Performance Improvement of Block Based NAND Flash Translation Layer

Siddharth Choudhuri
Center for Embedded Computer Systems
University of California, Irvine, USA
sid@cecs.uci.edu

Tony Givargis
Center for Embedded Computer Systems
University of California, Irvine, USA
givargis@uci.edu

ABSTRACT

With growing capacities of flash memories, an efficient layer to manage read and write access to flash is required. NFTL is a widely used block based flash translation layer designed to manage NAND flash memories. NFTL is designed to achieve fast write times at the expense of slower read times. While traditionally, it is assumed that the read traffic to secondary storage is insignificant, as reads are cached, we show that this need not be true for NAND flash based storage due to garbage collection and reclamation processes. In this work, we present two independent techniques that extend NFTL and improve the read throughput in particular. The techniques presented add a minimal amount of RAM overhead to a flash controller, while providing, on an average, a 22.9% improvement in page read times and a 2.6% improvements in page write times on a set of file system and rigorous synthetic benchmarks. The techniques presented are well suited for flash controllers that are typically space constrained and have minimal processing power.

Categories and Subjsect Descriptors: *D*.4.2 Operating Systems: Storage Management − *Secondary Storage*
General Terms: Design, Management, Performance
Keywords: NAND Flash, Block mapping, Storage

1. INTRODUCTION

The use of flash memory as a non-volatile storage medium is on the rise. The characteristics of flash memory such as low power, shock resistance, lightweight, small form factor and absence of mechanical parts has long been recognized as a storage alternative for mobile embedded systems [6]. There are two kinds of flash memories - NOR flash and NAND flash. NOR flash is processor addressable and commonly used for small amounts of code storage. NAND flash, on the other hand, is mostly used for data storage and scales from megabytes to gigabytes in terms of storage capacity.

NAND flash of varying sizes can be found in devices such as compact flash cards, USB storage devices, mp3 players, and many more. The proliferation of embedded and mobile devices along with increasing demand for storage in such systems has contributed to a rise in NAND flash storage capacities. According to the Gartner group estimates, the flash market grew from 1.56 billion in 2000 to 11.42 billion in 2005. This trend is expected to continue in the coming years [11].

Flash memories have certain properties that prevent them from being a direct replacement for conventional storage devices. Specifically, flash memories do not support in-place updates, i.e., an update (re-write) to a given location (known as a *page*) is not possible, unless a larger region (known as a *block*) is first erased. In order to overcome the lack of in-place updates, a hardware and/or software layer is added to the flash. This layer, along with the flash memory, mimics a secondary storage device. This layer is called the *flash translation layer*. The flash translation layer takes care of mapping a sector to a $\langle block, page \rangle$ on the NAND flash, thereby giving the file system a view of an in-place mass storage device (Figure 1a). For a large class of flash based devices,

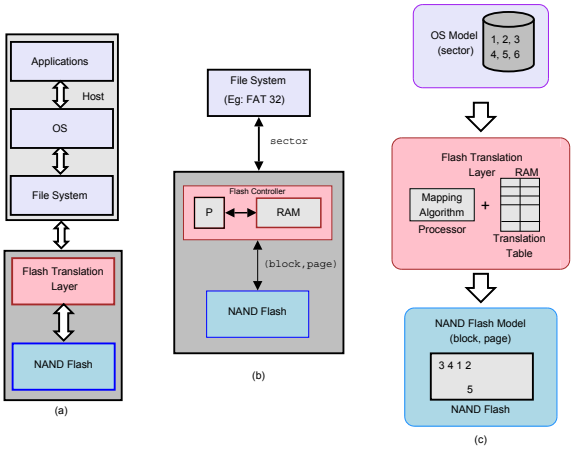

Figure 1: Flash Translation Layer

the flash translation layer is implemented as a controller in hardware. The controller consists of a low-end processor or microcontroller along with small amounts of RAM (Figure 1b). The controller is responsible for two important functions *(i)* Translating a read/write request from the file system (i.e., a *sector*) into a read/write operation on a specific $\langle block, page \rangle$ of the flash (Figure 1c), and *(ii)* Initiating garbage collection to erase dirty blocks and reclaim free space. The bulk of RAM serves as a placeholder for the translation data structure (e.g.: translation table). The size of RAM is driven by the flash capacity. However, with rising

CODES+ISSS'07, September 30–October 3, 2007, Salzburg, Austria.

flash capacities, scaling the RAM becomes unaffordable due to rise in cost, area and power consumption of the controller. Thus techniques are employed to manage the flash address translation using a limited amount of RAM. However, compromising on RAM size results in a performance trade-off in terms of read/write access time to the flash.

In this paper, we present two application agnostic techniques that improve the performance of NFTL, one of the most popular flash translation layers [4]. Our techniques require additional data structures to be stored in RAM. However, the cost of such data structures is minimal, deterministic and does not affect the space complexity with growing sizes of flash memory. The performance optimizations are a result of identifying NFTL bottlenecks by running and evaluating file system traces.

The rest of this paper is organized as follows. Section 2 introduces NAND flash. Section 3 illustrates NFTL and our proposed extensions. Section 4 provides details on benchmarks and traces. Results are provided in Section 5, followed by discussion of related work and conclusion.

2. BACKGROUND

A NAND flash consists of multiple *blocks*. Each block is further divided into multiple *pages*, which is the minimum unit of data transfer. Typically, the page size is 512 bytes, resembling the size of a sector in traditional hard disk drives. Blocks are usually either 8 KB or 16 KB in size (i.e., consisting of 16 or 32 pages). There are other variants of NAND flash known as large block NAND flash that have 2KB of page size. In this paper, we use small block NAND flash (i.e., page size is 512 bytes). The techniques presented, however can be applied to large block NAND flash. Each page also contains additional control bytes, also known as *Out Of Band* (OOB) data. Typically, the size of OOB data is 16 bytes. The primary purpose of the OOB data is to store Error Correcting Code (ECC). Most flash translation layers use the OOB data to store housekeeping information (such as inverse page table, status flags) along with the ECC. This information is used to reconstruct the translation table when the device is powered up. NAND flash has certain characteristics that impose restrictions on how it can be used: *(i)* a page, once written, cannot be re-written unless it is erased; *(ii)* an erase cannot be done on a per-page basis. The minimum unit of erase is a block; *(iii)* a block has a limited number of erase operations (typically, 100,000) after which it becomes unusable . Erase is costly i.e., it is slow.

The above properties of NAND flash result in out-of-place updates and garbage collection. A page P starts off in a *free* (erased) state. Once data is written into page P, its state changes to a *valid* state. However, an update (re-write) to page P is made out-of-place i.e., to another page Q that is in free state.

Garbage collection is the process of reclaiming space by erasing the blocks that contain obsolete pages. Note that not all pages in a block might be obsolete, hence garbage collection takes care of moving valid pages into a different block before erasing the whole block. Due to the out-of-place updates and garbage collection, it is not possible to have a fixed association between a sector and a page. Table 1 depicts NAND flash access time characteristics from datasheets of two leading manufacturers [1, 2]. One of the striking characteristics of NAND flash is the disparity between erase, write and read times.

Table 1: NAND Flash Specifications

Characteristics	Toshiba 16MB	Samsung 16MB
Block size	16384 (bytes)	16384 (bytes)
Page size	512 (bytes)	512 (bytes)
OOB size	16 (bytes)	16 (bytes)
Read Page	52 (usec)	36 (usec)
Read OOB	26 (usec)	10 (usec)
Write Page	200 (usec)	200 (usec)
Write OOB	200 (usec)	200 (usec)
Erase	2000 (usec)	2000 (usec)

The mapping from sector to ⟨block, page⟩ is done by the flash translation layer. This translation is mostly done at a page or a block granularity. One of the early flash translation layers known as the FTL was proposed by [3]. FTL is a page based translation layer. In FTL, given a sector s, the translation table entry $T[s]$ contains the corresponding block, page pair $\langle b_0, p_0 \rangle$. Furthermore, the OOB data area of $\langle b_0, p_0 \rangle$ contains the sector number s (inverse map) and a flag indicating the valid status of the page p_0. An update to sector s results in finding the next free block, page pair $\langle b_1, p_1 \rangle$. Furthermore, the entry $T[s]$ is updated to $\langle b_1, p_1 \rangle$; the OOB data area of $\langle b_0, p_0 \rangle$ is marked as obsolete; the sector number s and valid flag are written into the OOB data area of $\langle b_1, p_1 \rangle$. Garbage collection takes care of reclaiming obsolete blocks and modifying the translation table for pages that are moved in the process of garbage collection. The number of entries in a translation table is equal to the total number of pages in flash. This scheme, though simple and efficient, does not scale well with the size of flash. For instance, the translation table size for 1GB flash with 512 bytes page size and 8 bytes per translation table would be 16MB ($(1GB/512) \times 8$). As flash controllers are resource constrained, FTL is not a feasible option.

3. TECHNICAL APPROACH

Page based mapping (e.g., FTL) is a fine-grained approach that is efficient but requires a large amount of memory. Block based mapping is a coarse-grained approach which is less efficient compared to the page based approach, but consumes less space, thereby presenting a viable option for resource constrained flash controllers.

3.1 Preliminaries

A well known block based translation layer, called NFTL, was proposed by [4]. In NFTL, a sector (also known as *logical block address*) is divided into a *virtual block address* (most significant bits) and a *page offset* (least significant bits). A block based mapping table maps the virtual block address into a physical block. This physical block is also known as the *primary block*. The first write to a sector is always done to a sector in the primary block. Subsequent updates to the same sector are made on another physical block, also known as the *replacement block*. Furthermore, an update to any page in the primary block is made to a new page in the replacement block. Each primary block is associated with one replacement block and writes to the replacement block are sequential starting with page 0. In case of reads, the replacement block is read backwards. If a page corresponding to the desired sector is not found in the replacement block, the primary block is searched. A survey of data structures (including the ones used in block and page based mapping) and garbage collection algorithms can be found in [7].

The following example illustrates NFTL in more detail. Consider a flash made of 8 blocks and 4 pages per block. Thus, the block address is 3 bits and the offset is 2 bits.

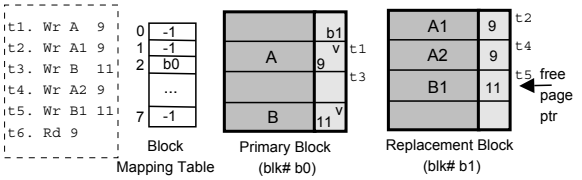

Figure 2: NFTL

A sequence of read and write operations ($t1 \ldots t6$) is shown in Figure 2. *Request* $t1$ `Wr A 9` implies a write to sector 9 (binary - 01001) with the data to be written being 'A'. The virtual block address is therefore {010} and the page offset is {01}. A free physical block (block number $b0$) is found and designated as the primary block for virtual block {010} (Figure 2 Block Mapping Table). Further, data 'A' is written to block $b0$ page 1. The sector number '9' is written in the OOB data area and it is marked as valid. *Request* $t2$: Since a rewrite to block 0, page 1 is not possible, another free physical block (block number $b1$) is found and designated as the replacement block for virtual block {010}. The data 'A1' is written to the first available free page in the replacement block, i.e., page 0. The replacement block information is stored in a pre-defined header area in the primary block namely, OOB data area 0. *Request* $t3$: The OOB data area corresponding to page offset {11} is read to see if it already has any valid data (in which case the write would be directed to the replacement block). *Request* $t4$, $t5$ being updates, are written to the replacement block. *Request* $t6$ is a read request to sector 9 (virtual block {010}). The mapping table directs the request to the physical block $b0$. Reading the header of block $b0$ reveals the presence of a replacement block. The most recent (valid) copy of the data can be found by reading the replacement block backwards starting from the "free page ptr". If there is no match found in the replacement block, the page exists in the primary block.

Over a period of time a replacement block might get full. In this case, a new physical block is chosen and valid data from the primary block and the replacement block are merged into the new block. The primary block and the replacement blocks are then erased. The new block becomes the primary block and the mapping table entry is updated to reflect this fact. This process of merging a physical block and its corresponding replacement block to create a new block is known as *folding*. NFTL use can lead to a situation when there are no free blocks left. A *Garbage Collection* (GC) process is initiated in such situations to initiate folding across all the blocks of flash. In summary, NFTL is optimized to do writes in a constant, shortest possible time, at the expense of reads that can take as long as searching the length replacement block OOB data in the worst case. Such a design is intentional due to the wide gap between read and write times. The techniques presented in this paper aim at reducing OOB data reads, thereby contributing to improved read and write throughput.

Note that in the above example there are two important pieces of meta-data information that are accessed frequently. The first is a frequent check to ascertain if a given page has valid data. This is a boolean information, thus having an in-RAM copy of a page's valid status can avoid OOB data reads. The second information that is frequently required is to ascertain a replacement block information that corresponds to a primary block. Our first technique - the *lookup table* maintains an in-RAM data structure for fast lookup of meta-data information. This is followed by our second technique, in which we exploit temporal locality to cache most recently accessed pages. On a cache hit, this technique leads to a constant time mapping from a sector to a $\langle block, page \rangle$.

3.2 Technique 1 - Lookup Table

In our first technique, we introduce two in-RAM data structures - *rep-block table* and *page-status bitmap*. The rep-block table accelerates the process of finding out the replacement block corresponding to a given primary block i.e., if it exists. The page-status bitmap accelerates the check to ascertain if a given page has valid data in it.

The rep-block table provides fast access to replacement block information. The rep-block table is indexed by the virtual block address. For a given virtual block, an entry in rep-block is the physical address of a replacement block if it exists. The rep-block table saves OOB read overhead by providing a faster in-RAM lookup. The rep-block improves NFTL by avoiding flash OOB reads in the following cases: *(i)* every page read results in OOB read in order to check the existence of a replacement block; *(ii)* a rewrite to a page requires the physical address of replacement block; *(iii)* accelerates the GC process by providing the replacement block address for every primary block that needs to be folded.

The page-status bitmap is a per page status indicator indexed by the page offset. For a given page offset, a 1 indicates that the corresponding page has been written at least once and a 0 indicates that the corresponding page has never been written (i.e., the page-status bitmap is a copy of per page valid flag). Note that this information can avoid OOB reads because: *(i)* every write request checks OOB to determine if the request should go to the primary block or the replacement block and *(ii)* every read request checks the OOB to determine the possibility of an illegal read request. The page-status bitmap accelerates both writes and reads.

The space overhead of the rep-block table is the same as the space requirements of the translation table. The space requirements for page-status bitmap is minimal - a bit for every page. Thus, for a given flash of size S, block size B and page size P, the rep-block table has (S/B) entries. The page-status bitmap requires (S/P) bits. For instance, a 1GB flash with 512 bytes page size, 8KB block size and 4 bytes per entry would require $\left(\left(2^{30}/2^{13}\right) \times 4\right) = 512KB$ of memory and the page-status bitmap would require $\left(2^{30}/\left(2^9 \times 8\right)\right) = 256KB$ of memory.

3.3 Technique 2 - Page Cache

The page cache is a configurable cache that holds the physical address of ⟨block, page⟩ of the most recently written sectors. This mapping can be used to locate a ⟨block, page⟩ directly, instead of reading OOB data, which, in worst case, could lead to OOB reads equal to the number of pages per block (during a read request). However, unlike the lookup table approach, the page cache relies on the temporal locality. Similar to the lookup table approach, the page cache entries are also copies of information that is already present in the OOB data area. Thus, there is no need to flush this information to the flash. The page cache can improve the mapping time from sector to block/page in the following cases: *(*i) during a sector read requests and *(ii)* during reads initiated by a fold operation. Note that the gains in both cases rely on temporal locality. Table 2 summarizes where

each data structure i.e., rep-block, page-bitmap and page cache can help improve the performance.

Table 2: Improvement Scope of Techniques

Technique	Read	Write	Rewrite	Fold	GC
rep-block	●	❍	●	❍	●
page bitmap	●	◗	●	❍	❍
page cache	◗	❍	❍	◗	❍

● Always ❍ Never ◗ May

4. EXPERIMENTS

Figure 3 depicts our experimental setup. A USB flash disk, formatted as a FAT 32 file system was connected to a PC running Linux kernel 2.6.16. The kernel was modified to sniff low level file read/write requests being issued to the USB flash and log the requests (sector, read/write operation) into a `/proc/flash` entry. A series of benchmarks were run to generate trace data. The trace, along with input parameters (flash characteristics - block size, page size, etc) is fed to our simulation framework. The simulation framework has two parts - a NAND flash simulation module providing basic read, write and erase capabilities. A desired flash translation layer implementation can be run on top of the NAND flash simulator.

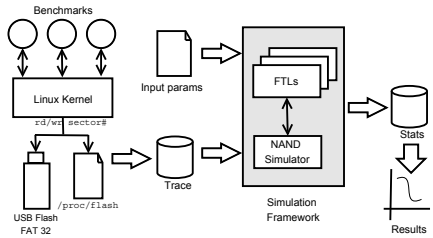

Figure 3: Experimental Setup

4.1 Benchmark Characteristics

As our work is not intended to be application specific, we chose the following benchmarks representing a variety of workloads. The *Andrew* benchmark [8] consists of five phases involving creating files, copying files, searching files, reading every byte of a file and compiling source files. The *Postmark* benchmark measures performance of file systems running networked applications like e-mail, news server and e-commerce [10]. Two versions of postmark were run. The short version created files in the range of 500 bytes to 9.77 KB and the longer version *Postmark - long*, created files in the range of 500 byte to 3 MB. The postmark benchmark has three phases involving creating files, running 500 transactions on files and deleting files. The *iozone* benchmark [12] is a well known synthetic benchmark. We ran iozone to do read, write, rewrite, reread, random read, random write, backward read, record rewrite (i.e., writing and rewriting to a particular hotspot within a file) and stride read. The file sizes ranged from 64KB to 32MB in strides of 2× (i.e., 64, 128, 512 ... 32768). Table 3 depicts the number of read and write requests in each benchmark.

Table 3: Benchmark Characteristics

Benchmark	Reads	Writes	Read %	Sect Range
Format	15	36	29.4	0 - 533
Andrew	2	3126	0.06	1 - 2867
Postmark	412	21153	1.9	2 - 10000
Postmark long	23694	1238135	0.9	1 - 65543
IOzone	3713	3089393	0.12	1 - 65588

4.2 File System Trace

A set of benchmarks were run in sequence to generate a file system trace. The first trace, called the *combo* trace was generated by running the following sequence: *format flash* → *andrew setup* → *andrew run* → *postmark setup* → *postmark run* → *postmark long run* → *iozone setup* → *iozone run*. The combo trace resulted in access of sectors ranging from 0 to 65588. A flash size of 34 MB [i.e., $(34MB/512bytes) = 69632$ sectors] was chosen so that it leads to a high utilization of 94.19% [$(65588/69632) \times 100$]. In order to study the effects of an underutilized flash, our second trace, *AtoPM* was derived. The AtoPM trace was derived by running benchmarks starting from *format flash* through *postmark run*. The flash utilization of the AtoPM trace is 23% with sectors ranging from 0 through 10000. We use the following notation to represent traces: *C8K* - combo trace, 8KB block size, *C16K* - combo trace, 16KB block size, *AP8K* - AtoPM trace, 8KB block size, *AP16K* - AtoPM trace, 16KB block size.

4.3 Disparity in Flash Read-Write Ratios

It has been well known that file system accesses to disk are write dominated as reads are cached in the main memory [14]. This is also reflected in Table 3 which shows a low read percentage for individual benchmarks. However, in the case of flash, the number of page reads issued to the flash (hardware) are typically greater than the number of reads issued by the file system. These additional page read operations are due to the folding and garbage processes triggered by the file system write requests. For example, a write to a replacement block that has no free page available leads to a fold operation. The fold operation results in several page reads. Thus, flash disk activities, unlike traditional disks, are not essentially write dominated. Table 4 shows the trace read-write characteristics for NFTL. As can be seen in Table 4, there is a disparity in the percentage of read requests issued by the file system versus those issued to the actual flash.

Table 4: Trace Characteristics

Trace	File Sys. Reads	File Sys. Writes	File Sys. Read %	Flash Reads	Flash Writes	Flash Read%
C8K	27731	4351879	0.63	4904441	9228589	34
C16K	27731	4351879	0.63	5327138	9651286	35
AP8K	421	24351	1.7	6772	30702	18
AP16K	421	24351	1.7	5503	29433	15

5. RESULTS

We present our experimental results and analysis in this section. Table 5 summarizes the performance improvements over NFTL due to each of our techniques. The metric for comparison in Table 5 is the average read time per sector and the average write time per sector. For each trace in Table 5, the number of entries in the page cache is set to a fixed number such that the in-RAM page cache size equals to the in-RAM lookup table size. This gives a common ground to compare the two techniques. The average write time calculation includes the time due to fold, garbage collection and the time spent in OOB reads and OOB writes. Note that the fold and garbage collection times are included in calculating write times as these two operations are always a consequence of a write request. The average read time includes time spent in OOB reads due to possible searches in the replacement block. The following analysis is made out of Table 5:

(1) The performance gain is device specific. For the same trace, the performance benefits differ due to differences in page read and OOB read times (Table 1).

(2) The performance gains from page cache is consistently less than the gains from lookup table approach. We believe

Table 5: Performance Improvements - Average Read/Write Access Times

					NFTL		Lookup Table				Page Cache				
Trace	Util	Blk KB	Folds	GC	Write (usec)	Read (usec)	Write (usec)	Gain %	Read (usec)	Gain %	Write (usec)	Gain %	Read (usec)	Gain %	
Combo	94%	8	309282	644	1230.51	64.44	1203.96	2.16	44.37	31.15	1226.85	0.30	56.95	11.62	Samsung
		16	167556	762	1134.34	74.27	1102.50	2.81	54.19	27.03	1130.69	0.32	60.91	17.99	
AtoPM	23%	8	782	0	676.61	58.71	651.79	3.67	38.71	34.07	676.38	0.03	58.00	1.21	
		16	386	0	584.96	63.03	564.15	3.56	43.03	31.73	584.58	0.07	60.89	3.39	
Combo	94%	8	309282	644	1317.05	125.89	1231.96	6.46	73.71	41.44	1307.52	0.72	106.43	15.46	Toshiba
		16	167556	762	1219.48	151.43	1149.01	5.78	99.25	34.46	1210.00	0.78	116.77	22.89	
AtoPM	23%	8	782	0	721.84	111.04	671.36	6.99	59.04	46.83	721.25	0.08	109.19	1.67	
		16	386	0	629.04	122.28	583.09	7.31	70.28	42.53	628.05	0.16	116.72	4.55	

that this is partly due to the fact that page cache relies heavily on the temporal locality and the fact that our page cache is not based on any sophisticated algorithms (e.g., LRU based eviction). A more sophisticated algorithm may give better gains at the cost of implementation complexity and additional space overhead. Our page cache scheme, being simple, is well suited for flash controllers that are based on microcontrollers or low-end processors with limited RAM. The average improvements are: lookup table approach 4.8% for writes, 36.1% for reads; page cache approach 0.30% for writes, 9.8% for reads.

(3) The performance gain due to writes is minimal compared to the performance gain due to reads. The following reason explains this trend - NFTL is optimized for writes, taking a fixed amount of time (i.e., page data write + OOB data write) for writes that fit either a primary block or a replacement block. However, a write may also lead to folding and/or garbage collection due to lack of free pages. Both block erase and page writes take time that is an order of magnitude more than other operations like page read or OOB data read (Table 1). Thus, the average write time is dominated by page data write time and block erase time. Figure 4 shows the time spent in different activities in case of a write request (normalized to 100%). Notice that in Figure 4, the majority of time (over 80%) is spent on erase and page data writes. Therefore the only scope of improvements is on the rest 20% (shown as "others" in Figure 4), leading to gains that are considerably less compared to gains from read requests.

(4) The lookup table approach yields better read performance for smaller block sizes. In the lookup table, the gain of finding the physical address of replacement block is amortized by the number of OOB data reads incurred during a backwards search to locate a required page. Therefore, the advantages of the lookup table are smaller in a flash with larger block size, since a larger block holds more replacement pages compared to a same size flash with smaller block size. The page cache approach, however, shows a reverse trend. The gains in the case of larger blocks size is higher compared to a smaller block size. This can be attributed to the fact that in the case of larger block size, a page lives longer in the replacement block before a fold operation erases it. This yields better temporal locality. Additionally, the overhead of searching a larger block is higher and a cache hit in such cases avoids numerous expensive OOB data lookups, resulting in higher gain percentages.

The performance improvements from the lookup table and the page cache are due to avoiding unnecessary OOB data lookup in traditional NFTL. Table 6 provides a device independent look at the number of OOB data reads that were avoided for each technique. Though a significant percentage of OOB lookups are avoided, the same percentages are not reflected in Table 5. The reason is due to the fact that the OOB data read access takes less time compared to other operations (Table 1). Thus the overall gain percentage in Table 5 is lesser compared to Table 6, due to a smaller contribution of the OOB read time to the overall page read time. Figure

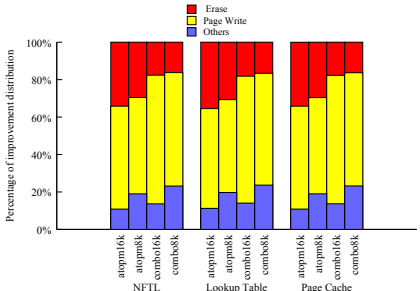

Figure 4: Distribution of Write Times

Table 6: Device Independent Gains

	NFTL	Lookup Table		Page Cache	
	Reads	Reads	Gain	Reads	Gain
C8K	18739977	6820423	63.00%	17124427	8.60%
C16K	17963617	7403132	58.78%	16340587	9.03%
AP8K	63445	23547	62.88%	62862	0.91%
AP16K	63149	24038	61.93%	62133	1.60%

5 shows the variation in read times due to varying cache sizes on the x-axis, starting with no page cache. The four page cache sizes are chosen to be 50%, 100%, 200% and 400% of the size of the lookup table as a base reference. Note that, after a certain threshold, increasing the page cache size results in a point of diminishing returns. Thus, increasing the number of entries does not necessarily improve the read performance after a certain threshold. From our simu-

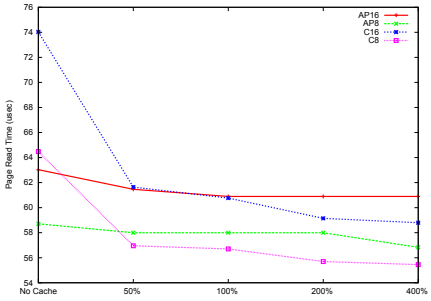

Figure 5: Page Cache Size vs. Read Time

lation of a 34 MB NAND flash, the additional RAM space requirements due to lookup table is the sum of sizes required for the rep-block and page cache bitmap - 26112 bytes for 8KB block size and 17408 bytes for a 16KB block size flash.

The size of the page cache can be adjusted based on need. For our simulations, we set the page cache to be the same size as the lookup table. Considering that the mapping table itself requires 17408 bytes for 8KB flash and 8704 bytes for 16 KB flash, our approaches provide a viable option at the cost of fixed RAM space requirements. The approaches presented are based on caching OOB data and exploiting locality. Thus, the issue of data inconsistency due to abrupt shutdowns is not an issue.

Scalability: In order to study the affect of our approach on large NAND flash memory, we ran iozone benchmark on a 1GB flash with 95% utilization (details omitted for lack of space). iobench was configured to do only reads and writes (to simulate file transfers). The average improvement in read was 31.6% and write 5.19%. Similar to the traces presented in this Table 5, the gains due to reads are higher compared to writes. The only noticeable difference in the 1GB configuration is that the ratio of reads is close to 49%. The reason behind this could be the large amounts of folds in the 1GB flash and that iozone was configured to do only reads and writes. **Reliability:** The rep-block table and the page-status bitmap are updated along with updates to the actual OOB data. Note that, both the rep-block and the page-status bitmaps are copies of information that is written to the OOB data area. The page cache is constructed during write operation. Thus, there is no need to update this information to the flash i.e., any abrupt shutdown or power loss will not lead to inconsistency in data structures. **Overhead:** The techniques presented lend themselves easily to small microcontrollers that are used in flash and also the data structures can be easily reconstructed at device startup along with the translation table.

6. RELATED WORK

Space efficient schemes for flash translation layer has been a topic of research. In [5], the authors propose a sector to page translation layer based on buddy system [13]. The allocation and de-allocation of pages is managed by binary trees and two linked lists that keep track of free and occupied pages. At the node of the binary tree is a data structure that stores information in units of a "physical cluster" which is a set of contiguous sectors. The scheme relies on write requests being sequential in nature. The drawback of this approach is that the RAM space requirements is non-deterministic and can grow out of proportions depending on nature of requests. In order to keep a bound on the RAM space requirements, the data structures are flushed to the flash when the RAM size grows beyond a certain threshold − thus making a part of flash act like a swap area for the in-RAM data structures. This can lead to variable access time. The RAM space requirements for a 16 GB flash in [5] is shown as 17 MB. In case of NFTL, the mapping tables require 4 MB and 4.12 MB for a lookup table. In CNFTL [15], the authors propose a mapping scheme based on multiple levels of indirection. A physical block is broken down into segments, frames and pages, a frame being the minimum unit of read and write. The drawbacks of this approach are *(i)* minimum unit of write is a frame which could result in internal fragmentation; *(ii)* a write request can be mapped only to sectors belonging to a segment. This could lpagesead to frequent garbage collection operations and, *(iii)* neither writes or reads are optimized, a write request could lead to linear search of the OOB data areas of every frame in a given segment. Moreover, the number of OOB data reads increases sharply with a rise in frames per segment. For a 34 MB flash configuration (similar to our experiments), the RAM space requirements of CNFTL to store only the mapping tables is 13872 bytes for a 16 KB block size (compared to 8704 bytes in NFTL which provides a faster write time). In [16], the authors propose a two level (both page and block) adaptive scheme to speed up read accesses. This scheme has memory requirements for maintaining hash tables, doubly linked LRU lists in addition to the mapping tables. The approach proposed in this work delays the fold operations of NFTL, hence, it is not well suited for flash memories that have a high utilization. In our work we show results under conditions of over 90% utilization. In [9] the authors propose a space efficient flash translation layer that is application specific, tuned for multimedia workloads.

7. CONCLUSION

NFTL is designed for fast writes and reads of varying speeds. In this work we presented a system designer with two techniques that extend NFTL by improving the read and write throughput. The *lookup table* extension incurs a fixed RAM overhead, whereas the *page cache* extension presents a system designer with a configurable option. The RAM space requirements of these extensions are minimal and deterministic. The techniques presented are application agnostic and lend themselves easily to be implemented on flash controllers that are typically both space constrained and have minimal processing power. We also show that unlike traditional disk based storage, in NAND flash based storage, both reads and writes contribute to the flash data traffic.

8. REFERENCES

[1] Samsung 16M x 8 bit NAND flash memory, K9F2808U0B. `http://www.samsung.com`, 2006.

[2] Toshiba 128 MBIT CMOS NAND EEPROM TC58DVM72A1FT00. `http://www.toshiba.com`, 2006.

[3] A. Ban. Flash file system. *US Patent 5,404,485*, Apr 4, 1995.

[4] A. Ban. Flash file system optimized for page-mode flash technologies. *US Patent 5,937,425*, Aug 10, 1999.

[5] L.-P. Chang and T.-W. Kuo. Efficient management for large-scale flash-memory storage systems with resource conservation. *Trans. Storage*, 1(4):381–418, 2005.

[6] F. Douglis, R. Caceres, M. F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber. Storage alternatives for mobile computers. In *Operating Systems Design and Implementation*, pages 25–37, 1994.

[7] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2):138–163, 2005.

[8] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[9] H. Jo and et al. FAB: Flash-aware buffer management policy for portable media players. *IEEE Trans on Consumer Electronics*, 52(2):485 –493, May 2006.

[10] J. Katcher. Postmark: A new file system benchmark. Technical report, Network Appliance Inc, TR 3022, 1997.

[11] G. Lawton. Improved flash memory grows in popularity. *Computer*, 39(1):16–18, 2006.

[12] W. Norcutt. IOZONE benchmark program. `http://www.iozone.org`.

[13] J. L. Peterson and T. A. Norman. Buddy systems. *Commun. ACM*, 20(6):421–431, 1977.

[14] C. Ruemmler and J. Wilkes. UNIX disk access patterns. In *Usenix Conference*, pages 405–420, Winter 1993.

[15] Y.-L. Tsai, J.-W. Hsieh, and T.-W. Kuo. Configurable nand flash translation layer. *SUTC*, 1:118–127, 2006.

[16] C.-H. Wu and T.-W. Kuo. An adaptive two-level management for the flash translation layer in embedded systems. *ICCAD*, pages 601–606, 2006.