Exploiting Internal Parallelism of Flash-based SSDs

Seon-yeong Park*, Euiseong Seo†, Ji-Yong Shin*, Seungryoul Maeng* and Joonwon Lee‡
*Korea Advanced Institute of Science and Technology †Ulsan National Institute of Science and Technology
‡Sungkyunkwan University

Abstract—For the last few years, the major driving force behind the rapid performance improvement of SSDs has been the increment of parallel bus channels between a flash controller and flash memory packages inside the solid-state drives (SSDs). However, there are other internal parallelisms inside SSDs yet to be explored. In order to improve performance further by utilizing the parallelism, this paper suggests request rescheduling and dynamic write request mapping. Simulation results with real workloads have shown that the suggested schemes improve the performance of the SSDs by up to 15% without any additional hardware support.

Index Terms—Flash memory, I/O scheduling, parallelism, Solid-State Drives (SSDs).

1 Introduction

REPLACING hard disk drives, where performance improvement has faced the physical limitation due to mechanical positioning components, flash-based solid-state drives (SSDs) are becoming the mainstream high performance storage devices because of their fast random access speed and superior throughput, as well as low power consumption.

An SSD uses multiple flash memory packages which are connected to a flash controller through parallelized multi-channel buses. By increasing the bus channels, the performance of SSDs has been improved rapidly over the last few years, so that cutting-edge SSDs now show sustained read throughput of up to 250MB/s and write throughput of up to 200MB/s. However, the strong demand for storage devices with even better performance remains due to the stiff increase of processor parallelism from the multicore technology.

Inside the current SSD architecture, there are other parallelisms which have been of little concern yet. The state-of-the-art flash memory packages have independently working multiple dies inside, and each die contains multiple planes which operate simultaneously [1][2]. This integration provides a large storage capacity within restricted space and improves the performance. However, this parallelism can be fully utilized only when the planes on a single die perform operations of the same type at the same time. Moreover, a plane provides a command pipeline to expedite the process of queued requests [3]. However, this command pipelining achieves a performance boost only when consecutive requests are of the same type. Therefore, the effectiveness of plane-level parallelism strongly depends on the pattern of request sequence.

Due to the nature of flash memory, which does not support in-place updates, SSDs use a hard disk emulation layer called an FTL (Flash Translation Layer). The FTL maps externally seen logical pages to internally used physical pages. Because the FTL does not consider the queue status of outstanding requests at plane-, die-, and flash package-level, uneven request distribution among planes, dies, or flash packages may occur during write request mapping. Unsurprisingly, performance significantly decreases with such uneven write request distribution, which causes long waiting time in a queue for both read and write requests.

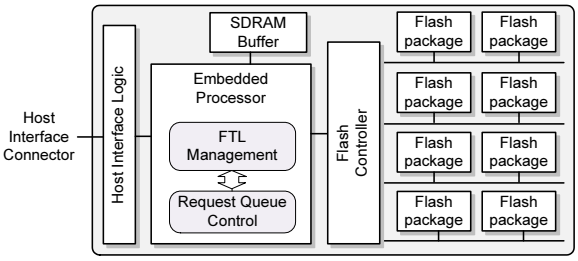


Fig. 1. SSD block diagram

In this paper, we suggest *request rescheduling* and *dynamic write mapping* approaches to exploit the internal parallelism of SSDs to boost the performance by overcoming the problems described. Request rescheduling reorders the outstanding requests in the request queue to improve the plane-level parallelism more effectively. The dynamic write mapping algorithms reduce queue delays of requests inside SSDs reflecting the status of each parallel components. These schemes are expected to result in shorter response times for SSDs.

2 Background

SSDs are storage devices that use flash memory as their storage media. They have the standard host interfaces as well as the form-factors of ordinary hard disks.

Both read and write operations on flash memory are processed by a *page* unit, which is usually 2KB in size. Different from traditional storage media, such as hard disks or DRAM, flash memory pages must be erased before rewriting on them. However, the unit of an erase operation is different from that of a read or write operation. An *erase block* generally consists of 64 consecutive physical pages. Because of this behavioral difference of flash memory, SSDs employ the FTL, which maps a logical page to a physical page. The FTL allocates free physical pages which have already been erased, to write new data when a write request arrives.

As shown in Figure 1, an SSD has an embedded processor that manages the FTL and external interface, along with a flash controller, which connects flash memory packages through a multi-channel bus. A flash controller can issue a command through each channel independently.

Manuscript submitted: 28-Dec-2009. Manuscript accepted: 03-Feb-2010. Final manuscript received: 09-Feb-2010.
This work was supported by the Korea Science and Engineering Foundation(KOSEF) grant funded by the Korea government(MEST). (No. 2009-0080381)

Posted to the IEEE & CSDL on 2/26/2010
DOI 10.1109/L-CA.2010.3 1556-6056/10/$26.00 © 2010 Published by the IEEE Computer Society

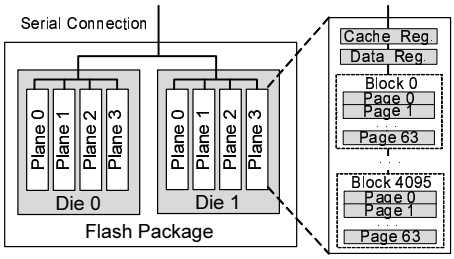


Fig. 2. Flash package organization

Each flash package consists of multiple dies as shown in Fig. 2. Each die contains multiple planes, which have physical pages inside. Although each die is able to perform a read, write or erase operation independently, all the planes on a die can only carry out the same type of operation at one time. A command that makes two planes work simultaneously is called a *two-plane command*, and a command that makes n planes work at the same time is called an *n-plane command*. Currently, the most widely used configuration is a die with two or four planes with the two-plane command. The increased number of planes will also inflate the importance of effective use of n-plane commands for the performance of SSDs.

Every plane contains a register called the data register, which temporally stores a page before issuing a read or write command. When a plane processes a write command, data is first transferred from the controller to the data register, which usually takes about 50μs. After that, the data stored in the data register is written to the corresponding physical page, which takes about 250μs. On the other hand, when reading a physical page, the data is read from a physical page into the data register, which takes about 25μs, and then transferred from the data register to the controller, again, which takes about 50μs.

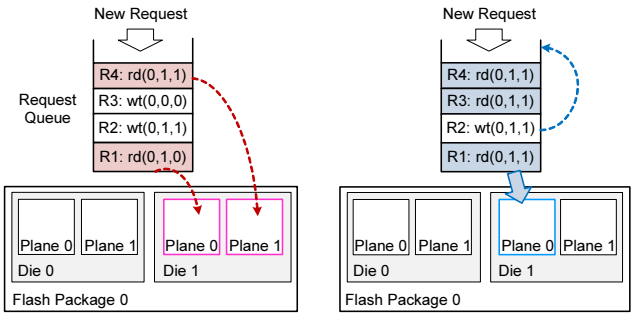
In each plane, there is another register called the cache register for pipelining the consecutive commands. While processing a series of write commands, the cache register temporally stores data to write until the data register becomes available after finishing writing the previous data. On the other hand, when processing consecutive read commands, the cache register is used to store data temporally before sending it to the flash controller. While a page is read from a physical page to the data register, the cache register transfers the previous read data that was sent from the data register. This command pipelining improves the performance significantly when the consecutive commands are of the same type.

The host interfaces of recent SSDs provide outstanding request queues by Native Command Queuing (NCQ) standard of SATAII interface. Generally, the SSDs with the NCQ feature can store up to 32 outstanding requests. For traditional hard disks, request queues are used to reduce disk head movement by reordering the outstanding requests. However, in comparison, methods for managing outstanding request queues in SSDs have not yet been as actively researched up until now.

3 Parallelism-aware Request Processing

3.1 Request Rescheduling

Although both the n-plane command and the command pipelining of flash packages have significantly enhanced the performance of flash memory packages [3], the ordinary request sequence, which has no consideration for those features, will not fully utilize them. Therefore, we suggest two request



(a) Multi-plane rescheduling (b) Pipeline rescheduling

Fig. 3. Request rescheduling schemes

rescheduling schemes, *Multi-plane rescheduling* and *Pipeline rescheduling*.

Multi-plane rescheduling reorders outstanding requests in order to assign requests of the same type to as many planes in a die as possible. Fig. 3 (a) depicts an example of the multi-plane rescheduling scheme. If the requests in the request queue were issued sequentially as listed in the queue, requests R1 and R2 cannot be executed by using a two-plane command although the target plane of R2 is idle because read and write commands cannot be executed on a single die at the same time. Thus, R1 and R2 must be executed sequentially. However, by scheduling R4 with R1, both requests are allowed to run simultaneously with a single two-plane read command.

When a flash package starts to execute a request, the multi-plane rescheduling scheme looks up the next request in the queue that is of the same request type, and has a different target plane on the same die. If there exists a request in the queue that meets those two conditions, the flash controller executes both requests with an n-plane command at one time.

The command pipelining does not bring a performance boost when read and write commands are interchanged with each other because the next command can not start until both the data and cache registers become available after finishing the previous command. To boost the effectiveness of command pipelining in a plane, consecutive requests should have the same operation type where possible. Pipeline rescheduling reorders outstanding requests in the request queue so as to reduce the number of operation type changes.

Fig. 3 (b) presents an example of the pipeline rescheduling scheme. Requests R1, R2, R3, and R4 are headed to plane 0 in die 1. However, R1, R2, and R3 cannot be pipelined because R2 is a write request while both R1 and R3 are read requests. By rescheduling R2 after R4, the requests of R1, R3 and R4 all have the same operation type. Therefore, the time to transfer data to the flash controller for R1 and R3 can be overlapped by the time to read physical pages for R3 and R4, respectively.

When there is data dependency between two outstanding commands, both of the suggested rescheduling schemes preserve the order of those commands to maintain the integrity of the data.

3.2 Dynamic Write Mapping

When a write request arrives, the FTL decides the physical location of pages for the request. To obtain a free physical page to write new data, the FTL determines which package, die, and plane will be used with a predefined algorithm. The algorithm can make decisions based on the logical page

addresses of requests [4]. However, because this static mapping algorithm is blind to the request queue status or the parallelized architecture inside flash storage components, it does not fully exploit the parallelisms in reducing queue delays. Our approach, the *dynamic write mapping* scheme, distributes the outstanding write requests with the consideration of dynamic queue status changes or parallel architectures inside SSDs.

We explore three different heuristic algorithms for the dynamic write mapping approach. First, the *shortest-queue-first* algorithm maps the write requests to the least loaded die among whole packages. Second, the *shortest-estimation-time-first* algorithm maps the write request to the die with the shortest estimated completion time of the queued requests in each die. These two policies prioritize the response time of the write requests. Thus, we expect that these algorithms lessen the write response time but, because consecutive read requests can flow into few numbers of less loaded packages and dies, they may have slow response time of read requests. Third, the *multi-level stripe* mapping considers the internal structure of flash storage components when deciding write locations. It distributes the requests according to the parallelism level of the components. It prioritizes channel striping and, within the same channel, it performs package striping, and then die and plane striping. For example, the i-th request is assigned to the channel number (i mod total number of channel). In turn, the request will be assigned to the package number ((i / total number of channel) mod total number of packages per channel) within the channel number (i mod total number of channel). In the same manner, die and plane number are assigned to the request. This mapping policy fully utilizes the parallelism of storage components when distributing the write requests, and the read requests will also be expected to benefit from its parallelism-aware distribution.

Dirik and Jacob [5] used a simple write mapping algorithm that maps a write request to free packages in a round-robin manner based on the sequence number of the write request. However, different from their algorithm, our approach takes the sub-package-level parallelisms into consideration. In addition to that, our approach considers not only the sequence number, but also the dynamic changes of load distribution due to the existence of pre-issued read and erase operations.

4 Evaluation

4.1 Environment

We implemented an SSD simulator with parameters listed in Table 1 and integrated with Disksim 4.0 [6]. Both synthetic and real workloads are used in our evaluation. The real workloads to be used in our evaluation were collected from the *iozone* filesystem benchmark, the *Openmail* e-mail messaging application, the *SYSmark* PC benchmark, and a web browser. We assume a multicore processor environment, thus these workloads are modified such that I/O requests are gathered from four processes running on a quad-core processor. Table 2 presents the characteristics of the workloads evaluated.

4.2 Results

The effectiveness of the suggested request rescheduling schemes are evaluated with synthetic workloads. Fig. 4 (a) shows the improvement of average response time according to the changes of read-to-write proportion of the workloads. The multi-plane scheme is evaluated with two- and four-plane command enabled hardware configurations, respectively.

TABLE 1
Parameters used in the SSD simulator

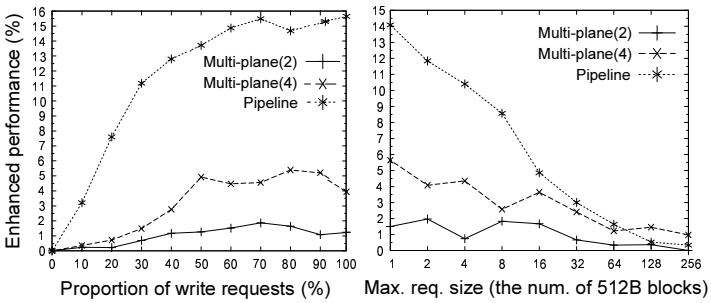
Parameter	Value
Total capacity	120GB
The number of channels	16
The number of packages per channel	2
The number of dies per chip	4
The number of planes per die	4
The number of pages per block	64
Internal bus speed (x8)	25 ns
Page size	2 KB
Page read time	25 us
Page write time	250 us
Block erase time	1.5 ms
Host to SSD bus	SATA II (300MB/s)
Request queue depth	32
FTL scheme	Page-based mapping
Garbage collection policy	Background running with threshold of 5% of total number of blocks

TABLE 2
Workload characteristics used in the evaluation

Name	Total storage (GB)	RW ratio (read:write)	Average/median request size (KB)			Consecutive requests (#)	
			Total	Rd	Wt	Rd	Wt
iozone	57.8	0.48 : 0.52	44.1 /128	45.1 /128	43.1 /128	1.6	1.7
openmail	59.1	0.63 : 0.37	5.8 /16	5.4 /16	6.5 /16	4.5	2.6
sysmark	100.1	0.58 : 0.42	30.0 /36	24.7 /32	37.2 /128	15.0	11.1
web	66.7	0.24 : 0.76	24.0 /14	8.5 /8	28.9 /24	17.7	55.3

The rescheduling schemes have insignificant effects when most requests are reads because they only have few opportunities for reordering read requests with write requests. On the other hand, when the proportion of write requests is becoming high, the performance is gradually enhanced up to 15.6% with the pipeline scheme and up to 5.4% with the multi-plane scheme. The performance enhancement does not shrink when the write requests are dominant. This is due to the large amount of partial writes which induce a large amount of internal page read operations. By reordering these internal read operations with write operations, the effectiveness of our schemes remain under write-dominant situations.

Fig 4 (b) illustrates the performance enhancement according



(a) Varying read/write ratios (req. size = 2KB)

(b) Varying request size limits (read:write = 0.5:0.5)

Fig. 4. Response time improvement of synthetic workloads by each request rescheduling scheme

TABLE 3
Average response time improvement by request rescheduling and dynamic write mapping from the result of the SSD in Table 1 (Numbers in parentheses are differences from that without rescheduling.)

	Request Rescheduling (%)						Dynamic Write Mapping (%)		
	Without rescheduling			Rescheduling			Shortest-	Shortest-est.	Multi-level
Workload	n-plane	Cache reg.	Both	Multi-plane	Pipeline	Both	queue-first	-time-first	Stripe
iozone	4.38	2.49	8.70	8.39 (4.01)	7.74 (5.35)	11.88 (3.18)	-0.67	0.86	2.49
openmail	21.86	24.68	27.48	24.53 (2.67)	30.22 (5.54)	31.70 (4.22)	17.42	19.98	16.32
sysmark	5.46	5.82	11.12	5.98 (0.52)	7.38 (1.56)	13.29 (2.17)	14.49	14.57	14.64
web	19.69	15.03	34.73	20.55 (0.86)	15.41 (0.38)	35.87 (1.14)	6.18	6.12	6.24

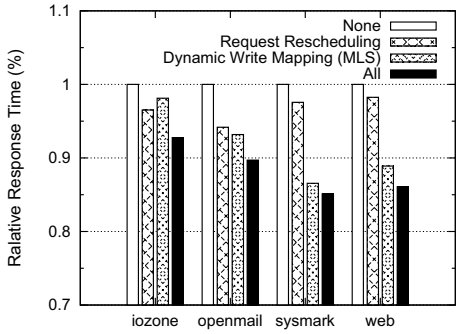


Fig. 5. Normalized response time of applying both request rescheduling and dynamic write mapping together

to various request sizes. As the average request size is getting smaller, the effectiveness of request rescheduling is becoming larger because smaller request size induces more frequent switches between page read and write operations in die-level, which lead to more chances for request rescheduling.

In both of Fig 4 (a) and (b), the improvement from multi-plane rescheduling is smaller than that from pipeline rescheduling. However, if the number of planes continues to grow, the effectiveness of multi-plane rescheduling will increase as the number of integrated planes on a die grows. In addition to this, when the size of a page doubles, since the requests will be split into a smaller number of pages, we expect that there will be more rescheduling opportunities that lead to better performance improvement.

The evaluation results using the four real workloads are illustrated in Table 3. The values in Table 3 denote the improvement of average response times from that of the default configuration without both n-plane command sets and cache registers. Because `openmail` has small request size, shown in Table 2, it has more opportunities for reschedule than others. Although iozone has large request size, the average number of consecutive requests is small. This means that it has frequently interleaved read and write requests. Because of this, it makes better performance improvement than `sysmark` and `web`.

The suggested dynamic write mapping algorithms are compared with the static stripe mapping. `sysmark` gains the greatest improvement by applying dynamic write mapping because its repetitive read and write requests on restricted regions cause hot-spot dies, and the dynamic write mapping algorithms effectively spread the requests over whole dies. In `web`, because write requests are dominant, there are many chances to apply the dynamic write mapping algorithms.

The effectiveness of multi-level stripe mapping increases when request size is large, because interleaving large requests over multiple planes, dies, or packages increase the chances of using parallelized read operations when the data is read in the future. On the contrary, when the request size is small, the multi-level stripe shows the least improvement, as presented in the `openmail` results. The large amount of small requests raises the chances of uneven distribution of read requests, and also write requests does not have the benefit of reducing the response time from the multi-level stripe algorithm. Therefore, the algorithms with consideration of the load distribution perform better than the multi-level stripe algorithm.

Fig. 5 shows the overall performance after applying both request rescheduling algorithms and the multi-level stripe write mapping algorithm. After applying both of the suggested schemes together, the response time of `iozone`, `openmail`, `sysmark` and `web` are reduced by 7.2%, 10.3%, 14.9%, and 13.9%, respectively.

5 Conclusion and Future Work

In spite of the significantly greater performance of SSDs compared to that of hard disks, the rapidly increasing parallelism of multicore processors is still firmly retaining the strong demand for the improvement of SSDs as high performance storage devices. This study suggested request rescheduling and dynamic write mapping schemes to further improve the performance of SSDs through enhancing parallelism.

Request rescheduling improves the plane-level concurrency and the effectiveness of the command pipelining. Dynamic write mapping reduces the response time of write requests by spreading write requests over the entire dies in an SSD. We showed that an SSD can benefit up to about 15% performance improvement by applying both approaches together.

Based on the findings of our study, we will pursue research on dynamic data redistribution. By redistributing stored data among planes reflecting the actual reading patterns, which are collected on the fly, the degree of parallelism is expected to be further enhanced.

References

[1] Micron, "Improving performance using two-plane commands introduction," Tech. Rep. TN-29-25, October 2008.
[2] Samsung Electronics, *K9GAG08B0M (2G x 8 Bit NAND Flash Memory) Datasheet*, April 2006.
[3] Micron, "Nand flash status register response in cache programming operations," Tech. Rep. TN-29-26, June 2007.
[4] J.-Y. Shin, Z.-L. Xia, N.-Y. Xu, R. Gao, X.-F. Cai, S. Maeng, and F.-H. Hsu, "Ftl design exploration in reconfigurable high-performance ssd for server applications," in *International conference on Supercomputing*, June 2009, pp. 338–349.
[5] C. Dirik and B. Jacob, "The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization," in *International Symposium on Computer Architecture (ISCA)*, June 2009, pp. 279–289.
[6] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger, "The disksim simulation environment version 4.0," Tech. Rep. CMU-PDL-08-101, May 2008.