SOS: Software-Based Out-of-Order Scheduling for High-Performance NAND Flash-Based SSDs

Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim
Department of Computer Science and Engineering, Seoul National University, Korea
{shanehahn, chamdoo, jihong}@davinci.snu.ac.kr

***Abstract*—We propose an efficient software-based out-of-order scheduling technique, called SOS, for high-performance NAND flash-based SSDs. Unlike an existing hardware-based out-of-order technique, our proposed software-based solution, SOS, can make more efficient out-of-order scheduling decisions by exploiting various mapping information and I/O access characteristics obtained from the flash translation layer (FTL) software. Furthermore, SOS can avoid unnecessary hardware-level operations and manage I/O request rearrangements more efficiently, thus maximizing the multiple-chip parallelism of SSDs. Experimental results on a prototype SSD show that SOS is effective in improving the overall SSD performance, lowering the average I/O response time by up to 42% over a hardware-based out-of-order flash controller.**

I. Introduction

NAND flash-based SSDs are expected to replace a large share of the traditional HDD market in the near future because of their several advantages (such as low power consumption, high performance, and high shock resistance) over HDDs. In particular, modern high-end SSDs achieve impressive I/O performance, outperforming HDDs by an order of magnitude. For example, the I/O bandwidth of PCIe SSDs is more than 12 times higher than enterprise HDDs. Furthermore, the high cost of SSDs, which was considered a major disadvantage of SSDs, has been steadily lowered due to the continuous shrinks of the NAND fabrication process as well as the increased bit density per single memory cell.

In order to achieve high performance, modern high-end SSDs exploit multiple-chip parallelism aggressively, since the bandwidth of a single NAND flash chip is limited to several tens of megabytes (e.g., 40 MB/s for writes) [1]. By employing a multi-channel and multi-way architecture that allows several flash chips to operate concurrently, host I/O requests can be distributed to multiple chips and executed in parallel. Therefore, host I/O requests can be serviced quickly.

To decide the execution order of multiple host I/O requests sent to an SSD, two different approaches are used [1], [2]. In-order scheduled SSDs service incoming I/O requests in the order of their arrival times, whereas Out-of-order scheduled SSDs service I/O requests by their data dependencies. By more aggressively exploiting I/O parallelism, using out-of-order scheduling can achieve higher performance over in-order scheduling. Out-of-order scheduling is usually implemented by a hardware controller because software implementation can cause a sequential performance bottleneck, which, in turn, can degrade an achievable speed-up from multiple chips [3].

In this paper, we show that, in modern high-end SSDs, software-based out-of-order I/O scheduling can be more efficient than hardware-based out-of-order scheduling. There are two main motivations for software-based out-of-order scheduling. First, if out-of-order scheduling is supported by software, multiple-chip parallelism can be more efficiently exploited. Furthermore, software-based out-of-order scheduling can eliminate unnecessary write operations more easily by exploiting available information at the software level. On the other hand, hardware-based out-of-order controller makes decisions only with limited hardware-level information only, and thus it can miss many potential optimization opportunities during runtime. Second, even though software-based out-of-order scheduling incurs higher computational overheads than hardware-based one, its performance penalty is not significant. Furthermore, the performance benefit of improving I/O parallelism is sufficient enough to outweigh the performance penalty caused by software overheads.

Based on these observations, we propose a **s**oftware-based **o**ut-of-order **s**cheduling technique, called SOS, which performs out-of-order scheduling at the software level. SOS increases the parallelism of multiple chips by rearranging I/O requests stored in storage queues using the mapping table information at the software level. This helps us to balance the size of multiple I/O queues, thereby improving the overall performance of SSDs. SOS also eliminates unnecessary write operations by preventing useless write requests for the same logical page from being serviced. This benefit can be realized by exploiting both logical and physical page information available at the software level, which cannot be identified at the hardware level. We implemented the proposed SOS technique in an FPGA-based SSD prototype [4], and then evaluated its effectiveness, using various benchmark programs. Our evaluation results show that SOS reduces I/O response time by up to 42% over hardware-based out-of-order scheduling, while incurring negligible computational overheads.

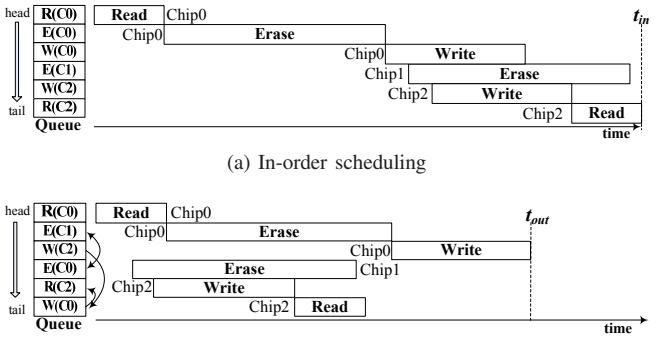
This paper is organized as follows. In Section II, we first review in-order and out-of-order scheduling with a main focus on a hardware-based out-of-order scheduling technique. In Section III, we explain how SOS overcomes the limitations of a hardware-based approach. Experimental results are given in Section IV, and we conclude with a summary in Section V.

II. Out-of-Order Scheduling in SSDs

A. In-Order Scheduling and Out-of-Order Scheduling

The out-of-order scheduling technique is proposed to improve the parallelism of SSDs by relaxing the ordering constraints of multiple I/O requests. The in-order scheduling technique assigns I/O requests to idle flash chips in the order of their arrival times. On the other hand, as long as data dependencies are not violated, out-of-order scheduling rearranges

978-1-4799-0218-7/13/$31.00



(a) In-order scheduling

(b) Out-of-order scheduling

Fig. 1: A comparison of in-order and out-order scheduling.

I/O requests in a storage queue based on the latencies of I/O requests, thus greatly improving the parallelism.

Fig. 1 compares out-order scheduling with in-order scheduling. Suppose that there are six I/O requests in a storage queue: `R(C0)`, `E(C0)`, `W(C0)`, `E(C1)`, `W(C2)`, and `R(C2)`. Here, `R`, `W`, and `E` indicate the types of flash operations, i.e., page read, page write, and block erase operations, respectively. `C0`, `C1`, and `C2` indicate target chips (i.e., `Chip0`, `Chip1`, and `Chip2`, respectively) where I/O requests are to be executed. As depicted in Fig. 1(a), the in-order scheduling technique picks up I/O requests from a storage queue in the order of their arrival times and then assigns them to a target chip. If the chip is busy, it waits until the chip becomes idle. Unlike in-order scheduling, out-of-order scheduling changes the positions of I/O requests in a storage queue, as shown in Fig. 1(b), according to their latencies, and then issues them to target chips. As a result, out-of-order scheduling reduces the completion time of I/O requests by t_{in} - t_{out}.

The proposed SOS technique is based on out-of-order scheduling which has been rapidly adopted in recent SSDs because of its superior performance. In the following subsection, we describe the hardware-based out-of-order technique in detail.

B. Hardware-Based Out-of-Order and Its Limitations

Fig. 2 illustrates the overall architecture of the hardware-based out-of-order scheduling module which we call HOS shortly. The host system issues read and write requests (host I/O requests) with the logical addresses of a file system. After receiving host I/O requests, the flash translation layer (FTL), which is an intermediate software layer between a host system and flash chips [1], divides host I/O requests into several flash requests with a page size. Then, the FTL performs address translation that maps the logical page addresses (LPAs) of flash requests to physical page addresses (PPAs) in flash memory. Logical-to-physical mapping information is kept in the mapping table of the FTL. This address translation is necessary to overcome the 'out-place-update' nature of flash memory [1]. A target chip in which flash requests are to be served is decided by the FTL during address translation. The FTL delivers flash requests with *physical addresses* to HOS on the hardware side. For the efficient support of out-of-order scheduling, HOS maintains multiple queues which are separately managed for individual flash chips. After receiving flash requests from the FTL, HOS inserts them to corresponding hardware queues.

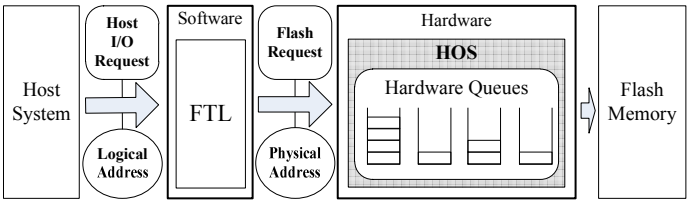


Fig. 2: An overall architecture of an SSD with a hardware-based out-of-order scheduling module.

Whenever an idle chip is found, HOS fetches a flash request from a corresponding queue and executes it on a target chip.

Even though HOS exhibits higher I/O parallelism than the in-order scheduling technique, it has two types of serious problems that limit the overall performance of SSDs.

Skewed queue problem: HOS cannot fully exploit the potential parallelism of multiple chips. In HOS, write requests can be distributed as evenly as possible because the FTL can decide a target chip where requested data are to be written during address translation. However, (1) when read requests whose target chips are already decided are flocked in certain hardware queues or (2) when block erase operations (triggered by garbage collection) that take a long time for completion are placed in hardware queues, the lengths of hardware queues are inevitably skewed. In this paper, we call it a *skewed queue problem*. If a skewed queue problem is frequently observed, it means that multiple chips in a storage device are not fully utilized. Thus, the performance of SSDs is greatly degraded.

Skewed queue problems occur frequently as the number of channels and ways in a flash device increases (i.e., as the number of flash chips that operate in parallel increases). For example, flash chips that remain idle are rarely observed when there are relatively small numbers of flash chips. However, the chip utilization is greatly lowered as the number of flash chips increases (see Fig. 7). Considering that recent SSDs employ more flash chips to overcome the decreasing performance of high-density NAND flash memory, this skewed queue problem will be a serious bottleneck that limits the SSD performance.

One of the promising approaches that address the skewed queue problem is to move write requests from busy queues (i.e., queues that contain lots of requests) to non-busy queues (i.e., queues that contain few or no requests). However, this rearrangement of flash requests involves some modifications on the mapping table which is managed by the FTL. This is because the physical page addresses of flash requests must be changed before they are moved to different queues (or different flash chips). As depicted in Fig. 2, the FTL and HOS are operated separately, and thus it is difficult for HOS to update the mapping table in the FTL.

Useless write problem: In out-of-order scheduling, the FTL sends write requests to HOS as soon as possible so that host I/O requests are rapidly processed [3]. This improves user-perceived performance by maintaining sufficient room in hardware queues which can be used for temporarily storing bursty write requests issued during a short time period. However, it inevitably incurs lots of useless page writes. For example, suppose that a write request `W0` for a logical page '`101`' is issued from a host system. The FTL assigns `W0` to `Chip0` using its own allocation strategy (e.g., a round-robin policy),

quickly forwarding it to HOS. Further suppose that another write request `W1` for the same logical page '`101`' is issued again before `W0` is materialized to `Chip0`. The target chip of `W1` is decided to be `Chip1` by the FTL, and it is sent to HOS. In that case, `W0` is uselessly written to `Chip0` because its data become obsolete due to the new data of `W1`. We call this problem a *useless write problem.*

As depicted in Fig. 2, HOS manages flash requests in hardware queues using only their physical addresses (that specify a chip, a block, and a page where flash requests are served). Thus, HOS does not know the logical page numbers of flash requests in hardware queues. From the perspective of HOS, `W0` is different from `W1` even if they have data for the same logical page '`101`'. For this reason, HOS cannot eliminate useless page writes (i.e., `W0`). If the FTL informs HOS which request is a duplicate one, HOS can remove useless page writes. But, it is also infeasible because the FTL is unaware of the status of hardware queues; that is, the FTL does not know which flash requests still stay in hardware queues.

Both of skewed queue problem and useless write problem could be addressed if more advanced functions are added to hardware modules. For example, if all the functions of the FTL are implemented as hardware, the information sharing between a scheduling module and a mapping module can be easily done. This approach, however, greatly increases the hardware cost. Furthermore, it is still unknown whether the efficient hardware implementation of the FTL is possible or not. As another candidate, more rich interfaces can be added between the FTL and HOS for more efficient information sharing between two modules. However, this will increase the design complexity of a storage device. For instance, storage designers need to consider more complicated hardware/software co-design issues.

III. Design and Implementation of SOS

Software-based out-of-order scheduling is a feasible solution that addresses both the skewed queue problem and the useless write problems without additional hardware resources and high design cost. If the out-of-order scheduling module is implemented as software, it can directly access the mapping table of the FTL, overcoming the skewed queue problem. Thus, the inter-queue request rearrangement can be easily made. The useless write problem can be easily addressed as well because the out-of-order module figures out which flash requests are useless by looking up software queues and the mapping table in the FTL.

Running the out-of-order module as software inevitably incurs additional computational overheads. According to our implementation study, however, the its overheads were not significant; the time taken for executing the software out-of-order module was less than 10 μsec per flash request even in an embedded processor running at 400 MHz. Considering the benefits of increasing the performance of a storage device, software overheads are negligible and cannot be a serious obstacle in realizing the benefits of software-based out-of-order scheduling.

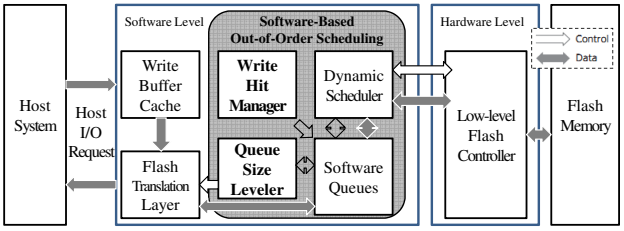


Fig. 3: An overall architecture of an SSD with software-based out-of-order scheduling

A. Overall Architecture of SOS

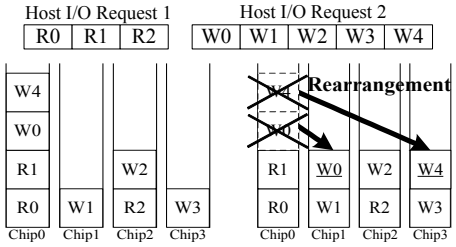
Fig. 3 shows the overall architecture of an SSD with the proposed SOS technique. The data of a host I/O request arrive from a host system are stored in a write buffer cache first, and then the FTL divides host data into several flash requests with a page size. After mapping the logical addresses of flash requests to physical addresses in NAND flash memory, the FTL stores the data of flash requests to corresponding queues. The dynamic scheduler monitors the status of flash chips. Whenever an idle chip is observed, the dynamic scheduler dispatches a flash request from a corresponding queue and then sends it to the low-level flash controller responsible for executing flash requests on target chips.

The two modules, a queue size leveler and a write hit manager, have important roles in SOS. First, the queue size leveler detects a skewed queue problem and then rearranges flash requests so that they are evenly distributed across queues. On the other hand, the write hit manager is designed to eliminate useless write problems by canceling unnecessary writes.

B. Queue Size Leveler

The queue size leveler (QSL) is designed to balance the size of multiple I/O queues. Although the FTL distributes flash write requests to multiple chips as evenly as possible by considering I/O parallelism, the deviation of the size of multiple I/O queues becomes significantly high due to the following two reasons: (1) localized read requests and (2) block erase requests. First, unlike write requests, read requests must be served by a flash chip where requested data have been written before. For this reason, the FTL cannot decide target chips where read requests are served during address translation. If lots of read requests flock to a certain queue, other requests in the same queue are inevitably delayed. Second, a block erase operation that is triggered by garbage collection takes a very long time (e.g., 3 ms) in comparison to read (e.g., 400 μs) and write (e.g., 1 ms) operations. Thus, if a block erase operation is placed on a certain queue, newly arrived requests must wait until a block erase operation is completed. As a result, the size of a queue increases.

The queue size leveler of SOS solves such a skewed queue problem because write requests can be easily moved to other queues by modifying the mapping table of the FTL even if they are already enqueued. Some readers may think that the FTL in HOS can handle the skewed queue problem effectively if the FTL knows the status of hardware queues; the FTL can assign incoming flash requests to non-busy queues, so as to make the distribution of flash requests on queues more even.



(a) Before I/O request rearrangements (b) After I/O request rearrangements

Fig. 4: An example of resolving the skewed queue problem.

However, this cannot be an ultimate solution to the skewed queue problem. Suppose that lots of localized read requests come after many writes are added to hardware queues. In that case, the skewed queue problem cannot be avoided because write requests already inserted into queues cannot be moved to other queues in HOS.

In SOS, QSL is triggered by the dynamic scheduler whenever an empty queue is found. QSL sees if there are other queues that have more than one write request. If it finds write requests, it moves to an empty queue after an update on the physical page address in the mapping table. Then, the dynamic scheduler forwards the newly moved request to the low-level flash controller.

Fig. 4 illustrates an example of how QSL addresses the skewed queue problem. There are two host I/O requests. One is a read request and the other is a write request. Fig. 4(a) shows that the read request consists of three flash read operations, `R0`, `R1`, and `R2`. The write request requires five flash write operations, `W0`, `W1`, `W2`, `W3`, and `W4`. We assume that there are four flash chips, `Chip0`, `Chip1`, `Chip2`, and `Chip3`, which can operate in parallel. Fig. 4(b) illustrates that two of the three flash read requests, `R0` and `R1`, flock to `Chip1` and the five flash write requests are evenly distributed to four chips in a round-robin fashion. Thus, the completion of the host write request is delayed until flash write requests in `Chip0` are finished. As shown in Fig. 4(c), with QSL, two flash write requests, `W0` and `W4`, are moved from `Chip0` to `Chip1` and `Chip2`, and the time taken to complete the host write request can be reduced by two write operation times.

C. Write Hit Manager

In order to prevent obsolete data from being written uselessly, the write hit manager (WHM) of the SOS technique detects duplicate writes and eliminates them so that they are not unnecessarily written to flash memory.

When a write request arrives, the FTL first divides a host write request into several flash write requests with a page size and then performs address translation that decides a physical page address where a requested page is to be written. The mapping table in the FTL is composed of 'logical-to-physical' mapping entries that indicate physical page addresses (PPAs) to which logical page addresses (LPAs) are mapped. Each mapping entry has an additional flag, called a *completion flag*, which indicates whether or not a requested logical page is completely written to a physical page. Before a flash write request is enqueued into a queue, the completion flag of a logical page entry in the mapping table is set to '0'. After the flash write request is finished, the completion flag is set to '1'.

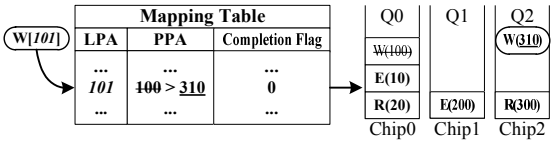


Fig. 5: An example of eliminating a useless write operation.

When an overwrite occurs on a certain logical page, the completion flag of that page can be '0' or '1'. If the completion flag is '1', the FTL inserts a flash write request into a target queue after address translation, making the completion flag '0'. On the other hand, if the completion flag is '1', it means that a flash write request for the same logical page is already stored in a certain queue. In this case, the FTL invokes the write hit manager, WHM, and then passes the physical page address of a previously requested page to WHM. Using this physical page address, WHM easily knows which queue has a previously requested page. WHM then removes that page from a corresponding queue. After removing an unnecessary page write request, the FTL inserts a newly arrived write request into a queue, updating the mapping table.

Fig. 5 illustrates how WHM eliminates unnecessary page writes using a simple scenario. Here, we assume that there are three flash chips, denoted by `Chip0`, `Chip1`, and `Chip2`, and three queues `Q0`, `Q1`, and `Q2` for `Chip0`, `Chip1`, and `Chip2`, respectively. `Q0` has three flash requests `R(20)`, `E(10)`, and `W(100)`, while `Q1` contains one flash request `E(200)`. `Q2` has one flash request `R(300)`. `R`, `W`, and `E` indicate page read, page write, and block erase operations, respectively. The number in the parenthesis refers to a physical page number for page read or write operations and, for a block erase operation, this number indicates a physical block number.

Suppose that a page write request for a logical page '101' comes to the FTL. The FTL first looks up the completion flag of the logical page '101' in the mapping table. As depicted in Fig. 5, the completion flag is '0'. This means that a write request for the logical page '101' already exists in a queue and it is not finished yet. The FTL easily figures out the physical page address (i.e., 100) of the previously requested page, and then informs WHM of that number so that it eliminates the obsolete page. As shown in Fig. 5, WHM notices that `W(100)` in `Q0` has obsolete data for the logical page '101' and removes it from `Q0`. Finally, the FTL maps a newly arrived page to a physical page '301' and sends it to `Q2`.

IV. Experimental Results

A. Experimental Setup

In order to evaluate the performance of the proposed techniques on a real platform, we implemented the proposed software-based out-of-order scheduling technique in our FPGA-based SSD prototype, called BlueSSD [4]. BlueSSD supports 4 buses and 4 ways, so it has 16 flash chips that can operate in parallel. A NAND flash memory chip used for our evaluation is Micron's MT29F4G08ABADAWP. A block contains 64 pages and the size of a page is 2 KB.

The benchmarks used for our evaluation include two micro-benchmarks, *Bonnie++* and *Postmark*, and three real-world

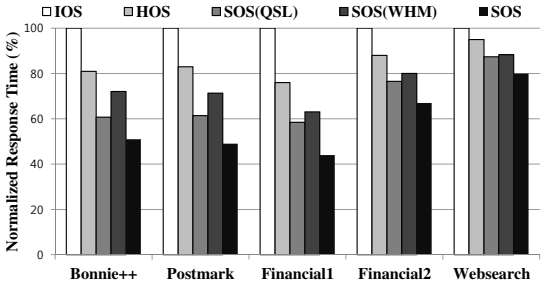


Fig. 6: Normalized I/O response times over IOS.

workloads, *Financial1*, *Financial2*, and *WebSearch*. To repeat the same workload patterns under a variety of storage configurations, we extracted block-level I/O traces while running benchmarks in a real system and then replayed them in our prototype SSD platform for evaluation.

We compared the performance of the proposed SOS technique with two existing techniques, in-order scheduling and hardware-based out-of-order scheduling techniques which are denoted as IOS and HOS, respectively. In our evaluation, IOS is implemented as a software module. Implementing the hardware modules of HOS is not easy due to its inherent characteristic. Thus, we realized HOS by rearranging the sequences of I/O requests in I/O traces according to the out-of-order scheduling algorithm. The rearranged I/O traces were replayed, using the in-order scheduling algorithm. This evaluation methodology was somewhat advantageous to HOS because the performance overhead caused by out-of-order scheduling was excluded.

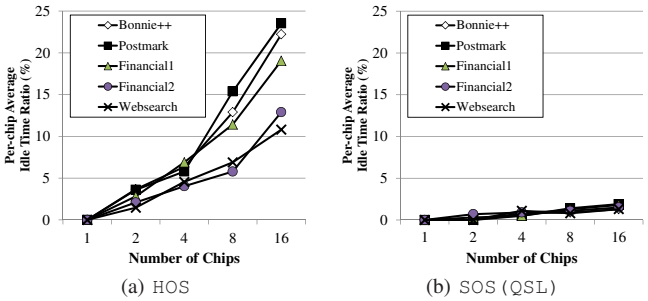
B. Performance Evaluation

Fig. 6 shows the I/O response times under five different SSD settings, which were normalized to IOS. Here, SOS(QSL) is SOS with the queue size leveler (QSL), while SOS(WHM) is SOS with the write hit manager (WHM). SOS is SOS with both QSL and WHM. 4 buses and 4 channels were used for our evaluation, so that 16 flash chips can be operated concurrently.

As depicted in Fig. 6, SOS(QSL) exhibited 16% to 42% lower I/O response time than that of HOS. SOS(QSL) outperformed IOS by up to 56%. As expected, this performance benefit of SOS(QSL) can be realized by improving the parallelism of multiple chips in a storage device. For a more detailed analysis, we analyzed the ratio of the per-chip average idle time to the benchmark execution time. Here, the per-chip idle time is the length of the time during which a flash chip remains idle while other flash chips are busy. Fig. 7 shows our experimental results while varying the number of flash chips from 1 to 16. The higher the per-chip average idle time ratio, the more the skewed queue problem was observed which reduces the multiple-chip parallelism. When the number of flash chips was 1, the skewed queue problem was rarely observed in HOS and SOS(QSL). However, in the case of HOS, as the number of flash chips increased to 16, the per-chip average idle time ratio sharply increased by up to 23.5%. Conversely, the degree of I/O parallelism was maintained in

Benchmark	*Bonnie++*	*Postmark*	*Financial1*	*Financial2*	*WebSearch*
Hit ratio	11.7%	14.3%	17.6%	9.2%	7.1%

TABLE I: Write hit ratios with SOS(WHM).



(a) HOS (b) SOS(QSL)

Fig. 7: Per-chip average idle time ratio.

SOS(QSL) regardless of the number of flash chips; the idle time ratio was lower than 2%.

Fig. 6 shows that SOS(WHM) improves I/O response times by 5% to 17% over HOS by eliminating the number of useless page writes. We analyzed how many page writes are canceled by the write hit manager in storage queues. Table I displays the write hit ratios of five different benchmarks. Here, the write hit ratio represents the ratio of the number of eliminated page writes to the number of original page writes sent from a host system. As shown in Table I, we observed that 7.1%-17.6% of original page writes can be removed by SOS(WHM).

We finally evaluated the performance improvement when two different techniques, QSL and WHM, are used together. The benefits of two techniques were maintained when they were simply integrated. As shown in Fig. 6, the integrated version of QSL and WHM, SOS, reduced I/O response time by up to 42% and 56% over HOS and IOS, respectively.

V. Conclusion

We proposed a new software-based out-of-order technique, called SOS. We introduced two kinds of serious problems, a skewed queue problem and a useless write problem, which inevitably occur when a scheduler's decision is made only at the hardware level. By effectively exploiting the information available at the software level, the proposed SOS technique overcomes the problems of hardware-based out-of-order scheduling without significant computational overheads. Our evaluation results showed that SOS improved I/O response times by up to 42% over HOS.

VI. Acknowledgement

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Ministry of Education, Science and Technology (MEST) (No. 2012-0006417). This research was supported by WCU (World Class University) program through the National Research Foundation of Korea funded by the Ministry of Education, Science and Technology (R33-2012-000-10095-0). The ICT at Seoul National University and IDEC provided research facilities for this study.

References

[1] N. Agrawal et al.,"Design Tradeoffs for SSD Performance," *in Proc. USENIX Annual Technical Conference*, 2008.

[2] J. U. Kang et al.,"A Multi-Channel Architecture for High-Performance NAND Flash-Based Storage System," *J. Systems Architecture*, vol. 53, no. 9, pp. 644-658, 2007.

[3] E. H. Nam et al.,"Ozone (O3): An Out-of-Order Flash Memory Controller Architecture," *IEEE Trans. Computers*, vol. 60, no. 5, pp. 653-666, 2011.

[4] S. Lee et al.,"BlueSSD: An Open Platform for Cross-Layer Experiments for NAND Flash-Based SSDs," *in Proc. International Workshop on Architectural Research Prototyping*, 2010.