Data Reliability in SSD

SANTA CLARA UNIVERSITY

By
Tina Jain
He Shouchun
Ajay Rai
(TEAM3)

Table of Contents

1. Introduction 3
1.1 Objective 3
1.2 What is the problem? 3
1.3 Why this is a project related to this class? 4
1.4 Why other approach is not good? 4
1.5 Why you think your approach is better? 4
1.6 Scope of Investigation: 4
1.7 Statement of the problem 5
2. Theoretical bases and literature review 5
2.1 Definition of the problem 5
2.2 Theoretical background of the problem 5
2.3 Related research to solve the problem 6
2.4 Advantage of those research 6
2.5 Disadvantage of those research 6
2.6 Our solution to solve this problem: 6
2.7 Where our solution is different from other solutions: 7
2.8 Why our solution is better: 7
3. Hypothesis 7
4. Methodology 8
5. Implementation 9
6. Data Analysis and Discussion 12
7. Conclusions and recommendations 13
8. Bibliography 13
9. Appendices 14

1.Introduction

SSD can be thought of as an oversized and more sophisticated version of the humble USB memory stick. Like a memory stick, there are no moving parts to an SSD. Rather, information is stored in microchips. A hard drive uses a mechanical arm with a read/write head to move around and read information from the right location on a storage platter. This difference is what makes SSD so much faster. A typical SSD uses what is called NAND-based flash memory. This is a non-volatile type of memory. We can read and write to an SSD all day long and the data storage integrity will be maintained for well over 200 years. SSD does not have a mechanical arm to read and write data, it instead relies on an embedded processor called a controller to perform a bunch of operations related to reading and writing data. The controller is an embedded processor that executes firmware-level code and a very important factor in determining the speed of the SSD. SSDs are typically less susceptible to physical shock, much quieter, have lower access time, and less latency.

1.1 Objective

The objective is to study the features of SSD, overcome the data loss in SSD and work out a solution to extend its life by some optimization of disk access algorithm on OS level.

1.2 What is the problem?

Today SSD's have become much cheaper and affordable. They replace the traditional hard disk as external memory. Most of the SSD's use NAND- based flash memory which has high storage density, fast access times, low power requirements in operation and excellent shock resistance. Though NAND Flash memory has many benefits but there are few limitations as follows:

- Due to production yield constraints, NAND Flash is shipped with the number of bad blocks that cannot be used.
- Good blocks may eventually wear out due to the limited write endurance of flash cells, even with the best wear leveling strategy .This results in data loss and make it unreliable.
- Frequent reads degrade the quality of data and results in data noise.

Unfortunately, the file systems of the three most frequently used personal operating systems (Microsoft Windows, Linux, and Mac OS), the NTFS, EXT-4 (and EXT-3), HFS+, have not been optimized for SSD in considering the write cycle limitation issue. Related work focusses on FAT file system and on the reliability with sacrifice of performance and

usability of SSD (reduced the disk size available to end users). Our approach focusses on the file system like NTFS, EXT-4 and HFS+ required by personal operating systems.

1.3 Why this is a project related to this class?

The file system is an important part of any operating system. After all, it's where users keep their stuff. SSD's are replacing existing HD because of their high storage density, fast access times and low power requirements in operation. Most of the SSD are NAND based, so data reliability is not achieved due to existence of bad blocks. Our aim is to achieve data reliability in SSD similar to HD.

1.4 Why other approach is not good?

- The other approach focus on the old file system like FAT or FS on embedded systems.
- The other solutions often focus on the reliability with sacrifice of performance and usability (reduced the disk size available to end users).

1.5 Why you think your approach is better?

- We focus our solution on optimizations for SSD on the Linux file Systems. We can also implement our solution on other operating systems like Windows, MAC. However the other solutions focus on the old file system like FAT or FS on embedded systems.
- Our solution targets to minor changes to existing FS and seek for backward compatibility to the existing FS and Operation systems.
- Our solution not only aims to the improvement of reliability, but also the performance and usability of the SSD. The other solutions often focus on the reliability with sacrifice of performance and usability (reduced the disk size available to end users).

1.6 Scope of Investigation:

- Study of SSD features
- Study the desktop OS about disk write access

- Study of desktop file system (EXT3 and EXT4)
- Design an algorithm to find solution for failure and thereby extending the life of SSD.

1.7 Statement of the problem

SSD have many advantages over hard drives but its life span is short which results in data loss. Thus data reliability is a main concern in SSD.

2. Theoretical bases and literature review

2.1 Definition of the problem

Currently, the Solid State Disks (SSD) are not costly as before and widely used on personal computers for their high performance, low weight than magnetic disks. However, there are lots of SSD failure and data loss reports on Internet. When you search SSD failure on Google, there are about 6,070,000 results and most of them are about the failure of SSD.

Almost all the SSDs in the consumer market are made of NAND. Unfortunately, the file systems of the three most frequently used personal operation systems (Microsoft Windows, Linux, and Mac OS), the NTFS, EXT-4 (and EXT-3), HFS+, have not been optimized for SSD in considering the write cycle limitation issue. So there are potential risks of data loss for end users after a long period of running these operation systems on their SSDs.

2.2 Theoretical background of the problem

By checking implementations of the File System of the three most important personal operations, we can find that the most critical parts of the File System are always been frequently written again and again:

On NTFS, the Master File Table (MFT) stores all the file information. The file attributes are stored in this area. So once the file information is changed, the related records in MFT will be changed accordingly. That makes the MFT are almost the most frequently changed area on the NTFS partitions. If the blocks reach the write cycle limit, the file information may be lost.

On EXT3 and EXT4, the journal files are the mostly frequently updated files. The frequently update of the journal files will cause certain blocks changing to bad and make the journals unusable.

2.3 Related research to solve the problem

The papers that we referred to address the problems are as follows:

1. Dataset Management-Aware Software Architecture for Storage Systems Based On SSDs.
2. Data Quality Management for Flash Memory based Read Intensive Embedded and Multimedia Applications.
3. Research on a New Real-time Bad Block Replacement Algorithm.
4. Extending the Lifetime of NAND Flash Memory by Salvaging Bad Blocks
5. A Methodology for Extracting Performance Parameters in Solid State Disks (SSDs)
6. Research on Reliability Improvement of NAND Flash Memory in FAT File System

2.4 Advantage of those research

- Bad Block information is maintained in BBM table so that the information about bad block is not lost while performing erase operation.
- The data loss in Flash memory is prevented by fixing the number of reads allowed after a write.

2.5 Disadvantage of those research

- For FAT, the system need to scan the whole FAT table at the boot time, which make boot very slow
- Compatibility issue: The solution can only be used on certain new products, and cannot be applied on legacy products, systems
- No Market value: FAT is passing away from the market

2.6 Our solution to solve this problem:

To overcome the write cycle limit of SSD, we just need to spread the writing operation to different blocks evenly so that no block will quickly reach the write cycle limits and become bad.

By checking the Linux file system implementation, we can identify that certain areas on the disks and certain files are frequently written again and again. So that we can make some changes in the device-independent software layer and the device drivers layer of the operation system to solve the write cycle limit problem of SSD.

Below are the details of the solution:

- Identify the frequent write operations to certain disk areas and files, and try to dispatch the operators to other disk areas to make almost all blocks have similar chances to be written.

- When the writings to certain blocks reach a certain number, we can mark these areas to “hibernate”. And the next write operation to the same files will be done on other blocks.
- The solution will make minor changes to existing file system which will offer backward compatibility to them. And it will not cause performance issue.

2.7 Where our solution is different from other solutions:

- We focus our solution and optimizations for SSD on the Linux file system. Later we can implement our solution on other operating system (Microsoft Windows, MAC) file systems. However the other solutions focus on the old file system like FAT or FS on embedded systems.
- Our solution targets to minor changes to existing FS and seek for backward compatibility to the existing FS and Operation systems.
- Our solution not only aims to the improvement of reliability, but also the performance and usability of the SSD. The other solutions often focus on the reliability with sacrifice of performance and usability (reduced the disk size available to end users).

2.8 Why our solution is better:

Firstly, our solutions can be widely used on personal computers which have SSD installed. Now the Microsoft Windows, Linux and the Mac OS have dominated more than 99% of the PC operation market, so the solution introduced paper have great practical value to the end users. The solutions found in other paper focused on the FAT file system which had almost been out-of-date or some embedded file systems.
Secondly, our solutions only bring minor changes to the file systems and make them much backward compatible to the existing file systems. The changes brought by other solutions are huge, not backward compatible.
Thirdly, our solutions still can guarantee the system performance, and will not waste the disk spaces.

3. Hypothesis

- The SSD manufacturers had not added any special optimizations for certain operation systems like Microsoft Windows, Linux, and Mac OS in SSD controller. If this hypothesis is not true, our solution may conflict with the solutions by manufactures and cannot work efficiently.

- The SSD produced by all vendors should expose the same interface to the OS, and the interfaces are almost the same as those of magnetic disks. If this hypothesis is not true, we need to study the features of each of the major SSD products in the market and work out case by case solutions for each of them.

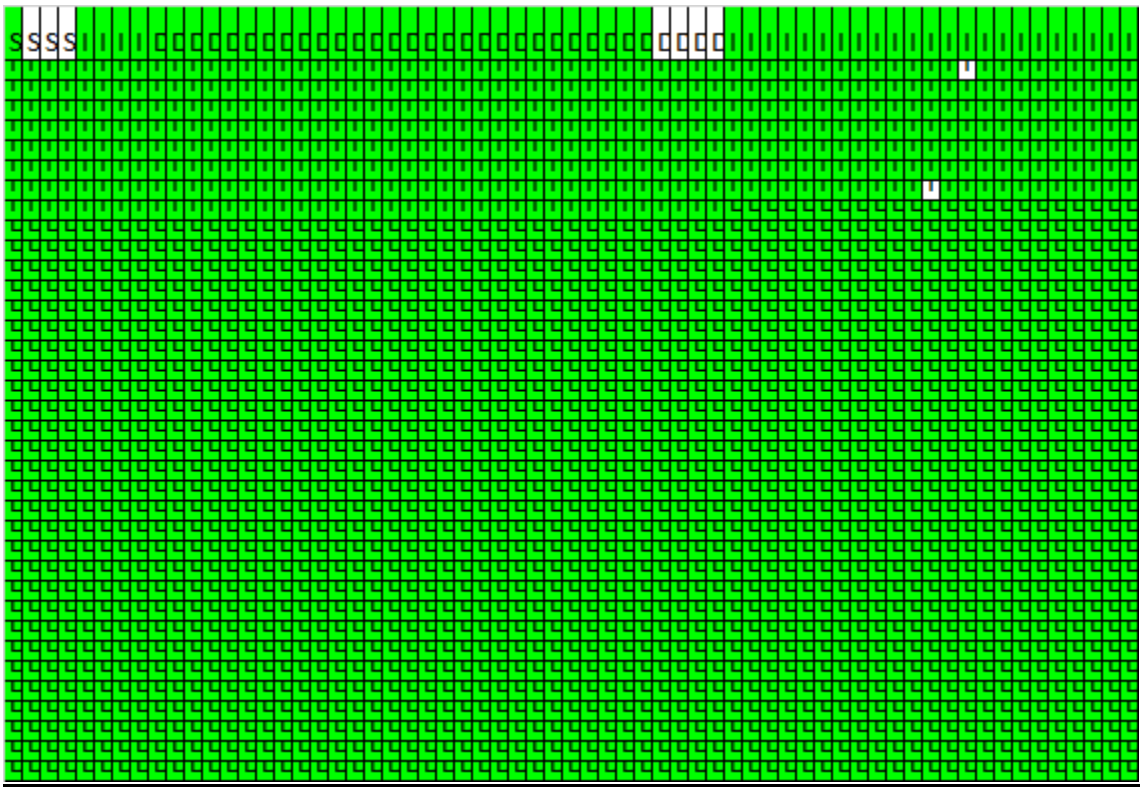
4. Methodology

- Study the Flash memory and SSD features, and the causes of SSD failures and data loss, such as write cycle limit.
- Study the Linux file system (EXT4) in detail and identify the negative effect they bring to the SSD.
- Based on above two points, we found out the solutions for Linux file system about how to prevent the data loss in daily use and extend the life of SSD.
- Design an application which simulates the access of SSD usage on current file system and control of our improved solutions under the same file system writing requests, and then compare their effect to the SSD. The application may provide visual effect to show the difference. Input data: The disk parameters, time of simulation, pre-defined file system operations, etc.

4.1 How to generate input data:

- Our methodology consists of below points :
  1. Firstly we got an estimate of file space usage with the block size by the command: du –a –L –X aaa –B 4096. The –a option displays an entry for each file in a file hierarchy.
  2. tune2fs command (see appendix) adjust tunable file system parameters on ext2/ext3/ext4 file systems. This command gave us information about Inode count, Inode size, block size, Journal inode. This information is required for our simulation.
  3. We have two separate scripts one for system related files and the other for user files. System related files (e.g. /etc, /usr/, /dev) are much more stable than user files (like .c files, .java files). So for system related files we just perform add operation as they are not frequently updated.
  4. For user files the script randomly picks up the file with the block size and performs the operation likes 98% update, 1% ADD, 1% Delete and generate the output with the file name (full path), Block size and operations.
  5. This is the input to our simulation. As we have lot of updates so if the block exceeds the maximum access we select the free block and mark the current block as hibernate.

Simulation output:



Output data: The disk blocks condition graph for comparison (shows above).

- Tools used for implementation and verifying - Eclipse IDE + Java/C programming language.

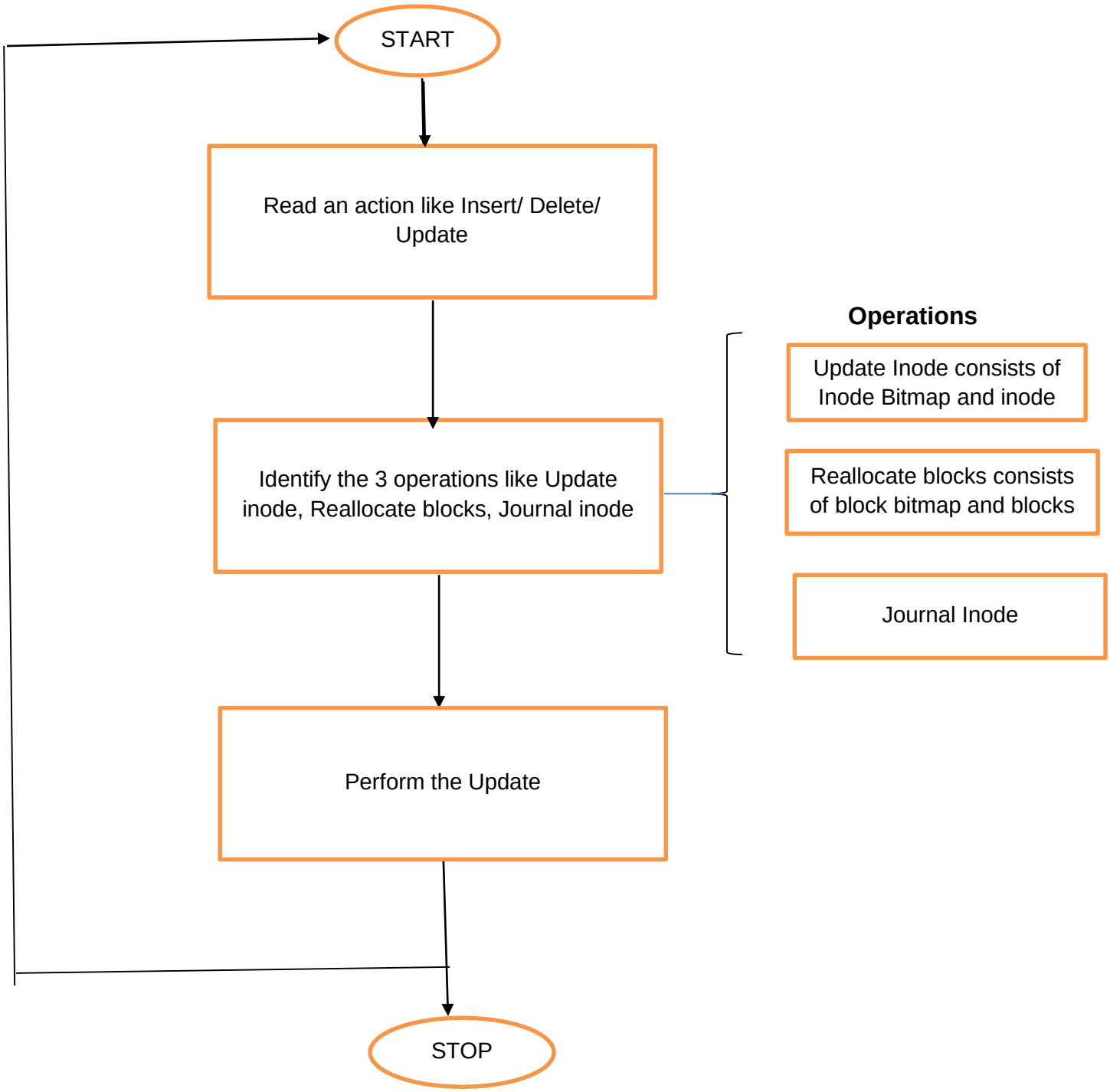
5. Implementation

5.1 Code

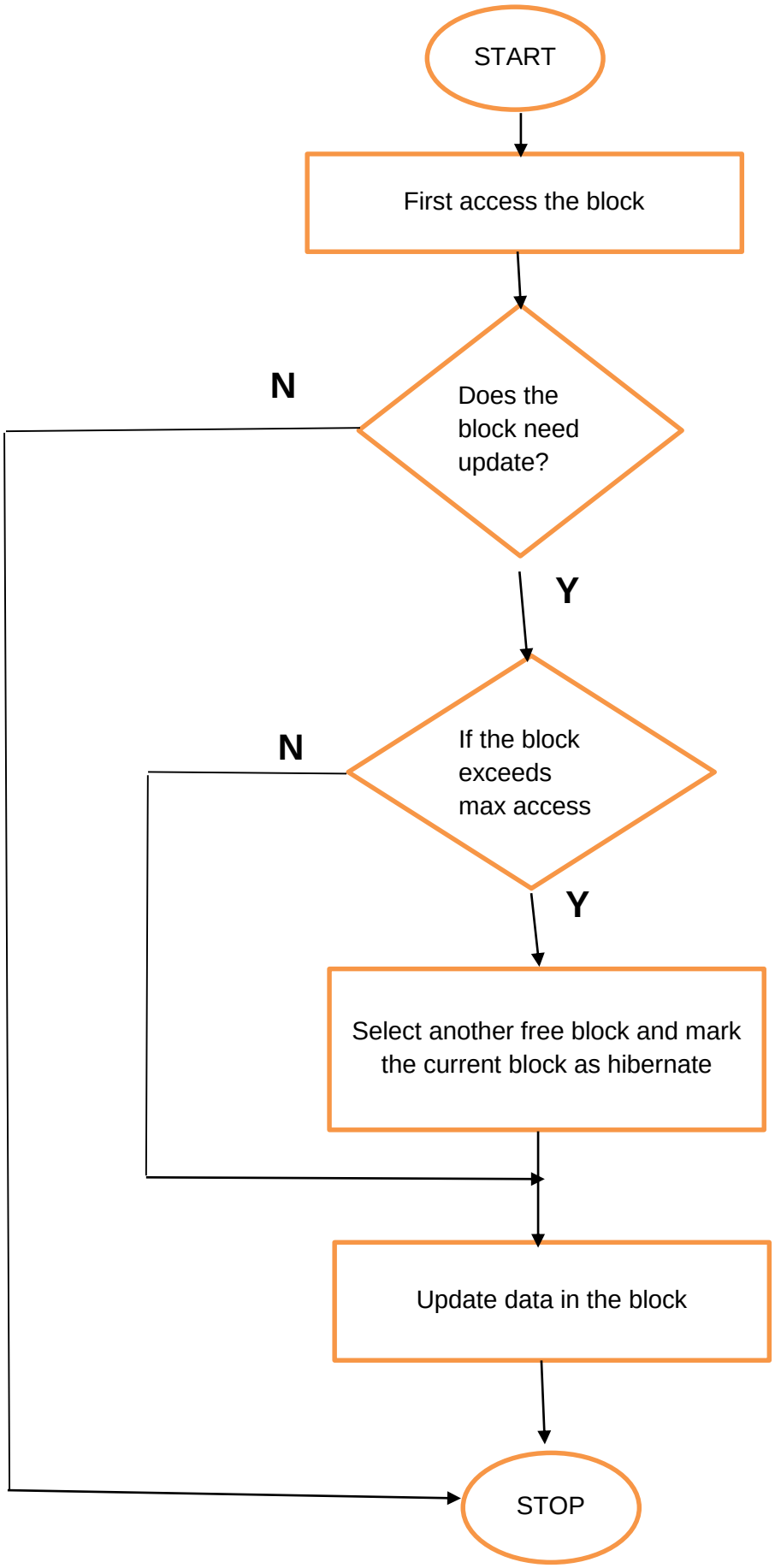
In order to keep this report precise and concise, the full source code for this application is provided in the 'P3' folder with this submission. For individual module implementation please see the Appendices section.

5.2 Flow chart

Overall System flowchart



Detailed Flowchart

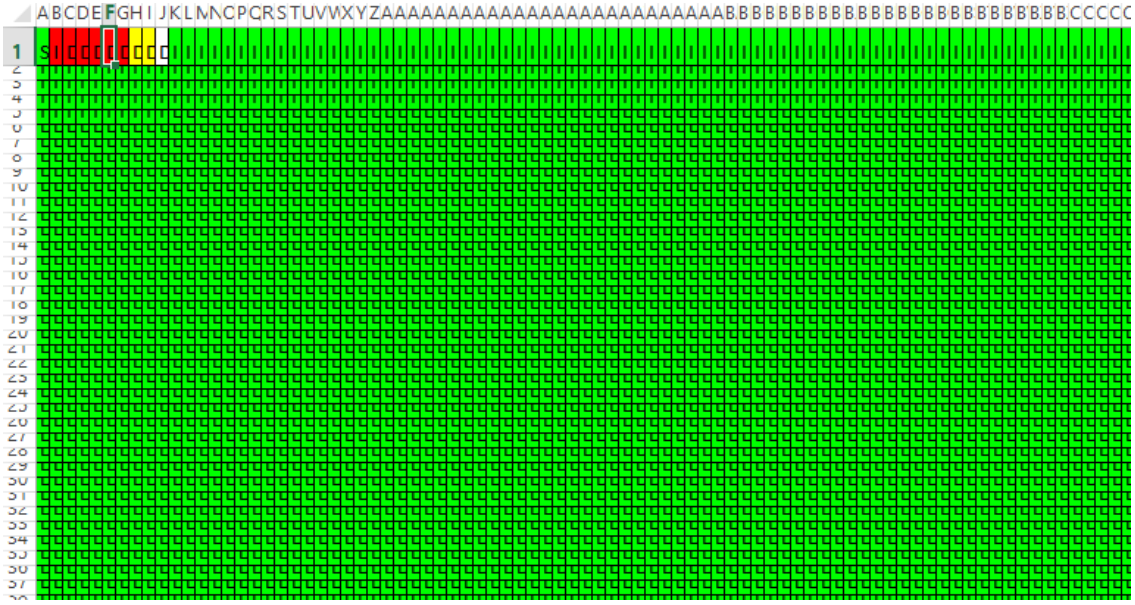


6. Data Analysis and Discussion

6.1 Solution 1: Existing approach:

In the existing solution the pointer in the disk checks which block is free. Once it finds the free block it writes into the block and then move forward to find another free block. The current solution does not keep the write count of the block (does not count how many times the block is accessed).This result in critical state which is shown in below diagram (Red color).

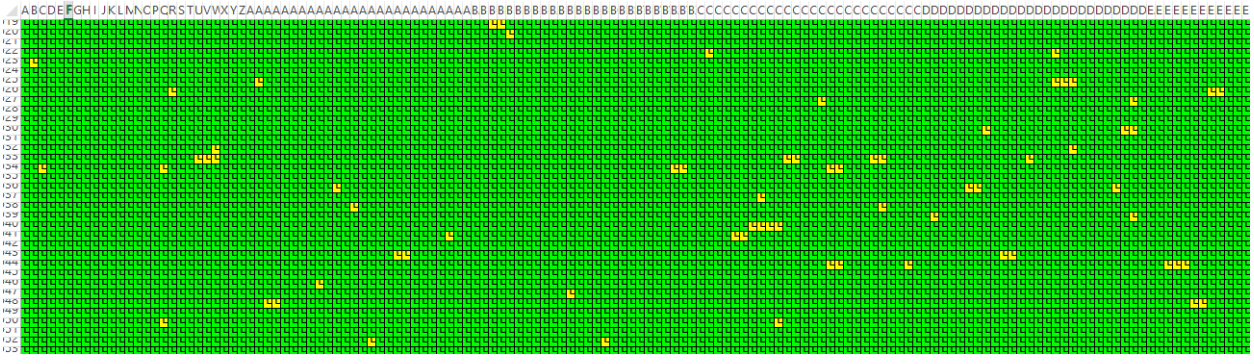
1. Critical State



The below diagram shows the unsafe state of the block as it was accessed multiple times.

2. Unsafe State

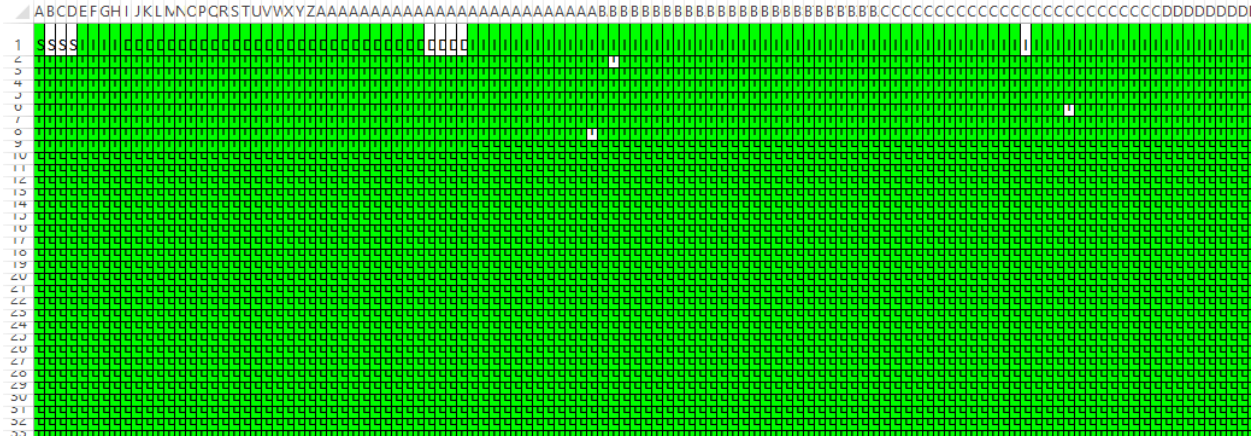
The yellow color depicts the unsafe state of the blocks.



6.2 Solution 2: Our approach/Solution:

In our solution we identified the frequent write operations to certain disk areas and files, and tried to dispatch the operators to other disk areas to make almost all blocks have similar chances to be written. When the writings to certain blocks reached a certain number, we marked these areas to “hibernate”. The next write operation to the same files will be done on other blocks. Our solution made minor changes to existing file system thus offering backward compatibility to them and did not cause any performance issue.

The below diagram shows the simulation result of our solution.



Also we read the kernel code and found out the key functions for allocating the inode, handling the write located in the file (See appendix for functions and file name).

7. Conclusions and recommendations

We found that our approach/solution works very well with no critical and unsafe states in the blocks. Our solution offered better reliability and performance compared to existing solution. This was verified by comparing the simulation results of existing approach and our approach. We read the kernel code and found out the functions which can be modified to achieve data reliability so that we can extend the life of SSD’s.

Currently we focused on one operating system i.e. Linux (EXT4) but going forward we can implement our solution in other operating systems like (MAC OS, Windows NT).

8. Bibliography

[1] SSD introduction on Wikipedia: http://en.wikipedia.org/wiki/Solid-state_drive
[2] Arstechica: http://arstechnica.com/information-technology/2012/06/inside-the-ssd-revolution-how-solid-state-disks-really-work/
[3] Flash memory on Wikipedia: http://en.wikipedia.org/wiki/Flash_memory
[4] Flash memory introduction on HowStuffWorks.com: http://www.howstuffworks.com/flash-memory.htm
[5] Microsoft MSDN: http://technet.microsoft.com/en-us/library/cc758691(v=ws.10).aspx
[6] NTFS.com: http://www.ntfs.com/
[7] EXT-4 wiki (official): https://ext4.wiki.kernel.org/index.php/Main_Page
[8] Apple “legacy technote 1150”: http://developer.apple.com/legacy/library/#technotes/tn/tn1150.ht

9. Appendix

1. The command **tune2fs –l /dev/sda1** gave us the below output
   tune2fs 1.42 (29-Nov-2011)
   Filesystem volume name: <none>
   Last mounted on: /
   Filesystem UUID: 38b5548e-917e-46c3-ba46-4fcf372fc911
   Filesystem magic number: 0xEF53
   Filesystem revision #: 1 (dynamic)
   Filesystem features: has_journal ext_attr resize_inode dir_index filetype needs_recovery extent flex_bg sparse_super large_file huge_file uninit_bg dir_nlink extra_isize
   Filesystem flags: signed_directory_hash
   Default mount options: (none)
   Filesystem state: clean
   Errors behavior: Continue
   Filesystem OS type: Linux
   Inode count: 2444624
   Block count: 9771528
   Reserved block count: 488576
   Free blocks: 7327287
   Free inodes: 2072900
   First block: 0
   Block size: 4096
   Fragment size: 4096
   Reserved GDT blocks: 1021
   Blocks per group: 32768

Fragments per group: 32768
Inodes per group: 8176
Inode blocks per group: 511
Flex block group size: 16
Filesystem created: Mon Feb 4 08:19:26 2013
Last mount time: Tue May 28 10:02:28 2013
Last write time: Fri May 24 13:32:22 2013
Mount count: 3
Maximum mount count: 21
Last checked: Fri May 24 13:32:22 2013
Check interval: 15552000 (6 months)
Next check after: Wed Nov 20 12:32:22 2013
Lifetime writes: 159 GB
Reserved blocks uid: 0 (user root)
Reserved blocks gid: 0 (group root)
First inode: 11
Inode size: 256
Required extra isize: 28
Desired extra isize: 28
Journal inode: 8
First orphan inode: 270038
Default directory hash: half_md4
Directory Hash Seed: 3f849baf-ca75-436c-8290-ffe402858d8e
Journal backup: inode blocks

2. The functions from the Kernel code :

a. Function for allocating the inode located in file "/ext4/ialloc.c" :
   void ext4_free_inode(handle_t *handle, struct inode *inode);
b. Functions for handling the write located in file "/fs/ext4/file.c" :
   static int ext4_file_open(struct inode * inode, struct file * filp);
   static ssize_t ext4_file_write(struct kiocb *iocb, const struct iovec *iov, unsigned long nr_segs, loff_t pos);