An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems*

Li-Pin Chang and Tei-Wei Kuo
{d6526009,ktw}@csie.ntu.edu.tw
Department of Computer Science and Information Engineering
National Taiwan University, Taipei, Taiwan, ROC 106

Abstract

Flash memory is now a critical component in building embedded or portable devices because of its non-volatile, shock-resistant, and power-economic nature. With the very different characteristics of flash memory, mechanisms proposed for many block-oriented storage media cannot be directly applied to flash memory. Distinct from the past work, we propose an adaptive striping architecture to significantly boost the system performance. The capability of the proposed mechanisms and architecture is demonstrated over realistic prototypes and workloads.

1 Introduction

Flash memory is now a critical component in building embedded or portable devices because of its non-volatile, shock-resistant, and power-economic nature. Similar to many storage media, flash memory, especially NAND flash, is treated as ordinary block-oriented storage media, and file systems are built over flash memory in many cases [9, 10]. Although it has been a very convenient way for engineers in building up application systems over a flash-memory-based file system, the inherent characteristics of flash memory also introduce various unexpected system behavior and overheads to engineers and users. Engineers or users may suffer from degraded system performance significantly and unpredictably after a certain period of flash-memory usage. Such a phenomenon could be exaggerated because of the mismanagement of erasing activities over flash memory.

A flash memory chip is partitioned into blocks, where each block has a fixed number of pages, and each page is of a fixed size, e.g., 512B. The size of a page fits the size of a disk sector. Due to the hardware architecture, data on a flash are written in an unit of one page. No in-place update is allowed. Initially, all pages on flash memory is considered as "free". When a piece of data on a page is modified, the new version must be written to an available page somewhere. The pages store the old versions of the data are considered as "dead", while the page stores the newest version of data is considered as "live". After a certain period of time, the number of free pages would be low. As a result, the system must reclaim free pages for further writes. Because erasing is done in an unit of one block, the live pages (if they exist) in a recycled block must be copied to somewhere else. The block could then be erased. Such a process is called garbage collection. The concept of out-place update (and garbage collection) is very similar to that used in log-structured file-systems (LFS) [6] [1]. How to smartly do garbage collection may have a significant impact on the performance a flash-memory-based storage system. On the other hand, a block will be "worn-out" after a specified number of erasing cycles. The typical cycle limit is 1,000,000 under the current technology. A poor garbage collection policy could also quickly wear out a block and, thus, a flash memory chip. A strategy called "wear-leveling" with the intention to erase all blocks as evenly as possible should be adopted so that a longer lifetime of flash memory could be achieved.

On the other hand, flash memory is still relatively slow, compared to RAM. For example, the throughput in writing to a "clean" (un-written) NAND flash memory could reach 400 $\sim$ 500KB per second. However, it could be degraded by roughly 20% when garbage

*Supported in part by a research grant from the ROC National Science Council under Grants NSC90-2213-E-002-080

[1]Note that there are several different approaches in managing flash memory for storage systems. We must point out that NAND flash prefers the block device emulation approach because NAND flash is a block-oriented medium (Note that a NAND flash page fits a disk sector in size). There is an alternative approach which builds an LFS-like file system over flash memory without the block device emulation. We refer interested readers to [11] for more details.

collection happens. If there is significant locality in writes, the overheads of garbage collection could be much higher because of wear-leveling. The throughput could even deteriorate down to less than 50% of the original performance. The performance under the current flash memory technology might not satisfy applications which demand a higher write throughput.

The purpose of this paper is to investigate the performance issue of flash-memory storage systems with a striping architecture. The idea is to use I/O parallelism in speeding up a flash-memory storage system. As astute readers may notice, such an approach in I/O parallelism could not succeed without further investigation on garbage collection issues. In this paper, we shall propose an adaptive striping mechanism designed for flash-memory storage system with the consideration of garbage collection.

In the past few years, researchers [1, 5, 7] have started investigating the garbage collection problems of flash-memory storage systems with a major objective in reducing both the number of erasings and the number of live-page copyings. In particular, Kawaguchi and Nishioka [1] proposed a greedy policy and a cost-benefit policy. The greedy policy always recycles the block with has the largest number of dead pages. The cost-benefit policy assigns each block a value and runs a value-driven garbage collection policy. Kawaguchi and Nishioka observed that with the locality of the access pattern, garbage collection could be more efficient if live-hot data could be avoided during the recycling process, where hot data are data which are frequently updated. Chiang, et al. [7] improved the garbage collection performance by adopting a better hot-cold identification mechanism. Douglis, et al. [4] observed that the overhead of garbage collection could be significantly increased when the capacity utilization of flash memory is high, where the capacity utilization denotes the ratio of the number of live pages and the number of total pages. For example, when the capacity utilization was increased from 40% to 95%, the response time of write operations may drop by 30%, and the lifetime of a flash memory chip may be reduced up to a third. Chang and Kuo [3] investigated the performance guarantee of garbage collection for flash memory for hard real-time systems. A greedy-policy-based real-time garbage collection mechanism was proposed to provide a deterministic performance.

Distinct from the past work, we focus on the architecture of a flash-memory storage system. A joint striping and garbage-collection mechanism for a flash-memory storage system is proposed with the objective to significantly boost the performance. We propose to adopt a striping architecture to introduce I/O parallelism to flash-memory storage systems and an adaptive bank assignment policy to capture the characteristics of flash memory. We formulate a simple and effective garbage collection policy for striping-based flash-memory storage systems and investigate the effects of striping on garbage collection. The contributions of this paper are as follows:

- We propose a striping architecture to introduce I/O parallelism to flash-memory storage systems. Note that storage systems are usually the slowest systems for many embedded application systems.
- We propose an adaptive striping-aware bank assignment method to improve the performance of garbage collection.
- We investigate the tradeoff of striping and garbage collection to provide insight for system design.

The rest of this paper is organized as follows: In Section 2, the motivation of this paper is presented, and the architecture of a striping NAND-flash storage system is introduced. Section 3 presents our adaptive striping architecture. The proposed policy is evaluated by a series of experiments over realistic workloads in Section 4. Section 5 briefly introduces our policy for garbage collection. Section 6 is the conclusion.

2 Motivation and System Architecture

NAND flash is one kind of flash memory specially designed for block storage systems of embedded systems. Most flash-memory storage systems, such as *Smartmedia*^{TM}, and *CompactFlash*^{TM}, now adopt NAND flash. The operation model of NAND flash, in general, consists of two phases: setup and busy phases. For example, the first phase (called "setup" phase) of a write operation is for command setup and data transfer. The command, the address, and the data are written to proper registers of flash memory in order. The second phase (called "busy" phase) is for busy-waiting of the data being flushed into flash memory. The operation of reads is similar to that of writes, except that the sequence of data transfer and busy-waiting is inverted. The phases of an erase is as the same as those of a write, except that no data transfer is needed in the setup phase. The control sequence of read, write, and erase are illustrated in Figure 1(a).

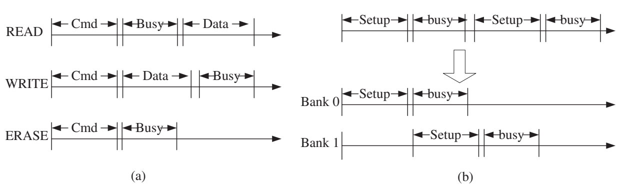


Figure 1: (a) Control Sequences of Read, Write, and Erase. (b) Parallelism for Multiple Banks.

Interval	Documented (μs)	Measured (μs)
E_{setup}	0.25	31
E_{busy}	2,000	1,850
W_{setup}	26.65	616
W_{busy}	200	303
R_{setup}	26.55	348
R_{busy}	10	Negligible

Table 1: NAND-Flash Performance (Samsung K9F6408U0A 8MB NAND flash memory)

The purpose of this paper is to investigate the performance issue of NAND-flash storage systems for embedded applications because of the popularity of NAND flash. While the setup phase of an operation is done by a driver and consumes CPU time, CPU is basically free during the busy phase. Because of the design of NAND flash, the busy phase of a write (/ an erase) usually takes a significant portion of time for each operation, compared to the elapsed time of the first phase. The performance characteristics of a typical NAND flash over an ISA bus is summarized in Table 1, where W_{setup} (/R_{setup} /E_{setup}) and W_{busy}(/ R_{busy} / E_{busy}) denote the setup time and the busy time of a write (/read/erase), respectively.

The motivation of this research is to investigate the parallelism of multiple NAND banks to improve the performance of flash-memory storage systems, where a bank is a flash unit that can operate independently (Note that a dedicated latch and a decode logic are needed for each independent bank.). When a bank operates at the busy phase, the system can switch immediately to another bank such that multiple banks can operate simultaneously, whenever needed, to increase the system performance, as shown in Figure 1(b). For example, a write request, which consists of 3 pages writes, needs 3*(606+303) = 2,727 μs to complete without striping. But it requires only 3*606+303 = 2,121 μs to complete if we have two independent banks. As a result, the response time is reduced by 23%. Note that E_{busy} is far more than E_{setup} over an ISA-based NAND flash. When a higher speed bus

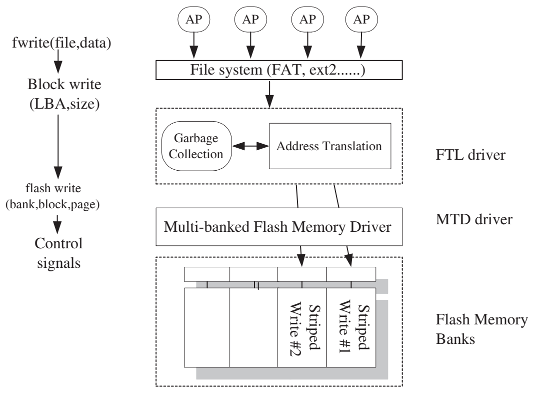


Figure 2: System Architecture.

other than ISA is adopted, W_{setup} and E_{setup} (and R_{setup}) could be much reduced, and we surmise that the performance of the system can be further improved under the idea of striping. As astute readers may notice, performance improvement will be much more significant if $W_{setup} < W_{busy}$, since a higher parallelism can be achieved.

An ordinary NAND-flash storage system consists of a NAND-flash bank, a Memory-Technology-Device (MTD) driver, and a Flash-Translation-Layer (FTL) driver, where a MTD driver provides functionality such as read, write, and erase. A FTL driver provides transparent access for file systems and user applications via block device emulation. When a striping NAND-flash storage system is considered, the MTD and FTL drivers must be re-designed accordingly to take advantage of multiple banks. The system architecture is illustrated in Figure 2. The modification of a MTD driver can be done straightforwardly: Basically, a MTD driver may no longer block and wait for the completion of any write or erase because several banks are now available. We shall discuss it again later. The major challenges of a striping architecture are on the FTL driver. In this paper, we will address this issue and propose our approaches for striping-aware garbage collection.

3 An Adaptive Striping Architecture

In this section, a multi-bank address translation scheme is presented, and then different striping policies for the proposed striping architecture are presented and discussed.

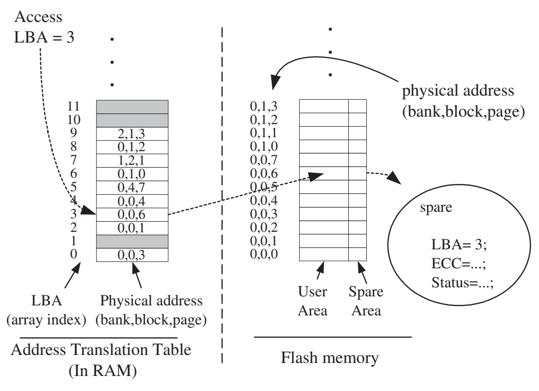


Figure 3: Address Translation Mechanism.

3.1 Multi-Bank Address Translation

The flash-memory storage system provides the file system a collection of logical blocks for reads/writes. Each logical block is identified by a unique logical block address (LBA). Note that each logical block is stored in a page of flash memory. Because of the adoption of out-place update and garbage collection, the physical location of a logical block might change from time to time. For the rest of this paper, when we use the term "logical block", it is for a logical block viewed by the file system. When we say "block", we mean the block of flash memory, where a block consists of a fixed number of pages, and each page can store one logical block.

In order to provide transparent data access, a dynamic address translation mechanism is adopted in the FTL driver. The dynamic address translation is usually accomplished by using an address translation table in main memory, e.g., [1, 5, 7, 9]. In our multi-bank storage system, each entry in the table is a triple (*bank_num*, *block_num*, *page_num*), indexed by LBA. Such a triple indicates that the corresponding logical block resides at Page *page_num* of Block *block_num* of Bank *bank_num*. In a typical NAND flash memory, a page consists of a user area and a spare area, where the user area is for the storage of a logical block, and the spare area stores the corresponding LBA, ECC, and other information. The status of a page can be either "live", "dead", or "free" (i.e., "available"). Whenever a write is processed, the FTL driver first finds a free page and then writes the written data and the corresponding LBA to the user area and the spare area, respectively. The pages store the old versions of data of the written logical block (if it exists) are considered as "dead". The address translation table is updated accordingly. Whenever a system is powered up, the address translation table is re-built by scanning the spare area of all pages. As an example shown in Figure 3, when a logical block with LBA 3 is accessed, the corresponding table entry (0,0,6) shows that the logical block resides at the 7th page (i.e., (6+1)th page) of the first block on the first bank.

3.2 Bank Assignment Policies

Three kinds of operations are supported on flash memory: read, write, and erase. Reads and erases do not need any bank assignment because they are already stored in specific locations. Writes would need a proper bank assignment policy to utilize the parallelism of multiple banks. When the FTL driver receives a write request, it will break the write into a number of page writes. There are basically two types of striping for write requests: static and dynamic striping:

Under the static striping, the bank number of each page write is derived based on the corresponding LBA of the page as follows:

Bank address = LBA % (number of banks)

The formula is close to the definition of RAID-0. Each sizable write is striped across banks "evenly". However, the static bank assignment policy could not actually provide even usages of banks, as shown in the experimental results in Section 4.2. As a result, a system adopts the static bank assignment policy might suffer from a large number of data copyings (and thus a degraded performance level) and different wearing-out time for banks because of the characteristics of flash. The phenomenon is caused by two reasons: (1) the locality of write requests, and (2) the uneven capacity utilization distribution among banks.

To explain the above phenomenon, we shall first define that an LBA is "hot" if it is frequently written; otherwise the LBA is "cold" (we may refer written data as hot data if their corresponding LBA's are hot for the rest of this paper). Obviously, hot data will be invalidated sooner than cold data do, and they usually left dead pages on their residing banks. Since a static bank assignment policy always dispatches write requests to their statically assigned banks, some particular banks might have many hot data on them. As a result, those banks would need to do a lot of garbage collecting. On the other hand, the banks which have a lot of cold data on them would have a high capacity utilization, since cold data would reside on their assigned banks for a longer period of time. Due to the uneven capacity distribution among banks, the performance of garbage collection on each bank may also vary since the performance highly depends on the capacity utilization [4, 6].

To resolve the above issue, we propose a dynamic bank assignment policy as follows: When a write request is received by the FTL driver, we propose to scatter page writes of the write request over banks which are idle and have free pages. The parallelism of multiple banks is achieved by switching over banks without having to wait for the completion of issued page-writes. The general mechanism is to choose an idle bank that has free pages to store the written data. One important guideline is to further achieve the "fairness" of bank usages by analyzing the attributes (hot or cold) of the written data: Before a page write is assigned a bank address, the attributes of the written data must be identified. We propose to write hot data to the bank that has the smallest erase-count (which is number of erases ever performed on the bank) for the consideration of wear-leveling, since hot data will contribute more live page copyings and erases to the bank. The strategy in writing hot data prevents hot data from clustering on some particular banks. On the other hand, cold data are written to the bank that has the lowest capacity utilization to achieve a more even capacity utilization over banks. The strategy in writing cold data intends to achieve a more even capacity utilization distribution since cold data will reside at their written locations for a longer period of time. Because flash memory management already adopts a dynamic address translation scheme, it is very easy and intuitive to implement a dynamic bank assignment policy in FTL. We shall explore the performance issues of the above two policies in Section 4.2.

3.3 Hot-Cold Identification

As we mentioned in the pervious section, we need to identify the attributes of the written data. The purpose of this section is to propose a hot-cold identification mechanism. Note that the hot-cold identification mechanism does not intend to capture the whole working set. Instead, the mechanism aims at analyzing if the requested LBA's are frequently written (updated).

The proposed hot-cold identification mechanism consists of two fixed-length LRU lists of LBA's, as shown in Figure 4. When the FTL driver receives a write request, the two-level list will examine each LBA associates with the write to determine the "hotness" of the written data: If a LBA already exists in the

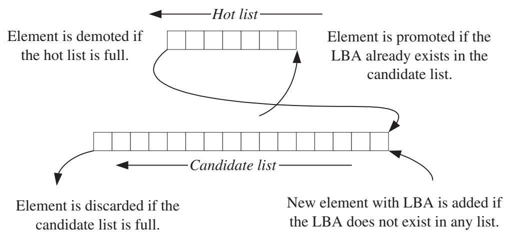

Hot list
Element is demoted if the hot list is full.
Element is promoted if the LBA already exists in the candidate list.
Candidate list
Element is discarded if the candidate list is full.
New element with LBA is added if the LBA does not exist in any list.

Figure 4: Two-Level LRU Lists.

first-level LRU list, i.e., the hot list, then the data is considered hot. If the LBA does not exist in the hot list, then the data is considered cold. The second-level list, called the candidate list, stores the LBA's of recently written data. If a recently written LBA is written again in a short period of time, the LBA is considered hot, and it is promoted from the candidate list to the hot list. If the hot list is already full (when a LBA is promoted from the candidate list to the hot list), then the last element of the hot list is demoted to the candidate list. If the promoted LBA already exists in the hot list, it will be moved to the head of the hot list. Under the two-level list mechanism, we only need to keep tracks of hot LBA's, which is a relatively small subset of all possible LBA's [2]. This mechanism is very efficient and the overhead is very low, compared with the mechanism proposed in [7] which always keeps the last access time of all LBA's.

4 Experimental Results

A multi-bank NAND-flash prototype was built to evaluate the proposed adaptive striping architecture. Since the performance of garbage collection highly depends on the locality of access pattern, we evaluated the prototype under three different access patterns: trace-driven, sequential, and random. The traces of the trace-driven access pattern were collected by emulating web-surfing applications over a portable device, and the characteristics of the traces are shown in Table 2. Sequential and random access patterns were generated by modifying the requested LBA's of the gathered traces. Other parameters (arrival time, request size, etc.) remained the same. In the following experiments, the trace-driven access pattern was used unless we explicitly specified which one was used. The capacity of the NAND-flash-based storage system was set as 8MB, the block size of the flash memory chips was 16KB, and the number of banks varied over experiments. The performance of the evaluated NAND flash memory (e.g., the setup time and the busy time of operations) is summarized in Table 1. Note that some of the experiments were performed over a software simulator when the hardware was not available. For example, those experiments require a NAND flash memory which had a block size equal to 8KB.

The primary performance metrics of the experiments is on the soft real-time performance of the flash-memory embedded systems, e.g., the average response time of write requests. We shall measure the efficiency of striping, although the average response time of writes also indirectly reflects that of reads. As astute readers may point out, the performance improvement of striping could be even more significant if the performance metrics also consider the response time of reads because the intervals of R_{setup} and R_{busy} are more close (see documented figures in Table 1), compared to those of W_{setup} and W_{busy}. The rationale behind this measurement is to provide an explicit impacts of striping on the system performance over writes (which are much slower than reads) and also to reflect the efficiency of garbage collection, because it introduces live page copyings.

Another performance metric is the number of live page copyings because it directly reflects the performance of garbage collection. A fewer number of live page copyings indicates the better efficiency of garbage collection and a smaller number of block erasings. A small number of block erasings implies the better endurance of flash memory and a less impact on writes' response time. Note that we did not measure the blocking time of flash-memory access here because the performance of the systems in handling flash-memory access could be observed by the primary performance metrics, i.e., response time of write requests.

The performance evaluation consisted of two parts: The first part evaluated the performance improvement of the proposed striping architecture under different bank configurations. The second part evaluated the performance of the proposed dynamic bank assignment policy versus a RAID-similar static bank assignment policy.

4.1 Performance of the Striping Architecture

Under a striping architecture, the performance improvement depended on the degree of parallelism because a higher degree indicated a higher concurrency of operations and a better throughput. This part of the experiments evaluated the performance improvement of the proposed striping architecture under different bank configurations. We must point out that the setup time of an operation implied the best parallelism we can achieve under a striping architecture. The setup time depended on the hardware interface we used to communicate with the flash memory, and a shorter setup time implied a potentially higher degree of parallelism because the system could switch over banks quickly. In this part of performance evaluation, we used typical NAND-flash over an ISA bus with characteristics shown in Table 1. When a higher performance bus such as PCI is used, we surmise that the performance improvement of a striping architecture could be even more significant.

File system	FAT32
Applications	Web Browser & Email Client
Sector Size/Page Size	512 Bytes
Duration	3.3 Hours
Final Capacity Utilization	71% (Initially Empty)
Total Data Written	18.284 MB
Read / Write Ratio	48% / 52%
Mean Read Size	8.2 Sectors
Mean Write Size	5.7 Sectors
Inter-Arrival Time	Mean: 32 ms Standard Deviation: 229 ms
Locality	74-26 (74% of Total Requests Access 26% of Total LBA's)
Length of the Two-Level LRU List	512 (Hot) / 1024 (Candidate)

Table 2: Characteristics of Traces.

Figure 7(a) shows the average response time of writes under various numbers of banks of an 8MB-flash storage system. The X-axis reflects the number of write requests processed so far in the experiments. The first experiment measured the performance improvement by increasing the numbers of banks in the experiments. Figure 7(a) shows that the system performance was substantially improved when the number of banks increased from one to two because of more parallelism in processing operations. However, when the number of banks increased from two to four, the improvement was not so significant because W_{setup} was larger than W_{busy}. Note that the garbage collection started happening after 3,200 write requests were processed in the experiments (due to the exhaustion of free pages). As a result, there was a significant performance degradation after the garbage collection activities began. Additionally, we also evaluated the performance of the cost-benefit policy [1] in the one-bank (not striped) experiments to make a performance comparison. The results showed that our system had a shorter response time than the cost-benefit policy did after the garbage collection activities began, even

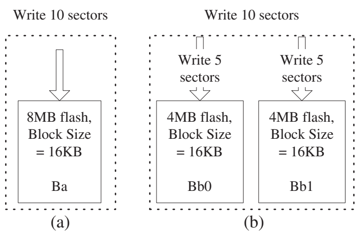

Write 10 sectors
Write 10 sectors
Write 5 sectors
Write 5 sectors
8MB flash, Block Size = 16KB
Ba
4MB flash, Block Size = 16KB
Bb0
4MB flash, Block Size = 16KB
Bb1
(a)
(b)

Figure 5: An Example of 10-Sector Writes with/without Striping.

the striping mechanism was not activated. It demonstrated that our garbage collection policy was also very efficient. We shall discuss our garbage collection policy in Section 5.

Figure 7(b) shows the number of live page copyings during garbage collection. The larger number of live page copyings, the more system overheads were. Note that a write request might consist of one or more page writes. Therefore, in the one-bank experiment (as shown in Figure 7(b)), the system wrote 37,445 pages ($\approx$ 18MB, please see Table 2) , and made copies of 5,333 pages for garbage collecting. We should point out that Figure 7(b) reveals the tradeoff between striping and garbage collection: We observed that a system with a large number of banks could have a large number of live page copyings. The tradeoff had a direct impact on the performance of the 4-banks experiments, as shown in Figure 7(a): That is, the average response time of the 4-bank system was a little bit longer than that of the 2-bank system after 8,000 write requests were performed. To identify the cause of the phenomenon, we provide an example as shown in Figure 5 for explanation:

In Figure 5(a), 10 pages were written to a 8MB flash bank called B_a. If the 8MB flash bank was split into two 4MB flash banks, and a striping architecture was adopted, 5 pages would be written to each of B_{b_0} and B_{b_1}, as shown in Figure 5(b). Although the bank size (and the amount of data written to B_{b_0} and B_{b_1}) were reduced by a half, however, the block size remained 16KB. Theoretically, we could consider that the block size of B_{b_0} and B_{b_1} as being doubled. The garbage collection performance would be degraded when the block size increased, as reported in [4, 7], since potentially more live pages could be involved in the recycling of blocks. Such a conjecture could be proved by observing the live-page copyings under striping when the page size was 8KB (a half of 16KB), as shown in Figure 7(b).

As shown in Figure 7(b), the extra live page copyings could be removed by using a NAND flash memory which has a smaller block size. However, the block size of a typical NAND flash memory currently in market is 16KB. Another approach to remove the extra live page copyings is to lower the overall capacity utilization of the flash memory. For example, with 1MB extra free space added, the number of live page copying of a 4-bank system is significantly reduced. As a result, we concluded that when several banks were adopted together without fixing the size of a flash-based storage system, the system performance improvement would be much more significant because the number of live page copyings might not increase under the new configuration.

Figure 7(c) shows the software-simulated performance improvement of striping when a more efficient bus was adopted for NAND-flash storage systems. We reduced the W_{setup} from 606us to 50us and remained W_{busy} as 303us (as shown in Table 1), where the documented W_{setup} was those shown on the data sheet [8], and 606us was that measured on our prototype. It was shown that the performance improvement of 4-bank striping was now much more than that of 2-bank striping, as shown in Figure 7(a), because a higher parallelism was possible.

4.2 Performance under Different Bank Assignment Policies and Workloads

The second part evaluated the performance of the proposed dynamic bank assignment policy versus a RAID-similar static bank assignment policy under different numbers of banks.

Figure 7(d) shows the performance of the flash memory storage system under the static bank assignment policy and the dynamic bank assignment policy. The X-axis reflects the number of write requests processed so far in the experiments. The garbage collection started happening after 3,200 writes were processed in the experiments. When garbage collection started happening (after 3,200 write requests), the dynamic bank assignment policy greatly outperformed the static bank assignment policy. The rapid performance deterioration of the static bank assignment policy was because of the uneven usages of banks. Figure 7(e) shows the erase count and the capacity utilization of each bank of a 4-bank system. We could observe that the dynamic bank assignment policy could provide a more even usages of banks. A better performance was delivered, and a longer overall lifetime of flash memory banks could be provided.

Experiments were also performed to compare the performance of different policies under different access patterns. Two additional access patterns were evaluated: a random access pattern and a sequential access pattern. The experiments were performed on a 4-bank system, and the results are presented in Figure 7(f). Under the sequential access pattern, there was no noticeable performance difference between the dynamic bank assignment policy and the static bank assignment policy. That was because data were sequentially written and invalidated. Under the random access pattern, since data were written and invalidated randomly, the performance of the garbage collection was pretty poor. But the dynamic bank assignment policy flexibly dispatched page writes among banks, as a result, the dynamic bank assignment policy outperformed the static bank assignment policy under the random access pattern.

5 Remark: Garbage Collection Issues

A flash memory storage system could not succeed without the support of a good garbage collection policy. In our system, we implemented a hot-cold aware garbage collection policy to incorporate with the proposed striping mechanism, by fully utilizing the hot-cold identification mechanism described in Section 3.3. We shall discuss important issues for garbage collection in this section.

5.1 Hot-Cold Separation

Any block in flash memory may be erased to recycle dead pages in the block if any available space is needed. During the recycling process, any live pages in the block must be copied to some available space. Pages that store hot data usually have a high chance to become dead in the near future. As a result, an efficient garbage collection policy must prevent from copying hot-live data. It is necessary to identify the "hotness" of data and store them at proper place so that the copying of hot-live pages can be minimized. An improper placement of hot data could result in a serious degradation of the system performance. In this section, we shall introduce a separation policy in locating pages for hot and cold data.

The bank assignment policy (please see Section 3.2) determines which *bank* the write will be used, and the block assignment policy determines which *block* the write will be used. The purpose of the hot-cold separation policy is to propose a hot-cold aware block assignment policy. In order to reduce the possibility in mixing hot and cold data inside the same block, we propose to store hot data and cold data separately. Each bank is associated with two pointers: "hot pointer" and "cold pointer". The hot pointer points to a block (called hot block) that is currently being used to store hot data. The cold pointer points to the block (called cold block) that is currently being used to store cold data. The proposed method in the separation of hot and cold data is illustrated in Figure 6. An experiment was performed over a 4-bank system without the the hot-cold identification and separation mechanism. As shown in Figure 7(b), the mechanism did substantially reduce the number of live page copyings for garbage collection.

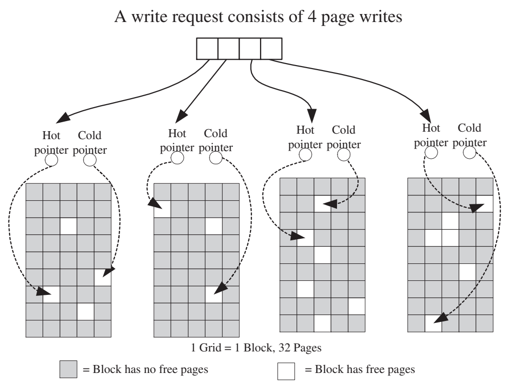


Figure 6: Hot-Cold Data Separation with Hot/Cold Pointers.

Page Attribute	Cost	Benefit
Hot (live)	2	0
Cold (live)	2	1
Dead	0	1
Free	N/A	N/A

Table 3: The Cost and Benefit Functions of Pages

5.2 A Block-Recycling Policy

A highly efficient cost-benefit policy for block reclaiming is proposed in this section. Distinct from the cost-benefit policies proposed in the previous work, e.g., [1, 7] [2], the value (called "weight" in our system) of each block is only calculated by integer operations in this paper. The adoption of integer operations in a cost-benefit policy is very useful in some embedded system without any floating-point support.

When a run of garbage collection begins, the weight

[2]The cost-benefit value is calculated by $\frac{age*(1-u)}{2u}$ in [1] and $\frac{u+c}{1-u}$ in [7].

of each block is calculated by the attributes of each page in the block. The weight is calculated by the following formula:

$$weight = \sum_{i=1}^{n}(benefit(p_i) - cost(p_i))$$

The block with the largest weight should be recycled first, where n is the number of pages in a block, and P_i is the $i-th$ page in the block. Functions $Cost()$ and $Benefit()$ of pages are shown in Table 3. During block recycling, the handling of a live page consists of one read and one write, and the cost of a live page is defined as 2, regardless of whether the data on the page is hot or cold. Here 2 stands for the cost of the read and the write of live page copying. As we pointed out earlier, the copying of a live-hot page is far uneconomic than the copying of a live-cold page. As a result, we set the benefit of copying a live-cold page and a live-hot page as 1 and 0, respectively. The benefit of the reclaiming of a dead page is one because one free page is reclaimed. It's cost is zero because no page copying is needed. Note that a free page is not entitled for reclaiming. The weight calculation of blocks can be done *incrementally* when the attribute of pages changes (e.g., from live to dead).

6 Conclusion

This paper proposed a dynamic striping architecture for flash memory storage systems with the objective to significantly boost the performance. Distinct from the past work, we focus on the architecture of a flash memory storage system. We propose to adopt a striping architecture to introduce I/O parallelism to flash-memory storage systems and an adaptive bank assignment policy to capture the characteristics of flash memory. We formulate a simple but effective garbage collection policy with an objective to reduce the overheads of value-driven approaches and to investigate the tradeoff of striping and garbage collection to provide insight for further system design. The capability of the proposed methodology was evaluated over a multi-bank NAND flash prototype, for which we have very encouraging results.

For future work, we shall further explore flash memory storage architectures, especially over interrupt-driven frameworks. With interrupt-driven flash controllers, an even more independent and large scale system architecture is possible in the future. More research in these directions may be proved very rewarding.

Acknowledgements

We would like to thank Samsung Electronics Company and M-System for their kindly providing us the NAND flash evaluation board and the related circuit design example, and Pou-Yuan Technology Corp. for her technical support.

References

[1] A. Kawaguchi, S. Nishioka, and H. Motoda,"A Flash Memory based File System," Proceedings of the USENIX Technical Conference, 1995.

[2] C. Reummler and J, Wilkes, "UNIX disk access patterns," Proceedings of USENIX Technical Conference, 1993.

[3] L. P. Chang, T. W. Kuo,"A Real-time Garbage Collection Mechanism for Flash Memory Storage System in Embedded Systems," The 8th International Conference on Real-Time Computing Systems and Applications, 2002.

[4] F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J.A. Tauber, "Storage Alternatives for Mobile Computers," Proceedings of the USENIX Operating System Design and Implementation, 1994.

[5] M. Wu, and W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Storage System," Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, 1994.

[6] M. Rosenblum, and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," ACM Transactions on Computer Systems, Vol. 10, No. 1, 1992.

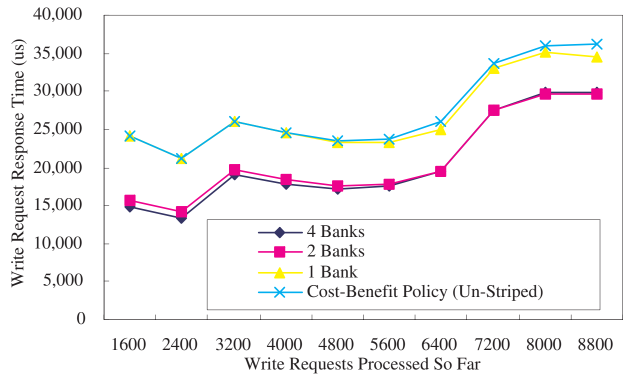
[7] M. L Chiang, C. H. Paul, and R. C. Chang, "Manage flash memory in personal communicate devices," Proceedings of International Symposium on Consumer Electronics, 1997.

[8] Samsung Electronics Company,"K9F6408U0A 8Mb*8 NAND Flash Memory Datasheet".

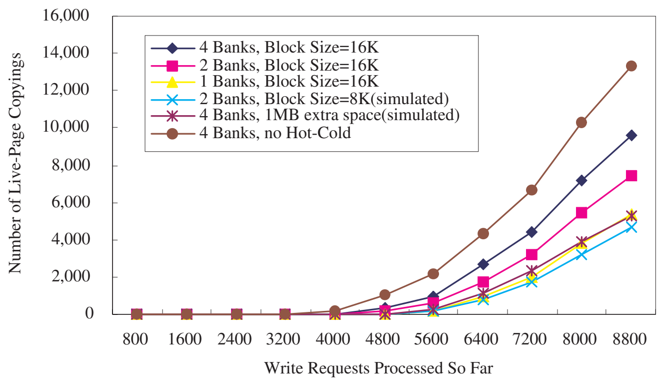
[9] SSFDC Forum, "$SmartMedia^{TM}$ Specification", 1999.

[10] Compact Flash Association, "$CompactFlash^{TM}$ 1.4 Specification," 1998.

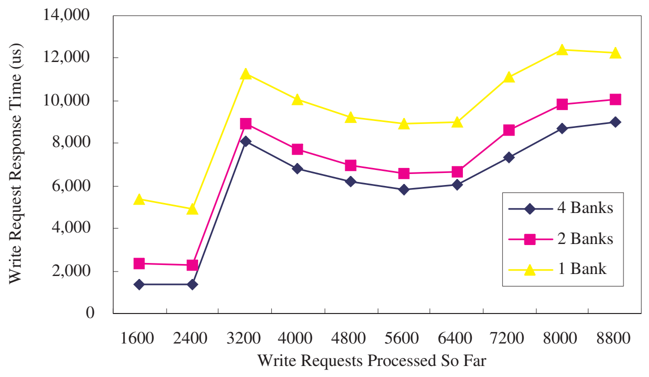
[11] Journaling Flash File System (JFFS) and Journaling Flash File System 2 (JFFS2), http://sources.redhat.com/jffs2/jffs2-html/



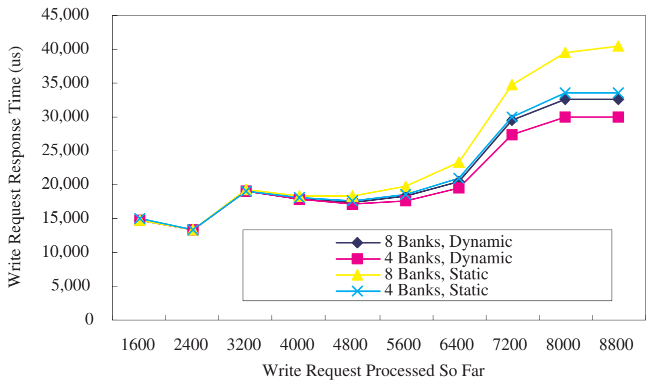
(a)Average Response Time of Writes under Different Bank Configurations



(b)Performance of Garbage Collection under Different Bank Configurations



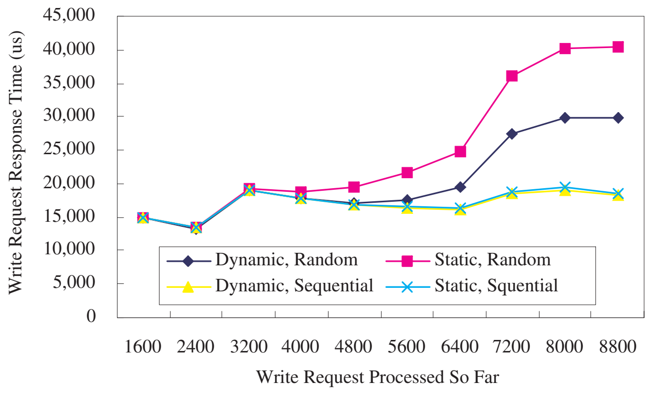
(c)Performance Improvement with Less Flash Setup Time (Simulation).



(d) Average Response Time of Writes under Different Bank Assignment Policies

	Bank 0	Bank 1	Bank 2	Bank 3
Erase counts (Dynamic)	350	352	348	350
Erase counts (Static)	307	475	373	334
Capacity Utilization (Dynamic)	0.76	0.76	0.76	0.76
Capacity Utilization (Static)	0.72	0.81	0.77	0.74

(e)The Usages of Banks under the Dynamic and Static Bank Assignment Policy (A 4-Bank System).



(f)Average Response Time of Writes under Different Access Patterns and Bank Assignment Policies (A 4-Bank System).

Figure 7: Experimental Results.