공학석사 학위논문

Software-based Out-of-Order Scheduling Technique for High-Performance SSDs

2013년 8월

서울대학교 대학원

전기컴퓨터공학부

한 상 욱

Software-based Out-of-Order Scheduling Technique for High-Performance SSDs

지도교수 김 지 홍

이 논문을 공학석사 학위논문으로 제출함
2013년 8월

서울대학교 대학원
전기컴퓨터공학부
한 상 욱

한상욱의 석사 학위논문을 인준함
2013년 8월

위 원 장	하 순 회	[illegible] (인)
부위원장	김 지 홍	(인)
위 원	– ii McKay	(인)

Abstract

We propose an efficient software-based out-of-order scheduling technique for high-performance NAND flash-based SSDs. Unlike an existing hardware-based out-of-order scheduling technique in SSDs (which is implemented in a physical address space), our proposed software-based solution, SOST, can make more efficient out-of-order scheduling decisions, taking advantage of both logical address information and physical address information of multiple I/O requests simultaneously without a significant implementation cost. By exploiting various mapping information and I/O access characteristics available from the flash translation layer (FTL) software, SOST can avoid unnecessary hardware-level operations and manage inter- and intra-queue request rearrangements more efficiently, thus maximizing multichip parallelism of SSDs. Furthermore, SOST can easily support user- or OS-specified I/O request priorities which are important in time-sensitive real-time applications. Experimental results on a prototype SSD show that SOST is effective in improving the overall SSD performance, lowering the average I/O response time by up to 43% over the out-of-order flash controller.

Keyword : SSD, NAND flash, Out-of-order, scheduling
Student ID : 2011-20952

목　　차

Chapter 1. Introduction

Section 1. NAND flash memory

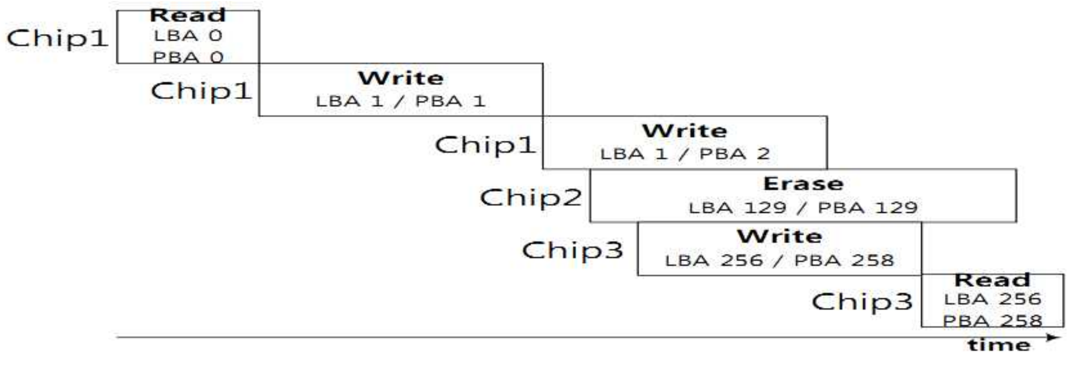
NAND flash memory based devices such as Solid State Disks (SSDs) are becoming more popular in the market. For example, NAND flash memory has already become the most popular storage technology in mobile devices because of its low power consumption, high density and high resistance to shock. These advantages are possible mainly because SSDs operate without any mechanical movement. Also, the high cost of SSDs, which was once considered as a disadvantage, has been gradually lowered as the manufacture process shrinks and the number of bits stored in a single cell increases. As a result, flash memory based SSDs are now increasingly adopted not only in personal computers such as laptops but also in server systems.

NAND flash memory, however, has some characteristics which prevent its direct use as a storage device. A NAND flash chip consists of blocks, each of which contains a set of pages that can be accessed individually by read and program operations. Unlike HDDs, the NAND flash memory architecture does not allow the in-place update of data and all the pages in a block must be erased before any update. In order to present the same storage interface as an HDD and to overcome the limitations of flash memory, a software layer called an FTL is used in various flash storage devices.

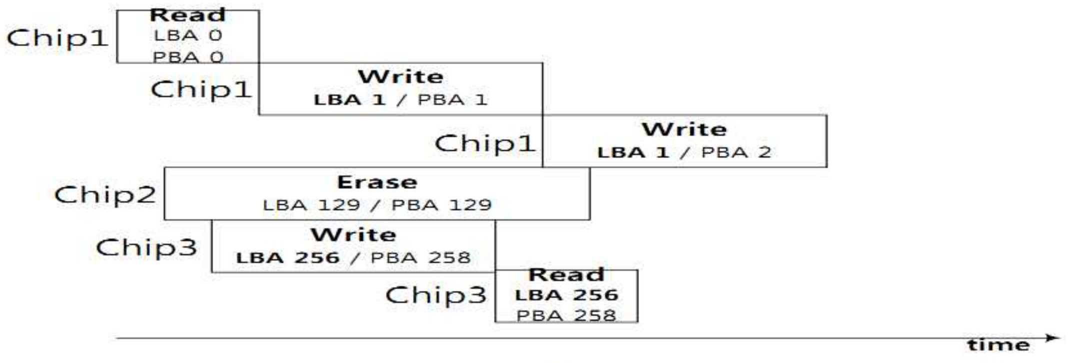
While a simple FTL generates one flash request at a time, a more advanced FTL needs to exploit multiple flash memory chips efficiently by generating multiple concurrent flash requests. Thus, the parallel servicing of flash requests becomes more important if the storage device interface allows the host computer to have multiple read and write requests. Also, handling the time-sensitive requests becomes more critical if the multiple requests are generated concurrently by the FTL because the priority information of a request is not provided at the FTL level. The out-of-order scheme can be a perfect solution to support the multichip-level parallel servicing and the prioritized request management issues.

Section 2. Out-of-order execution

NAND flash-based device consists of multiple chips and each chip can perform only one flash operation at a time. There are write, read and erase flash operations for NAND flash memory and their latencies differ. When NAND Flash device receives I/O requests from file system, requests are divided with page size in order at a software layer called an FTL. FTL also translates the logical block addresses (LBA) of requests to physical block addresses (PBA). Each request divided with page size is allocated to chips by exploiting the multichip parallelism. If some chips finish allocated requests earlier than other chips or don't get any request, they remain idle state until the requests of the other chips will be completed. When some requests are flocked to one chip, as Fig. 1 illustrates, the performance can be limited in the in-order execution model. In order to exploit multichip parallelism, the following requests should be assigned before previous request finished for decreasing the time which chips remain idle. Therefore, the out-of-order algorithm is the perfect solution to increase performance by exploiting the multichip parallelism.



(a) In-order execution



(b) Out-of-order execution

Fig. 1: Models of in-order and out-of-order execution.

Section 3. Motivation

Unlike an existing hardware-based out-of-order scheduling technique in SSDs (e.g. O3 flash controller [1]), in this paper, we argue that out-of-order scheduling of NAND operations can be more efficiently supported by a software implementation. There are two main motivations for our proposed software-centered out-of-order scheduling support.

First, if out-of-order support could be implemented by a software module, it can take advantages of inter-request dependences more efficiently by exploiting inter-request dependences both at the logical address space and at the physical address space. As will be described in later sections, there are several distinctive advantages if a storage device can make scheduling decisions based on information available from two separate address spaces. On the other hand, if out-of-order scheduling is supported by a hardware controller, out-of-order controller's decisions are mostly limited to the physical address space, thus missing many optimization opportunities observed during run time.

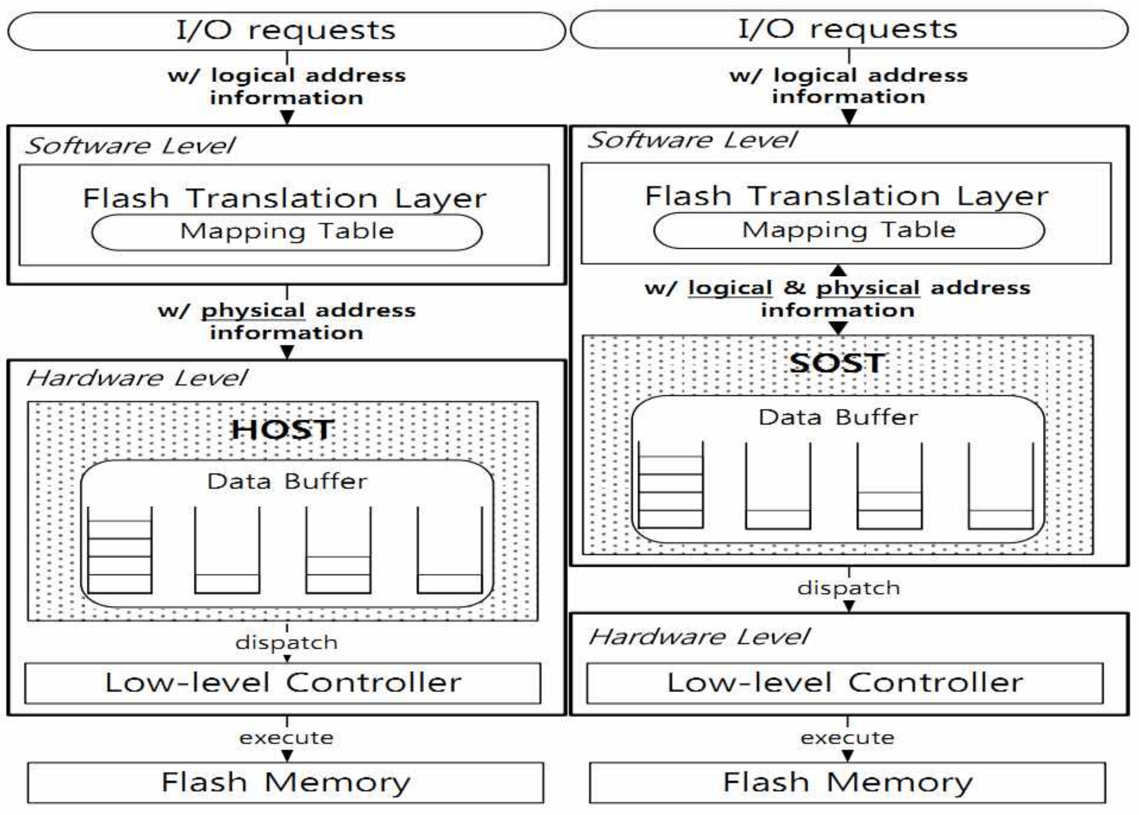
Second, a software implementation of out-of-order scheduling does not incur a significant performance penalty over a more costly hardware implementation. This typical claim is based on the NAND flash memory's characteristics. Due to reliability, NAND flash memory's read/write latency tends to get worse as the manufacturing process gets smaller. For example, the write latency increases from 300 us of 120-nm NAND flash memory [2] to 2290 us of 20-nm [3]. Also the read latency increases from 25 us of 120-nm [4] to 119 us of 20-nm [5]. At the same time, modern SSDs employ a high-performance multicore CPUs (e.g., SAMSUNG 830 SSD has triple-core CPUs and each CPU works at 300 Mhz clock speed [6]). For these high-performance multicore CPUs, scheduling decisions by an out-of-order software module does not take more than 1/100-th of one NAND read operation, thus making a software-based out-of-order scheduling technique almost free.

Chapter 2 Out-of-Order Scheduling in SSDs

Section 1. Hardware-based Out-of-order Scheduling

The hardware-based out-of-order scheduling technique (HOST) was designed by Nam et al. and they realized it at a flash memory controller that improved the performance of a flash storage system by executing multiple flash operations in an out-of-order manner. In the HOST, data dependencies are the only ordering constraints on the execution of multiple flash operations. This allows HOST to exploit the multichip parallelism inherent in flash memory more effectively than interleaving.

Fig. 2(a) illustrates the architecture of the HOST. The host computer generates read and write requests (I/O requests) with logical addresses and FTL translates host requests into flash requests at the software level. FTL delivers flash requests with physical addresses to HOST placed at the hardware level. After HOST receives flash requests, it executes them in the out-of-order manner to the NAND flash memory. This HOST was able to achieve 46 to 88 percent lower response time compared to the in-order execution model which allows



(a) Hardware-based out-of-order scheduling technique (b) Software-based out-of-order scheduling technique

Fig. 2: Out-of-order architectures.

a partial overlap between flash operations by using the pipeline method.

They adopted a hardware-oriented approach for HOST because the flash latencies of read, write, and erase operations are relatively short (in the range of a few tens to hundreds of microseconds) when they designed HOST. They supposed that if they involved FTL in scheduling flash operations, it would introduce a serial bottleneck for speeded-up operations with multiple chips. They also mentioned that software scheduling requests can be a sensible option for HDDs when media access latencies are very long (in the range of milliseconds).

However, as Fig. 3 illustrates, since the manufacture process shrinks and the number of bits stored in a cell increases, flash latencies have dramatically increased. The number of cores used in SSDs has also increased from single to triple and the clock frequency of each core increases from 75 Mhz to 300 Mhz [7, 8].

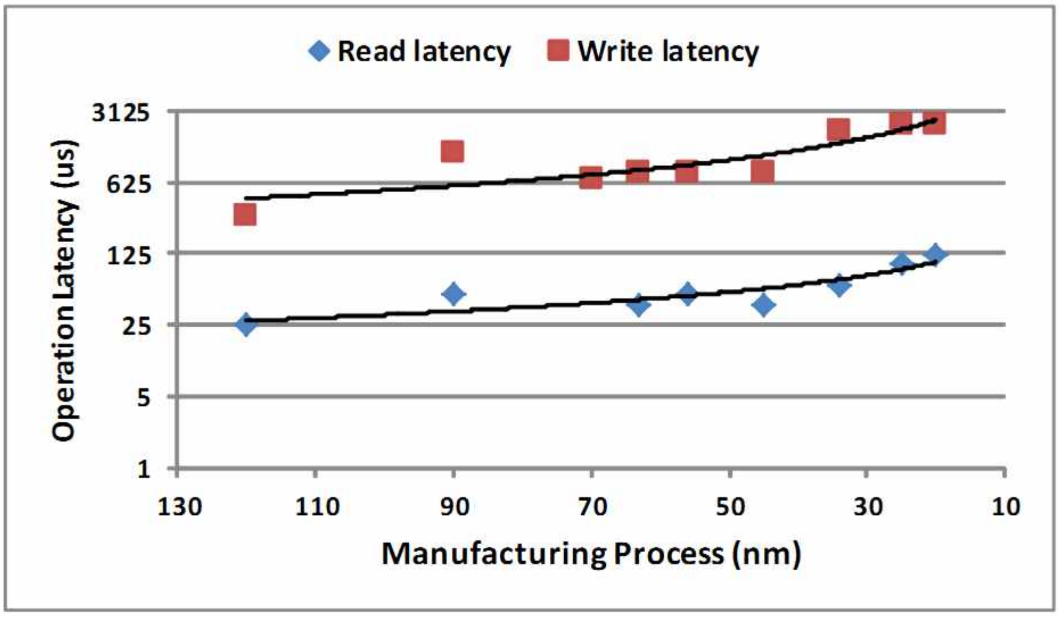


Fig. 3: Flash latencies with shrinking manufacturing process.

We measured the average clock cycles which represent the software overhead between flash operations from various benchmarks. We were able to calculate the time of the software overhead with measured average clock cycles between flash operations from various benchmarks by multiply them and clock frequency of CPU in SSDs. Fig. 4 illustrates that software overhead has been decreasing as the manufacture process shrinks and the flash latencies increase.

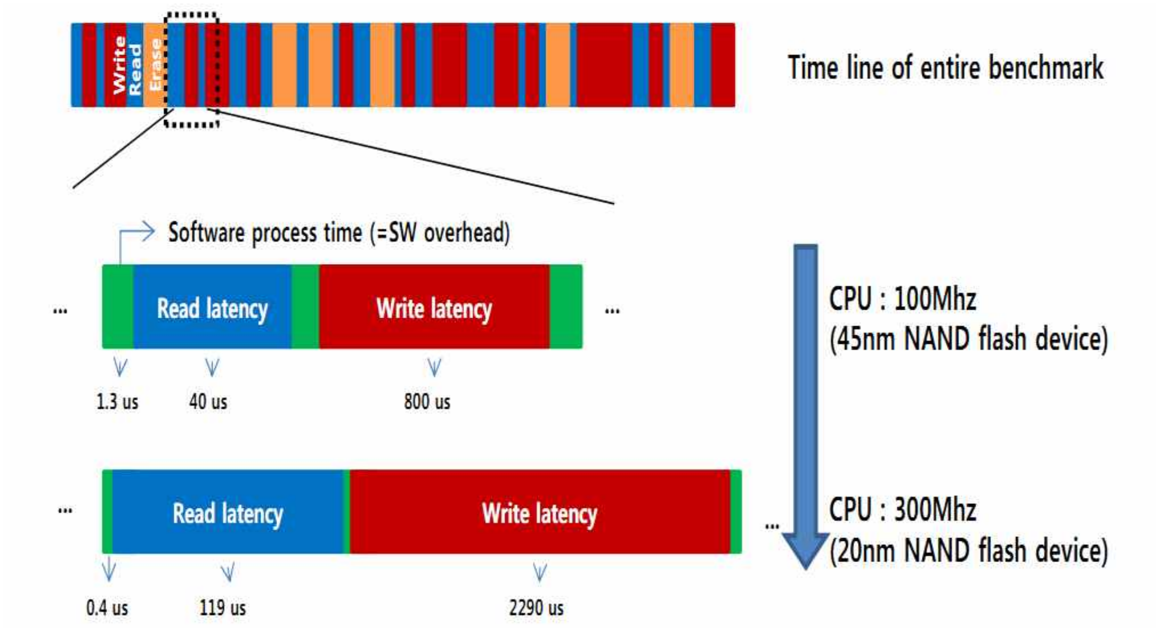


Fig. 4: Portion of software overhead with increasing flash latencies.

As a result, at the FTL level, not at the controller level, scheduling requests for the out-of-order execution is no longer a bottleneck than in the past. Besides, realizing these logics on the hardware-level controller costs more than as a software one.

Section 2. software-based Out-of-order Scheduling

Unlike the hardware approach, the software-based out-of-order scheduling technique (SOST) is able to easily access and modify information maintained by the FTL because it is placed at the software level with FTL as Fig. 2(b) illustrates. By taking advantages of software, SOST can reduce unnecessary hardware-level operations and adopt new algorithms for performance improvement which HOST hardly afford to do.

Even when flash requests are allocated to chips by exploiting the multichip parallelism, their process time of flash requests assigned can be varied due to the locality of read and erase requests as well as different read, write, and erase operation latencies. Fig. 2(b) shows the situation that 4 flash requests are flocked to one chip while other chips have less than 2 requests. Those chips with a relatively short process time of flash requests can remain in an idle state until other chips with a longer process time complete their operations. At the software level, performance improvement can be induced by reallocating the flocked requests to other chips to decrease the total response time. HOST is not able to adopt this method which uses the logical addresses of flash requests because it handles requests only with their physical address information. Although the FTL is able to dispatch requests to the controller with not only their physical addresses but also logical addresses, the reallocation of requests needs the process which accesses and modifies data of the mapping table and this process is extremely difficult and costly for the controller.

We can recognize that there is overwrite which writes data to the same LBA at the chip1 in Fig. 1. However, HOST can't recognize this overwrite because it handles requests only with the physical address information. Hence, HOST executes the later write request of PBA 2 after finishing the previous write request of PBA1 and these processes are very inefficient. Another data dependency occurs at the chip3 which executes read operation on the data of previous write request.

At the software level, we can improve efficiency by reading buffered data directly unlike the hardware approach which has to wait until the previous write request of buffered data is completed.

In this paper, a software module of the out-of-order execution technique which uses above methods is proposed to achieve the overall performance improvement of the SSD. We named this software-based out-of-order scheduling technique as SOST and implemented this SOST and tested its performance using an FPGA-based development platform. Our evaluation of the performance of this SOST shows a 14 to 42 percent improvement in response time compared to HOST.

Chapter 3. Design and Implementation of SOST

Section 1. Architecture of SOST

The SOST performs the out-of-order execution of multiple flash operations at the software level by exploiting multichip parallelism. Fig. 5 shows the overall architecture of the SOST which consists of a queue for data buffer, a dynamic scheduler, a queue size leveler, a hit manager, and a user-defined priority handler. The FTL allocates the logical address of a request to the physical address after dividing up the received request data into several pages with the size of page in the NAND flash memory. Then, the allocated request data are added to the queues as the data buffer on a page basis, which store the information and the data of a request. The dynamic scheduler in the SOST dispatches the pages buffered in the queue separately to each chip so that the operations are dispatched to each chip independently and the remaining are sent to other chips. When the target chip becomes free, the dynamic scheduler executes the next flash operation addressed to that chip in the queue.

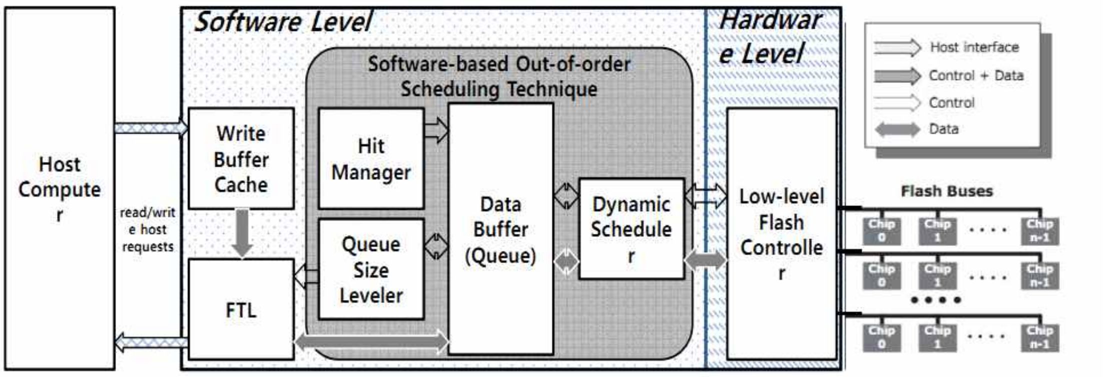


Fig. 5: Architecture of Software-based Out-of-order Scheduling Technique.

The above architecture is almost similar to that of HOST except the reordering process. HOST should create the illusion that flash operations belonging to the same stream have been completed by the use of a reorder process because HOST is located below the FTL. On the other hand, the FTL can communicate to commit a transaction with the file system directly. Hence, SOST is able to ask a request from the file system directly by communication with the FTL and perform an out-of-order execution. For example, the request to the file system after buffering the previous write request at the queue unlike the HOST. Therefore, it does not need to undergo the reordering process.

SOST includes additional functions for the performance improvement. Queue size leveler reallocates the requests flocked to busy chips to other idle chips to decrease the total response time. Hit manager manages the read-hit and overwrite requests by reading buffered data directly or removing them for efficiency improvement.

Section 2. Queue Size Leveler

The size of a target queue indicates the operation time of a target chip and the largest size of all queues refers to the total operation time of all chips. As shown in Fig. 6(b), the total response time of queues is 4 write operation time as the largest size (1 Read request is including 10 read operations of which latency is same as 1 write operation). The Queue Size Leveler (QSL) is designed to balance the sizes of queues for performance improvement.

By this balancing, the QSL can reduce the time that some chips remained free while the others have a lot of work to do. Although the FTL distributes the request data to the NAND flash memory by exploiting bus-level and chip-level parallelism, the deviation of the queues can be reduced by other factors such as the high-locality read request and the long-latency erase request. If the deviation of queues increases, the chance that free chips wait for other busy chips, i.e., the idle time, also increases and this can lead to the increased operation time and performance degradation. The QSL reallocates the remaining pages of busy chips to the queues of free chips and can reduce the idle time of free chips.

In case a chip is free, the dynamic scheduler wakes from the sleep state to execute the request in the queue of the free chip. If the queue of the target chip is empty, the QSL is triggered by the dynamic scheduler. The QSL checks other queues whether they have more than one write request. If the QSL finds that chip, it reallocates a write request to the queue of a free chip by moving it and updating the physical address in the mapping table of the FTL. After the reallocation and searching, the QSL ends its process and the dynamic scheduler executes the reallocated page on the free chip. With this QSL scheme, we can reduce the total response time from 4 write operation time to 2 write operation time in the Fig. 6(c).

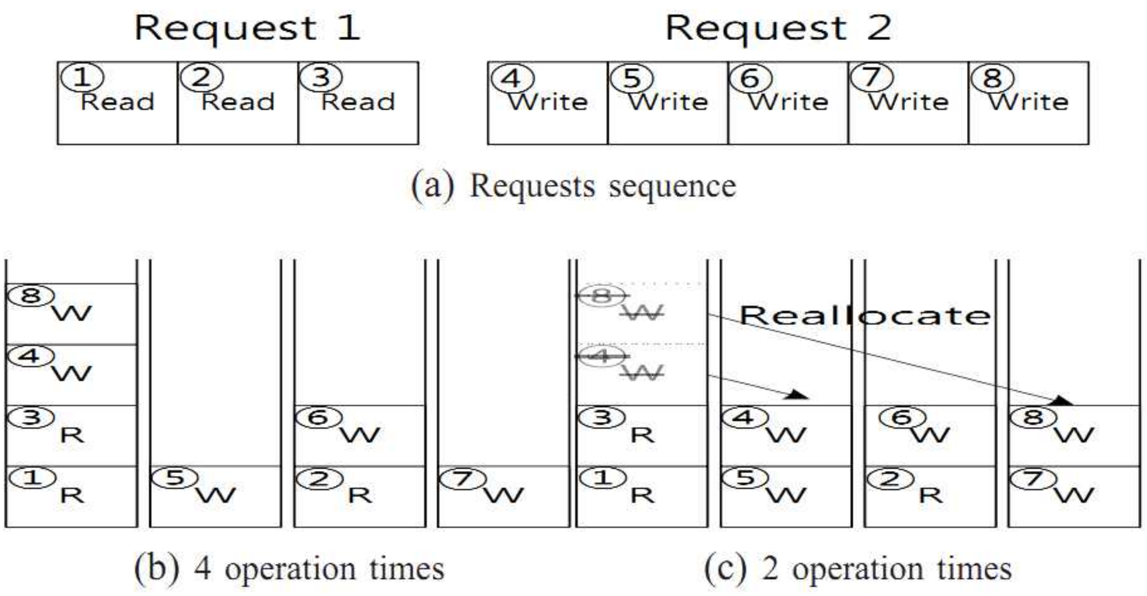


Fig. 6: Queue size leveler.

In order to reallocate the write page from the queue of the busy chip to that of the free one, not only moving the data to the target queue but also updating the address mapping table with the new physical block address in the FTL are needed. These processes of the QSL are not hard tasks for the software FTL. However, it would be much more complicated and costly to do the same tasks for the hardware controller such as the HOST.

We also consider the other scheme that the FTL distributes a write request to the queues of the chips by balancing the sizes of the queues. We call this scheme 'queue-aware FTL' and it has an advantage because of its structure which does not need the searching time unlike the QSL. However, the read and erase requests which increase the deviation of the queue sizes can not be distributed by the FTL due to their locality. Additionally, since they also occur after the allocation of the FTL is completed and the FTL can not reallocate a page when some chips are idle, the performance of the queue-aware FTL becomes worse than that of the QSL. That is mainly why we balance the queue sizes at the queue level rather than at the FTL level.

When we designed the QSL algorithm, we concerned the policy of what request to move to a free chip. In case a chip becomes free and more than one request remains in the queue, the QSL has to choose a request to reallocate. Since the FTL distributes the write requests by exploiting the multichip parallelism, reallocating the write request indicates that it can lead to limit the parallelism as illustrated in Fig. 7. The parallelism limitation of a write request also increases the response time when the read request occurs on the written data. The performance improvement of a single request by the QSL is much higher than the performance degradation of a request by the limitation parallelism because a write response time is five to ten times longer than a read one. However, if the read requests occur more than dozens of times on the reallocated data, there can be a side effect of the QSL that the performance degradation can become more significant than the improvement. In order to minimize the side effect of the QSL, we use the information of the hit ratio which stores the read ratio and is managed by the hit manager, which is referred in the following section, to select the low-read-ratio request to reallocate. Therefore, we add a new policy to select the reallocation page of the read request which will be read less than others in order to minimize the limitation parallelism side effect of the QSL.

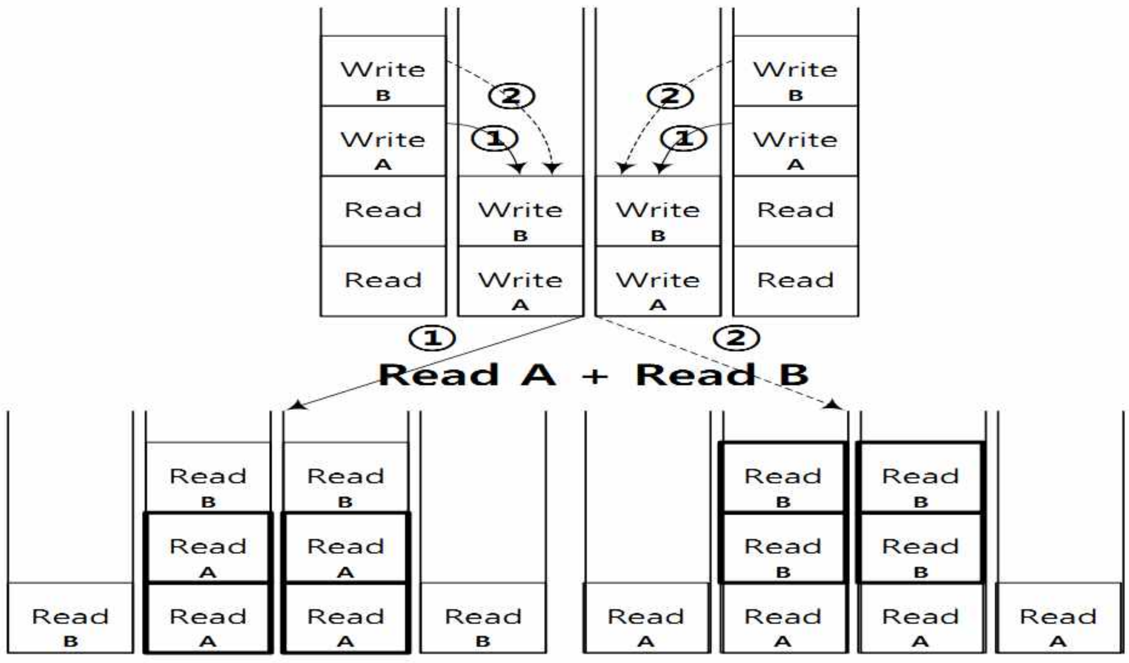


Fig. 7: Effect of queue size leveler.

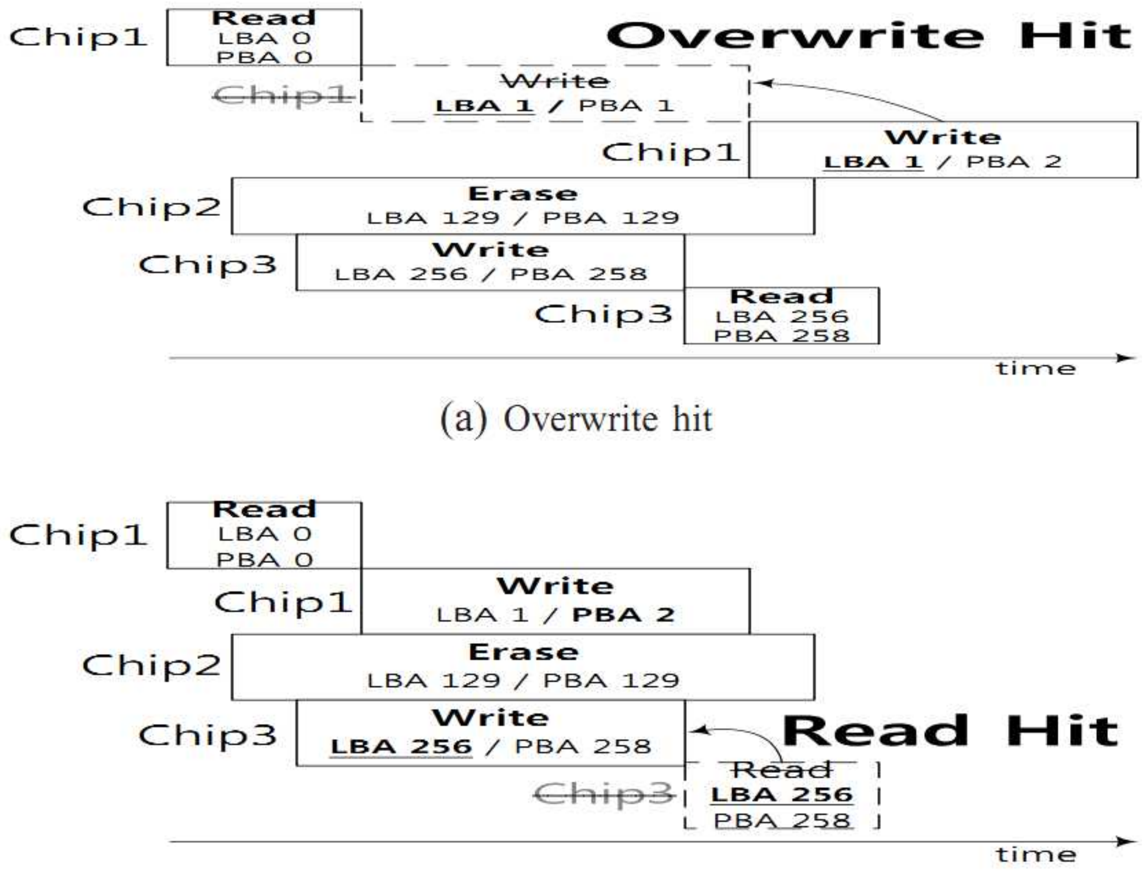
Section 3. Hit Manager

Since queues exist inside the SOST, write requested data are buffered at the queue after an eviction from the WBC. While the WBC tries to keep its data as long as possible, this queue executes an operation of the first page. Although an operation of the first page of this queue is executed whenever any chip becomes available unlike the WBC which tries to keep its data as long as possible, We can improve the performance by managing read and write requests for the data already stored in the queue during the time when the data remain in the queue.

The data of the write request mapped by the FTL are buffered in the queue on a page basis. If the read request for the data with the same logical block address, which are already in the queue, comes to the FTL without the hit manager, this read request should wait until the operation of the data in the queue is completed as illustrated in Fig. 8.

In this figure, the LBA means a logical block address. However, if the same situation occurs with the hit manager, there is no need to wait till the previous write request is completed because the hit manager changes the address of the read request data from the NAND flash memory to the D-RAM which stores the data not written yet. With this method, we can reduce the waiting time for both write and read requests because the time taken to read data from the

D-RAM is significantly faster than that from the NAND flash memory.



(a) Overwrite hit

(b) Read hit

Fig. 8: Hit manager.

If the hit ratio of a page in a queue is high, we can regard the data in the page as read-hot data. Handling read-hot data to increase the read hit for performance improvement is the role of the WBC. However, the WBC with an LRU algorithm has some limitations because of restricted space and short flush time. In the hit manager, the data which are evicted from the WBC and read by the following read request several times are returned to the WBC by the policy of the hit manager. Since both the WBC and queues are placed at the FTL level, returning the read-hot data of a write request from the queue to the WBC at the FTL level is much simpler than the same process at the controller level.

In order to move the read-hot data from the queues to the WBC at the FTL level, the hit manager selects read-hot pages in the queues and allows the WBC to evict the same number of pages to the selected read-hot pages in order to secure the space in the WBC. After securing the needed space, the hit manager copies read-hot pages to the WBC and converts the read hit-ratio of the queues into the hit-ratio of the WBC.

In addition, our hit manager manages not only the read hit but also the write hit. If the write request for the data already in the queue with the same logical block address comes to the FTL without the hit manager, the previous write request operates on the NAND flash memory and it becomes invalid before the later write request is executed. However, there is no need to execute the previous write request and make it invalid because the hit manager removes the previous write request from the queue and updates its status of the mapping table as empty. With this, we can reduce the previous write request time and improve the lifetime of the NAND flash memory.

If the LBA of a new read request matches that of a page in the queues, it means that the read hit occurs and the read hit ratio of that LBA accordingly increases. Similarly, if the LBA of a new write request matches that of a page in the queues, it means that the overwrite hit occurs and the overwrite hit ratio of that LBA also increases. In our work, the read hit ratio is used as a criterion for the QSL to select a page to be reallocated to a free chip. Also, a combination of the read hit ratio and the overwrite hit ratio is defined as the total hit ratio, and it is used as a criterion for the user-defined priority handler to classify write requests.

Chapter 4. Experiments

Section 1. Experiments Environment

We implemented both the in-order execution model, the HOST and the SOST using BlueSSD [14]. BlueSSD, which is based on the Zarkov system [15], i.e., an SSD prototype developed by UCSD, is extended to provide more efficient and flexible hardware/software design environments. BlueSSD has a Xilinx XUPV2P FPGA board [16] and one PowerPC405 processor on its board. It consists of 32 flash packages and up to four identical buses. Each of the four buses consists of up to 8 packages and up to 16 chip-enables. The read operation latency of NAND flash chip used in the experiments is 128 to 256 us and the write operation latency is 1.6 to 2.29 ms. The erase operation latency is also 5 ms. The 100 Mb Ethernet controller allows us to transfer data from/to the host PC.

We compared the performances of the in-order execution model, the HOST, and the SOST using BlueSSD. PowerPC405, i.e., the processor of BlueSSD, runs the Linux 2.6.25.3 kernel that is used as a platform for firmware such as an FTL so that the benchmarks can be processed in the Linux environment. We realized the HOST at BlueSSD by extracting traces from the benchmarks and modifying the time stamps and the sequence of the requests.

The benchmarks used for the performance test are bonnie++, postmark, financial1, financial2, and websearch. Financial2 and websearch benchmarks have read-dominant traces, while bonnie++ and financial1 have write-dominant ones. The postmark benchmark has a balanced trace among these benchmarks.

Section 2. Experiment Result

Fig.9 illustrates the total response time for all benchmarks and the results are normalized at one hundred corresponding to the benchmark time of the in-order execution model. In the extreme case, where the workload does not contain read-dominant operations, the SOST performs almost 36% better than the HOST. These results show the effectiveness of both the QSL and the hit manager because the QSL manages high-locality read requests and the hit manager increases the hit ratio of read-hot write data.

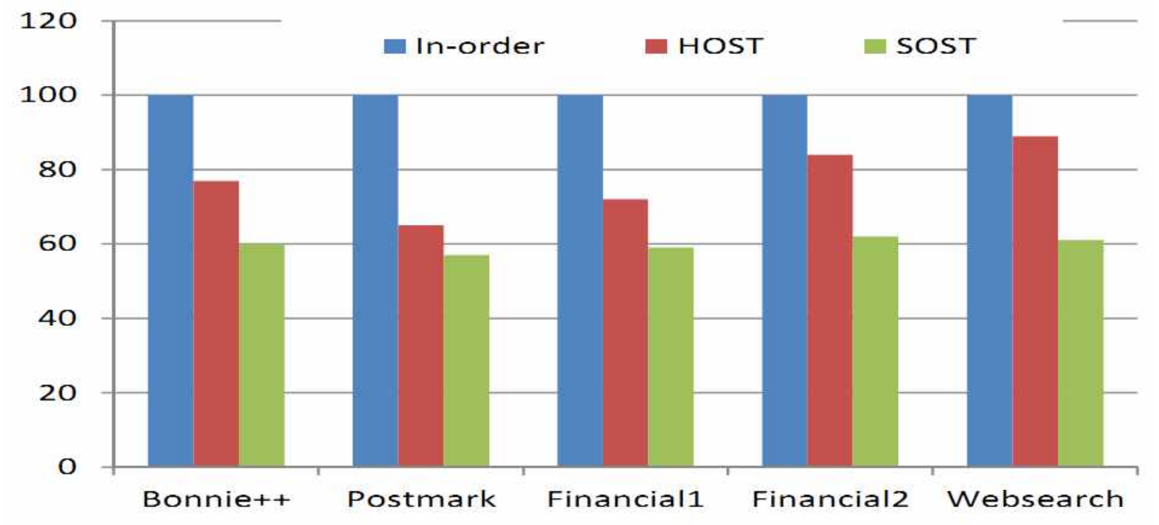


Fig. 9: Performance evaluation results.

Chapter 5. Conclusion

Section 1. Conclusion

We realized the SOST adopting the out-of-order scheme that improves efficiency by maximizing the use of multichip parallelism. The SOST improved the efficiency of the flash memory in the following ways. First, by adopting the queue size leveler, the deviation of the queue size was controlled and the idle time of chips was reduced. This directly resulted in the improved total response time. Second, by realizing the hit manager, unnecessary flash operations on the remaining data in the queue were minimized. This also resulted in the improved total response time while increasing the NAND flash memory lifetime. Third, by adopting the user-defined priority handler, the realistic priority setting at the user level is delivered so that requests can be managed with different priorities at the FTL level to minimize the response time to a target request. Performance evaluation with various benchmarks using an FPGA implementation of the SOST showed that its performance of response time is improved by 14 to 42 percent compared to that of HOST. In addition, we can emphasize that the SOST is more flexible and costs less than the hardware like HOST.

Section 2. Future Work

We plan to furthur apply this out-of-order scheme on the file system to investigate a new optimization model at a higher level. We also consider the software-level optimization for limited resource environments such as mobile system.

References

[1] E. H. Nam, et al., "Ozone (O3): An Out-of-Order Flash Memory Controller Architecture," IEEE Transaction on Computer, vol. 60, no. 5, pp. 653-664, 2011.

[2] J. Lee, et al., "A 1.8V 1Gb NAND Flash Memory with 0.12 um STI Process Technology", in Proceeding of the International Solid-State Circuits Conference, vol. 1, pp 104 - 450, 2002.

[3] M.Goldman, et al., "25nm 64Gb 130mm'2 3bpc NAND Flash Memory", in Proceeding of the International Memory Workshop, 2011.

[4] Samsung Corp., "S470 Series SSD", 2011.

[5] Samsung Corp., "830 Series SSD", 2011.

[6] S.J.Lee, et al., "BlueSSD: An Open Platform for Cross-layer Experiments for NAND Flash-based SSDs", WARP, 2010.

[7] A.M.Caulfield, et al, "Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications", ACM Sigplan Notices, 2009.