DLOOP: A FLASH TRANSLATION LAYER EXPLOITING INTRA-PLANE COPY-BACK OPERATIONS

A Thesis
Presented to the
Faculty of
San Diego State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Computer Science

by
Abdul Rahman Abdurrab
Fall 2011
SAN DIEGO STATE UNIVERSITY

The Undersigned Faculty Committee Approves the

Thesis of Abdul Rahman Abdurrah:

DLOOP: A Flash Translation Layer Exploiting Intra-Plane
Copy-Back Operations

Tao Xie, Chair
Department of Computer Science

Carl Eckberg
Department of Computer Science

Santosh Nagaraj
Department of Electrical and Computer Engineering

28/2011
Approval Date
Copyright © 2011
by
Abdul Rahman Abdurrab
All Rights Reserved
DEDICATION

I dedicate this thesis to my brother, sister and parents Abdurrab Bin Mohammed and Khursheed Unnisa Begum, whose prayers made this possible. I would like to express my deepest appreciation to my elder brother Abdul Aziz Abdurrob and his wife Asma Faheem who helped make my stay miles away from home much easier.

Lastly, I offer my obeisance to Allah S.W.T for guiding me with in.
ABSTRACT OF THE THESIS

DLOOP: A Flash Translation Layer Exploiting Intra-Plane Copy-Back Operations

by

Abdul Rahman Abdurrab
Master of Science in Computer Science
San Diego State University, 2011

Recent technological advances in the development of flash-memory based devices have consolidated their leadership position as the preferred storage media in the embedded systems market and opened new vistas for deployment in enterprise-scale storage systems. With increasing capacity, throughput and durability, NAND flash memory based solid state disk (hereafter, flash SSD) has started replacing hard disk drive (HDD) in laptops and desktop systems. The increasing capacity of NAND presents the issue of address translation. Flash Translation Layer (FTL) is one of the most important components of SSD whose main purpose is to perform logical to physical address translation in a way that is suitable to the unique physical characteristic of the Flash memory. In this research, we propose a new FTL called DLOOP (data log on one plane), which fully exploits fast intra-plane copy-back operations supported by modern flash SSDs. The basic idea of DLOOP is to allocate logs (updates) onto the same plane where their associated original data stay so that valid page copying operations triggered by garbage collection can be carried out by intra-plane copy-back operations without occupying the external I/O bus. To the best of our knowledge, DLOOP is the first page-mapping FTL that achieves high performance through utilizing the internal parallelism provided by modern flash SSDs. Further, we largely extend a well-recognized simulation environment DiskSim3.0/FlashSim to implement DLOOP. Finally, we conduct comprehensive simulations to evaluate DLOOP using realistic enterprise-scale workloads. Experimental results demonstrate that DLOOP consistently outperforms a classical hybrid FTL named FAST and a state-of-the-art FTL called DFTL.

Keywords: Flash memory, SSD, FTL, parallelism, storage system
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>viii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>ix</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2 RELATED WORK AND MOTIVATION</td>
<td>7</td>
</tr>
<tr>
<td>3 A SIMULATOR – DISKSIM AND FLASHSIM</td>
<td>17</td>
</tr>
<tr>
<td>4 DESIGN AND IMPLEMENTATION</td>
<td>23</td>
</tr>
<tr>
<td>5 PERFORMANCE EVALUATION</td>
<td>34</td>
</tr>
<tr>
<td>5.1 Performance Metrics</td>
<td>38</td>
</tr>
<tr>
<td>5.2 Experiment Results</td>
<td>38</td>
</tr>
<tr>
<td>6 CONCLUSION AND FUTURE WORK</td>
<td>64</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>65</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 3.1. DiskSim Code Statistics ................................................................. 22
Table 3.2. FlashSim Code Statistics ................................................................. 22
Table 5.1. Simulation Parameters ................................................................. 35
Table 5.2. Statistics – Real World Traces....................................................... 36
LIST OF FIGURES

PAGE

Figure 1.1. Internals of hard disk drive..............................................................1
Figure 1.2. SSD logic components. .................................................................5
Figure 2.1. Flash translation layer and other components.................................7
Figure 2.2. Internal structure of Samsung SSD. ...............................................14
Figure 3.1. Hardware design for SSD..............................................................18
Figure 3.2. Interleaving for read/write requests...............................................19
Figure 3.3. FlashSim architecture.................................................................20
Figure 4.1. Inter-plane and intra-plane copy operation.................................24
Figure 4.2. Working of the DLOOP algorithm (a). .......................................25
Figure 4.3. Working of the DLOOP algorithm (b). .......................................28
Figure 4.4. Merge using copy-back operation used in DLOOP algorithm.........30
Figure 4.5. The DLOOP algorithm.................................................................32
Figure 5.1. Performance impact of SSD capacity on three schemes. ..............39
Figure 5.2. Performance impact of number of extra blocks on three schemes...43
Figure 5.3. Performance impact of flash page size on three schemes..............47
Figure 5.4. Performance impact of planes per die on three schemes...............50
Figure 5.5. Performance impact of SSD capacity on three schemes for synthetic traces..........................................................53
Figure 5.6. Performance impact of number of extra blocks on three schemes for synthetic traces..........................................................57
Figure 5.7. Performance impact of flash page size on three schemes for synthetic traces..........................................................59
Figure 5.8. Performance impact of planes per die on three schemes for synthetic traces..........................................................62
ACKNOWLEDGEMENTS

It is difficult to overstate my gratitude to my M.S. supervisor, Dr. Tao Xie. Throughout my research period, he provided encouragement, sound advice, good teaching, good company, and lots of good ideas that helped make my research easy and interesting. I am thankful for the countless hours he invested in painstakingly revising drafts for the thesis. I would also like to thank the other members of the committee for their time and effort. I take this opportunity to thank the other team member Abhinav Sharma of our storage systems lab for all his help. I also thank all those who have contributed to the DiskSim simulator and the SSD Extension to DiskSim.
CHAPTER 1

INTRODUCTION

In today’s world which is connected by the Internet, data is produced at the scale of terabytes and petabytes. New services like social networking, new technologies like high definition videos, data availability and security, requirement of inexpensive high capacity and high performance mass media storage is ever demanding. The most popular medium of storing this data is hard disk drive (HDD), a non-volatile secondary storage media.

HDD is a non-volatile, random access device for digital data. The HDD is comprised of rigid rotating platters on a motor-driven spindle with a protective enclosure. A platter has a magnetic surface and can store information on both side. An HDD has a read/write head on each side of the platter to retrieve and store data. Data is magnetically read from or written to the platter by read/write heads that float on a film of air above platters. The heads are attached to actuator motor which positions the read/write head assembly across the spinning disk [1], [2]. Figure 1.1 [3] shows the internals of the HDD.

The disk is organized logically into tracks and each track is further divided into sectors. Each sector holds 512 bytes of data and this is the smallest unit for read and write request from the host. There are different interfaces available to connect HDD to host, e.g. SCSI, SATA, SAS, fiber channel, PCI-E, etc. [3] Accessing a particular sector from HDD depends on the location of the sector and read/write head. When a request arrives to read or write a piece of data, read/write head has to seek for that particular sector. The time for the head to reach the sector is called seek latency. Transfer rate of HDD depends on seek latency, the transfer rate is high if seek latency is low and transfer rate is low if seek latency is high. As seen from Figure 1.1, HDD is made mostly from mechanical parts, actuator motor, actuator arms, spindle motor, etc. Mechanical device has its own limitations. Disk capacity has been increasing at a rate of about 60% a year while disk access latency has been improving only by 10% per year [4]. To decrease the seek time and improve the access data rate, platter size has to be reduced and rotations of platter per seconds has to be increased, which requires more power and dissipates more heat. This makes HDD less energy efficient. Also, HDD makes noise due to its mechanical parts and causes it to vibrate. Shock resistance is also an important factor for HDD, as heads or platter can be damaged and data can be lost due to any unwanted force applied to the device.

Flash memory on the other hand does not have all the above mentioned drawbacks. Flash memory is a type of electronically erasable programmable read-only memory (EEPROM), memory chips that retain information without requiring power [5]. NAND flash memory has been deployed as a non volatile storage media for mobile phones and digital cameras, mainly because of its characteristics such as high performance, low electronic power consumption and physical stability. Moreover, as the density has increased and its price has decreased, flash based storage devices are now considered to have tremendous potential as an alternative storage medium that replaces hard disk. There are two major types of flash memory in the current market. NAND and NOR flash memory [6]. NAND flash memory is mainly designed for data storage because it has higher density and the only allows access in units of sectors. Most SSDs available on the market are based on NAND flash memories. NOR flash memory is for EEPROM replacement. Because NOR flash adopts standard memory interface, the processor could directly execute programs stored on it (such an ability is also called eXecute-In-Place, XIP). NAND flash memory can be classified in
two categories. Single-Level Cell (SLC) and Multi-Level Cell (MLC) NAND [7]. A SLC flash memory cell stores only one bit, while a MLC flash memory cell can store two bits or even more. Since SLC can store one bit, it can be in two states which is programmed or erased. Likewise, the MLC which have 2 bits per cell can have 4 states. However, there is reduced sensitivity between the states of MLC and is likely to be more prone to errors [8]. Compared to MLC, SLC NAND usually has 10 times longer lifetime and lower access latency. However, considering cost and capacity, most low-end and middle-level SSDs tend to use high-density MLC NAND to reduce production cost. SSDs don’t have any mechanical components, e.g. motor driven spinning platters, moving head or motor based actuator assembly. This provides more reliability by not having mechanical components which may fail and also consume less energy than HDD. This also provides faster access than HDD by eliminating seek latency and rotational latency. Its physical properties make it vibration-tolerance and shock-resistance.

Flash memory has several unique characteristics that introduces challenges. Unlike HDD, flash memory does not allow any page to be updated in place just by overwriting it. Therefore, a write operation is available after a time-consuming erase operation is executed [9], [10]. Unfortunately, the basic unit of an erase operation is not a page but a block which is much larger than page. This is because the erase operation is very expensive and are performed at a coarse granularity to normalize this expense. Hence, in order to overwrite any page in a block, other pages in the same block should be erased. Thus, it is necessary to preserve other valid pages in the block being erased, which incurs additional read/write operation. The characteristic of the flash memory is called “erase-before-write” limitation and is the main source of the performance bottlenecks in flash memory [11], [12]. Read/write operations are performed at a finer granularity. Flash memory allows limited number of erase operations before it gets worn out and become unreliable. SLC NAND flash technology allows upto 100K for consumer applications and 5-10K for enterprise applications [13]. To manage these operations an SSD uses a module called as the Flash Translation layer [14], [15], [16], [17], [18], [19].

Flash Translation layer (FTL) is a software middle layer between the standard storage interface protocols such as ScSi and ATA/IDE and Flash packages of SSD. FTL emulates a hard disk and exposes and array of logical sector to the upper-level file or database systems
The FTL takes care of three functions [11]. The first one is logical–to-physical address translation. This layer performs out-of-place updates which in turn helps to hide the erase operation in the flash memory. The mapping table is stored either in the fast on-board SSD RAM or in the SSD itself. The second function of the FTL is to take care of garbage collection. The role of the Garbage Collector is to reclaim invalid pages within blocks by erasing the blocks. For this purpose, the SSD is usually over-provisioned with a certain amount of clean blocks in allocation pools. Each write operation of a page will invalidate the previous occupied physical page and the new data is appended to an available clean block. When running out of clean blocks, the GC scans flash memory and recycles the invalidated pages. This process amplifies the number of read/write operations that the SSD actually receives from host. This means that if heavy random write requests arrive from host, then the FTL has to write pages to new free locations and erase more invalid blocks. This in turn actually increase the number of write requests. The third function of the FTL is to maintain wear leveling. Due to locality on most workloads, writes are often performed over a subset of blocks. Thus some flash memory blocks may be frequently overwritten and tend to wear out earlier than other blocks. FTLs usually employ wear-leveling mechanism to shuffle cold blocks with hot blocks to even out writes over flash memory blocks.

The overview of SSD internals is shown in Figure 1.2. Data goes through different components of SSD before it reaches to flash memory. SSD consists different components, mainly controller, flash chips, DRAM and Host Interface Logical unit. SSD controller provides and interface to the host, and firmware for SSD to execute. It also contains embedded processor, ROM, RAM, error correction code (ECC), wear leveling and other feature like components to act in power failure. Buffer is a high speed RAM memory component bridging the speed difference between high speed host and slow data transfer rate to flash memory, and helps to increase the overall throughput. Flash Memory Components are individual flash memory chips which actually store the data.

Each flash memory chip is divided into dies, each die is divided into planes, planes are further divided into blocks are divided into pages. Size of the page varies by manufacturer and is usually between 2K to 4K of size. Number of pages per block varies between 64 and 128 pages [13]. Reading a page from flash memory cell to the plane register of the corresponding plane takes 25 microseconds while writing a page on to the media takes
200 microseconds from the plane register [20]. This doesn’t give a very good overall performance. However, the second part which involves writing from register to memory can be interleaved.

There has not been a lot of research which study the FTL algorithm with respect to the physical characteristics of the SSD. There has been some work to study the concurrency operations in Flash memory but there is nothing that links it with the FTL algorithms. FTL algorithms are mostly studied to improve features like address lookup schemes, reducing garbage collection and merging over head, etc. We try to find a new avenue to improve FTL algorithms by understanding and using special commands which are used in flash memory. Copy-back is one of such operation. Although it has been mentioned in earlier research [9], nobody has really tired to utilize it fully by using it in conjunction with and FTL algorithm. We propose such a scheme called DLOOP (Data log on one plane) to take advantage of the copy-back scheme which increases interleaving capacity to boost the performance of data transfers in SSD. The basic idea of DLOOP is to have extra blocks distributed on a plane wise basis. The way this is helpful is that, when an update operation occurs, the new data goes in the block which lies in the same plane as the one. This offers three advantages. The first one is that, the data for update operations can still be striped across different planes the
same way the original data was striped. This would not be possible if the new data were written to one section of the SSD were the log blocks are concentrated. Striping the data is a way to interleave requests which gives the effect of parallelism and hides latency. The other major advantage is the extensive use of copy-back operation. Using the copy-back operation to do intra-plane transfers during garbage collection operation makes the external serial bus available for other operations. Also, the data to be copied doesn’t need to be shifted out from the plane register to the buffers present in the controller, they are just copied internally from the plane register to the destination page. The third advantage is that all the requests will be distributed more evenly to the planes of the SSD achieving an implicit wear-leveling at plane level. We show that by making all merge operations intra-plane based DLOOP improves response time significantly and distributes the requests more evenly across the different planes. After a comprehensive set of experiments, we notice that the performance boost of DLOOP against DFTL varies between 5% - 80%, mostly on the upper side for majority of the cases. The improvement over FAST is mostly over 90%. Also, form the figures later we see that DLOOP has much better distribution (lower standard deviation) of incoming requests over all the planes in the SSD. The DLOOP algorithm is evaluated against both real world and synthetic traces under different conditions like varying SSD size, varying extra blocks, etc.

This thesis is organized into following chapters. Chapter 2 discusses related work and motivation. Chapter 3 describes the DiskSim-3.0, a simulator which was used to conduct simulations and gather results. Chapter 4 explains the design and implementation of DLOOP scheme. Chapter 5 shows experimental results. Finally, a conclusion and a brief discussion of future work are discussed in Chapter 6.
CHAPTER 2

RELATED WORK AND MOTIVATION

In this section, we provide an overview of flash memory and FTL and how they are classified into different types. They are classified on how they come up with their address translation schemes. Figure 2.1 shows the general organization of a NAND-type flash memory system.

![Diagram of flash translation layer and other components.]

The host system view flash memory as a hard disk like block device and thus issues reads and writes commands along with the logical sector addresses and data [15]. FTL translates the commands into low-level operations, such as read, write and erase using physical sector addresses. During the address translation FTL looks up the address-mapping table. When issuing overwrites, FTL redirects the physical address to an empty location thus avoiding erase operations and later changes this information in the mapping table. Besides
the address translation from logical sectors to physical sectors, FTL carries out several important functionalities, such as guaranteeing data consistency uniform wear-leveling and garbage collection [21]. For uniform wear leveling, FTL should make every physical block erased as evenly as possible without performance degradation. Coming back to the address mapping issue, the mapping between the logical address and physical address can be managed at the page, block or hybrid level [22], [23]. In page-level mapping, a logical page can be mapped to any physical page in flash memory [17]. In this respect, this mapping approach is very flexible. However, its main disadvantage is that the size of mapping table is too large to be completely present in SRAM. In block-level mapping, a logical page address contains a logical block number and an offset. The mapping table maintains only the mapping information between logical and physical blocks, while the offset of the logical and physical block are kept identical. Even though the size of mapping table is small every time an update requests comes in, the over write is done at the same offset in the new physical block. The other valid pages from the original data block has to be copied to this new physical block. This was serious pitfall and performance bottleneck, mainly because of inflexibility in address mapping. In order to overcome the disadvantages of this several schemes have been proposed. One of such earlier scheme is called BAST [24]. This scheme uses two sets of blocks. The first set is called the data blocks for which block mapping addressing tables are used. The other set of blocks is called the log blocks and page mapping scheme is used for these blocks. The first time a write operation occurs, they go in the data blocks based on the logical address from which two pieces of information are extracted, the data block number and the offset in that data block. Whenever another write request comes in at the same address, a collision occurs since there is already data present in the data block. The way BAST resolved this collision was to allocate a log block for that particular data if none was present. After allocating the log block, data would be written into the first sector of this new block. This entry is stored in the page mapping table and data in the data block is invalidated so that any future reference would be from the log block. If any other update request comes for the same address it would be written into the next available page in the log block and the page in the log block would be invalidated. If another collision happens in the same data block but for a different page it’s going to be written in the next available page in the log block. If a collision happens which falls in another data block, then a new log block is
assigned if available, or one is made available by erasing an existing log block often referred to as the victim block. Before an erase operation can take place all the valid pages from the log block and the data block has to be copied onto a new physical block which will now become the new data block and the other two blocks, the old data block and the log block will be returned to free block pool so they can be reused. One interesting point is that the merge operation can be optimized if the victim log block satisfies a condition. If all the writes in the victim log block are sequential and the total number of written pages is equal to the capacity of the block, we can do what is called a switch merge operation. This is done by exchanging the victim log block with its data block. This is an optimization because it avoids copying of valid pages from the victim block and data block to the new physical block.

Having said this, the BAST scheme suffers two performance problems. The first one is that since each data block has an assigned log block, for every update request that comes in to a new data block, a victim log block has to be selected and merged with the data block. If this happens very often when all the log blocks are not even completely filled, this can result in block thrashing. Let’s assume a series of single page update requests all from different data blocks. For every incoming requests a new log block has to be assigned and once the system runs out of all the log blocks for every incoming request, a victim block with just one valid page has to be selected an merged with the data block which is expensive. Since this happens with every requests it’s called block thrashing. Another performance problem arises from the block-level associativity. Assume that the overwrite operations for one logical block occur successively and after every few requests this log block referred to as hot block has to merged with the data block even though the other log blocks are idle [16]. When intensive overwrites for one hot block occur during a given time window, the log block scheme might result in increased write operations. This is why BAST shows very low spatial utilization. To overcome this shortcomings Lee et al. proposed a new FTL addressing scheme known as FAST [15]. To overcome the problem of space inefficiency they used a fully associative approach in mapping logical sectors to log blocks. In this approach, a logical sector can be placed in any log block. Since there is a high degree of associativity between logical pages and log blocks, this increases the chance to find an empty page in the log block. The other optimization is that, this can avoid merge operations and also delay the merge operations until there is no empty page in the log blocks. Further, they have one log block
reserved for sequential writers while the rest are marked as random blocks. The reason for having the sequential block is to promote switch merge. Since a lot of workloads suggest of having sequential data, it is likely that many switch operations may arise from them. For every request that comes in, it is checked to see if it starts at the block boundary with page number 0. If this is the case, the sequential block is written, else the requests goes to the random block. If the sequential block is not empty it is merged with its corresponding data block. For every sequential requests that comes to the same block here after will be written to the sequential log block and when this blocks becomes full a switch merge occurs. Even though the FAST overcomes the shortcomings of the BAST algorithm it can still suffer from the worst case time problem. Also, it might not capture all switch-merge opportunities. Since FAST is a fully associative scheme, all the pages in the random log blocks can come in from different data blocks. Since there are 64 pages in a block, and assuming worst case scenario, all 64 pages correspond to different data blocks. Let’s say this block was selected by garbage collector as a victim block. All the valid 64 pages have to be merged from this block into corresponding data blocks using full merge. This would lead to 64 full merge operations just to free up one log block, but again this is the worst case analysis. The other issue is that most of the workloads have multiple sequential streams. Since the FAST algorithm uses just one sequential log block it doesn’t effectively recognize multiple sequential streams and looses switch merge opportunities [25]. Since being fully associative looses any locality present in workloads there have been schemes designed to take advantage of locality. One of such algorithm is the SUPERBLOCK algorithm [26]. They define a superblock as a set of adjacent logical blocks that share data blocks along with log blocks. Block mapping is maintained at the superblock level but they allow logical pages with in superblock to be freely located in one of the allocated data blocks or log blocks by maintaining the page-level mapping information within the superblock. During garbage collection, they separate hot page from cold pages and put them into different data block using this hybrid technique. The reason for separating the data as hot blocks and cold blocks is to exploit block-level spatial locality. This type of locality indicates that the pages in the same logical block are likely to be update again in the near future. The log block captures this locality by redirecting update requests of the same logical block into the associate log block. They do this by page-mapping N logical blocks into N+M physical blocks. N is the number of logical blocks composing a
single superblock, which is identical to the number of Data blocks allocated for superblock. Therefore this N is determined by superblock size while M is dynamically changed according to the number of currently available log blocks. If a new log block is allocated the superblock, M increases by one. Meanwhile, M is decreased when garbage collection merge data blocks along with log blocks. To do this the FTL scheme utilizes the page-level mapping inside the superblock, which means the pages belong to N logical block can be situated in N + M physical blocks. The page-mapping table is organized in three levels. The first level is indexed using superblock number which is mapped to a page middle directory that holds four entries of a table whose each entry contains the physical block and physical page number. Even though super-block tried to exploit the temporal locality of changing data it doesn’t really give very good performance. Firstly, the three level mapping tables are complex to understand and hard to maintain. The other drawback is that they separate cold and hot data in different blocks with in a superblock but this does not effectively distinguish hot pages from cold pages. Also, another shortcoming is that page-level mapping information within a superblock should be maintained. Another scheme named LAST (Locality-Aware Sector Translation) which tries to separate the type of requests to achieve lesser full merge operations was proposed [18]. The modeled a “locality detector” which would identify if a request was part of a sequential access or random request and use different buffers to process these requests. Furthermore, they partitioned the random requests into hot and cold pages. To overcome the inability of FAST to serve multiple sequential schemes, they came up with multiple sequential blocks to hold each incoming sequential stream and perform a switch merge when these were full. They locality detector works based on examining the characteristics of the incoming requests. They observe that small writes have little sequential locality but high temporal localities where as large write has relatively high sequential locality. So based on these observations the locality detector examines the size of an incoming request and places it in sequential and random buffers. There are short-coming about such a detector as well. It cannot effectively identify small sized sequentially requests when the small-sized write has sequential locality. Moreover, maintaining sequential log blocks using block-mapping scheme requires streams to be aligned with the starting page offset of the log block in order to perform switch-merge. This can be a problem for workloads with dynamically changing requests streams. All the schemes that we have seen
up until now use a hybrid mapping scheme with a mixture of log blocks and data blocks. This is because like we saw earlier, maintaining a page-level mapping can be expensive in storing all the entries in SRAM. To avoid this cost, hybrid mappings have to employ mechanisms to avoid full-merges, promote switch merges and they do this by separation of hot//cold data and sequential/random requests. Recently Youngaee et al. came up with an algorithm named DFTL (Demand based Page-mapped FTL) to use page mapping tables efficiently to get rid of full-merges completely [14]. However, since it’s not possible to store the complete table in SRAM they design to store only a part of it as cache and the complete set of mapping tables on the SSD as part of data blocks itself. DFTL heavily relies on the presence of temporal locality in workloads to circumvent the costs associated by reading an entry from the SSD if its missing from SRAM. It dynamically loads and unloads the page-level mappings depending on workload patterns. Each page can store upto 512 address mappings. Also, to store the complete page-mapping table on SSD requires a very small percentage of the total space available on SSD. So apart from data pages, we also have translation pages which store all this mapping information. The entire set of translation tables are called as Global Mapping table. Since only a set of those are stored in cache, they are called as Caching Mapping table. To keep track of all the translation blocks in SSD, another table by the name Global Translation Directory is used which is always in SRAM. Each request that comes has an associated logical address to it, which is looked up in the CMT to see if a valid entry exists and if does, the requests is serviced based on the physical address. If the address is not present, it is read in from the translation page in the SSD and put in the CMT. If the SRAM was already full, one of the entry gets evicted using segmented LRU Scheme. If the corresponding entry was dirty, it gets written back to the a new entry of the translation page invalidating the old one and if need be the GTD is updated. The extra read/write account for overhead of this scheme. The worst-case scenario is that there are two translation page reads, one for the victim chosen and the other for the original request. To reduce their over head they use strategies like lazy copying and batch overhead when they have to evict an entry from the CMT or do GC. Lazy updates is done when the GC chooses a data block as a victim. This information is not written back to translation page in the SSD but just to the CMT. Moreover, since multiple valid data pages in the victim may have their virtual-to-physical address translations present in the same translation-page, this can happen
in a single batch update. DFTL algorithms get completely rid of full merge operations and they claim that for the workloads they have used, over 90% requests where handled from cache.

The above FTL schemes were considered with the address translation part. FTL requests are also supposed to maximize throughput of an SSD by interleaving requests across independent blocks on SSD. Before we go any deeper, let’s have a look at the standard Samsung chip. Samsung’s K9ZZG08UXM series NAND-Flash part has good internal architecture and we are conducting our research abased on this architecture [9].

Samsung SSD architecture is organized into multiple identical elements (also known as packages) like shown in Figure 2.2. Each element contains in the SSD has its own data line connecting to the controller but shares the control line. Having independent data line for each element, these elements can be accessed in parallel and more data transfer rate can be achieved. Two dies in each element are serially connected which allows interleaving between read, write and erase operations. Granularity of read/write and erase operations is different while time required to perform each operations is different. Thus, these operations can be interleave between two dies through serial connection and can fully utilize architecture of SSD. Each die in the element is further divided into four planes, each plane is divided into blocks and blocks are further into pages. Page is the smallest unit and is of size 4 KB. Read and write operations at the page size. Since erase operations take a lot longer than read and write operations they are done on a per-block basis to normalize this large time. Blocks are distributed on pair of planes, from blocks 0 to 4095, even blocks reside on Plane 0 while odd blocks reside on Plane 1. The serial interface over which flash packages receive commands and transmit data is a primary bottleneck for SSD performance. The Samsung park takes roughly 100us to transfer a 4 KB page from the on-chip register to an off-chip controller. This includes the 25 us required to move data into the register from the NAN cells. When these operations process serially they can produces 23 MB/sec. If interleaving is employed with in a die, the maximum bandwidth from a single part improves to 40 MB/sec. Write operations on the other hand require the same 100us serial transfer time per page reads, but 200 us programming time. Without interleaving, this gives a maximum of 13 MB./sec. Interleaving the serial transfer and time and program operation doubles the overall bandwidth. Interleaving can provide considerable speedups but they do so with serious
Figure 2.2. Internal structure of Samsung SSD.

constraints. So, for example, operations on the same flash plane cannot be interleaved. This suggest that same-package interleaving is best employed for a choreographed set of related operations, such as a multi-page read or write. The Samsung parts support a fast internal copy-back operation that allows data to be copied to another block on-chip without crossing the serial pins. This optimization comes at a cost: the data can only be copied within the same flash plane. Two such copies may themselves be interleaved on different planes and the result yields similar performance to the full-interleaved inter-package copying without monopolizing serial pins. Our DLOOP algorithm tries to dominate on this feature and tries over-come the limitations discussed above. Dirk et al. All did a comprehensive study of system-level organization choices for solid-state disks [20]. They study the system-level organizations, vary the number of busses speeds, width of busses and degree of concurrent access allowed on each bus and compare the device-level tradeoffs. They say that one way to hide the write latency in flash memory is to interleave sequential writes by dividing flash array into banks where each bank can read/write/erase independently. Their results show significant improvement in both read and write requests when banking is increased from 1 to 2 and 2 to 4 but flatten out later due to bus contention. They also show that write latency can be hidden by ganging blocks across individual flash banks and create superblocks. They also examine the performance improvements in SSDs by having multiple channels to different banks, instead of shared ones. The results show that it’s not always beneficial to do so if the
I/O bus is slow. Also, read requests prefer faster channels while write requests prefer multiple channels. Shin et al. studied different allocation schemes of pages for reconfigurable high performance SSD [25]. They use static and dynamic allocation schemes to compare their results amongst other things. Static allocation can be convenient if allocation can distribute the requests evenly to each plane which will parallelize each request. However, improperly skewed distribution of requests can degrade the performance. They consider a variety of striping mechanism and gauge their performance. Amongst the striping patterns were plane/die/chip wise striping. They try to determine the right granularity of striping to minimize page migrations and increase parallelism. Dynamic allocation can be done to distribute the requests more evenly. One such scheme involves migrating data to an area with more space. Migrating cold data to a flash module with a small number of valid pages will balance the number of hold and cold data. After conducting a bunch of experiments with static allocation in different configurations and with dynamic allocations they figure out the pros and cons of each scheme and how each scheme is suitable for a particular workload and their best conditions to use. They conclude that for sequential IOs a wide striping static – based allocation works well while for random IOs dynamic allocations deal well. To achieve maximum parallelism within SSD, utilizing static allocations with block striping unit is better than employing page striping unit with dynamic allocations.

The industry is still facing performance related issues with maximizing throughput once the SSD is full. The performance deteriorates as the SSD starts getting old. This is because once the SSD is full and all update operations happen to log blocks which cause merge and erase operations soon enough. These operations are overheads and the primary cause of delays. The algorithms mentioned above tackle two different set of problems differently. Algorithms like FAST and DFTL try to reduce merge costs while other algorithms study concurrency and interleaving as system-level issue without considering other aspects of FTL. We believe that the problem of unsustainable performance can be dealt in different ways. One way is to exploit plane-level parallelism. Since the lowest level of parallelism can be achieved at a plane level, we try to distribute incoming requests in different planes. Since copy-back operations can really boost the performance of interleaved requests we try to maximize chances of this happening. We propose an algorithm DLOOP which works by integrating page-level mapping along with optimizations to employ
concurrency and interleaving. This is done by maintaining log blocks at a plane level. The allocation pool in this case will be at the plane level. Algorithms like FAST which are fully associative loose locality while other algorithms like LAST and superblock employ certain log blocks for a given number of data blocks. Our decision to maintain extra blocks at plane level is to utilize the copy-back operations that take place at plane-level. A merge operation can proceed internally without transferring the data out of the plane register to the controller and back. Also, while the copy operation is taking place, the external channel is available for other operations and so are the other planes. Using copy-back operations for merge requests during garbage collection can really help boost the overall performance of the SSD as shall be seen in the coming chapters.
CHAPTER 3

A SIMULATOR – DISKSIM AND FLASHSIM

In order to study the performance of our DLOOP algorithm and compare it with the other FTL algorithms, we use a standard SSD simulator named FlashSim [27] which was built by extending a well-known simulator named DiskSim [28]. DiskSim is well-regarded simulator in the storage community. DiskSim is an event-driven simulator which has been extensively used in different studies and validated with several disks models. FlashSim was built to evaluate the DFTL algorithm and was designed with a modular architecture with the capability to model a ssd based environment. DiskSim supports all aspects of storage subsystem architecture. DiskSim includes different modules to simulate HDD accurately. It is able to simulate components like controllers, caches, flash devices and various interconnects. DiskSim has been validated as a part of the comprehensive system-level model and also as a standalone subsystem [9]. It is been validated against five different disk drives from three different manufacturers. To support the flash drive in the integrated simulator, the basic infrastructure required for implementing the internal operations like page read, page write, block erase were added. The main aspects of the Flash drive is the FTL which is supposed to handle three different responsibilities. The first one is to handle address translation from logical addresses to physical addresses on the flash drive. The second responsibility is to do garbage collection when the number of free blocks drops to a certain threshold and return them back to the free block pool. It is also the responsibility of the FTL algorithm m to do wear-leveling which ensures that all the blocks on the SSD wear out evenly. FlashSim already has the DFTL and the FAST algorithm implemented which will be used in our analysis and performance evaluation with the DLOOP algorithm. The original DiskSim-3.0 module was built to handle requests in a serial order. It doesn’t natively support parallelism over multiple planes/dies. However we modify it to support interleaving capability. The hardware used to simulate the SSD is given as shown in Figure 3.1. As seen from figure, we see that each bus channel connect to several flash dies that are grouped in a package. Each bus channel function independently and in parallel.
When a request arrives at a SSD controller, the controller might interleave this request along with other requests that might have arrived earlier, or will or will arrive at a later time. Figure 3.2 shows the interleaving processing events for one bus channel. The operating timing diagram for read and write requests shows how interleaving can be performed. To read a page, the controller issues a read command to the command register specifying the block and page address. The page data will be accessed in 25 us and will be loaded into the data register. Once the plane register has the data it will be read out of the serial bus. There is an idle period between the time a control request for read is sent and the till the data from the page to the data register is transferred. A read command can be interleaved in this period. Once the plane register has the data it will be read out of the serial bus. Only after the first read request is complete, the data from the second read request can be transferred from page register to the controller. Figure 3.2(b) shows how write interleaving takes place. First, the bus channel must be locked for control channel to inform
block and page will be receiving the data and then the data gets loaded into the plane register. Once this is done, the page program takes place which takes around 200us. During this time the bus channel is free to handle other read or write. To simulate the interleaving process, we had to modify the Disksim source code and add structures to support and report these operations. The code was also modified to support copy-back operations. These are operations which happen in the same plane. During the garbage collection process there are merge operations taking place across blocks, and when the source block and destination block belong to the same plane, a copy-back operation saves time by doing an internal transfer of data. The data is moved from the source page to the plane register and back to destination block without any buffer copy mechanisms. This functionality was added to the flash.c module which has the nand_page_read() and nand_page_write() functionality. When the GC invokes these function calls, the physical addresses of the pages to be copied are examined and based on this info the appropriate delay is added. Apart from interleaving and copy-back, the other modifications include changing the way the physical addressing scheme works for incoming requests based on Logical Block Addressing. Also, more metadata information had to be added to keep reference to a current free block and current free page for each and every plane.

Figure 3.3 shows a few files that should give an overall description of disksim/flashsim. The above is not an exhaustive list of files present in disksim, but just a few that have functions which constitute the main flow of disksm. Only a few functions are
Figure 3.3. FlashSim architecture.
mentioned for the source files mentioned. disksim_main.c is the file that has the main() function which starts up the simulator. The main function calls another function in disksim_main.c which reads the trace file and adds all the requests the queue. The function disksim_simulate_event() pulls out requests from the queue and passes it to io_internal_event(). Depending on the type of request, the request gets passed onto the correct module. This could be the driver, controller, bus and device. When the request is intended for the device it gets passed along to disksim_simpleflash.c This is the interface between disksim and flashsim. The authors of flashsim had to extend the simulator from this part onwards. The device is the endpoint of the whole simulation environment. The disksim_simpledisk.c provides an interface of what functions are to implemented to create a flash based device and the authors implemented it. The disksim_simpleflash.c passes the incoming request to the ssd_interface.c, which is the top level file from an SSD standpoint. It has the collection of all the FTL algorithms. Our modification takes place from this file onwards. The DLOOP algorithm has to be added to this file so that the ssd_interface can do some preprocessing and forward the request to the correct FTL module. The ssd_interface.c has references to all ftl modules and depending on what ftl algorithm is used in the parameter file, the right function gets called. There are separate files for each ftl module with the corresponding algorithms in it. We added our DLOOP algorithm to these ftl modules. The main functions that are to implemented by an FTL module are read, write and erase functions. The next file which is important and required a lot of modifications is the flash.c file. This function has the device access functions. The FTL file converts the logical address into physical addresses on the actual device and passes this information to the flash.c file. The flash.c file only gets the number of blocks to write/read/erase along with the physical addresses. The flash.c file has no information about the FTL and logical block address. It is an interface to the physical device. The important functions here are the nand_page_read and nand_page_write functions. Apart from this, there is one more modification we had to make. We added a priority_linkedlist.c file which implements a priority list to keep requests in order on how they can be processed by free channels. This was used to implement the interleaving feature that we mentioned earlier. The search_and_insert function implements a non-blocking version of insertion. An incoming request is added to the list in the right position depending on other existing requests in the queue. If this request it can be processed
immediately, it will be, else the call just returns back to processing other events/request until the channel and plane are free. The above was a general explanation of the modifications we made and Table 3.1 and Table 3.2 shows statistics for the lines of code we added or modified (LOCs).

**Table 3.1. DiskSim Code Statistics**

<table>
<thead>
<tr>
<th>DiskSim Code</th>
<th>New /Modified LOCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>disksim_global.h</td>
<td>10</td>
</tr>
<tr>
<td>disksim_iotrace.c</td>
<td>7</td>
</tr>
<tr>
<td>disksim.c</td>
<td>19</td>
</tr>
<tr>
<td>disksim_simpleflash.c</td>
<td>22</td>
</tr>
<tr>
<td>disksim_main.c</td>
<td>33</td>
</tr>
</tbody>
</table>

**Table 3.2. FlashSim Code Statistics**

<table>
<thead>
<tr>
<th>FlashSim Code</th>
<th>New /Modified LOCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssd_interface.c</td>
<td>102</td>
</tr>
<tr>
<td>flash.h</td>
<td>7</td>
</tr>
<tr>
<td>flash.c</td>
<td>413</td>
</tr>
<tr>
<td>dloop.c</td>
<td>45</td>
</tr>
<tr>
<td>fast.c</td>
<td>3</td>
</tr>
<tr>
<td>priority_linkedlist.c</td>
<td>155</td>
</tr>
</tbody>
</table>
CHAPTER 4
DESIGN AND IMPLEMENTATION

In this chapter, we present the DLOOP algorithm exploiting plane-level parallelism in SSD to improve its performance for update requests and maintains sequentiality in order of requests. The basic idea of DLOOP is to group data blocks and extra blocks together on the same plane such that when update operations occur they are written to the extra blocks in the same plane. The way this is beneficial is that when the garbage collection process kicks in, the merge operation happens with in the same plane using the copy-back technique and without having to utilize the external serial pins. Figure 4.1 shows how a typical copy operation looks when it happens on like a inter-plane operation and a intra-plane operation. Figure 4.1 shows that if the source and destination are addresses are on different planes how the requests proceeds. This can be divided into four sequences as pointed out in the figure. The first part is to copy data from the page to the plane register. The second part is to copy the data from plane register to the host buffer via the controller. This operation uses the external main channel for the duration of the whole transfer. The third part is to copy this data from the host buffer to the plane register which again uses the channel. The final step is to actually program the nand page using the data from the register. This operation owns the channel for two host operations. One the other hand we see that from second figure, the intra-plane operation happens in two steps. The first step is to copy the data from the source page to the plane register and the second part is to program this data into the destination page. Using the intra-plane operation avoids two host operations where the data has to travel all the way to the host and back, thus saving time and keeping the channel free for other operations. Also, since the requests occur in the same plane, there can multiple such copy-back operations happening in different planes at the same time. The copy-back operations can be interleaved across different planes as shown in the figure. DLOOP banks on this advantage and tries to maximize

With the help of an example let’s see how DLOOP algorithm handles incoming write requests and update operations. Figure 4.2 shows the order of requests that are coming into
the controller. The logical block addresses are 4204, 4205, 4206 and 4207. The FTL is a part of the controller that is supposed to do address translations. The requests that come in the controller have certain pieces of information that the FTL uses to carry out its work of address translation, garbage collection and wear-leveling. This information is the logical address of the request, the type of request which could either be read/write, the number of bytes requested. If there are multiple such requests and their logical addresses are consecutive, they are called sequential requests and if they are not consecutive, they are called random requests. The FTL also has an SRAM which is used to cache a few mappings and used as a look up. This cache is free when the SSD is empty. As requests arrive at the controller, if the logical addresses are not present in the SRAM they are read from the SSD and stored in the Cache Mapping table (CMT) This is how the cache fills up until a point
Figure 4.2. Working of the DLOOP algorithm (a).

where it can’t hold any new mapping information. For this example we assume that the mapping information has been read, either from the CMT or from the SSD itself. Lets says the incoming request was of 14 KB. Incoming requests have been aligned on page boundaries which is of size 4 KB. Even if the requests are shorter, the page has to be read and the extra information has to be discarded. The original requests gets subdivided into four individual sub-requests of write each. Having seen the architecture of the Samsung NAND chip, the DLOOP algorithm stripes the request across multiple planes on the die to alleviate waiting time at the plane. If all the four requests were to be assigned to the same plane, while
the first request was finishing to write the request to the NAND cell (100 us), the other requests couldn’t have proceeded. Striping the request reduces the waiting time. It takes only around 25 us to move the data from the host interface to the plane register. Once this is done the controller can send across another request to a plane which is available to process more requests. This is because the plane is the smallest unit which can handle a request individually. Once this is done the controller can send across another request to a plane which is available to process more requests. This is because the plane is the smallest unit which can handle a request individually. The plane which gets the incoming request is decided by the logical block address:

\[
\text{plane\_no} = (\text{LBA} \% \text{no\_of\_planes\_in\ SSD})
\]

The basic idea behind this formula is to spread the LBA’s as evenly as possible across planes. This way successive LBA requests go into different planes. This works even when the total number of LBA is more or less than the number of physical pages on the SSD depending on whatever configuration is used. Apart from the regular data blocks which add up to the total size of the SSD, there are also the extra blocks which are accessible only by the FTL. Like mentioned earlier, the extra blocks in DLOOP are divided on a plane basis. The number of extra blocks are usually 3% of the total number of data blocks and they are divided equally in all planes. Any remaining blocks which are left over after division are placed in plane 0, plane 1 and so on. So coming back to the figure, the requests gets striped across each individual plane namely plane 0, plane 1, plane 2 and plane 3. The waiting time here was minimum since each plane was available to handle the request. Every plane has a reference to a “current block” which is responsible for serving incoming requests for that plane. It also has a reference the a “current free page” which is used for programming incoming requests. The pages are written sequentially in this current block and once this block is full, a new block is assigned as current block and the first page is used as current free page.

Since every plane has a reference to the current free block and current free page, this extra information has to be stored in the SRAM. DLOOP takes some extra space to store this information and this is updated according to the incoming requests. However, storing this information on the plane level offers two advantages. The first one is that the sequentially of the requests are still maintained even in the extra blocks unlike other operations. The update
operation again happens at full speed since they are interleaved among all the planes. The other advantage is seen when these pages are read in. When a read request comes in for the same logical pages, the information is looked in the CMT and since the earlier update operations happened in the different planes, the read operation is serviced from all the four planes. The read requests happens at maximum throughput as well since these requests are interleaved as well. Assuming the SSD is empty, the incoming requests get assigned to the first block and first page from the current block in all the planes. The current free page reference is updated for all the planes.

In the mapping table stored in the SRAM, there is a valid bit corresponding to each mapping. In this case all the above four mappings have a valid bit set. Now assume that some time later an update operation came in on the same logical addresses. Since the logical address was already served in a previous case, we assume that it’s still in the cache and hasn’t been evicted yet. After reading the values from the mapping table, we invalidate them by unsetting the valid bit. This is shown by a dark black marking on the page in Figure 4.3. The light colored marking show that the page is still valid. When the update operation of LBA 4204 happens, we see that the current block has already an entry of the same logical address on it. The entry in the earlier block was already invalidated earlier. We invalidate the entry in the current block table. This block however will still take other incoming requests. The current free page marks to an free location and shall be programmed with this write which will be set as valid. For the next LBA 4205, we see that the current block has two invalid pages in it. The older mapping of this LBA on the plane will be invalidated and the new data will be written in the current free page. This new mapping information will be added in the SRAM. If the current free page is the last page in the block, a new block will be selected and that block will be referred to as a new block and the first page will be current free page. It is to be noted that every time a new block is allotted, we check the remaining the number of free blocks in the pool and once this falls below a certain threshold, which is two in our case, the GC process is invoked and the block with the maximum invalid pages is reclaimed and added to the free block pool. If the earlier update operations were all sent to a extra block which was on a single plane, the write and read requests couldn’t have been interleaved for future operations even when they were sequential. Hence the DLOOP algorithm which keeps data blocks and extra blocks together at the plane level helps
Figure 4.3. Working of the DLOOP algorithm (b).
sequential workload to sustain their performance. Since the extra blocks are spread out on multiple planes, this has other advantages as well. The waiting time is spread out evenly across all the planes rather than on just the one in which extra blocks are. The drawback with other algorithms is that since log blocks are concentrated in one plane/die, the activity in this region increases and so does the waiting time for these elements. With DLOOP this is evenly distributed across multiple planes. Also, this helps implicitly in wear-leveling at the plane-level since extra blocks are maintained at this level. In other algorithms, only those plane(s) might be used more in which log blocks are present, but in DLOOP since extra blocks are spread at the plane-level, all planes will be used more evenly than in the previous case which helps in taking care of wear-leveling at the plane level.

Moving on to the next part, the DLOOP algorithm does a lot better than the other algorithms when it comes to merge operations. It is in some sense similar to the other algorithms where they try to group data and log blocks. In the superblock technique the reason for grouping the blocks is to exploit the block-level spatial locality which is associated in having data of a single file, index, bitmap. In their case, their theory is that if the data in one of the block is updated the data in the other blocks will follow. This increases the block utilization factor for them. In the case of a reconfigurable FTL they have an N + K mapping scheme. The purpose here is to reduce the number of merge operations and improve efficiency. They have a variable range of values for N and K depending on the workload. DLOOP on other hand groups all data blocks along with extra blocks in the same plane. This is done to make use of the copy-back operation and to merge without using the serial pins. So say for example that a block has four pages and there are two blocks which are full and both of them have two valid pages in them. This is shown is illustrated in Figure 4.4. Let’s say that the lower threshold for free blocks has been reached and garbage collection has been invoked. The victim block has been selected as B1. The regular way that data is copied from a victim block to a new block which is in a different plane happens in phases. First the host instructs that the page of data be written from the NAND memory to the plane register and then it gets transferred all the way up the controller and then from there gets written back into the plane register of the new plane and then the NAND memory of the new block. This uses the external channel for the duration of reading/writing one page for all the pages copied. If the data to be copied is in the same plane, the merge operation happens in the following way.
The valid data is written from block B1 into the copy-back plane register present in the same plane and then it gets written out from there internally to a new block available (B3) in the same plane. With the DLOOP algorithm, we make sure that all the merges that happen take place intra-plane which means that we save the time it takes to copy each individual page from the NAND cell till the host and back to the destination page. A block usually has 64 pages, and say theoretically that such a block with all valid pages was selected for merge operation. With the DLOOP algorithm we save 64 reads and writes from the SSD controller which is a significant time saving. Not only this, we also leave the channel available for other
operations to take place while this plane is busy. The controller can schedule other requests as long as they are destined to other planes this whole time. The controller has to send only command/control data to the plane to invoke a copy-back merge operation in the victim plane and the time taken to deliver such a request on the channel is very less when compared to data operations. Hence the main channel is available most of the time to other requests. So using DLOOP, we can save time and increase throughput using the copy-back operations which can boost performance highly.

Now we will describe how the algorithm works. A lot of this is directly adopted from the DFTL algorithm Figure 4.5 outlines the working of the algorithm. When a request comes to FTL, it is first checked to see if the mapping information is present in CMT. If it is, the physical addresses are read and the request is sent downward to the SSD. If it’s not present, which happens when the cache is empty or that particular mapping is not present, the request is sent to the SSD so that the mapping tables can be read from. Depending on the logical addresses of the incoming request the GTD (Global Translation Directory) in the FTL decides which plane to send the request to. The mapping information is spread across multiple planes based on the logical address. If it’s a new write request, the physical address can be anywhere inside the plane. Every plane has a reference to a current free block where it has to write new requests. If there are sequential requests, the mapping data can be looked up by interleaving requests and later the actual read/write data can be interleaved. Whenever a new request comes in, if the mapping information is not present in cache it is then bought into cache. If the cache is full, an existing entry is removed from the cache and the victim is selected by using a segmented LRU algorithm. If this mapping information is dirty, as in it has been updated while it was still in cache; this information has to be propagated to the SSD. The GTD (Global Translation Directory) which has the information of all the translation pages on the different planes of SSD is consulted and the and the old translation mapping is invalidated from that block and sent to the right plane to be written out. When an update request comes in, the entry if present in the CMT is invalidated. The request then proceeds to the same plane in which the old data was present. It is written in one of the extra blocks in that plane. This information is then updated in the CMT. Once a garbage collection process is invoked and the victim block is selected, the valid data moves from the old block to the new block and since this is a page mapping scheme, information about every page has
1. **Input:** Request Logical Page Number \((\text{request}\ lpn)\), request’s size \((\text{request}\ size)\), request type \((\text{request}\ type)\)

2. **Output:** NULL

3. **while** \(\text{request}\ size \neq 0\):

4. **if** \(\text{request}\ lpn\) is not present in Cache Mapping Table (CMT) then:

5. **if** CMT is full:

6. select \(\text{victim}\ lpn = \text{select}\_\text{victim}\_\text{entry}()\):

7. **if** \(\text{victim}\ lpn\) has been updated:

8. consult GTD before writing out page to the flash

9. **else:**

10. entry present in mapping table on flash is already latest. nothing to do

11. **end**

12. erase the victim page

13. **end**

14. consult GTD to find the location of the translation page which holds mapping information of the current \(\text{request}\ lpn\)

15. load entry of translation page into the CMT

16. **end**

17. **if** \(\text{request}\ type\) is read:

18. read from corresponding \(\text{request}\ ppm\)

19. **elseif** \(\text{request}\ type\) is write:

20. **if** \(\text{request}\ ppm\) doesn’t exist:

21. calculate \(\text{plane}\_\text{no}\) based on \(\text{request}\ lpn\):

22. write to \(\text{current}\_\text{free}\_\text{block}\) in that particular plane

23. **else:**

24. \(\text{plane}\_\text{no} = \text{get}\_\text{plane}\_\text{no}()\) of previous page

25. write request to \(\text{current}\_\text{free}\_\text{block}\) in the same plane

26. **end**

27. **elseif** \(\text{request}\ type\) invokes GC:

28. \(\text{victim}\_\text{block} = \text{select}\_\text{victim}\_\text{block}()\)

29. **for** each \(\text{page}\) in all pages in \(\text{victim}\_\text{block}\):

30. read page into plane register using copy-back

31. write page back in \(\text{curr}\_\text{free}\_\text{block}\) using copy-back

32. **end**

33. **end**

34. \(\text{request}\ size \--\)

35. **end**

---

**Figure 4.5. The DLOOP algorithm.**
to be written to new translation pages. If this information is present in cache, it is just updated in cache and taken care upon while eviction. If it’s not available in cache it is written to the translation pages directly. This all happens if the victim block is the data block. If the victim block is the translation block, the information has to be updated in the GTD which keeps track of the physical addresses of the translation mapping blocks in each plane. The GTD is present in the cache. Since the merge operation which happens by copying valid data from the victim block to the new physical block happens in the same plane using the copy-back scheme, the external channel is still available most of the time for processing other requests. Also, since all update requests go into the same planes the original data was, this implicitly wear-levels blocks on a per plane basis without an external mechanism.
CHAPTER 5

PERFORMANCE EVALUATION

In this section, we evaluate the performance of our DLOOP algorithm along with two existing commonly used algorithms FAST and DFTL. Like mentioned before, FlashSim will be used to conduct the simulations for all the three algorithms. The simulation will be conducted under different parameters and different workloads. FlashSim can be configured using multiple configurations. One such configuration is to simulate SSDs of various sizes. The performance of an SSD varies depending on its size. Enterprise SSDs come in different sizes. In our experiments we use SSDs of the sizes of order 4, 8, 16, 32, 64 GB. This means that each SSD will have different number of plane/dies and packages. SSDs usually have a few extra blocks which are used to store updated data in them for out of place write operations. These extra blocks are used as log blocks to store updated data in the flash drive. These blocks are merged on a regular basis with other data blocks when their number drops below a certain threshold. The numbers of such extra blocks are usually a fraction of the number of data blocks. Varying the number of such extra blocks can affect the performance of the SSD since the merge operation is triggered at different time points depending when they reach that given threshold. We vary the number of extra blocks to see how it affects the performance. We conduct simulations with increasing number of extra blocks which are in the order of 3%, 5%, 7%, and 10% of the total number of data blocks. Another parameter that is configurable is the page size of each block. A higher page size would mean that more data would be read/written on every incoming request. This would mostly be advantageous for workloads with large requests size, since this would reduce the number of requests send to the flash drive. On the other hand, for workload with small requests which don’t usually end on page boundaries this would mean reading and writing unnecessary data along with every request. Experiments are conducted to see how page size influences behavior in SSDs. The simulations are conducted with page sizes of 2, 4, 8 and 16 KB. One of the other configuration setting that we varied was the number of planes per die. The primary reason to do this was to see how this effects the interleaving capacity of the system and hence the
overall performance. Decreasing the number of planes in a die would decrease the response
time of data transfers for merge operations since each plane operates as a single entity at
which parallelism can be performed. This would lower the contention on this particular die.
However, increasing the number of planes per die would increase data’s spatial locality as
more planes would be in the same die. The different parameters with which FlashSim is
configured is shown in Table 5.1

**Table 5.1. Simulation Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value (Default) – (Varied)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of planes in a die</td>
<td>(4) – (2, 4, 8, 16)</td>
</tr>
<tr>
<td>Page size (KB)</td>
<td>(4) – (2, 4, 8, 16)</td>
</tr>
<tr>
<td>Flash block size (page)</td>
<td>(64)</td>
</tr>
<tr>
<td>Flash SSD capacity (GB)</td>
<td>(8) – (4, 8, 16, 32, 64)</td>
</tr>
<tr>
<td>Percentage of free blocks (%)</td>
<td>(3) – (3, 5, 7, 10)</td>
</tr>
<tr>
<td>Block erase latency (μs)</td>
<td>(2000)</td>
</tr>
<tr>
<td>Page read latency (μs)</td>
<td>(25)</td>
</tr>
<tr>
<td>Page write latency (μs)</td>
<td>(200)</td>
</tr>
<tr>
<td>Chip transfer latency per byte (μs)</td>
<td>(0.025)</td>
</tr>
</tbody>
</table>

We use real-world traces to study the impact of different FTLs on a wide spectrum of
workloads. The selection of traces was done to cover different types of workloads. A
workload can be characterized by type of requests it contains. One of such parameter is the
ratio of read to write requests. A workload can be read dominant, write dominant or be
balanced between both. A workload is also characterized by the frequency of requests it
receives. This is measured as inter-arrival time between requests, which can be either read or
write. A workload with low inter-arrival time is to have high IOPS (Input Operations per
second) and the one with high inter-arrival time has low IOPS. Another parameter that is
considered is the size of each request which can be small, spanning less than a page to large
ones which usually occupy multiple pages. Apart from this, a workload usually consists of
requests which can be sequential accesses or purely random in nature. Depending on the type
of system the workload was captured on, each trace can have different distribution of data.
When we did our experiments, we tried to get trace files that covers all the types of requests as discussed above.

We evaluate our DLOOP algorithm with five real world traces: Financial-1, Financial-2 [29], TPC-C [30], Exchange [31], Build [31] which has been widely used for researches conducted by different authors. The statistics of the traces are given in Table 5.2.

<table>
<thead>
<tr>
<th>Workloads</th>
<th>Financial1</th>
<th>Financial2</th>
<th>TPC-C</th>
<th>Exchange</th>
<th>Build</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of writes</td>
<td>4,099,354</td>
<td>653,082</td>
<td>123,060</td>
<td>69,492</td>
<td>200,462</td>
</tr>
<tr>
<td>Number of reads</td>
<td>1,235,633</td>
<td>3,046,112</td>
<td>244,399</td>
<td>80,164</td>
<td>237,452</td>
</tr>
<tr>
<td>Requests per second</td>
<td>122</td>
<td>90</td>
<td>3000</td>
<td>17</td>
<td>45</td>
</tr>
<tr>
<td>Average request size (KB)</td>
<td>3</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>8</td>
</tr>
</tbody>
</table>

Financial1 and Financial2 are taken from OLTP applications running at two large financial institutions. Financial1 is a write dominant trace which contains around 76% of write requests and 24% of read requests. On the other hand Financial2 is a read dominant trace amount to 82% of reads and 18% of writers. The average request size of Financial1 trace is 3KB and 4 KB for Financial2 trace. The number of requests per second for both the trace files are around the same which is 122 IOPS for Financial1 and 90 IOPS for Financial2.

TPC-C trace is trace collected on a storage system connected to a Microsoft SQL Server via storage area network. TPC-C has about 66% of read requests and 34% of write requests. The average request size is 8 KB. TPC-C is a very intensive workload and the requests are mostly random. The requests in this trace arrive at 3000 IOPS which is a lot more than the Financial traces. The fourth trace is known as the Exchange trace has been collected over at the Microsoft Exchange mail server for several users. These traces are broken down into 15 minutes intervals. The exchange trace has almost even distribution with respect to the number of reads and writes. Percentage of reads requests are 53 and percentage of write requests are 47. The average request size is around 12 KB. Since this is a mail server which doesn’t do a lot of crunching the intensity is lesser than the other servers we have seen. It has around 17 IOPS. The final trace known as Build trace is taken from a Windows Build server. This system compiled and produced complete builds every day for a 32-bit version of the
Windows Server operating system. This trace also has been broken down into 15 minute interval. The build trace is also pretty evenly distributed with read requests totaling up to 54% and write requests coming up to 45%. The average requests size is around 8 KB. This has around 45 IOPS.

Apart from the real world traces we also evaluate DLOOP algorithm against synthetic traces. The synthetic traces can be generated with different parameters of our choice which will help us in removing any sort of uncertainty that could have been present in real world traces. The Intel Open Storage Toolkit [32] was developed for storage research and allows us to vary many trace parameters such as percentage of read and write requests, percentage of random requests, benchmark size, etc. Since the traces are generated by controlling each variable, we can be certain of how the change in one parameter can affect the algorithm by keeping other parameters in check. The idea is to see how these algorithms behave under varying workloads. Keeping this in mind, four synthetic traces were generated to see how the performance of DLOOP and other algorithms differ when certain parameters are changed.

The first synthetic trace that was generated has a 75% of reads and 25% of writes. This trace will be used to evaluate how the algorithms work when the majority of requests are reads. The average request size is 32 KB. This trace is however pretty load intensive and has around 1200 IOPS. The second synthetic trace has a majority of write requests amounting to 75% of the total workload size. The request size is 64 KB and like the previous trace, this trace is also pretty intensive and has around 1150 IOPS. The increase in the number of writes will show us how good/bad the algorithms behave with increase in write requests which are more intensive than the read requests. It is noted that although write requests don’t usually have a large overhead, it is the accompanying operations like merging of data and garbage collection which are responsible for a lot of extra overhead which might bog down the overall performance. The next synthetic trace was generated keeping in mind the randomness of these incoming requests. It is a known fact that SSDs usually behave well with sequential requests and their performance drops with the rise of random requests. The workings of FTL algorithms can be heavily dependent on the pattern of incoming requests. A few algorithms perform really well and rely heavily on the fact that the incoming stream of requests will be sequential, but when tested under random requests their performance might take a dip. Other algorithms are resilient to randomness in requests. The next two traces will help us evaluate
how randomness in disk requests can change the behavior of DLOOP and other algorithms. The third trace that was generated has 20% of randomness in the requests it. The percentage of write requests was kept to 75%. The reason we have more number of writes is because when we vary the randomness in requests, we want to make sure there is sufficient difference in the results we observe. This can happen when there is heavier workload which will help distinguish results better. The read requests were 64 KB and the writes were 128 KB in size. The IOPS for this trace was around 850. We increase the amount of randomness in requests to 80% in the final synthetic trace so that we see how this will affect our results when compared to the earlier trace. For purpose of stability, the number of writes and sizes of requests remained the same from earlier trace. The number of IOPS also didn’t change a lot. The four synthetic traces as explained above will be used to carry out simulations.

5.1 PERFORMANCE METRICS

The mean response time is a good metric for estimating the FTL performance since it captures the overheads due to address translations, merges and garbage collection. The mean response time is on a per request basis and basically measures the time it takes to complete each request on the flash drive. For the queuing delay, average waiting time is a good metric. This captures the time each request took in waiting before the request was serviced. This would include queuing delay at the serial-to-host interface, the die channel delay and the plane availability delay. Since the basic idea of DLOOP is to distribute extra blocks along with data blocks which improves the performance as a result of better interleaving, we capture the standard deviation of request distribution amongst the planes. A lower standard distribution indicates that the requests were distributed more evenly across multiple planes.

5.2 EXPERIMENT RESULTS

We conduct a group of simulations experiments to examine the effectiveness of the DLOOP strategy. The first group of experiments was done by varying the SSD size from 4 GB to 64 GB, by doubling the size every time. The motive was to see how varying disk size effects the performance of the three algorithms. The test results in Figure 5.1 shows the graphs of mean response time in milliseconds. As seen from the results we can interpret two things. One is that the DLOOP algorithm performs better than the other two algorithms for
Figure 5.1. Performance impact of SSD capacity on three schemes.
all traces and for all disk sizes. The other observation is that, as disk size increases the mean response time also decreases. The reason for that is, the larger the disk size, the more time it takes for the disk to get full before operations like garbage collection and wear-leveling kicks in. Garbage collection causes a lot of merge operations to happen which is usually a major cause of overheads in these devices. Coming back to Figure 5.1 for the mean response time, the DLOOP algorithm performs 70% and 90% better than DFTL and FAST algorithms for the 4 GB SSD. For the waiting time the DLOOP algorithms is 72% better than DFTL and well over 90% for FAST. As we move to higher sized SSDs, we see similar performance improvements for the DLOOP and DFTL algorithms and their individual response times have reduced by almost half. For the 64 GB SSD, the DLOOP algorithm has a mean response time of 0.43 ms and a waiting time of 0.2 ms while the DFTL algorithm has a response time of 2 ms with a waiting time of 1.7 ms. The FAST algorithm performs weakly with a mean response time of 11.4 ms for the 64 GB SSD. In the case of the Financial 2 trace, we see that the improvement of DLOOP over DFTL and FAST has lowered. The reason for this is that Financial 2 is a read dominant trace and read requests don’t incur in performance penalty like write requests do. Read requests don’t cause updates which eventually leads to merge operations, the main reason why DLOOP is successful. DLOOP performance improvement for a 4 GB SSD over DFTL is by 32% and for FAST is by over 90%. The waiting time improvement for DLOOP over DFTL is around 50% and for FAST is above 90%. For higher sized SSDs, DFTL algorithm improves at a faster rate than DLOOP which reduces the improvement to 21% for a 64 GB sized SSD. The improvement over FAST remains the same. The waiting time has also improved more for DFTL than DLOOP. DFTL has a 27% waiting time improvement against DFTL. For the TPC-C trace which is a high IO trace with a lot of random requests, we see that the performance of DFTL falls a lot. The DFTL algorithm picks up free blocks to write sequentially from the same plane.

This can be a problem if several of such requests come in quick successions, because they start adding up to the waiting queue of the that particular plane. However, since most of the requests get served off the cache mapping tables present in the cache itself, this latency gets hidden but not in this case since the requests are mostly random in nature. The FAST algorithm maintains a table between the logical block number and physical block number, so even if random requests comes in they can be mapped to a different plane and the random
nature of request doesn’t matter much. The DLOOP algorithm performs a lot better than DFTL because of the high waiting times at each plane. For the 4 GB SSD, it performs around 75% better than FAST with respect to response time and 80% better with respect to waiting time. As size increases we see that the DFTL improves a lot because of lesser update requests but still performs worse than DLOOP. The FAST algorithm has improved more than the DLOOP, but still DLOOP shows a 57% improvement in response time and 50% in waiting time for the 64 GB SSD. For the Exchange trace which has even distribution of reads and writes, for the 4 GB SSD, the DLOOP performs 65% better than DFTL in response time and 75% in waiting time while more than 90% for FAST. The gap remains almost the same for higher sized SSDs like the 64 GB SSD, DLOOP performs 62% better in response time and 70% in waiting time than DFTL. It performs 80% better in response time and waiting time when compared to FAST. Coming to the final trace which is the Build trace, for the 4 GB SSD, the DLOOP algorithm has a performance gain of 45% in response time and 43% of improvement in waiting time against DFTL and more than 90% when compared to FAST. As we go higher in size, the DFTL performance increases more than DLOOP, like for the 64 GB SSD the performance difference between DLOOP and DFTL is now around 31% and 45% for response time and waiting time respectively. The FAST algorithm still lags behind a lot. Now we look at the standard deviations plots which are basically the log of the number of requests that each plane receives over the entire simulation. For the Financial 1 trace, we see that DLOOP has better distribution when compared to DFTL and FAST. For the 4 GB SSD, DLOOP has a value of 11.6 compared to 12.8 for DFTL and 12.1 for FAST. As the SSD size increases the distribution improves for all the 3 algorithms. At 64 GB, the DLOOP value stands at 10.8 while DFTL stands at 12.1 and 11.1 for FAST. We see similar results for the Financial 2 trace. The distribution is more even with DLOOP when compared to DFTL and FAST and as size increases this distribution improves because requests are distributed more evenly to more planes that come with higher SSDs. We also see that FAST started better than DFTL in the 4 GB SSD but ended higher than DFTL in the 64 GB SSD. However they both perform worse than DLOOP. For the TPC-C trace we see that the DLOOP has lesser deviation than FAST and DFTL and as size increases the difference increases. The difference in a 4 GB SSD was 0.8 for DFTL and 1 for FAST, but for a 64 GB SSD the difference is 1.2 for DFTL and 1.6 for FAST. For the Exchange trace with the DLOOP algorithm, we see
that the distribution gets more even for higher sized SSDs. It’s 11.1 for the 4 GB SSD and 9.7 for 64 GB, but that’s not the case with DFTL algorithm. The deviation increases from 11.2 for the 8 GB to 11.7 for the 16 GB SSD and then reduces for the 32 and 64 GB. For the FAST algorithm the deviation falls through the 32 GB, but increases back for 64 GB. This increase is DFTL is because even as the number of planes increases from 16 to 32, the DFTL algorithm wasn’t able to spread out requests more efficiently resulting in unutilized planes which increased standard deviation. Similarly the FAST algorithm wasn’t able to efficiently use the extra 64 planes it got when size increased from 32 to 64 GB. For the Build trace we still see that DLOOP has overall better distribution than DFTL and FAST. The difference reduces between DFTL and FAST as we move to higher sized SSDs. The difference for a 4 GB SSD was 1 between DFTL and FAST and it reduced to 0.3 for a 64 GB SSD. DFTL on the other hand, has an increase in deviation from 4GB to 8 GB SSD and from 16 GB to 32 GB because of underutilization of extra planes.

The second group of experiments was done by varying the number of extra blocks the SSD has and the results are shown in Figure 5.2. The goal was to study the impact of how increasing/decreasing the number of extra blocks would affect the overall performance of the SSD. The extra blocks are used to support update operations and after a while when they reach a certain threshold they are to be merged with the data blocks. The number of extra blocks are usually in some small percentage compared to the data blocks that are used. This percentage generally varies depending on several factors including FTL algorithm used, SSD size, manufacturer, etc. Like mentioned earlier we vary the number of extra blocks between 3-10% of the data blocks. The SSD size is kept constant through the experiments. Note that the SSD size is only comprised of the data blocks and not the number of extra blocks. We select the size of SSD to be 4 GB. The reason for selecting a small sized SSD was to notice the impact clearly of varying number of extra blocks. The data blocks are used up more quickly in a small sized SSD and extra blocks come into play at an earlier time when compare to higher sized SSDs. From Figure 5.1 of Financial 1 trace we see that the DLOOP algorithm performs better than DFTL and FAST for all the different configurations. As the number of extra blocks increase we definitely see improvement in the mean response time and the reduction in plane waiting times. For DLOOP this difference doesn’t seem to be very large when normalized to a per request time. However for DFTL we see a good amount of
Figure 5.2. Performance impact of number of extra blocks on three schemes.
reduction in the mean response time and waiting time as the number of extra blocks increase. The reason for this is that, the more number of extra blocks that are used, the later the merge operations are invoked which and a lot of the initial requests happen without any overhead of erase and copy operations. For the DFTL algorithm we see that the mean response time and waiting time increases as we move from 7% to 10%. The reason for this that the DFTL algorithm initially stores its page mapping information in the first few blocks of plane 0. These are not moved until merge operations take place. In the case of increasing extra blocks, these are accessed more frequently from plane 0 which increases the contention for a longer time before they are moved out to a different plane and hence the increase in waiting times. The performance improvement of DLOOP over DFTL for 3% extra blocks is 66% and for FAST is over 90%. This performance improvement decreases to around 60% for 7% extra blocks while it says the same for FAST. The waiting time reduces by 75% and over 90% for the 3% extra blocks when compared to DFTL and FAST respectively. They decrease to around 65% in case of the DFTL for the 7% extra blocks. For FAST, its still very high.

Coming to Financial 2 which is mostly a read dominant trace, we see that the increase in extra blocks has very slight effect both in DLOOP and DFTL algorithms with respect to performance improvement in response time and waiting times. FAST has more improvements but still overall it has a high mean response time and waiting time. The performance improvement of DLOOP over DFTL in mean response time is around 37% for 3% extra blocks and reduces a little bit to around 35% for a 10% extra blocks. The waiting time improvement for 3% extra blocks over DFTL is 60% and is the same for higher number of extra blocks as well. Coming to FAST there is more than 90% improvement in mean response time and waiting times for 3% extra blocks. Even though these metrics which include both mean response time and waiting improve as number of extra blocks increases, its still very high compared to the other two algorithms to make a difference. For the TPC-C trace, like we saw in the earlier set of figures, the DFTL performs badly for reasons mentioned above. The change in number of extra blocks decreases the performance of DFTL because of the additional waiting time in plane 0. The DLOOP algorithm reduces the mean response time by 69% and waiting time by 78% when compared to FAST algorithm for 3% extra blocks. As we get more extra blocks, FAST algorithm improves but DLOOP algorithms remains about the same. For 10% extra blocks the DLOOP algorithm’s performance
advantage reduces to 62% for mean response time and 72% for waiting time. For the DFTL algorithm the performance is more than 90% and keeps on increasing for higher extra blocks. Moving on to the Exchange trace the DLOOP and DLOOP algorithm perform mostly the same irrespective of the number of extra blocks that they have mostly because the trace is not heavy on workload and the extra blocks don’t do a lot of good to them. FAST on the other hand starts with a high mean response time and waiting time and then improves considerably with the increase in the number of extra blocks. DLOOP performs 78% better than DFTL in terms of mean response time and 85% in terms of waiting time for 3% extra blocks. The performance advantage remains around the same for higher number of extra blocks. The FAST algorithm starts with high times which are worse than 95% when compared to DLOOP but as extra block increase, its starts improving and at 10% extra blocks the performance has considerably improved but still relatively high when compared to DLOOP. DLOOP beats FAST by 87% in mean response time and 90% in terms of waiting time. The last build trace shows similar results as exchange. DLOOP improves mean response time by 73% and waiting times by 84% for 3% extra blocks when compared to DFTL. FAST waiting times for 3% extra blocks are pretty high. Moving to more extra blocks the performance difference doesn’t change much between DLOOP and DFTL. Even though FAST’s response time and waiting time has improved, DLOOP still performs a lot better than FAST and outperforms it by more than 90%. Moving on the plot of standard deviation we see that for the Financial 1 trace the DLOOP has better distribution when compared to DFTL and FAST. With 3% of extra blocks we see that DLOOP has a value of 11.3 compared to 12.2 for DFTL and 11.8 for FAST. As we move towards to higher extra blocks, we see that DFTL’s distribution improves even though very slightly. On the other hand for DFTL and FAST algorithm, the deviation increases. The reason why this happens in DFTL and FAST is because that a lot of page/block mapping requests come to plane 0 and this is not copied to other planes until merge happens which increases the requests at plane 0 more than others. We see similar results for Financial 2 as well. The distribution for DLOOP doesn’t improve much, but for DFTL and FAST it drops as number of extra blocks increases. The difference in the logarithmic values between DFLOOP and DFTL for 3% extra blocks is around 0.9 and increases to 1.7 for 10% extra blocks. For FAST its around 0.5 for 3% and increases to 0.9 for 10% This behavior is responsible for increasing the response times for DFTL
algorithms as we saw in earlier results for 4 GB SSD. However they both perform worse than DLOOP. For the TPC-C trace we see that the DLOOP has lesser deviation than FAST and DFTL. There is not much change in deviation as the extra block increases. The difference between DLOOP and DFTL is around 0.7 and it remains the same. Deviation on the FAST algorithm increases till 7% and then drops back. The difference between DLOOP for 3% extra blocks is around 0.6 and increases till 0.8 for 7% and comes back 0.7 for 10%. For the less intensive Exchange trace we see that deviation for DLOOP is better than the other two algorithms and it stays the same for higher extra blocks. The difference between DLOOP and DFTL for 3% extra block is 1. It's around the same for FAST. However, the deviation increases slightly as extra blocks are added to DFTL and now the difference becomes 1.2 for 10% extra blocks. For the build trace the distributions is about the same as for the Exchange trace. DLOOP remains constant about 11.1 while DFTL starts with 12.2 and increases slightly to 12.4 FAST this time again improves slightly by starting off from 12.1 and reducing down to 11.9.

The third set of experiments are conducted to see how page size effects the behavior of the algorithms. The page size was varied from 2 KB to 8KB for a constant SSD size of 8 GB as shown in Figure 5.3. The disk size remained the same to isolate changes from variation in size and only to capture changes from page size. The first graph shows the trace results of Financial 1. This shows that as the page size increases the mean response time and waiting time decreases. It also shows that for every page size DLOOP algorithm performs better than DFTL and FAST algorithm. For the 2KB page size, the DLOOP algorithm reduces the mean response time by 77% and waiting time by 74%. The improvement time when compared against FAST is really high and 90% upwards. As we move to higher page sizes, the performance of DFTL improves at a faster rate than DLOOP. For the 16 KB page size the performance improvement of mean response over DLOOP is 58% and for waiting time is 70%. DFTL sees more performance gain because increasing the page size sees a reduction in number of requests sent to the disk. This also takes place even in DLOOP but the difference is that it reduces the extra waiting time that was seen in smaller sized pages. DLOOP on other hand didn’t have a lot of waiting time to begin with. The reason why mean response time and waiting times don’t reduce by half every time the page size is doubled is because that all the data that is read/written in a request might not be useful. The FAST algorithm also
Figure 5.3. Performance impact of flash page size on three schemes.
improves compared to lower page sizes but performance is still bad and worse by more than 90%. For the next trace which is a read-dominant trace Financial 2, the performance improvement of mean response time on DLOOP over DFTL is 65%, and waiting time it is 45%. FAST performs bad even in this case and the performance improvement is above 90%. As page size increases, like earlier DFTL seems more performance improvements. For the 16 KB page size DLOOP outperforms DFTL in mean response time by 30% and in waiting time by 40%. The FAST algorithm still underperforms with the difference in performance being over 90%. For the TPC-C trace the DFTL performs bad for the 2KB and 4KB page sizes but does better for the 8 and 16 KB page size. For this trace, DLOOP performs 69% better in mean response time compared to the FAST algorithm and 69% better in waiting time. It performs more than 90% better in both mean response time and waiting time than FAST algorithm. Moving to higher page sizes of 8 KB and 16 KB, DFTL has improved a lot. Now in comparison for the mean response time for 8 KB, the DLOOP algorithm performs 82% better than DFTL and 53% better than FAST. For the 16 KB page size the difference between DLOOP and DFTL has reduced to 44% for mean response time and 45% for waiting time. For FAST the difference between mean response time is 19% and waiting time is 15%. For the next trace which is Exchange, DLOOP improves the mean response time by 81% and waiting time by 85% when compared to the DFTL algorithm. FAST times are pretty high and we see improvement of 90% upwards. For higher page sizes the DFTL and FAST algorithm improves pretty quickly. For the 8 KB page size the mean response time improvement of DLOOP over DFTL has decreased to 73% and 86% for FAST. For the 16 KB page size the mean response time gain of DLOOP over DFTL has decreased even further to 63% and 87% waiting time. DLOOP mean response time over FAST is now 59% and waiting time improvement is 80%. For the final trace in this set, the Build trace follows the earlier trace. The performance improvement of DFTL algorithm over DFTL is 53% for mean response time and 66% for waiting time. The FAST times are still high and result in poor performance. As page size increases all three algorithms show improvement. DFTL and FAST algorithms improve faster than DLOOP. For the 16 KB page size DLOOP performance improvement for mean response time over DFTL is 37% and for waiting time is 66%. The FAST algorithm for this page size has gotten a lot better when compared to smaller page sizes but still is lesser than DLOOP. DLOOP outperforms FAST in mean response time
by 51% and 70% for waiting time. We now look at the graphs involving the log of standard deviation to see how requests deviations changes as per increasing page size. For the Financial-1 trace, we see that DLOOP has better distribution when compared to DFTL and FAST. With 2 KB page size we see that DLOOP has a value of 12.2 compared to 13 for DFTL and 13.9 for FAST. As we move towards to higher page sizes we see that DLOOP’s distribution improves along with DFTL and FAST. For the 16 KB page size the value for DLOOP, DFTL and FAST is 11.05, 11.9 and 12.9 respectively. DLOOP does equally better for all three page sizes. We see similar results for Financial-2 as well where As the page size increases all the three algorithms DLOOP DFTL and FAST improve distribution of requests. The difference between DLOOP and DFTL for 2 KB page size was 0.5 while difference with FAST was 0.7, but as page size increased DFTL and FAST improved more than DLOOP and the difference for a 16 KB page size between DFTL and FAST was 0.1. For the TPCC trace, the distribution for DLOOP is better than the other two algorithms. The difference between DLOOP and DFTL for 2 KB is 0.5 while the difference between FAST is 1.4. The difference reduces by some amount for higher sized page size as the distribution improves for all the three traces. The difference between DFTL for a 16 KB page size is 0.3 and for FAST is 0.9 when compared to DLOOP. For the less intensive Exchange trace we see that deviation for DLOOP is better than the other two algorithms and it stays the same for higher sized pages. The difference between DLOOP and DFTL for 2 KB page 1. It’s around 0.6 for FAST. For the 16 KB page the difference is about the same. All the traces improve distribution at the same rate. For the build trace, the DLOOP starts worse than DFTL but better than FAST for the 2KB page size. DLOOP value is 12.1 while DFTL’s is 11.8 and FAST is 12.4. DLOOP reduces diversity for higher page sizes but DFTL doesn’t do this in a linear fashion. This is probably because that even though the number of requests decreases for higher page sizes, but they might not be able to spread out evenly as a result of this. At the 16 KB page size DLOOP has a lead of 0.2 over DFTL and 0.3 over FAST.

For the final set of experiments, we examine how the number of planes per die affects the performance of the three algorithms as shown in Figure 5.4. The SSD size will remain the same selected as 8 GB to isolate the changes. Since the number of planes per die change, this would actually result in different number of dies/packages for the same SSD size. Doing so increases/decreases contention on two types of channels. One is the channel connecting the
Figure 5.4. Performance impact of planes per die on three schemes.
planes and the other is the one connecting the dies. We try to examine how much these changes contributes to the final response time and total waiting times due to the multiple queues. The first trace is the Financial-1 trace and shows that the DLOOP algorithm performs better than the other two algorithms. For two planes per die DLOOP performs 56% better than DFTL with respect to mean response time and 70% with respect to waiting time. Compared with the FAST the improvement is huge and over 90%. Moving to higher planes per die only increases the contention on the main channel, but this leads only to a slight decrease in the performance in both DLOOP and DFTL. The performance improvement stays the same even for 16 planes per die. For the Financial-2 trace the mean response improvement for the DLOOP algorithm is 41% when compared to DFTL and 67% in waiting time. The performance improvement over FAST is huge and over 90%. As we move to higher sized number of planes per die, the performance of DLOOP deteriorates very slightly but stays the same for DFTL. The mean response time improvement is now 38% and waiting time improvement is 65% when compared to DFTL. When compared to FAST the difference is still pretty huge. For the TPC-C trace the DFTL algorithms performs poorly. The performance difference between DLOOP and DFTL is well over 90% for all the experiments. When compared to FAST, DLOOP performs 84% better in mean response time and 89% better in terms of waiting time. For higher planes per die the performance remains about the same. The next trace is the Exchange trace in which we see that DLOOP does 80% better in mean response time when compared to DFTL and 87% in terms of waiting time. The FAST algorithm still performs badly with over 90% improvement. For higher planes the DLOOP’s mean response time increases slightly and now the improvement is 78% for mean response time and 85% for waiting time. The performance improvement for FAST is still very high and over 90%. For higher number of planes per die the mean response time and waiting time of DLOOP reduces very slightly. The performance improvement for 16 planes per die is now 42% for mean response time and 64% for waiting time against DFTL. Coming to the graphs which shows the plane distribution, the graph shows pretty consistent results for the three algorithms. For the Financial-1 trace, DLOOP has the least deviation of 11.73 followed by DFTL and FAST which have 12.8 and 13.8 respectively.
Since the number of planes remain the same and only the planes per die differ for this set, we will have the same plane distribution for all the variations. For the Financial 2 trace DLOOP has log of deviation 11.6 along with FAST. DFTL has a deviation for 12.3. The next trace is TPC-C and DLOOP has a value of 12 while DFTL and FAST have a deviation of 12.1 and 12.2. For the next trace which is Exchange, DLOOP has a deviation of 10.9 which is still the least, while DFTL and FAST have 11.2 and 11.4 respectively. For the final trace Build, we see that FAST has a better distribution than DLOOP. FAST has a value of 11.1 while DLOOP and DFTL have values 11.8 and 11.9. Even though that FAST has better distribution it still suffers from long waiting times for channel and plane acquisition.

The next set of simulations are evaluated against synthetic workloads that were generated as explained earlier. The goal is to evaluate how DLOOP performs under varying types of request and if its behavior is dependent on the sequentiality/randomness of these incoming requests. The experiments were conducted against all the four synthetic traces. The experiments shown in Figure 5.5 were done by varying the SSD size from 4 GB to 64 GB, by doubling the size every time. The motive was to see how varying disk size effects the performance of the three algorithms. The test results in Figure 5.5 shows the graphs of mean response time in milliseconds. As seen from the graphs for the Read Dominant trace, we see that DLOOP algorithm performs better than DFTL and FAST algorithms. For the 4 GB SSD the difference between DLOOP and DFTL is around 26% and more than 90% for FAST. It must be noted that FAST algorithm performs poorly for synthetic traces because of the intensive workload and the low inter-arrival time between incoming requests. Due to this, the requests keep adding up in the waiting queue and the response time for the current request along with all the other requests that are waiting to be serviced increases linearly. Also, since these requests are spaced out pretty evenly in time, there is no elongated periods of inactivity which would help FAST recover by servicing all the pending requests in this time. The waiting time improvement of DLOOP against DFTL is 62% and for FAST is over 90%. As we move to higher sized SSDs, the mean response time of the algorithms decreases because of the addition of more blocks that delays the invoke of the garbage collection process. The difference however remains close to same and is 29% for DFTL and over 90% for FAST. The waiting times of DLOOP and FAST decreases as size increases while remains the same for DFTL. The difference for waiting time is around 75% and over 90% for FAST. As we
Figure 5.5. Performance impact of SSD capacity on three schemes for synthetic traces.
move to a write dominant trace we see that the mean response time increases and so does the waiting times for all the algorithms. This is because write workloads are more intensive and invoke a lot update operations which later result in garbage collect operations. The merging and copying of data around adds to the overhead. For the 4 GB SSD, the DLOOP algorithm performs 72% better than DFTL and a lot better than FAST. For the 4GB SSD, DLOOP performs close to 90% better in terms of waiting times. As we move to higher sized SSD, we see that performance of DLOOP improves but DFTL remains almost the same. This shows that DLOOP is able to take advantage of the increasing blocks more than the DFTL algorithm. The DLOOP algorithms is 75% better than DFTL in terms of mean response time and more than 90% in terms of waiting times for 64 GB. Even though the numbers of FAST improve, they were very high to begin with. FAST algorithm deteriorates with high volume of write requests. Coming to the traces where we try to see how randomness in incoming requests can change the behavior of the FTL algorithms, we have a trace where the majority of requests are sequential writes. We see that the performance of DLOOP and DFTL is not very far apart in terms of mean response time. The DLOOP algorithm performs 5% better than DFTL for all sizes of SSD and over 90% when compared to FAST. When coming to waiting times, the DLOOP performs 20% better than DFTL for the 4 GB SSD and 35% better than the 64 GB SSD. The performance gap widens for higher sized SSD. FAST is still worse. This trace and the next one are not as workload intensive than the above two traces. This is because we wanted to see how changing the randomness of trace would affect the algorithms performance and hence we tried to suppress other factors to could have played in. The last trace has a majority of random requests. We see similar results like we observed earlier. A 5% improvement of DLOOP over DFTL for all sizes of SSD and over 90% over FAST. The waiting times are 24% and 34% better than DFTL for 4 GB and 64 GB size respectively. The reason we didn’t see significant performance deterioration in both DLOOP and DFTL is because both of them are page based requests. The performance hit is seen by FAST algorithm when moving from a sequential workload to a random workload because FAST is a hybrid based algorithms. Algorithms which use block based mapping or hybrid based mapping have the intrinsic shortcomings because of the full merge operations that have to be performed during the garbage collection process. However, page mapping scheme don’t have to do this and therefore don’t see a lot of difference when the pattern is changed from
sequential to random. Now we look at the standard deviations plots which are basically the 
log of the number of requests that each plane receives over the entire simulation. For the 
trace with majority of read traces, we see that DLOOP has better distribution when compared 
to DFTL and FAST. For the 4 GB SSD, DLOOP has a value of 9 compared to 14.1 for DFTL 
and 11.5 for FAST. As the SSD size increases the distribution improves for all the 
3 algorithms. At 64 GB, the DLOOP value stands at 7 while DFTL stands at 11 and 10.8 for 
FAST. Even though DFTL is still slightly higher, FAST it has improved more than DLOOP 
and FAST. The reason why DFTL is higher than FAST is because that DFTL always assigns 
incoming requests to one block it selects and unless it writes to all 64 pages to the current 
block it wouldn’t select another block. Depending on what plane the other new block is, the 
successive requests go there. This is why we see more requests getting assigned unevenly to 
one set of planes. For FAST, when requests first come in, they are distributed across planes 
based on the their LBA to PBA mapping stored in the FTL table. However, when update 
operations take place all requests go a predetermined log blocks and requests might start 
getting uneven from this point forwards. For the trace with majority writes we see that for the 
smaller sized SSD DLOOP performs a little worse than it did for read dominant trace. This is 
because write dominant trace generate more internal merge requests due to GC and has 
requests with higher block sizes. As size increases, the DLOOP does better and reaches level 
matching the earlier read dominant trace. DLOOP performs better than DFTL and FAST. 
DLOOP and FAST perform worse than they did earlier because write traces generate more 
merge requests incurred due to garbage collection and the resulting assignment of requests to 
plane might not be even. This is not a problem with DLOOP because all merge requests take 
place in a plane will get merged into the same plane. However, if merge happens more 
frequently in one set of planes, the distribution might be a little skewed. Moving on to the 
trace which has more sequential requests we see similar results as earlier. The distribution is 
more even with DLOOP when compared to DFTL and FAST and as size increases, the 
distribution improves as well. We also see that FAST started better than DFTL in the 4 GB 
SSD but ended higher than DFTL in the 64 GB SSD. However they both perform worse than 
DLOOP. For the final trace in which most of the requests were random, we see that the 
DLOOP has lesser deviation than FAST and DFTL and as size increases the difference
increases. The difference in a 4 GB SSD was 4.4 for DFTL and 1.8 for FAST, but for a 64 GB SSD the difference is 6.2 for DFTL and 6.3 for FAST.

The next set of experiments is to evaluate how the algorithms behave when the number of extra blocks are changed from Figure 5.6. From the first trace which has a majority of reads we see that DLOOP algorithm performs better than both DFTL and FAST. As the number of extra blocks increase, we don’t see any perform changes in the DLOOP and DFTL algorithms. The DLOOP performs around 22% than DFTL for all sizes. The DLOOP algorithm performs more than 90% than the FAST algorithm and even though the FAST improves performance at higher sizes, it doesn’t help much. As per the waiting times, DLOOP performs 60% better than DFTL for all percentage of extra blocks and more than 90% better than FAST. Moving to the next trace with majority writes, we see similar behavior. DLOOP performs 72% better than DFTL algorithm and more than 90% than FAST. The difference remains same through all the different number of extra blocks. The difference in waiting time between DLOOP and DFTL is close to 90% and is more than 90% for FAST. For the following trace with majority of sequential requests, we see that DLOOP performs a little better than DFTL with a margin of 5% and over 90% better than FAST. This holds true for all percentage of extra blocks. As per the waiting times, DLOOP performs 14% better than DFTL and over 90% better than FAST for all percentage of extra blocks. We see similar results for both mean response time and waiting times for trace with random requests. DLOOP does 5% better than DFTL for mean response time and 90% over FAST. For waiting times, we see 18% improvement for DLOOP over DFTL and still 90% improvement over FAST for all percentage of extra blocks. Coming to standard deviation of requests distribution for increasing percentage of extra blocks, we see that for the trace with majority reads, DLOOP has better distribution than DFTL and FAST. For the trace with majority reads we see that DLOOP has a value of 9.08, DFTL has a value of 14.1 and FAST has a value of 11.3 for 3% extra blocks. As the number of extra blocks increases, the distribution increase slightly for DLOOP and stays the same for DFTL. For the FAST the distribution worsens a bit for the 5% and 7% extra blocks due to underutilization of planes and improves a bit for 10%. For the trace with write requests, we see that the distribution is a little more skewed when compared to the read trace. However, in comparison of DLOOP with the other two algorithms there is similar behavior. DLOOP has best distribution amongst all the three
Figure 5.6. Performance impact of number of extra blocks on three schemes for synthetic traces.
algorithms and improves very slightly moving from 3% extra blocks to 10% extra blocks. DFTL distribution remains same throughout while distribution of FAST increases as the number of extra block increases. For the next two traces with sequential and random requests we see that the DLOOP has better distribution in comparison with DFTL and FAST. For both the traces, the deviation for DLOOP lowers slightly as we the number of extra blocks increases. For the DFTL, the distribution remains the same irrespective of the number of extra blocks. This is because the way DFTL assigns new blocks to incoming requests has little to do with the time when the garbage collection process has been invoked. FAST on the other hand has worse performance when the number of extra blocks increases. This is because all the extra blocks are concentrated in one plane which results in over utilization of that plane when compared to other planes.

The third set of experiments were conducted to see how page size effects the behavior of the algorithms using synthetic traces. The first figure from Figure 5.7 shows that the mean response time of the DLOOP is better than DFTL and FAST for all page sizes. For the smallest page size 2 KB, the DLOOP algorithm is better than 28% when compared to DFTL and over 90% to FAST algorithm. The waiting time of DLOOP is 60% better than DFTL and over 90% than FAST. As page size increases the mean response time and the waiting time decreases for all the algorithms. The waiting time of DFTL and FAST improves more than DLOOP because they were high to begin with. For the 16 KB page size the performance improvement of DLOOP to DFTL has lowered to 11% and 42% over FAST. FAST benefits from large page size because this reduces the number of incoming requests considerably reducing to lower mean response and waiting times. For waiting times, the performance improvement of DLOOP over DFTL is 40% and 85% over FAST. Moving on the next trace which has a majority of writes, we see that the mean response time and waiting times have increased for all the algorithms as when compared to earlier trace. However, for the smallest page size of 2 KB DLOOP is better than DFTL by 70% and over 90% than FAST. This difference is reduced for the higher page size. The difference for a 16 KB page size with DFTL is 5% and around 50% for FAST. In case for waiting times for a 2 KB page size DLOOP has a performance gain of around 90% over DFTL and more than 90% for FAST. However for the 16 KB page size the difference now is 60% for DFTL and little over 90% for FAST. The next trace which as a majority of sequential requests, the DLOOP and DFTL
Figure 5.7. Performance impact of flash page size on three schemes for synthetic traces.
perform closely. The performance gain of DLOOP over DFTL for a 2 KB page size is 8% and this reduces to 5% for a 16 KB page size. The mean response time of FAST is pretty high for a 2 KB page size but reduces considerably for 16 KB page size and now the difference is 70%. For waiting times, the DLOOP has 44% lower waiting times than DFTL for a 2 KB page size, but just 10% performance gain for 16 KB page size. For the FAST the waiting times are pretty high for the 2 KB page size and even though the reduction has been considerably high for a 16 KB page size, they are still relatively high. The final trace which has random requests follow the same pattern of results. The performance gain of DLOOP over DFTL for a 2 KB page size is 8% and this reduces to 5% for a 16 KB page size. For the waiting times, the DLOOP has 44% lower waiting times than DFTL for a 2 KB page size and reduces to 24% for a 16 KB page size. Even though the waiting times of FAST reduces for higher page size, it is still relatively high when compared to DLOOP. Coming to standard deviation plots we see that for the first trace which majority of read requests, the deviation improves for all algorithms as page size increases. DLOOP has better distribution when compared to DFTL and FAST. With 2 KB page size we see that DLOOP has a value of 9.4 compared to 13.7 for DFTL and 11.5 for FAST. As we move towards to higher page sizes we see that DLOOP’s distribution improves more than DFTL and FAST. For the 16 KB page size the value for DLOOP, DFTL and FAST is 5.9, 11 and 9.1 respectively. DLOOP does equally better for all three page sizes. We see similar results for trace with majority of write requests. In general, the distribution is a bit more uneven when compared to earlier distribution. As the page size increases all the three algorithms DLOOP, DFTL and FAST improve distribution of requests. The difference between DLOOP and DFTL for 2 KB page size was 4.3 while difference with FAST was 2.1, but as page size increased DLOOP and DFTL improved more than FAST and the difference for a 16 KB page size between DLOOP and DFTL is 4.7 and 3.7 for FAST. For the traces with majority of sequential requests and random requests, the distribution follows the same pattern. The DLOOP is better than DFTL and FAST by 4.3 and 2.1 respectively for a 2 KB page size. The difference stays almost the same for higher page size.

For the final set of experiments, we vary the number of planes per die to see how this effects the performance of the three algorithms. The SSD size will remain the same selected as 8 GB to isolate the changes. Since the number of planes per die change, this would
actually result in different number of dies/packages for the same SSD size. We try to
examine how the performance changes with synthetic traces and can be seen in Figure 5.8.
For the first trace which has a majority of read requests, we see that the performance of
DLOOP decreases as the number of planes per die increases. The performance of DFTL also
decreases very slightly. DLOOP which started better than DFTL for 2 plane per die has
become almost similar to DFTL for 16 plane per die. FAST didn’t see a lot of effect in
changing the number of planes per die. For the trace with majority writes we see almost
similar results. The performance of DLOOP starts to worsen as the planes per die increases
from 2 to 4 to 8 and is effected the most for 16 planes per die. DFTL which starts much
higher than DLOOP is close to DLOOP in 16 plane per die case. The gap has lowered from
73 % to less than 2%. FAST on the other than which is worse than 90% under all cases
doesn’t see change. For the next two traces we see that the performance of DLOOP and
DFTL is very close. Also, the performance of DLOOP deteriorates very slightly as the
number of planes increases. For next two traces which have a majority of sequential requests
and random requests, the improvement of mean response time of DLOOP over DFTL was
between 4 – 7 %. FAST had very high response times. The reason for the dip in performance
when the planes per die increase is because of contention of main channel. The more the
contention the more the request has to wait before it is served. Coming to the waiting time,
for the trace with majority of reads we see that the waiting time increases as the planes
increases as seen from mean response time, but the major hit is when the planes per die
increases from 8 to 16. DLOOP started 80% better than DFTL at 2 planes per die but is just
3% better for 16 planes per die. DFTL’s performance hasn’t changed much as the planes per
die increases and neither has FAST’s which has a high waiting time. For the next trace which
is the one with the write requests, we see that performance deteriorates as planes per die
increases and is worst at 16 plane per die for DLOOP. DFTL has little change in its waiting
time and so does FAST. For the next two traces which have majority sequential and random
trace in it, we see similar results. Their waiting time decreases smoothly as planes per die
increases unlike previous two traces. The performance difference starts with 40% between
DLOOP and DFTL for 2 plane per die and decreases to around 10% for 16 plane per die.
FAST is much worse over 90%. Coming to the standard deviation plot we don’t see any
change in the distribution as we increase the number of planes per die, since the total number
Figure 5.8. Performance impact of planes per die on three schemes for synthetic traces.
of planes for a given size don’t change. For all the traces we see that DLOOP has a better distribution than DFTL and FAST. For the trace with majority read requests the DLOOP has a lead of 5 over DLOOP and 3 over FAST in terms of log of standard deviation. For majority of write requests we see that the distribution has increased for all the three algorithms when compared to earlier trace. The difference between them is however the same. For the following traces of sequential requests and random requests we see similar distribution. DLOOP has the best distribution with a value of around 9.5, while DFTL and FAST have values 13.8 and 11.8 respectively.
CHAPTER 6

CONCLUSION AND FUTURE WORK

In this thesis, we proposed a new scheme to do address translation in the FTL. The basic idea was to maximize interleaving capacity of the SSD by parallelizing operations between individual planes of the SSD. As the SSDs start getting full, the write operations are written to the log/extra blocks which invokes the process of garbage collection frequently. This usually hinders the interleaving process. Also, interleaving is best applied for requests which are sequential. Random requests can hardly be interleaved. With all these problems in mind, we propose a scheme which basically promotes intra-plane transfers by constraining the allocation pool of extra blocks for update operations. This way we make sure that for all update operations have intra-plane transfers. This helps in two ways. The first one is that, data which has to be programmed doesn’t have to be transferred into controller buffers. It can be directly be copied in the new block of the same plane. This means that the external serial channel will also be available to handling other requests thereby increasing the availability. Also, since extra blocks are spread across every plane, the requests can be spread out more evenly which will reduce the waiting time of each individual request.

In my current thesis, we assumed that the request distribution across all planes will be even and so the extra blocks are spread out evenly across all the planes in the dies. However, this is not always true. We know that there are studies which classify data in hot blocks and cold blocks. Since data in hot blocks is updated more frequently, it would make more sense to have more extra blocks dedicated to such hot blocks to reduce the frequency of garbage collection. If we could classify the hotness of data on a plane-wise basis, more extra blocks would be assigned to such hot areas. The scheme used in this thesis was pretty static in which the extra blocks were distributed equally across planes. We plan to implement a dynamic scheme in which we change the number of extra blocks in a plane depending on the update frequency of data in a plane. This will also help in recycling the blocks more efficiently and reduce the garbage collection overhead.
BIBLIOGRAPHY


C. Dirik and B. Jacob. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization. Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA’09), Austin, TX, 2009. ACM.


