Framework for Efficient and Flexible Scheduling of Flash Memory Operations

Bryan S. Kim[1], Yonggun Lee[2*], Sang Lyul Min[1]
[1]Seoul National University, [2]SK Hynix
bryansjkim@snu.ac.kr, yonggun.lee@sk.com, symin@snu.ac.kr

***Abstract*—Flash memory-based storages are used in a wide range of systems from small mobile devices to large-scale system servers. The performance demand from applications and the technology of flash memory vary widely from one system to another, making it difficult to design a universal flash memory scheduler for all systems. In this paper, we present a framework for efficient and flexible flash memory scheduling and compare a software scheduler based on the framework against an optimized hardware scheduler. The throughput of the software scheduler achieves more than 97% of that of the hardware scheduler across workloads we consider. We further highlight the extensibility of the framework with a case study on fair queueing scheduling.**

Keywords— Flash Memory; Flash Memory Scheduling; Framework for Flexibility

I. INTRODUCTION

Nowadays flash memory-based storage systems are used ubiquitously from small mobile devices to large-scale servers. Flash memory's advantages of small size, resistance to vibration, and low power consumption make flash-based storage an ideal replacement for mechanical hard disk drives in mobile systems. On the other hand, flash memory's low latency and collectively massive parallelism make it suitable for high-performance storage in large-scale systems. As flash memory manufacturers push for lower $/GB with multi-level cell technology [1] and 3D stacking [2], flash storage systems will continue to expand into a wider range of applications.

However, the wide range of applications that use flash storage and the ever-advancing flash memory technology make it difficult to design a flash memory scheduler that universally performs well across all systems. Depending on the requirement for the system, the scheduler may need to maximize total throughput or equalize fairness of internal management tasks known collectively as the flash translation layer (FTL) [3]. On the underlying technology perspective, new features of flash memory add another layer of complexity in scheduling. Access time for pages differ with multi-level cell technology [1, 4], and multiple pages must be programmed together with one shot programming technology [5]. A scheduler must not only be optimized for performance but also be flexible in order to adapt to changing performance demands and flash memory technology.

In this paper, we present a framework for efficient and flexible scheduling that can easily adapt to changes in application demands and technology trends with comparable performance to an optimized hardware scheduler. The contributions of this paper are as follows:

- We present an efficient and flexible flash memory scheduling framework with an abstraction interface that hides the low-level details of flash memory.
- We demonstrate that a software scheduler based on the framework performs comparably against an optimized hardware scheduler, achieving a throughput of more than 97% of that of the hardware scheduler.
- We demonstrate the extensibility of the framework with a case study on fair queueing [6] scheduling that achieves fairness among concurrent streams of flash memory operations.

The remainder of the paper is organized as follows. Section II gives a background on flash memory and prior work in flash memory scheduling. Section III presents the framework for efficient and flexible scheduling and the overall architecture of the flash memory subsystem. Section IV describes the evaluation platform and presents the experimental results. Lastly, section V concludes and discusses future work.

II. BACKGROUND

A. Flash Memory

Flash memory's density has been exponentially increasing and the organizational complexity has been increasing as well. For simplicity, we abstract the complexity of flash memory organization and describe its interface focusing on core functionality. A flash memory subsystem consists of *chips* that perform flash operations independently. Multiple chips may be connected to a shared *channel*, and all IO activities including command, address, and data transfer communicate via the channel. Within each chip is a set of *blocks,* and within each block is a set of *pages*. A block is the unit for flash erase operations while a page is the unit for flash program and read operations. The non-volatile memory where the data is stored is called the *flash array*, and the IO buffer that acts as a staging area for data to and from the chip is called the *page register*.

*His work is completed when he was with Seoul National University.

This work was supported in part by SK Hynix, and the National Research Foundation of Korea under the BK21 Plus for Pioneers in Innovative Computing (21A20151113068) and the PF Class Heterogeneous High Performance Computer Development (NRF-2016M3C4A7952587). Institute of Computer Technology at Seoul National University provided research facilities for this study.

978-1-5386-1768-7/17/$31.00 ©2017 IEEE

B. Flash Memory Scheduling

Ozone flash controller [7] presented three flash memory scheduling models: sequential, decoupled, and out-of-order. The sequential scheduling model executes each flash memory operation in the order of arrival with no overlap between operations. The decoupled model is an extension of the split transaction model [8] that allows overlapping of operations but requires initiation and completion of operations to be in the order of arrival. Finally, the out-of-order model fully exploits parallelism by scheduling request out-of-order, only constrained by data dependency. While Ozone outlined three scheduling models and implemented them in hardware with a performance–resource tradeoff, it was geared toward maximizing total throughput with limited capabilities for controlling quality-of-service.

QoSFC [9] presented a virtual time-based flash memory scheduler for multiple concurrent FTL tasks with varying priorities depending on the state of the system. In effect, it not only improved the average response time but also addressed the long tail latency problem of flash memory-based storages. However, QoSFC is resource inefficient due to computation overheads for managing virtual times and is not flexible as the scheduling algorithm is integrated with the controller.

SOS [10] is a software-based out-of-order scheduler that improves overall performance by load balancing operations among multiple chips and eliminating duplicate writes belonging to the same logical address. These techniques, however, are complementary to existing flash memory schedulers, and SOS does not take advantage of the software flexibility in deploying a wide range of scheduling policies.

III. Framework for Efficient and Flexible Scheduling

In this section, we describe the framework for efficient and flexible scheduling and the architecture of the flash memory subsystem. The flash memory subsystem is composed of the flash memory scheduler and the flash memory controller. The flash memory scheduler is responsible for exploiting parallelism in the flash memory subsystem while meeting performance demand for each FTL task. On the other hand, the flash memory controller handles IO signaling in compliance with the timing constraints of flash memory. The flash memory abstraction interface hides the complexity of both components and enables the scheduler and the controller to be independent of each other. Both the scheduler and the controller must not only be optimized for performance, but also be flexible to address changes in performance demands of tasks and technology advances in flash memory.

A. Flash Memory Abstraction Interface

The flash memory abstraction interface allows scheduling of flash memory operations without the explicit knowledge of the low-level details of flash memory. This is accomplished by expressing each flash memory operation such as erase a block, program a page, and read a page as a sequence of flash memory requests. Each flash memory request is scheduled independently by the flash memory scheduler and is executed by the controller in the scheduled order. Thus, each request is the unit of schedule for the scheduler and must be self-contained for the controller to process it. As shown in Figure 1, we categorize flash memory requests into three types: *activation*, *data*, and *status*, and show how each flash memory operation can be expressed as a sequence of requests.

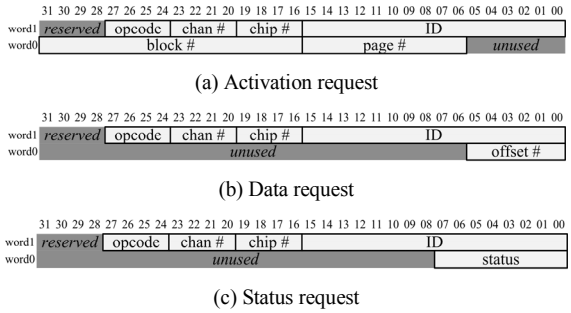


Figure 1. Flash memory requests. A sequence of flash memory requests can implement a flash memory operation such as erase, program, and read.

Erase a block operation is abstracted as a sequence of two requests: an *erase activation* request and a *status* request. The request for the erase activation contains all necessary information for the controller to perform an erase operation; this includes an opcode, a channel number, a chip number, and a block number. Status request reads the status register of the flash memory chip to determine whether or not the erase operation was successful.

Program a page operation is expressed as a sequence of four or more requests. The first request is a *program prolog activation* request that provides a page address in addition to all the other fields used in the erase activation request. One or more *input data* requests follow. Offset field of the data request indicates the data offset within the page in data chunks of large granularity (1KB or more). This not only allows transfer of data smaller than the page size, but also abstracts the page data layout of user data and out-of-band data including error correction code. A *program activation* request following input data requests writes the data from the flash's page register to the flash memory array, and finally, the *status* request returns the success/failure of the program operation.

Lastly, *read a page* operation is sequenced as two or more requests. The first request is a *read activation* request that initiates reading data from the flash memory array to the flash's page register. One or more *output data* requests follow the read activation request, and similar to the input data request, the offset field indicates data chunk offset within the page.

B. Flash Memory Scheduler

An efficient and flexible flash memory scheduler should allow a wide range of scheduling policies ranging from total throughput maximization to perceived fairness equalization. Hardware schedulers lack the flexibility as all possible (including future) scheduling policies need to be included at design time. In this subsection, we describe the overall architecture of an optimized hardware scheduler based on the out-of-order model [7] and present a software scheduler that mirrors the hardware scheduler.

Figure 2 illustrates the overall architecture and components of a hardware scheduler. Its main components are the *internal*

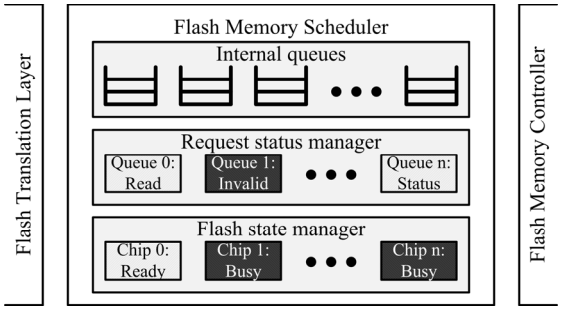


Figure 2. Overall architecture of a hardware flash memory scheduler. Situated between the FTL and the controller, the scheduler queues requests, maintains the status of available requests, and keeps track of the availability of chips.

queues, the *request status manager*, and the *flash state manager*. The internal queues hold requests addressed to the same chip. A first-in-first-out management of requests in the internal queue conservatively resolves data dependency, but a more complex re-ordering of requests can be used without violating the correctness. The request status manager maintains the status of requests at the head of each request queue. Information such as validity, opcode, and expected duration are needed to make optimal scheduling decisions. Lastly, the flash state manager keeps track of the ready/busy state for each chip. Activation requests such as erase, program, and read causes the flash state to become busy, and availability of the chip must be proactively checked by reading the status of the chip.

Given these components and structure, the scheduler selects a valid request for a chip that is ready. If the chip is not ready, that request cannot be scheduled and the status of the chip must be checked for its availability. Optimal scheduling policies differ depending on the performance goal of the system, but in general, a judicious scheduling policy prioritizes short duration requests (such as activation) over long duration requests (such as data transfer). Furthermore, status requests that check the readiness of chips must be carefully scheduled with activation and data requests not to waste channel bandwidth.

```
forever
  /* Queue FTL requests for each chip */
  while FTL request exists
    queue FTL request in internal queues
  /* Schedule activation and status requests */
  for all chips that are ready
    if activation request is available
      send request to controller
      if request is erase, program, or read
        mark chip as busy
    if status request is available
      send response to FTL with stored chip status
  /* Schedule one data request */
  for all chips that are ready
    if chip is ready and data request is available
      send request to controller
      break
  /* Check if busy chips have become ready */
  for all chips that are busy
    send status request
  /* Update chip state if it's ready */
  while controller response exists
    get controller response
    if chip is ready
      mark chip as ready
      store chip status
```

Figure 3. Pseudo code for flash memory scheduler. Some key features of this scheduler are: 1) FTL request and controller response handling, 2) prioritize activation and status requests over data requests, 3) schedule only one data request at a time, and 4) proactive ready/busy checking.

Based on the roles and responsibilities of a hardware scheduler, a pseudo code for a software scheduler is outlined in Figure 3. The outlined software scheduler, like its hardware version, 1) maintains an internal queue to allow requests to be serviced out-of-order, 2) schedules each request based on its request type, and 3) keeps track of the ready/busy state for each chip and schedules status requests to check availability.

C. Flash Memory Controller

The flash memory controller must be flexible enough to address different timing requirements of various flash memory chips and vendors while achieving efficiency at the IO signaling level. To address this issue, we implement the flash memory controller with a programmable microcode architecture [11]. The microcode instruction set is general enough that it supports MIPS-like instruction set, but specific enough for flash memory IO signaling efficiency. These specialized instructions include the following:

Assert/de-assert chip enable. Chip enable is asserted or de-asserted for the selected chip.

Write command/address. Command/address latch enable is asserted and the value is written to the channel.

Write/read data from/to FIFO. Bytes of data are transferred between data FIFOs and the channel. Mainly used for high bandwidth data transfers.

Write/read data from/to register. Data is transferred between a register and the channel. Mainly used for reading the status register.

Wait. Waits idly for a number of cycles.

Receive request / send response. The scheduled request is received into a register, and the response in a register is sent back to the scheduler.

IV. Evaluation

A. Evaluation Platform

We used a Xilinx ZC706 development board [12] to implement and evaluate our schedulers. Zynq SoC on the development board integrates two ARM Cortex A9 processors with the FPGA. We run a workload generator and the software scheduler on the two ARM processors, and implemented the hardware scheduler, flash controller and various other hardware components on the FPGA. The hardware scheduler is implemented based on the architecture shown in Figure 2, and the software scheduler's algorithm follows the pseudocode outlined in Figure 3. The overall architecture of the evaluation platform is shown in Figure 4. An in-house flash memory daughterboard connects to the ZC706 through an FPGA mezzanine connector. The specifications for the flash memory used are summarized in Table 1.

The workload generator runs on one of the ARM processors and generates flash memory requests for the

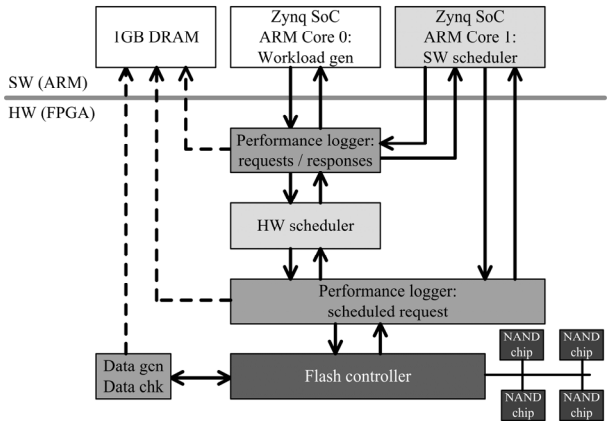


Figure 4. Evaluation architecture. The workload generator and the software scheduler runs on the ARM processors while the hardware scheduler, the controller, performance loggers, data generator and checker, and dedicated routing channels are implemented in the FPGA portion.

schedulers. It produces a sequence of flash memory requests with a configurable inter-arrival time between flash memory operations. The inter-arrival times have a uniform distribution given the minimum and maximum bound for the inter-arrival time. At each iteration, the generator randomly chooses a target block and selects a flash memory operation (erase, program, or read) given the ratios of the three operations.

Depending on the selected test mode, requests from the workload generator are routed to either the hardware scheduler in FPGA or the software scheduler running on the other ARM processor. Similarly, scheduled requests from either scheduler are routed to the flash controller. All routings are performed by separate and dedicated channels such that there are no interconnect and arbitration overheads.

In addition to the hardware scheduler, the flash controller, and the routers, the FPGA portion of the evaluation platform also contains performance loggers and data generator and checker for post-mortem analysis. All requests from and responses to the workload generator and scheduled requests from the schedulers are time-stamped and written to DRAM through a yet another dedicated channel. This information is later used to extract throughput of the system and the behavior of the scheduler during test. Data generator and checker use the address of the flash memory request to generate a pseudo-random data pattern based on linear feedback shift registers. For a program request, data pattern generated is sent to the flash controller to be written to flash; for a read request, the read data from the flash is compared with the generated data pattern. The error between the read data and the generated data pattern is logged and written to DRAM for analysis.

TABLE I. FLASH MEMORY SPECIFICATION

# of channels	1	**Read latency**	75µs (typ) 110µs (max)
# of chips per channel	4	**Program latency**	1.5ms (typ) 5ms (max)
# of blocks per chip	4096	**Erase latency**	5ms (typ) 10ms (max)
# of pages per block	256	**Channel bandwidth**	100MB/s
Page size	16KB	**Total capacity**	64GB

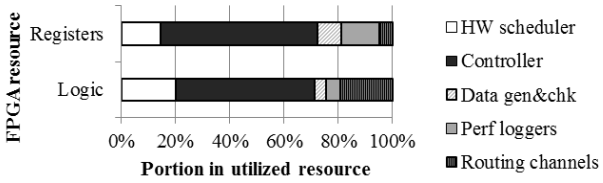


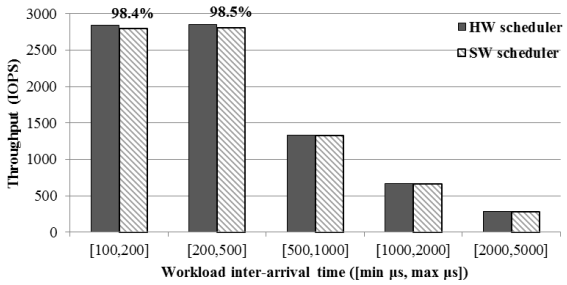
Figure 5. Resource utilization breakdown. The controller takes up most of the utilized resources due to its generalized microcode architecture. The hardware scheduler uses 20% of the occupied logic and 14% of the occupied registers.

The breakdown of resource utilization for the components implemented in FPGA is summarized in Figure 5. The flash memory controller takes up the majority of the resources for both logic and register. The generalized microcode architecture we chose to implement traded off resource footprint for flexibility. The hardware scheduler takes up 20% of logic and 14% of register resources that can be reduced if software scheduling is used instead. Resource utilization of the hardware scheduler is expected to further increase if complex algorithms such as fair queueing were to be implemented.

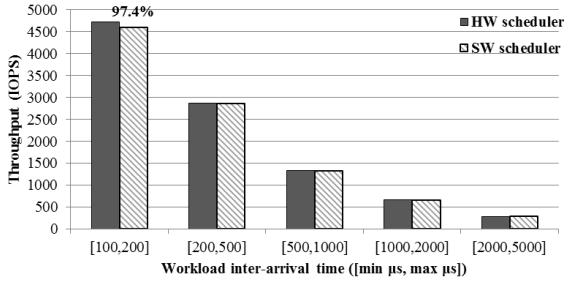
B. Performance Comparison

We compare the performance of the two schedulers under synthetic workloads with various intensity and composition in Figure 6. The percentage above the bar is the throughput of the software scheduler relative to the hardware scheduler's. 100% throughput is omitted in the graphs, and only at high workload intensity, it falls below 100%.

Experiment results summarized in Figure 6 show that the



(a) Erase:Program:Read = 1:256:128



(b) Erase:Program:Read = 1:256:512

Figure 6. Throughput comparison of both hardware and software schedulers under different workload intensity and composition setting. *x*-axis of each graph represent workload inter-arrival times, and each graph shows results with different workload composition.

software scheduler's throughput falls below 100% only when the total throughput has reached its maximum capacity. Experiment in Figure 6a) exercises a relatively program-heavy workload where the ratio of erase:program:read is 1:256:128. In this scenario, we can observe that the total throughput has saturated by the inter-arrival time of [200µs, 500µs] as the throughput no longer increases even when the inter-arrival time is shortened to [100µs, 200µs].

The performance of the software scheduler is at its worst at 97.4% when the workload is read-heavy with an inter-arrival time of [100µs, 200µs] as shown in Figure 6b). Flash read operations are an order of magnitude faster than programs, and the workload dominant with low latency operations gives the hardware scheduler an advantage. However, the current trend of increasing flash memory operation latencies and page size will reduce the software scheduler's disadvantage and close the already small gap of 2.6%.

C. Case Study: Fair Queueing

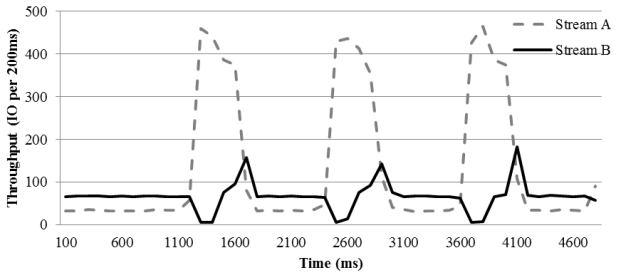
We demonstrate the extensibility of the scheduling framework and present a case study on fair queueing. In this example, we consider two streams *A* and *B* that each generates its own requests independently with its own inter-arrival time pattern. In this scenario, stream *A* suddenly increases its own workload intensity for a short duration and we observe its effect on the performance of stream *B*. The ratio of flash memory operations is the same for both streams at erase:program:read of 1:256:256.

The scheduler based on fair queueing differentiates requests based on their stream and queues them separately. Requests are scheduled based on virtual time that measures the progress of each stream, while issuing requests in a work-conserving manner. The simple scheduler outlined in Figure 3 and evaluated in the previous subsection, on the other hand, is stream-unaware and is unable to provide differentiated services.

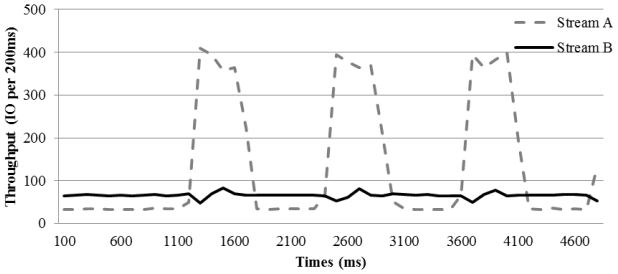
Figure 7a) shows a scheduler without stream-awareness, and the throughput of stream *B* is heavily affected by the increased activity of stream *A*. However, with fair queueing, as shown in Figure 7b), the throughput of stream *B* is isolated from the increased intensity of stream *A* and the scheduler provides overall fairness for both streams.

V. Conclusion

In this paper, we present a framework for efficient and flexible scheduling with an abstraction interface that hides the low-level details of flash memory from the flash memory scheduler. We demonstrate that a software scheduling based on the proposed framework achieves a throughput of more than 97% of that of an optimized hardware scheduler. Lastly, we demonstrate the extensibility of the framework by showcasing a fair queueing scheduler. Going forward, we plan to implement and run FTL tasks for a more realistic workload stimulus for the scheduler.



(a) Scheduling without stream-awareness



(b) Scheduling with fair queueing

Figure 7. Throughput comparison for a stream-unaware scheduler and a fair queueing-based scheduler. For the scheduler in a), the performance of stream *B* is affected by the changes in stream *A* and its throughput is reduced. For fair queueing in b), stream *B* is far less affected.

References

[1] C. Trinh et al., "A 5.6MB/s 64Gb 4b/Cell NAND flash memory in 43nm CMOS," in *IEEE Solid-State Circuits Conference*, 2009, pp 246-247.

[2] H. Tanaka et al., "Bit cost scalable technology with punch and plug process for ultra high density flash memory," in *IEEE Symp. on VLSI Technology*, 2007, pp. 14-15.

[3] E. Gal and S. Toledo, "Algorithms and data structures for flash memories," in *ACM Computing Surveys*, vol. 37, no. 2, pp. 138-163, June 2005.

[4] L.M. Grupp et al., "Characterizing flash memory: Anomalies, observations, and applications," in *IEEE/ACM Microarchitecture*, 2009, pp. 24-33

[5] skhynix.com, 'Products: Raw NAND,' 2017, [Online], Available: https://www.skhynix.com/eng/product/nandRaw.jsp

[6] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," in *SIGCOMM*, 1989, pp. 1-12

[7] E.H. Nam, B.S. Kim, H. Eom, and S.L. Min, "Ozone (O3): An Out-of-Order Flash Memory Controller Architecture," in *IEEE Trans. on Computers*, vol. 60, no. 5, pp. 653-666, May 2011

[8] B.S.Kim, E.H. Nam, Y.J. Seong, H.J. Min, and S.L. Min, "Efficient flash memory read request handling based on split transactions," in *Int'l Workshop on Software Support for Portable Storage*, 2009

[9] B.S. Kim and S.L. Min, "QoS-aware flash memory controller," in *IEEE RTAS*, 2017, pp. 51-62.

[10] S.S. Hahn, S. Lee, and J. Kim, "SOS: software-based out-of-order scheduling for high-performance NAND flash-based SSDs," in *IEEE MSST*, 2013,

[11] R. Razdan and M.D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *ACM Microarchitecture*, 1994, pp. 172-180

[12] Xilinx.com, "Xilinx Zynq-7000 All programmable SoC ZC706 Evaluation Kit", 2017, [Online], Available: https://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html