: SSD architecture [2] ..................................................................................................................... 7
Figure 3: Block Vs. Native File System........................................................................................................ 8
Figure 4: Block Vs. Page FTL....................................................................................................................... 9
Figure 5: One-to-one and many-to-one block associations in log FTL [4] ..................................................... 9
Figure 6: Switch, Partial and Full Merges [5] ............................................................................................ 11
Figure 7: Hierarchical organization of flash.............................................................................................. 12
Figure 8: Different RAID Configurations .................................................................................................. 16
Figure 9: i-RAID process ............................................................................................................................ 22
Figure 10: Relation between Parity Table and Hash Table ......................................................................... 22
Figure 11: Parallel actions in i-RAID ......................................................................................................... 23
Figure 12: Bloom Filters ............................................................................................................................. 27
Figure 13: HotDataTrap............................................................................................................................... 27
Figure 14: Two Level LRU ......................................................................................................................... 28
Figure 15: Parallel garbage collection ...................................................................................................... 34
Figure 16: Parallel invocation of Garbage Collection ................................................................................ 36
Figure 17: Performance of Parallel garbage collection on gcc ................................................................. 37
Figure 18: Performance of Parallel garbage collection on psu ............................................................... 37
Figure 19: Performance of Parallel garbage collection on spc ............................................................... 38
Figure 20: LRU Length Vs. Erase Count ................................................................................................. 38
List of Tables

Table 1: Comparison of SLC/MLC flash cells ................................................................. 3
Table 2: Read, Write & Erase operations on flash ......................................................... 4
Table 3: Properties of different FTL schemes .............................................................. 11
Table 4: Comparison of different hotness calculation methods .................................... 28
Table 5: Trace characteristics ....................................................................................... 29
Table 6: Performance of i-RAID on gcc .................................................................. 30
Table 7: Performance of i-RAID on psu ................................................................. 30
Table 8: Performance of i-RAID on spc ................................................................. 31
Table 9: Performance of i-RAID with access patterns on gcc .................................. 32
Table 10: Performance of i-RAID with access patterns on psu ................................ 32
Table 11: Performance of i-RAID with access patterns on spc ................................ 32
Table 12: Parallel garbage collection on gcc .......................................................... 36
Table 13: Parallel garbage collection on psu .......................................................... 37
Table 14: Parallel garbage collection on spc .......................................................... 38
Table 15: LRU length Vs. Erase Count on gcc ......................................................... 39
Chapter 1. Introduction

Even though Flash memories have better characteristics in terms of power, performance and robustness, the primary reason that their usage is not entertained in the industry is their cost. With improvements in the semiconductor processes and subsequent decrease in cost/GB of Flash memories, now a days they are deployed where high performance can be traded off with a little bit of extra cost [5]. Apart from cost, flash memories suffer from limited number of erase cycles that they can sustain. The fact that they cannot be modified in-place worsens the problem and requires additional methods to manage free space and to distribute the erases uniformly [2].

In addition to these inherent disadvantages, they are used in environments where they have to provide the conventional “Block-Storage” interface to the upper level layers in the storage stack [6]. So the conventional methods used on a disk drive cannot extract the maximum performance out of them. However using some non-conventional techniques these peculiar properties of flash can be exploited to provide better performance. Before going further let us see a little background on the working of flash memories and the terminology used in flash based storage.

1.1. Flash Memories overview

Flash memories are a form of non-volatile memories; meaning that data stored in them remains even after removing the power supply to the device. They come in two flavors – NAND and NOR [7]. All the references to flash in this work should be considered as NAND flash until and unless specified explicitly. The cross section of a transistor used to implement flash memory resembles a typical Metal Oxide Semiconductor (MOS) transistor with an additional gate; called Floating Gate that traps the electrons and makes the device "Non-Volatile" [7]. The field created
by the charge on the Floating Gate acts opposite to the field created by voltage on the Control Gate. Read action is performed by applying a voltage on the Control Gate and observing the current through the channel between the source and the drain. Programming/Writing is achieved by maintaining voltages in such a way that electrons tunnel from channel to the Floating Gate. Erasing is done by applying a high voltage on Control Gate to remove the trapped electrons in the Floating Gate [7].

![Figure 1: A flash transistor cell](image)

By default Flash Memory reads 1 as there is no charge on the Floating Gate. Erasing the device makes it 0. So, flash memories have to be erased before they can be programmed again. Several of these transistors are aligned to create several hierarchies. Grouping a bunch of transistors form a page, several pages combine to form a block, several blocks form a plane, planes form a die, dies form a chip and chips form a package.

Along with the data area, each page in a Flash Memory contains an additional area called Out-Of-Band (OOB) area which can be used to store metadata and Erasure Correction Code (ECC) [8]. Similarly, a page in every block can be allocated for storing metadata.

Because of their internal organization for increasing the storage density flash memory is architected in such a way that it can only be read/written in pages and erased in blocks. With
the difference in the unit of access of read/write and erase, the way that a write request happens in a flash device is entirely different than what happens in a disk based drive. On a write request, the new data is overwritten "in-place" in a conventional drive; whereas it is written to a new page in a block. Not only that, if the request spans less than a page, the remaining data of the page is read, modified and written to a new page in a block elsewhere. So, a write operation is dependent on the position of the previous write operation and is non-uniform; whereas a read operation is independent of the position of data requested. So, the way that the data is updated affects the erases that needs to be performed on the device. The limited number of erase/program operations that can be performed on a flash block usually depends on the technology of the underlying flash cells.

1.2. Comparison of SLC/MLC flash

A single transistor can represent a single bit of data as in Single Level Cell (SLC) devices or can represent more than one bit as in a Multi-Level Cell (MLC). SLC memories are fast, more durable (i.e. the number of erase operations that a single cell can sustain) and costly. MLC are slow, less durable and cheap. The following table enumerates the above mentioned facts in a tabular form.

<table>
<thead>
<tr>
<th>Property</th>
<th>SLC</th>
<th>MLC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Density</td>
<td>Less</td>
<td>More</td>
</tr>
<tr>
<td>Read/Write Speed</td>
<td>More</td>
<td>Less</td>
</tr>
<tr>
<td>Endurance</td>
<td>More</td>
<td>Less</td>
</tr>
<tr>
<td>Cost</td>
<td>More</td>
<td>Less</td>
</tr>
</tbody>
</table>

Table 1: Comparison of SLC/MLC flash cells

Because of their cost advantage; even high available commercial storage vendors use MLC flash which has less Mean time Between Failure (MTBF) than its SLC counterparts. Some additional attention is required in MLC based designs to improve their performance and MTBF.
1.3. Throughput of a single flash chip

Let us now see some read/write characteristics of sample flash chip [8]. A page of data written to a flash memory is transferred over a serial channel to a register in (50uS) and will be written to the chip in another 220uS. Similarly a page of data read from a flash memory will take 25uS to be read and another 50uS to be transferred on to the serial interface. Considering these numbers the read throughput that one can achieve is $\frac{1}{(50+25)} \times 2KB = 27.3$MB/Sec (considering no pipelining). Similarly the write throughput will be 7.5MB/Sec. These numbers represent several orders of magnitude lesser than a conventional Disk Drive. It is the internal organization of a Solid State Drive that makes it to give performance way better than a HDD.

![Diagram of Flash Memory Operations](image)

**Table 2: Read, Write & Erase operations on flash**

Before going to discuss about the internal details of a Solid State Drive (SSD) we need to know some flash related terminology. The following describes a brief introduction to the terminology used in the work.
1.4. Flash Terminology [5]

Logical Block Address (LBA) & Physical Block Address (PBA): LBA is the address requested by the host and PBA is the actual physical address of the location of the data. Similarly Logical Page Number (LPN)/Logical Block Number (LBN) & Physical Page Number (PPN)/ Physical Block Number (PBN) represent the logical and physical page/block numbers.

Hot & Cold blocks: Data blocks which are accessed often are classified as hot and those which are rarely accessed are called cold.

Bad Block Management: Because of high density packaging flash memories often contain manufacturing defects and are shipped with some bad blocks. The controller which manages these chips should be able to recognize these faulty blocks and keep a list of them so that it will not use them during normal operation.

Out-of-place update: Flash memories cannot be modified in-place. Data updates are written to a different area on the flash and is called an Out-of-place update.

Flash Translation Layer (FTL): As the write operations change the placement of the data on the device, the physical address location of a page of data is different than its logical address (address requested for). FTL is a table which maps these logical addresses to physical addresses.

Copy-On-Write (COW): As described a write operation writes data to a new location on the flash. If the updated data is less than a page then the remaining of the page has to be read from the old location, a new page has to be allocated and written with a full page of data. This process of reading old data on a write operation is called COW.
Wear Leveling (WL): As the number of erase operations that can be performed on a block of flash memory is limited (Write Endurance), the number of writes performed on the device has to be equally distributed among all the blocks in the device. This process of distributing write operations uniformly on the device is called wear leveling. Wear Leveling when done only on free blocks is called Dynamic and when done on combination of both active and free blocks is called Static wear leveling.

Garbage Collection/Cleaning (GC): A write request writes a data page to a new block and the old page is marked dirty. At some point of time the available free memory will be exhausted and will be filled with dirty pages. Garbage collection is the process of erasing occupied blocks so that free blocks are made available for future writes. The valid pages of the erased unit are copied to a newly allocated free space elsewhere in the device.

Garbage collection and wear leveling have conflicting requirements. For a better GC performance a block has to be recycled if it contains a lot of hot data. Wear leveling on the other hand dictates to erase a block which is static or cold in nature as cleaning a hot block repeatedly wears it more. There are several other factors affecting the performance of garbage collection and they are discussed later.

Write Amplification: Because of garbage collection, the number of pages written to the flash memory is greater than the number of pages written by the host. This increase in the number of writes is called write amplification. Write amplification depends on a factor called “Utilization” which indicates the percentage of active pages in a block which is going to be cleaned.
1.5. SSD Architecture

A typical solid state drive consists of a host interface, an array of Flash memory chips, one or more data buses (channels), a flash controller, and SRAM. The host interface helps in sending to and receiving data from the host. It is responsible for all the protocol handling and data exchanging at the host-SSD interface. Each SSD contains a bunch of flash memory chips to hold the data. The controller helps connecting these memory elements. Additionally the controller is responsible for managing an FTL, keep a track of bad blocks, handle garbage collection & wear leveling. For the controller in order to perform these activities effectively and efficiently it needs a scratch space RAM to store the metadata. This can be a part of the controller in some small SSD’s and a discrete memory module is some medium to large scale SSDs. This memory can also be used to cache read/write data to improve the performance of the device as a whole.

Figure 2: SSD architecture [5]

1.5.1. Native Flash File Systems and Block based File Systems

A file system (FS) is software which manages data on a storage device. A file system on a conventional disk drive locates, read and write data. Due to the peculiar properties of flash storage, a file system to manage a flash device should also provide address translation, bad block management, garbage collection and wear leveling. These additional functionalities can be implemented using an additional software layer (Block based FS) or can be integrated to a
normal file system (Native flash FS) to create a special file system. This work assumes that the underlying flash storage is managed by a Block based FS and all references to FS should be considered Block based. The following figure distinguishes the differences between the two.

![Diagram showing Block Based FS vs Native flash FS]

**Figure 3: Block Vs. Native File System**

### 1.5.2. Flash Translation Layer

One of the major responsibilities of a flash based FS is to make the Out-Of-Place update nature of flash memories invisible to the host. It maintains a mapping table to translate the logical addresses to physical addresses and the address mapping can be managed at several granularities - Page level mapping schemes maps a given logical page to a physical page, so the new page can be placed anywhere in the new block. On the other hand block level mapping schemes maps a logical block to a physical block. A page inside a Block level FTL is addressed using a block number and an offset. So if a page at a particular offset is overwritten, it has to be written at the same offset in a new block. This can lead to a lot of overhead during garbage collection and extra block erases. We can clearly see that a page level mapping requires more memory than Block level mapping but at an additional performance advantage.
Combining the advantages of both Block and Page level mappings evolved Hybrid level mappings in which "Log blocks" are maintained to contain the overwritten pages in the "Data blocks". When a page in a data block is overwritten it is written to a log block. The association between a data block and a log block can be one-to-one or many-to-one. In case of a one-to-one association there exists a log block for every data block and on the other hand there exists ‘n’ number of data blocks associated with a single log block in a many-to-one association. In this work we have used the one-to-one association of log block for every data block.
As a log block is the only place where an update goes; garbage collection has to be initiated when a log block fills or when there are no more log blocks that can be allocated for overwritten data on a data block. During garbage collection a log block and its associated data block are “merged” together to form a new data block and two free data blocks. There can be three different merge scenarios [10] which are explained below

Switch merge: In this particular merge operation the data block is completely overwritten sequentially into the log block such that the log block can be made the new data block. This has no overhead of copying the old data to the new block. The old data block is erased and hence the process creates a free data block.

Partial merge: Instead if the data block gets partially overwritten in a sequential order, the remaining pages in the data block are copied from the data block to the log block and the log block is made the new data block. The old data block is erased and a free block is created at the end of the process.

Full merge: In this form of merging the data block is overwritten randomly and the log is created in a non-sequential order. In such a situation a page of data from either the data block or the log block is read and written to the new data block, the process is continued till the new block is filled up with up-to-date data. This process creates two free blocks but requires the most number of copy operations compared to all the other merges.
The properties of the above mentioned FTL schemes are tabulated below.

<table>
<thead>
<tr>
<th>Properties</th>
<th>Page Based FTL</th>
<th>Block Based FTL</th>
<th>Hybrid FTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Space</td>
<td>High</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td>Response Time</td>
<td>High</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td>Cleaning Cost</td>
<td>Low</td>
<td>High</td>
<td>Medium</td>
</tr>
<tr>
<td>Wear Leveling</td>
<td>Low</td>
<td>High</td>
<td>Medium</td>
</tr>
</tbody>
</table>

Table 3: Properties of different FTL schemes

1.5.3. Internal Parallelism in Solid State Drives

As we have seen earlier, even though the bandwidth provided by a single flash chip is very less, a salient architectural feature of SSD that can be exploited for improving performance is its “internal parallelism”. As described earlier in this chapter an SSD is organized hierarchically into channels, chips, dies, planes, blocks and pages. Parallelism can be obtained at 4 different levels [1] i.e. planes, dies, chips and channels. In general all the hierarchies above planes operate independently. An incoming request can be serviced at the same time by one or more of the parallelisms.
Channel: Depending on the number of flash chips and the complexity of the design a flash controller is connected to more than one channel. All the channels work independently and provide parallelism if the request spans across multiple channels.

Chip: Each channel is connected to one or more chips which can operate in parallel. Requests can be interleaved on the chips to exploit parallelism.

Die: Typically each flash memory chip consists of two dies and data can be requested from them in parallel.

Plane: Two or more planes constitute a die and flash memories expose parallelism at this level as well.

We call any of these parallelizable flash elements as parallel domains in the rest of this work. Apart from these hierarchical structures some flash vendors provides "copyback" capability which helps to copy pages without crossing the chip boundary, thus saving the time which is required to transfer data onto the serial channel interface. These parallelisms cannot be exploited to 100% as there are certain constraints which restrict these benefits. They are

- All planes on a die have to perform the same action. This means that it is only possible to parallelize requests across planes of a die if and only if all the requests are of the same type.
- Copyback is restricted for copying pages only if the blocks are located on the same plane. This restricts that the "copyback" process to intra-plane.
It becomes very important to carefully utilize this parallel request handling nature of Solid State Drives as they can create bottlenecks if not used properly [1]. This utilization efficiency greatly depends on the placement of the data writes on the flash memory. A write request has to be interleaved properly among the available parallel domains such that both the write and the read latency are optimized. These two actions poses a serious problem to one another as it is required to write the whole data on a single parallel domain to get the benefits of sequential access; however it acts against the interest of read response as striping on more parallel domains makes it much faster to retrieve back (in parallel). There are two well-known methods used to distribute writes across the parallel domains [3]

- **Associative mapping**: In this method the stripe domain to be written is decided from the address of the write request (say address % domain_number), i.e. a given block address is associated with a particular parallel domain. This method has the disadvantage of uneven workload distribution which causes uneven wear.

- **Global mapping**: In this method the writes are distributed on the domains as they arrive. This method has the obvious advantage of better load distribution and better wear leveling.

The advantage of this parallel architecture does not stop here; it helps to create redundancy at a much finer level. This is further discussed in a later section.
Chapter 2. RAID architectures and the parity update problem

The ever increasing speeds of processors crave for more storage bandwidth, but the not so rapid increase in the response times of the disks has made the industry go for a logical organization of disk drives. The arrangement of the drives in the organization depends on the required level of parallelism, redundancy and storage space. RAID the acronym for Redundant Array of Inexpensive Disks is a disk technology that describes several ways of arranging the individual disk drives to achieve both performance and redundancy at the expense of additional inexpensive disk spindles. Before we go into the different available raid levels, let us define some terms which we will be using in this work.

2.1. RAID Terminology

Stripe and Stripe Unit: In order to load balance read/write requests on all the disks, the logical addresses of the data blocks are striped. For example if the striping granularity is a block, then if Block0 is on Disk0 then Block1 is on Disk1, Block2 is on Disk2, Block3 is on Disk3 and so on. Hence, a stripe can be defined as a set of data and parity block/s which occupy the same physical position on all the disks (The parity block/s store the parity data of the data blocks). Stripe Unit is the number of data blocks that constitute a data stripe.

Distribution Scheme: This defines how the data blocks are distributed on the devices. "Asis" indicates that the data blocks are distributed with no re-mapping where as "Striped" indicates that the data blocks are striped on all the devices.

Redundancy Scheme: Indicates the method of redundancy provided for disk failures. "Shadowed" indicates that there exists copies of disks; "Parity" indicates that a separate disk is
maintained for parity data and "Parity Rotated" indicates that the parity data is distributed on all the disks in the array.

There are several primary RAID levels and one can create a number of hybrid models mixing the primary levels. The following gives a brief introduction to several of the available raid levels used in the industry.

RAID0: This level of RAID just stripes the data at block level on all available drives so that the performance is increased without any effect on redundancy.

RAID1: This level of RAID shadows disks so that redundancy is achieved with no increase in performance. Using a combination of the above two models one can create a hybrid array which improves performance and provides redundancy at the same time. RAID(1+0) and RAID(0+1) are two systems with these properties.

Instead of complete mirroring (shadowing) the disks for redundancy, the following RAID methods create redundancy by using parity at different granularities. All these methods can recover from a single disk failure.

RAID2/RAID3: Bit/Byte level striping with data parity maintained on a dedicated drive. Spindle rotation is synchronized.

RAID4: Here the striping granularity is block level and similar to RAID2/RAID3 systems, parity data is maintained on a dedicated disk. Spindle synchronization is not required here i.e. each disk can act independently for data accessing.
RAID5: In RAID5 redundancy is maintained using parity as in RAID2/3/4. Striping is done at block level as in RAID4 but the distinguishing feature of RAID5 from RAID4 is that the parity data is distributed on all the disks rather than maintaining on a dedicated disk.

RAID6: Rather than maintaining a single block of parity this method maintains 2 blocks of parity per stripe. This results in coping with two disk failures simultaneously. There are several other proprietary methods used by different storage vendors to meet individual requirements.

Figure 8: Different RAID Configurations
2.2. RAID5 and the parity update problem

Among the available RAID architectures, this work utilizes RAID5 owing to its cost advantage for creating the redundancy. The conventional RAID schemes when implemented on Flash memories suffer from additional problems because of their peculiar properties. RAID4 architecture can be considered as an example of this; apart from congestion due to a single parity device, the implementation of RAID4 on SSDs suffers from increased wearing due to extensive overwrites on the "Parity Disk". Many methods have been proposed to reduce wearing using HDD as a parity store [12].

In addition to reliability, RAID4/RAID5 systems also increase I/O performance by parallelizing the Read/Write requests. It is to be noticed that the implementation of such a system has performance hit in itself (Work done to calculate and update the parity). As described above RAID5 uses parity encoding at block level for redundancy. The set of data blocks and their parity block is called a parity stripe. Read operations to individual blocks in a stripe are independent and can be performed in parallel. However, a small write operation on a single block effect the entire stripe. For every write, parity has to be calculated & updated and is derived as follows:

\[ P_{\text{new}} = P_{\text{old}} \oplus D_{\text{old}} \oplus D_{\text{new}} \]

Where \( P_{\text{new}} \): Updated parity, \( P_{\text{old}} \): Old parity, \( D_{\text{old}} \): Old data and \( D_{\text{new}} \): New data.

Thus a single block update requires two Reads (old block read\( (D_{\text{old}}) \), old parity read\( (P_{\text{old}}) \)) and two Writes (new block write\( (D_{\text{new}}) \), new parity write\( (P_{\text{new}}) \)). This deteriorates the throughput of the write operations. Depending on the number of blocks overwritten in a stripe there are two methods which can be used to reduce the parity update cost.
Read Modify Write (RMW): In this method new parity is calculated using the new data blocks, old data in the same data blocks and Old parity. This method is used when the number of updated blocks is less than half the number of blocks in the stripe.

\[ P_{\text{new}} = D_{\text{old}} \oplus D_{\text{new}} \oplus P_{\text{old}} \]

Reconstruct (Recon): In this method new parity is calculated using the new data and the remaining block of data in the stripe. This method of parity updating is used when the number of updated blocks in the stripe is more than half the number of blocks in the stripe.

\[ P_{\text{new}} = D_{\text{new}} \oplus D_{\text{rem}} \]

The parity update problem is more pronounced when the writes are small and random. However, if the writes are large covering nearly all blocks in a stripe then the burden of calculating and updating the parity is decreased.

The contents and mutual characteristics of data blocks in a stripe play an important role in deciding the penalty paid for parity update. We tried to exploit the access frequency/recency for creating the block associations and this will be explained further in a later section.
Chapter 3. Background Work

3.1. Device level redundancy in SSDs

There is quite a difference between system level and chip level reliability when it comes to Solid State Drives. Given that most of the failures caused in a conventional drive are due to moving parts [5], it is highly probable that the disk fails as a whole and makes unusable. On the other hand each block in a flash chip can fail individually without affecting the chip as a whole.

The major problem with flash memories is their limited erase/program cycles, and the situation is worsened by the adoption of cheap MLC into the construction of Solid State Drives which can sustain less number of Program/Erase (PE) cycles than their SLC counterpart. Furthermore flash memories experience errors that are caused by activity in nearby cells. These are called read/write/erase disturbed [13] errors depending on the activity on the nearby cell.

Another important phenomenon which affects data integrity is the Erratic Erase Phenomenon [13] which is caused during an erase operation and affects the data stored on a bit. This process is statistically distributed and cannot be removed completely. Apart from these, there are several Flash based failure mechanisms observed by the research community [13]. A study of these methods is out of scope of this work.

Error Correction Codes (ECC) present in the OOB area contains parity information to cope with the above mentioned errors. However, as the PE cycles increase some of the above mentioned error rates increase. This creates a necessity for an increase in the number of ECC bits to successfully guarantee the same level of reliability. But the good news is that they don’t occur very often and are distributed all over the chip.
On a Flash memory it is very rare that all the blocks wear out at the same time. Involving all blocks in a RAID like architecture impacts storage space as well as performance as it is required to manage parity for all the blocks involved. As a convenience to improve the performance and the storage capacity we propose to involve only those blocks whose erase count cross a threshold (A percent of the maximum that they can sustain). This way the parity maintenance can be restricted to only those blocks that need failure assistance.

Considering these reasons it makes more sense to have a finer level of redundancy in an SSD where the chance of a complete system failure is very minimal. Unlike a Hard Disk Drive which needs a full system redundancy it is possible in a Solid State Drive to have redundancy at lower levels. Additionally maintaining redundancy at a lower level will amortize the cost of drive electronics. For instance instead of having 4 disk controllers in a (3+1) RAID5 configuration, it is cheaper to have a single better controller driving all the flash chips in a single SSD with the same redundancy model & same storage capacity. However it is to be noted that the drive electronics form a single point of failure.

The level (chip/block/page) of redundancy can be decided depending on various factors including complexity of the design, performance and redundancy requirements. Wang [3] proposed an architecture for creating a redundancy internal to an SSD and also developed a simulator which is used in this research work. The architecture and the simulator are explained below.

3.2. i-RAID – Redundancy framework [3]

Wang [3] proposed a framework called i-RAID to represent a RAID5 organization inside an SSD. The stripe size is one block (Intuitive because this is the granularity at which erases take place). The distribution scheme is “striped” and the redundancy is “parity rotated” (explained in
As explained earlier all the blocks do not need redundancy and do not participate in i-RAID. i-RAID is activated on a block only when its erase count reaches a threshold. The model heavily utilizes the inner parallelism inherently present inside an SSD to hide the latency of the i-RAID operations. Flash blocks in a stripe are chosen from different flash domains so that read/write/erase operations can be interleaved at the same time. Let us now look at a few details of the simulator.

3.3. iRaider – Simulator for i-RAID [3]

The simulator used is based on Disksim [14] and a simulator developed at Pennsylvania State University [15]. The simulator as it is can only simulate a single plane of flash memory. It has been extended by Wang [3] to support parallel domains and i-RAID. In the present configuration of the simulator, a stripe is formed by two data blocks and a parity block and it is possible to make the stripe larger by having the parity calculated over more than two data blocks. The simulator uses Log-based FTL and a greedy policy for garbage collection.

The modified simulator maintains two lists called a candidate list and a raid list. When a clean block turns old it is added to the raid list, whereas active old blocks which are yet to form a raid stripe are kept in the candidate list. Adding a block to the raid list is guided by a variable called “iraid_threshold” which is an indicator of the number of erases that the block has sustained. The value of this threshold is decided depending on the workload; the max erase count of a block is scaled down depending on the erases generated by the workload. In general decreasing this threshold brings more blocks under i-RAID protection and conversely increasing it reduces the number of blocks being protected.
i-RAID is triggered whenever a block has to undergo a full merge operation. If the new data block formed by a merge operation (explained in chapter 2) is old enough it is paired with an active block from the candidate pool (peer block), and a clean block from the raid list (parity block) (A normal block is selected if we cannot get a suitable clean block from a raid list). The association between data, peer and parity blocks is maintained using an in-memory table called “parity table”. Due to the fact that not all the blocks participate in i-RAID this table is small and can be adjusted dynamically. In order to efficiently search i-RAID mapping for a given data block an assistant hash table is built on this parity table. The following figure illustrates the relationship between the two data structures.

![Diagram](image.png)

**Figure 9: i-RAID process**

![Diagram](image.png)

**Figure 10: Relation between Parity Table and Hash Table**
3.4. Block selection process for i-RAID

The blocks are selected in such a way that all three blocks i.e. data, peer and parity are located on different domains. Stripe formation is cancelled and the data block is added to the candidate list if we fail to allocate the blocks on distinct parallel domains.

Assuming that a stripe can be formed by the above process, data is read from either old data/log block & peer block simultaneously and similarly written to the new block & parity block simultaneously. This utilizes the parallel nature of flash to hide the latency of the peer read and parity write operations. It is to be noted that log and old data can be on the same parallel domain as both of them are not read at the same time. Doing these redundant reads and writes should not affect the requested data reads and write response times. Hence i-RAID is invoked only during full merges where it is required to read all pages in a block from either a data block or a parity block. The following picture shows this parallelization process.

The performance reduction resulted because of this parity creation (which includes peer reading) is very low as the process was able to successfully utilize the parallelism available inside an SSD. The number of available parallel domains plays a significant role in hiding this latency. If the number of domains is more it is more probable that i-RAID operations can find idle parallel domains, as they decrease the number of operations waiting in the queue corresponding to a domain which directly affects the response time of the request.
Chapter 4. Access Patterns for Block Selection

In the last chapter we explained creating redundancy internal to an SSD using i-RAID. The block selection procedure makes sure that the peer and parity blocks are on different domains to parallelize read/write actions. We now introduce how to use access patterns to form the associations between the blocks participating in i-RAID. As described earlier in Section 1.4 a hot block is one which is accessed frequently and a cold block is one which remains static during a given epoch. We will see how we can utilize access patterns during a stripe formation.

Consider a configuration in which we have two data blocks and a parity block. The mutual characteristics of the two data blocks can be

- Hot + Hot
- Cold + Cold
- Hot + Cold

Let us consider a hot block which is in association with a cold block. Whenever the data in the hot block is overwritten the cold block has to be added to the candidate pool and their parity has to be discarded. A new stripe needs to be formed with this cold block, a new candidate block and a free parity block. However if we had used access patterns to create association between two cold blocks we might have saved the extra erase and the extra write. Similarly we can gain performance when we associate two hot blocks to form an i-RAID stripe as it is more likely that both of them gets updated nearly at the same time. A proper association between the two data blocks is necessary to fight against the parity update problem.

The implementation of this association is quite straightforward. During the selection of a peer block we will select the peer block which matches with the hotness of the data block. Along with the constraint that the peer block should be from a different domain we have added an
additional constraint that the absolute difference in the hotness between the two blocks is less than a particular threshold. If both the conditions are not satisfied, the data block is added back to the candidate pool. The flow chart representation of such a scheme is represented below.

4.1. Flow chart representation

```
Start

Datablk hotness threshold

PeerInParallel Domain

(Data_HOT-Peer_HOT) < threshold

Add data blk to candidate list

Full_Merge_Iraid

Stop
```

Several methods have been studied to represent the hotness of a data block. Brief descriptions of some of them are listed below:
4.2. Related Work for calculating hotness

Time & frequency histories: This “ideal method” for calculating hotness maintains the time and frequency access information for each logical block. It incurs a lot of memory and processing power to find if a page is hot as it has to keep track of each and every data block. All the remaining methods try to either reduce the memory requirement or processing penalty.

Process based [16]: This method decides the hotness of a block based on the pid of the process generating the request. This kind of method needs the support of a native flash file system as block based storage devices are not provided with the information about a process.

Request size based: From previous research results [17] it is observed that the distribution of write request size has a bimodal tendency, most of the workload is dominated by either small accesses (4-8 Kbytes) or large accesses (128-256 Kbytes). Out of these small sized data writes are mostly hot. This method estimates the hotness of a block depending on the request size.

Single/Multiple Bloom filters [18]: In order to reduce the memory requirement of the above mentioned ideal method to maintain frequencies of all the data pages this method uses a hash table with counters to represent the hotness of the logical pages. As the length of the hash table is less than the number of pages it can cause hash collisions and can lead to false-positives. The probability of selecting a page false-positively is reduced by using multiple hash functions. Every logical page is hashed using all the functions and a block is hot if and only if all the hash counters mapped by it are greater than a given threshold. An aging mechanism is included by dividing the counter values with a constant after a certain number of requests.
HotDataTrap [19]: HotDataTrap develops on the above process by adding a recency bit and caches information about only a set of data blocks. The memory footprint of the process is reduced by using partial bits instead of saving the complete address and uses two hash functions instead of one to reduce hash table lookup time. The following figure gives an overview of the process. This method exploits both the spatial and temporal localities and assumes that false positives will be less because of the fact that most workloads are very restricted and do not occupy the entire logical address space.

![Figure 12: Bloom Filters](image1)

![Figure 13: HotDataTrap](image2)
Two Level LRU [20]: This scheme uses two lists to maintain hot blocks. Whenever there is a request on a block, it places it in a “candidate list”. Whenever there is a request to the same block the block is promoted to a “hot list”. Both the lists are maintained using LRU. The evicted blocks from the “hot list” are not directly sent to flash but placed in the ‘candidate list’.

![Two Level LRU Diagram](image)

**Figure 14: Two Level LRU**

The comparison of all the above mentioned methods is presented in a tabular form with comparative merit figures:

<table>
<thead>
<tr>
<th>Property</th>
<th>Request size based</th>
<th>Time &amp; Frequency Histories</th>
<th>Bloom Filter</th>
<th>Hot Data Trap</th>
<th>Process Based</th>
<th>Two Level LRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory req</td>
<td>Low</td>
<td>High</td>
<td>Medium</td>
<td>Medium</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Speed</td>
<td>Fast</td>
<td>Fast</td>
<td>Medium</td>
<td>Medium</td>
<td>Fast</td>
<td>Slow</td>
</tr>
<tr>
<td>Complexity</td>
<td>Easy</td>
<td>Easy</td>
<td>Complex</td>
<td>Complex</td>
<td>Easy</td>
<td>Easy</td>
</tr>
<tr>
<td>Accuracy</td>
<td>Inaccurate</td>
<td>Accurate</td>
<td>False Positives</td>
<td>False Positives</td>
<td>Inaccurate</td>
<td>Depends on list</td>
</tr>
</tbody>
</table>

**Table 4: Comparison of different hotness calculation methods**

The bottom line is that list based methods are processor intensive and table based methods are memory intensive. As all i-RAID operations occur at block granularities we just need to maintain the metadata at a block level. As the number of blocks in a flash is way less than the number of pages, we have chosen a direct table based approach without the use of any hash functions to be free from false positives [16].
Each block is associated with an ‘n’ bit hotness counter which indicates its hotness. A request to a Logical Page Number increases the hotness of its associated Logical Block Number by one. As the association will only be affected by writes, we will increment the counter only on a write request. The hotness of a block should be varied depending on the present condition of the workload. A block which was hot a while ago may become cold and hence the approach used to calculate the hotness should represent this change. In order to handle dynamic nature of workloads an “epoch” based hotness calculation is used. The epoch period is based on the number of write requests. After a certain number of write requests the hotness counter of all the blocks will be divided by a “dividing factor” such that they represent the hotness of a block better at that particular instant of time. In order to save on the memory the value of the counter is restricted to ‘n’ bits. The hotness counter of a block is saturated to $2^n$ even though there can be more writes on the block.

4.3. Performance Analysis

In order to evaluate the performance of the above proposed method, the simulator is run with three different traces. The first one is a trace generated while compiling the Linux kernel using gcc (called as gcc), the second one is a trace provided by Pennsylvania State University (called as psu) and the third one is a trace provided by Storage Performance Council [21] (called as spc). The characteristics of the three traces are listed below.

<table>
<thead>
<tr>
<th>Trace</th>
<th>Req. Count</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>%</td>
<td>Avg. Req. Size</td>
</tr>
<tr>
<td>gcc</td>
<td>0.13M</td>
<td>66.8</td>
<td>3.65 KB</td>
</tr>
<tr>
<td>psu</td>
<td>0.32M</td>
<td>9.5</td>
<td>27.55 KB</td>
</tr>
<tr>
<td>financial</td>
<td>0.5M</td>
<td>82.2</td>
<td>0.67 KB</td>
</tr>
</tbody>
</table>

Table 5: Trace characteristics
4.3.1. Performance of i-RAID

The following tables compare the performance loss due to i-RAID with various parameters and input traces. The first row (NO i-RAID) with no redundancy serves as a baseline.

<table>
<thead>
<tr>
<th>gcc</th>
<th>4 Domain</th>
<th>6 Domains</th>
<th>8 Domains</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>iraid_threshold</td>
<td>Avg Response</td>
<td>iraided blocks</td>
</tr>
<tr>
<td>NO i-RAID</td>
<td>7.492453</td>
<td>0</td>
<td>7.197662</td>
</tr>
<tr>
<td>2</td>
<td>7.950261</td>
<td>14193</td>
<td>7.528677</td>
</tr>
<tr>
<td>4</td>
<td>7.891761</td>
<td>12534</td>
<td>7.476474</td>
</tr>
<tr>
<td>6</td>
<td>7.654534</td>
<td>10553</td>
<td>7.282509</td>
</tr>
</tbody>
</table>

Table 6: Performance of i-RAID on gcc

<table>
<thead>
<tr>
<th>psu</th>
<th>4 Domain</th>
<th>6 Domains</th>
<th>8 Domains</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>iraid_threshold</td>
<td>Avg Response</td>
<td>iraided blocks</td>
</tr>
<tr>
<td>NO i-RAID</td>
<td>6.611712</td>
<td>0</td>
<td>6.037386</td>
</tr>
<tr>
<td>30</td>
<td>6.949579</td>
<td>7102</td>
<td>6.362322</td>
</tr>
<tr>
<td>60</td>
<td>6.786397</td>
<td>6413</td>
<td>6.216146</td>
</tr>
<tr>
<td>90</td>
<td>6.618191</td>
<td>5635</td>
<td>6.046496</td>
</tr>
</tbody>
</table>

Table 7: Performance of i-RAID on psu
4.3.2. Observations

From the data presented above we see that the performance loss due to i-RAID is less than 10% and can be concluded that the model effectively utilized the internal parallelism in an SSD.

4.3.3. Performance of i-RAID using Hot Cold associations

The following tables list the average performance improvement due to the introduction of access patterns in i-RAID.

<table>
<thead>
<tr>
<th>spc</th>
<th>4 Domain</th>
<th>6 Domains</th>
<th>8 Domains</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>iraid_threshold</td>
<td>Avg Response</td>
<td>iraided blocks</td>
</tr>
<tr>
<td>NO i-RAID</td>
<td>2.824962</td>
<td>0</td>
<td>2.670771</td>
</tr>
<tr>
<td>30</td>
<td>3.124286</td>
<td>7489</td>
<td>2.909282</td>
</tr>
<tr>
<td>60</td>
<td>3.120188</td>
<td>7467</td>
<td>2.891079</td>
</tr>
<tr>
<td>90</td>
<td>3.112567</td>
<td>7410</td>
<td>2.877373</td>
</tr>
</tbody>
</table>
## Table 9: Performance of i-RAID with access patterns on gcc

<table>
<thead>
<tr>
<th></th>
<th>enable_hotcold</th>
<th>iraid_threshold</th>
<th>4 Domain</th>
<th>6 Domains</th>
<th>8 Domains</th>
<th>% Change</th>
<th>Erase Count</th>
<th>Erase Count</th>
<th>Erase Count</th>
<th>% Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>0</td>
<td>2</td>
<td>7.950261</td>
<td>7.528677</td>
<td>7.318814</td>
<td>0.85</td>
<td>100353</td>
<td>101127</td>
<td>100843</td>
<td>1.36</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>7.916574</td>
<td>7.444237</td>
<td>7.24952</td>
<td>0.58</td>
<td>99090</td>
<td>99389</td>
<td>99718</td>
<td>0.93</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>4</td>
<td>7.891761</td>
<td>7.476474</td>
<td>7.235026</td>
<td></td>
<td>98487</td>
<td>99113</td>
<td>98788</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>4</td>
<td>7.820373</td>
<td>7.45133</td>
<td>7.213924</td>
<td>-0.18</td>
<td>97612</td>
<td>97950</td>
<td>98060</td>
<td>0.34</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>6</td>
<td>7.654534</td>
<td>7.282509</td>
<td>7.1333</td>
<td></td>
<td>95811</td>
<td>95902</td>
<td>96078</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>6</td>
<td>7.682833</td>
<td>7.292094</td>
<td>7.137084</td>
<td></td>
<td>95546</td>
<td>95538</td>
<td>95715</td>
<td></td>
</tr>
</tbody>
</table>

## Table 10: Performance of i-RAID with access patterns on psu

<table>
<thead>
<tr>
<th></th>
<th>enable_hotcold</th>
<th>iraid_threshold</th>
<th>4 Domain</th>
<th>6 Domains</th>
<th>8 Domains</th>
<th>% Change</th>
<th>Erase Count</th>
<th>Erase Count</th>
<th>Erase Count</th>
<th>% Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>psu</td>
<td>0</td>
<td>30</td>
<td>6.949579</td>
<td>6.362322</td>
<td>6.120159</td>
<td>2.19</td>
<td>769913</td>
<td>774632</td>
<td>780006</td>
<td>2.56</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>30</td>
<td>6.764629</td>
<td>6.181726</td>
<td>6.05399</td>
<td>1.36</td>
<td>707602</td>
<td>708392</td>
<td>713129</td>
<td>2.30</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>60</td>
<td>6.786397</td>
<td>6.216146</td>
<td>6.048989</td>
<td></td>
<td>682671</td>
<td>691707</td>
<td>705799</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>60</td>
<td>6.680834</td>
<td>6.106478</td>
<td>6.003813</td>
<td>0.08</td>
<td>628117</td>
<td>625289</td>
<td>628191</td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>90</td>
<td>6.618191</td>
<td>6.046946</td>
<td>5.967291</td>
<td></td>
<td>624297</td>
<td>621624</td>
<td>624485</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>90</td>
<td>6.613302</td>
<td>6.039488</td>
<td>5.964311</td>
<td></td>
<td>624297</td>
<td>621624</td>
<td>624485</td>
<td></td>
</tr>
</tbody>
</table>

## Table 11: Performance of i-RAID with access patterns on spc

<table>
<thead>
<tr>
<th></th>
<th>enable_hotcold</th>
<th>iraid_threshold</th>
<th>4 Domain</th>
<th>6 Domains</th>
<th>8 Domains</th>
<th>% Change</th>
<th>Erase Count</th>
<th>Erase Count</th>
<th>Erase Count</th>
<th>% Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>spc</td>
<td>0</td>
<td>30</td>
<td>3.124286</td>
<td>2.909282</td>
<td>2.759474</td>
<td>0.35</td>
<td>1068086</td>
<td>1070156</td>
<td>1070028</td>
<td>0.29</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>30</td>
<td>3.115683</td>
<td>2.897094</td>
<td>2.749664</td>
<td>0.15</td>
<td>1065065</td>
<td>1067124</td>
<td>1066825</td>
<td>0.25</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>60</td>
<td>3.128188</td>
<td>2.891079</td>
<td>2.756213</td>
<td></td>
<td>1067998</td>
<td>1068445</td>
<td>1069281</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>60</td>
<td>3.122088</td>
<td>2.890322</td>
<td>2.749717</td>
<td>0.46</td>
<td>1065369</td>
<td>1066216</td>
<td>1066132</td>
<td>0.29</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>90</td>
<td>3.142567</td>
<td>2.877373</td>
<td>2.755863</td>
<td></td>
<td>1067763</td>
<td>1065637</td>
<td>1066835</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>90</td>
<td>3.101186</td>
<td>2.885061</td>
<td>2.746347</td>
<td></td>
<td>1062863</td>
<td>1064195</td>
<td>1064033</td>
<td></td>
</tr>
</tbody>
</table>
4.3.4. Observations

From the above results it is clear that the average change in response time and erase count is very less and we concluded that using access patterns in the workload cannot have a major impact on the performance of the device. The reason for this is described next.

If we observe the model closely we can see that there is very little scope to improve performance. In a general RAID architecture performance is expected to improve due to access patterns because of the reduction in the number of parity updates. The current model cannot effectively reduce the number of parity updates. Though it works for cold-cold associations, it cannot work for hot-hot associations as there is no “buffer” in the system. Hence the proposed method to improve performance by reducing parity updates cannot create a significant performance effect.

Chapter 5. Parallel garbage collection in Solid State Drives

The i-RAID mechanism explained in the previous section was able to successfully exploit the parallelism present in a solid state drive. However, most of the domains remain idle when the block being merged is young and doesn’t need to be protected by i-RAID. Only the domains spanned by the log, old data and new data blocks are active during a normal merge operation. In order to utilize these unused domains during this idle time, the simulator is extended to perform merges in parallel.
5.1. Garbage Collection

The out-of-place update nature of flash memories necessitates data relocation. When a write update happens on a page of data, the data page is marked dead and is written elsewhere on the device. As the process goes on, the device reaches a point where it has no more free space to allocate. Garbage Collection is the process of creating free space by erasing occupied blocks after copying valid data from them. The process of garbage collection is critical in deciding the response time and the wearing of the device.
The garbage collection process in our log based FTL involves merging of data and log blocks using one of the three forms of merge operations i.e. switch, partial and full (explained in section 1.5.2). When a write request arrives; the simulator checks if a log block is associated with this data block. It assigns one if the data block does not have a log block. At any point of time the number of log blocks is very less in comparison with the number of data blocks and hence it is not possible to allocate a log block for every data block. Hence if all the log blocks are already assigned or if the log block associated with the data block is full, the simulator calls for a merge operation.

The selection of log blocks during a merge operation is crucial to improve the GC performance and the reliability of the device. If the writes are random and small, then it necessitates more merges than when the writes are large and sequential. Instead of greedily (maximum free pages) picking a log block, if we select a log block which is cold and abandoned we can reduce merges (as no one is actively using this log block). As the number of merges decrease, the wearing of the device decreases.

5.2. Parallel garbage collection using LRU

As a second part of this work we tried to enhance the simulator to improve the performance of the device by invoking multiple instances of GC in parallel. If the write request requires a full merge operation and if the block under consideration is young (i-RAID is not required) the simulator tries to invoke another full merge in parallel. The new data/log pair is chosen in such a way that the new merge operation can be done in parallel with the current merge operation. If this parallel operation succeeds two log blocks & two data blocks are freed and if not only a single log block & a single data block are freed. It is to be noted that we are “immaturely” merging the parallel data/log blocks which can increase the number of erases on the device.
An LRU is used to select log/data block pairs to distinguish hot blocks from cold blocks. As the LRU list is maintained only for log blocks, the list is very small and hence the processor/memory overhead is comparatively negligible.

5.3. Performance Analysis

The following results are obtained with the same traces used in the previous Chapter.

5.3.1. Parallel garbage collection using LRU

<table>
<thead>
<tr>
<th>gcc</th>
<th>8 Domains</th>
<th>6 Domains</th>
<th>4 Domains</th>
<th>Avg Change</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Serial</td>
<td>Parallel</td>
<td>Serial</td>
<td>Parallel</td>
</tr>
<tr>
<td>Response Time*10E4</td>
<td>32607.93</td>
<td>26015.16</td>
<td>33282.66</td>
<td>28422.15</td>
</tr>
<tr>
<td>Erase Count</td>
<td>40142</td>
<td>40409</td>
<td>40141</td>
<td>40271</td>
</tr>
<tr>
<td>Total Full Merges</td>
<td>19729</td>
<td>13145</td>
<td>19729</td>
<td>14527</td>
</tr>
<tr>
<td>Parallelisation Count</td>
<td>0</td>
<td>6781</td>
<td>0</td>
<td>5315</td>
</tr>
</tbody>
</table>

Table 12: Parallel garbage collection on gcc
Figure 20: Performance of Parallel garbage collection on gcc

Table 13: Parallel garbage collection on psu

<table>
<thead>
<tr>
<th>psu</th>
<th>8 Domains</th>
<th>6 Domains</th>
<th>4 Domains</th>
<th>Avg Change</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Serial</td>
<td>Parallel</td>
<td>Serial</td>
<td>Parallel</td>
</tr>
<tr>
<td>Response Time $\times 10^5$</td>
<td>75234.3</td>
<td>62706.4</td>
<td>75926.7</td>
<td>64888.9</td>
</tr>
<tr>
<td>Erase Count</td>
<td>48436</td>
<td>48850</td>
<td>48436</td>
<td>51632</td>
</tr>
<tr>
<td>Total Full Merges</td>
<td>24074</td>
<td>17113</td>
<td>24074</td>
<td>17888</td>
</tr>
<tr>
<td>Parallelisation Count</td>
<td>0</td>
<td>8306</td>
<td>0</td>
<td>7919</td>
</tr>
</tbody>
</table>

Figure 21: Performance of Parallel garbage collection on psu
Table 14: Parallel garbage collection on spc

<table>
<thead>
<tr>
<th>spc</th>
<th>8 Domains</th>
<th>6 Domains</th>
<th>4 Domains</th>
<th>Avg Change</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Serial</td>
<td>Parallel</td>
<td>Serial</td>
<td>Parallel</td>
</tr>
<tr>
<td>Response Time*10E5</td>
<td>90176.6</td>
<td>67611.6</td>
<td>91638.8</td>
<td>73214.9</td>
</tr>
<tr>
<td>Erase Count</td>
<td>237717</td>
<td>238900</td>
<td>237717</td>
<td>238602</td>
</tr>
<tr>
<td>Total Full Merges</td>
<td>117573</td>
<td>73785</td>
<td>117573</td>
<td>82750</td>
</tr>
<tr>
<td>Parallelisation Count</td>
<td>0</td>
<td>44829</td>
<td>0</td>
<td>35613</td>
</tr>
</tbody>
</table>

Figure 22: Performance of Parallel garbage collection on spc

53.2. LRU size Vs. Erase count

From the above results we see that the erase count increases as the number of parallel merges increases. To restrict the number of erases, we have introduced another variable that will limit the length of the LRU checked for finding a parallel partner.

![LRU Diagram](image)

Figure 23: LRU Length Vs. Erase Count

Normally the whole length of the LRU is searched for finding a parallel log/data pair. However with a restriction on the length of the LRU that will be searched, the number of parallel merges will be
decreased with a decrease in the erase count. The simulator is run using a gcc trace with 8 domains and the results are tabularized below.

<table>
<thead>
<tr>
<th>Fraction of LRU</th>
<th>Erase Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>40409</td>
</tr>
<tr>
<td>1/2</td>
<td>40352</td>
</tr>
<tr>
<td>1/4</td>
<td>40274</td>
</tr>
<tr>
<td>1/8</td>
<td>40257</td>
</tr>
<tr>
<td>1/16</td>
<td>40179</td>
</tr>
<tr>
<td>1/32</td>
<td>40172</td>
</tr>
<tr>
<td>1/64</td>
<td>40152</td>
</tr>
</tbody>
</table>

Table 15: LRU length Vs. Erase Count on gcc

5.3.3. Observations

As expected the response time improves with the increase in the number of parallel merges. Similarly the number of parallel merges increase with increase in number of domains and the erase count increases with increase in the number of parallel merges.

Compared to the baseline results (section 4.3.1), which uses maximum free page count to evict log blocks, the garbage collection process using LRU reduces the number of merges, which in turn reduces the erase count on the device.
Chapter 6. Conclusions and future work

6.1. Conclusions

This work explored the utilization of access patterns to improve the stripe creation process in a RAID5 architecture deployed internal to an SSD, however with a very little effect on the performance metrics.

A parallel form of garbage collection is implemented to utilize the unused domains when i-RAID is not activated, and again access patterns in the requested data are used to improve the process. The response time is reduced and the average erase count is reduced (compared to the greedy version of GC) improving the performance and extending the lifetime of the device.

6.2. Future work

Given the increasing usage of Solid State Drives in the industry, future work can be directed towards further understanding the performance variability under varying specifications of the device. In the current work we have used a model in which we haven’t distinguished the difference between the different parallelisms available (i.e. plane, die, chip, and package) on an SSD. A more complete model considering the constraints (Section 1.5.3.) on parallelization has to be implemented.

The need for redundancy of a flash block increases as the Program/Erase (PE) cycles on it increases. The length of a stripe determines the level of redundancy in RAID5 architecture. As the device gets old the stripe size can be reduced to increase the redundancy level, instead of using a constant stripe size. Currently the simulator is created for creating the redundancy with a constant stripe length; it can be extended to dynamically modify the stripe length dynamically depending on the age of the device.
**Bibliography**


   http://www.pdl.cmu.edu/DiskSim/.

15. A simulator for various FTL schemes. PSU Computer Systems Lab. [Online]
   http://csl.cse.psu.edu/?q=node/322.


