Unblinding the OS to Optimize User-Perceived Flash SSD Latency

Woong Shin
Seoul National University

Jaehyun Park
Arizona State University

Heon Y. Yeom
Seoul National University

Abstract

In this paper, we present a flash solid-state drive (SSD) optimization that provides hints of SSD internal behaviors, such as device I/O time and buffer activities, to the OS in order to mitigate the impact of I/O completion scheduling delays. The hints enable the OS to make reliable latency predictions of each I/O request so that the OS can make accurate scheduling decisions when to yield or block (busy wait) the CPU, ultimately improving user-perceived I/O performance. This was achieved by implementing latency predictors supported with an SSD I/O behavior tracker within the SSD that tracks I/O behavior at the level of internal resources, such as DRAM buffers or NAND chips. Evaluations with an SSD prototype based on a Xilinx Zynq-7000 FPGA and MLC flash chips showed that our optimizations enabled the OS to mask the scheduling delays without severely impacting system parallelism compared to prior I/O completion methods.

1 Introduction

Flash memory technology, in the form of flash solid-state drives (flash SSDs), is steadily replacing prior storage technology based on the value of affordable microsecond-level random access memory. However, the user-experienced performance of these SSDs has yet to reach its full potential since the time spent in the software portion of the I/O path has been magnified [5, 4, 7].

These software overheads were initially noticed in studies of next-generation memory technologies, such as PCM and STT-MRAM, which are expected to give nanosecond-scaled latencies, and ignited several efforts to address the problem [3, 9, 8, 6]. Yet, these studies do not apply well when considering flash SSDs because the latency of flash memory technology scales at microseconds. This large latency requires a significant amount of parallelism in both SSDs and host systems in order to benefit the users.

Flash memory latency, whether in nanoseconds or milliseconds, makes it difficult to decide whether to yield the CPU or not when the CPU awaits I/O completion since the scheduling delay can cause a non-negligible impact on performance. Blocking the CPU (i.e., polling [9]) would avoid such delays but would be at the cost of sacrificing parallelism. Recently, internal parallelism of SSDs (NAND channels, chips, dies, and planes) has been increased to meet the demands on performance, capacity, and costs. To utilize the capability of SSDs fully, the OS should multiplex higher numbers of I/O contexts (i.e., threads or state machines). This, in turn, increases the chance of scheduling.

We can minimize these scheduling delays based on accurate estimation of SSD latencies. However, queueing delays caused by high parallelism in SSDs and internal operations, such as garbage collection, make it impossible. To address this issue, we propose an optimization that enables the OS to make precise decisions on when to yield the CPU or not upon a new I/O request. The optimization eliminates or hides I/O completion scheduling delays while preserving system parallelism. This is done by placing latency predictors supported by an I/O behavior tracker inside the SSD. The tracker gathers information about the whereabouts of each I/O request and the state of each internal resource, such as DRAM buffers and NAND chips.

Here, latency predictors either aid the OS in determining the latency of the next I/O request or interrupt the OS when a pending I/O would finish in the near future. With such information, the OS can make decisions on whether to yield the CPU or not, or it can prepare itself to overlap the I/O time with the expected I/O completion scheduling delays. Such H/W and S/W interactions are done with an extended SSD interface, implemented as an in-band channel that is piggybacked on I/O completion paths.

To evaluate our proposal, we employed a Xilinx Zynq-

7000 SoC FPGA-based OpenSSD 2 Cosmos evaluation board [2] accompanied with a flash DIMM module based on MLC technology [1]. Evaluations on a prototype SSD showed that our method was capable of reducing the impact of scheduling delays while having a low impact on system parallelism.

2 Background and Motivation

In this paper, we aim to optimize user-perceived latency of flash SSDs by minimizing I/O completion scheduling delays without sacrificing system parallelism. Here, we observed that the fact that the OS was blind to the levels of SSD latency that it would experience was the main obstacle of tackling such delays. This motivated us to explore the design space in which the SSD actively inform the OS of its behavior upon performing system optimizations.

2.1 System Impact of Modern SSDs

Modern SSDs employ a significant amount of parallelism in order to keep up with the value of a high-performance random access storage device. Careless I/O control results in low resource utilization with hotspots. SSDs employ various techniques to spread I/O requests to achieve maximum utilization and performance. DRAM buffers, backed with high-capacity capacitors, are employed to serve as staging areas to perform such optimizations. Such performance considerations run deep in SSD design and have an impact on various SSD internal I/O tasks.

The higher degree of parallelism of modern SSDs burdens the host system with context-multiplexing overheads that introduce non-negligible scheduling delays. Multiple I/O contexts competing for CPU cycles vary user-perceived scheduling delays. SSD internal tasks also cause significant variability in latency observed from the host system. Even with the presence of DRAM hits or NAND read operations, which have (fairly) favorable latencies, the OS has to assume higher levels of latencies and neglect any optimizations based on lower latencies.

2.2 SSDs, Unblinding the OS

Blindness of the OS so that it is unaware of SSD latencies is the root of all evil that leads to suboptimal conservative approaches. For example, blocking a CPU core in the case of a DRAM buffer (or cache) hit (Figure 1-1 left) would eliminate the scheduling delay, but the OS yields the CPU assuming much higher latencies (Figure 1-1 right) and takes the penalties of scheduling delays because it has no information on such hits (at the question marks in Figure 1-1).

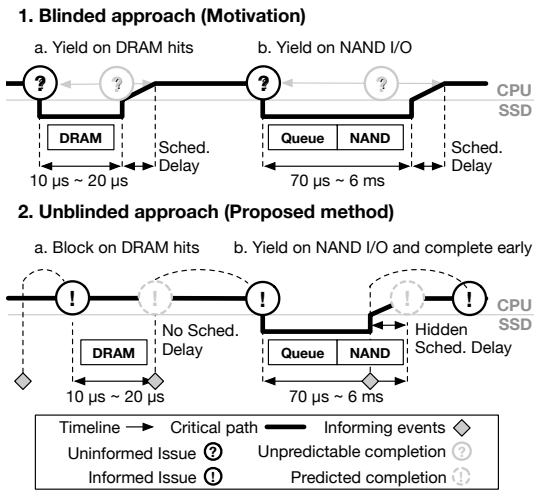

Figure 1: Minimizing the impact of I/O completion scheduling delays by having SSDs actively inform the OS

What if we have predictable latency? The negative impacts of scheduling delays can be minimized since we can make a best decision that benefits the system based on an accurately predicted latency (Figure 1-2). To this end, we were motivated to achieve predictable latency.

In this work, we positioned ourselves to define such predictability as being able to predict what comes next instead of trying to make SSD latencies adhere to a constant latency value. Here, we unblind the OS by informing it about SSD internals to enable accurate latency predictions based on such information. To achieve this, our proposal is to reinforce SSDs to inform the OS with appropriate information. The OS is informed to make predictions (at the bold exclamation marks) of the expected completion time of an I/O (the gray exclamation marks), making it possible to eliminate (Figure 1-2 left) or mask (Figure 1-2 right) the impact of scheduling delays from the critical path.

3 Design and Implementation

Based on our motivations, we implemented an I/O path in which the SSD actively cooperates with the OS in order to optimize user-perceived performance. The goal of the cooperation is to enable proper decisions, whether yielding or blocking a CPU upon an I/O request would be beneficial. Such cooperation is based on an accurate prediction of SSD latencies, which lies as the main challenge in our work. Our main strategy to the challenge is to predict the latencies within SSDs, not outside SSDs.

To achieve this, the I/O path has an I/O behavior tracker within the SSD controller S/W, which speaks

to the predictor in the OS device driver through an extended SSD interface. Our I/O path is based on modest changes only that are limited to S/W components and implemented both in the host OS and the SSD controller.

3.1 Predicting the I/O Time of SSDs

The most challenging part of our design is predicting the behavior of SSDs, which is highly variable. Our approach to this problem is to decompose SSD internals into individual components (DRAM buffers and NAND chips), each behaving in a simple way (compared to the whole system), and exploit the simple behavior to ease the prediction. This prediction activity is based on a simple model of SSD internals depicted in Figure 2-1 (a), where I/O requests first visit the DRAM buffer for opportunities of caching (reads) or aggregating (writes) and then are issued to the NAND array.

1) Classifying I/O Requests: In the I/O path, each I/O request is classified in terms of its destined components, and the prediction is based on the previous behavior of the individual components. The I/O behavior tracker, implemented within the SSD, tracks these component behaviors, which are translated into parameters of multiple behavioral models, each representing individual components (detailed in Section 3.2). Based on the models and the parameters gathered by the tracker, the OS classifies (predicts) the next I/O request at the I/O issue context (Figure 2-1 (a)) based on the criteria, as shown in Figure 2-4.

2) Remaining I/O Time: Even with the power of accessing internals of SSDs from an SSD controller, predicting the I/O time of an SSD is still challenging. While there are components with predictable latency, such as DRAM buffers, that a simple classification can help (upper row in Figure 2-4), predicting latencies of a NAND chip array (lower row in Figure 2-4) is difficult with the presence of multiple I/O requests colliding and queued up on resources (Figure 2-3 (i)), along with the inherent variability of the chips (Figure 2-3 (n)). To overcome the challenge, a predictor within the SSD considers only the remaining part of the I/O time (Figure 2-3 (m)) and predicts only for small-read operations, which have low variance in NAND I/O latency (Figure 2-2 (n)). Latency prediction begins only after a NAND I/O command (a single page read) is actually issued to a NAND chip (the beginning of Figure 2-3 (m)), effectively eliminating the queuing delays (Figure 2-3 (i)) from the prediction landscape. For cases when predictions can be inaccurate (write operations) or have marginal benefits (larger I/O), the predictor simply falls back to not predicting anything.

3) Precompletion: For the remaining I/O time, the SSD side latency predictor takes the moving average of three observed latency values as the prediction and notifies the host OS of a predefined period (precompletion window in Figure 2-3 (l)) before (Figure 2-1 (d)) the actual completion occurs (Figure 2-1 (e)).

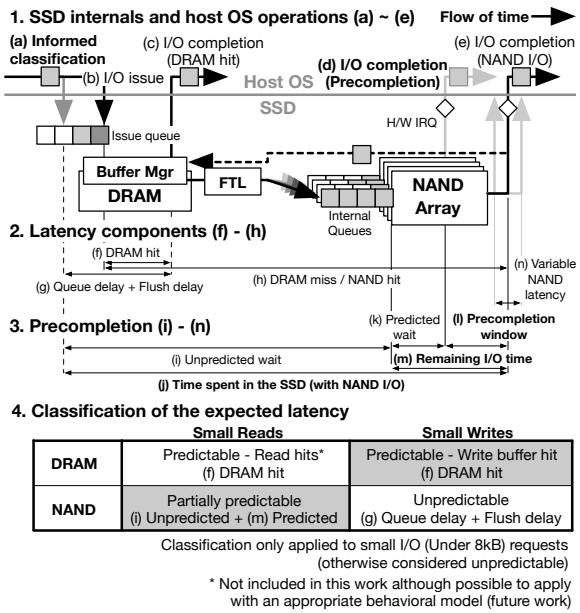

Figure 2: Our proposal is based on a simplified model of the internals of an SSD having a buffer/cache and an array of NAND chips (Time flows left to right, not at scale)

3.2 OS I/O Path Optimizations

The simple behavior of each individual component is modeled with coarse-grained implementation neutral behavioral models (Figure 3), which serve as an agreement between the latency predictor and the OS, in order to provide accurate predictions as well as protect SSD internals.

1) Applying Behavioral Models: We applied two models (Figure 3) to model the behavior of the DRAM buffer (left) and the I/O completion of NAND chips (right). The DRAM buffer model (Figure 3-left) is used by the device driver to determine whether a write request would result in a DRAM hit (under buffer full threshold) or a NAND I/O (exceeding buffer full threshold). The I/O completion model (Figure 3-right) is used by the latency predictor within the SSD to notify the OS that a predictable I/O operation is underway and that the actual completion would occur within a period called the precompletion window (Figure 2-3 (l)). The models are applied to cover the I/O requests classified into the gray areas of Figure 2-4, although this can be extended by defining additional behavioral models (i.e., read cache hits, read prefetching hits), which leads to our future work.

2) Eliminating Scheduling Delays: For shorter latencies, experienced when an I/O request hits the DRAM

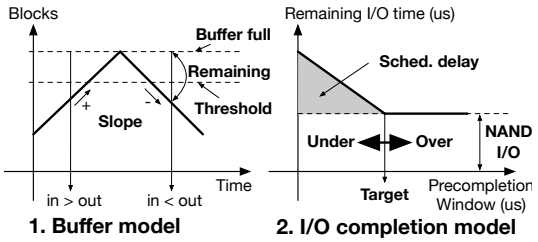

Figure 3: Behavioral models of SSD internals exposed to the OS

buffer within SSDs (Figure 2-2 (f)), the OS device driver knows that this will happen by comparing the remaining space of the buffer and the amount of write I/O it has to issue (Figure 3 left). This is possible since the exact amount of free space within the buffer is passed from the SSD through the completion of the previous completion. Upon buffer hits, the OS responds with blocking the CPU core (busy waiting [9]) for the I/O completion in order to eliminate the scheduling delay.

3) Hiding Scheduling Delays: For I/O requests headed for the NAND chips (Figure 2-2 (h)), the I/O path yields the CPU in order to preserve system parallelism at the cost of scheduling delays. To deal with this delay, the I/O path overlaps scheduling delays to hide the impact from the critical path. This is achieved by aligning the size of the precompletion window with the size of scheduling delays (target window size in Figure 3-right). If the precompletion window undershoots the target, scheduling delays are exposed depending on how much the window undershot (under: gray area in Figure 3-right). However, the window cannot overshoot since parallelism can be harmed due to the penalty of busy waits (over: right side of the target window in Figure 3-right). In addition, these scheduling delays can vary depending on system load, so the window should consider system load as well. Currently, the precompletion window is a fixed value given a priori that is planned to be reinforced with a dynamic feedback mechanism based on runtime measurements.

4 Evaluation

4.1 Experimental Setup

1) Implementation: We implemented a prototype SSD on top of the *"Greedy FTL firmware"*, which was included in the commercial distribution of the OpenSSD2 Cosmos evaluation board [2]. The implementation of our SSD was not a full-blown SSD, although it implements key features described in Section 3. One key limitation was that only a single I/O context (I/O depth 1) could be handled at a time while state-of-the-art SSDs are capable of handling 32 I/O requests (i.e., SATA 3.0) or more (i.e., NVM-Express) simultaneously.

2) Methodology: The OpenSSD2 evaluation board was connected to the host as an end point with a PCI-Express Gen1 x4 connection. The host system was equipped with an Intel i7-4770 3.30 Ghz Quad-core CPU (hyper-thread enabled), loaded with a custom block device driver that we developed on Linux 3.5.0. The I/O depth limitation limited the evaluation scenarios to a single thread competing with other parallel contexts, such as I/O threads or CPU threads. In the scenarios, we used Fio 2.1.3 for I/O threads (including the precompletion-based I/O thread) and a custom-built program that burns CPU cycles. To see the benefits of precompletions, we show the average latency without NAND latency and the throughput of the background task (CPU or I/O oriented) compared to when it was executed alone.

4.2 Results

In this study, we report 1) the effect of predicting DRAM buffer hits through classification, 2) the accuracy of device-side remaining time predictions, and 3) & 4) the impact of precompletions to project the impact on full-featured SSDs.

1) Classifying I/O Requests: The impact of I/O classification was verified by measuring the latency of a single I/O thread performing small random write operations. To limit the latency impact of DRAM buffer flushes and garbage collection overwhelming the average latency, we limit the frequency of buffer full situations and separately report buffer hit latencies. Our I/O path was able to reduce average latency up to 5.8 µs from the baseline. While the baseline experiences scheduling delays even when I/O requests hit the write buffer, I/O classification allows write buffer hits to be identified and minimizes the scheduling delays by blocking the CPU.

2) Predicting Flash Latency: To verify the I/O latency predictability of flash memory when in independent devices, we measured the latency of flash commands on an LP-DDR flash DIMM equipped with four MLC 25 nm 16 GB flash chips [1] provided by the OpenSSD2 project. This measurement was done inside the SSD on top of the flash controller logic, which includes ECC correction [1]. Flash latencies for read operations had very little variance (Table 1) compared to other operations (i.e., DMA engine). The error, which is a root of the sum of squared differences, was less than 1 µs.

3) Precompletion I/O vs I/O Threads: Light gray bars and lines in Figure 4 show the system impact of the completion schemes interacting with I/O threads. Polling with (`POLL_PRIO`) or without (`POLL`) task priority (nice-

[1] The high latency was due to the unoptimized implementation of the stock NAND controller of the OpenSSD2 project. Scheduling delays in this work, in terms of latency, are small with respect to the high latency of the evaluation board, although modern SSDs have lower latencies.

Table 1: Accuracy of Latency Predictions (three-value moving average)

H/W	Measured	Std. dev	Predicted	Error
Flash	352 µs	0.66 µs	352 µs	0.94 µs
DMA	9 µs	0.26 µs	9 µs	0.56 µs

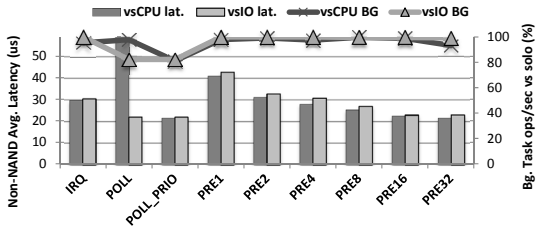

Figure 4: Precompletion I/O threads vs CPU threads

ness) showed the best latency, while interrupts (IRQ) showed scheduling delays up to 11 µs. The cost of reducing this amount of scheduling delays was the excessive CPU cycles causing 17.75% throughput degradation of the background I/O threads.

Precompletions solve this dilemma between latency and parallelism by masking the scheduling delays (11 µs) underneath the SSD I/O time while doing no harm to background threads. However, a precise precompletion window should be given based on observations on scheduling delays. In Figure 4, the best latency was achieved with a 16 µs precompletion window (PRE16). Having a window larger than 16 µs burns more CPU cycles, although this was marginal compared to poll-based methods.

4) Precompletion I/O vs CPU Threads: Dark gray bars and lines in Figure 4 show the interactions with CPU threads. With these interactions, we had to increase the priority of the polling thread (POLL_PRIO) since poll turns an I/O task into a CPU thread. The CPU scheduler, Linux CFS in this case, gave the poll thread an equal share of the CPU, so the poll thread (POLL) had difficulties in acquiring the CPU on time. The measured latency of (POLL) was significantly greater (1,488 µs), while there was no significant drop in background throughput. In contrast, interrupt-based I/O threads, which are threads other than POLL and POLL_PRIO, did not experience this problem.

Here, the cost of a shortened latency of POLL_PRIO was a significant drop in background CPU task performance, which recorded only 80% throughput (ops/sec) compared to the solo-run scenario. Yet, precompletion effectively reduced this latency without severely degrading CPU tasks. In addition, the right size of the precompletion window had to be used (PRE8) in this case.

5 Conclusion

In this paper, we presented a flash SSD latency optimization technique and reviewed the preliminary results toward minimizing the impact of scheduling delays. Our optimization exploits accurate latency predictions that were enabled by tracking behavioral parameters within the SSD. These predictions are based on simplified SSD behavioral models, which are agreed between the OS and the SSD a priori. The accuracy of such predictions is backed with the help of SSDs actively filling in the crucial parameters required for the models.

This was based on the insight that it is far easier to predict the behavior of individual components within an SSD than to predict the behavior of multiple components (the SSD as a whole) as a system. Based on this preliminary work, in future work we plan to evaluate the impact of our optimization on a full-featured SSD under high parallel workloads.

6 Acknowledgments

We would like to thank the anonymous USENIX HotStorage reviewers. This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT & Future Plannig (2015M 3C 4A7065646).

References

[1] SK Hynix / H27QDG8VEBIR-BCB 16GB 2bit MLC chip.

[2] The OpenSSD Project. Cosmos OpenSSD Platform. http://www.openssd-project.org/wiki/The_OpenSSD_Project.

[3] CAULFIELD, A., MOLLOV, T., AND EISNER, L. Providing Safe, User Space Access to Fast, Solid State Disks. ASPLOS'12.

[4] CAULFIELD, A. M., COBURN, J., MOLLOV, T., DE, A., AKEL, A., HE, J., JAGATHEESAN, A., GUPTA, R. K., SNAVELY, A., AND SWANSON, S. Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing. SC'10.

[5] FOONG, A., VEAL, B., AND HADY, F. Towards SSD-ready enterprise platforms. *ADMS'10*.

[6] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., ROSCOE, T., AND ZÜRICH, E. T. H. Arrakis : The Operating System is the Control Plane. OSDI'14.

[7] SEPPANE, E., O'KEEFE, M. T., AND LILJA, D. J. High performance solid state storage under Linux. MSST'10.

[8] VUČINIĆ, D., WANG, Q., GUYOT, C., MATEESCU, R., BLAGOJEVIĆ, F., FRANCA-NETO, L., MOAL, D. L., SAN, H., BUNKER, T., XU, J., SWANSON, S., DIEGO, S., CLARA, S., AND BANDI, Z. DC Express : Shortest Latency Protocol for Reading Phase Change Memory over PCI Express This paper is included in the Proceedings of the. FAST'14.

[9] YANG, J., MINTURN, D. B., AND HADY, F. When Poll is Better than Interrupt. FAST'12.