ABSTRACT
Flash memory is a storage class memory widely used in mobile computing systems due to its small size, low power consumption, fast access time, and high shock and vibration resistance. Flash memory based storage systems exploit chip-level parallelism and hide the latency of flash memory operations through request scheduling. However, conventional scheduling techniques are inadequate for handling series of small random requests, degrading the overall performance of the system. This paper presents a flash memory request handling technique based on split transactions to improve the read performance – in particular, the performance of multiple small read requests. This technique decouples the first and second phase of each flash memory read request and services them in a different manner. The first phase of the operation is issued at the earliest time possible, while the second phase is delayed until the operation latency expires or the next read request arrives to the same chip. This request handling technique based on split transactions has been incorporated into a prototype flash memory based storage system on an FPGA development board. Experimental results from this prototype show that for small random read requests, this technique improves read performance by up to 46%.

Categories and Subject Descriptors
B.3. Memory Structures; C.4 Performance of Systems

General Terms
Management, Measurement, Performance, Design

Keywords
Flash memory, Split transactions, Request scheduling

1. INTRODUCTION
In recent years, mobile computing devices such as MP3 players, PMPs (portable media players), and PDAs (personal digital assistants) have gained much popularity in consumer electronics. Flash memory is preferred over hard disk drives (HDDs) in such mobile computing devices due to its small size, fast access speed, low power consumption, absolute silence, and high resistance to shock and vibration [1].

The density of NAND flash memory has been doubling every year, a rate exceeding Moore’s Law [6]. With this trend, NAND flash memory becomes more affordable and its price more competitive against HDDs, accelerating the replacement of HDDs by NAND flash memory based storage systems [3, 4]. For example, flash memory solid-state disks (SSDs) that provide the same interface as HDDs are replacing HDDs in mobile and general-purpose computers.

NAND flash memory, however, has peculiar characteristics in terms of architecture and reliability that make a straightforward replacement of HDDs difficult [3, 7]. Architecturally, NAND flash memory prohibits in-place update of data. Instead, writing is performed by a program-page operation, which must be preceded by an erase-block operation that sets all the bits in the target physical block to 1. A block, the unit of erase operations, contains a set of pages, which is the unit of read and program operations. In terms of reliability, NAND flash memory allows some blocks to be ‘bad’ at manufacture time. Even good blocks may turn bad after a certain number of program-erase cycles. This physical endurance limit is typically about 10,000 ~ 100,000 cycles, depending on the type of NAND flash cells. NAND flash memory is also subject to bit-flipping errors, causing one or more bits in a block to be reversed, which necessitate the use of external error correction logic.

In order to hide the peculiarities of NAND flash memory, a flash memory based storage system employs a software layer known as the flash translation layer (FTL) [7, 8]. The first and foremost responsibility of the FTL is mapping table management resulting from the fact flash memory does not allow in-place update. Upon receiving a host write request, the FTL writes to a pre-erased area and remaps the logical host address to the physical flash memory address. Upon receiving a host read request, the mapping table is looked up for the corresponding physical flash memory address and the data is subsequently read from flash. Other FTL responsibilities include bad block management and wear-leveling.

On the hardware side, flash memory request scheduling employed by hardware acceleration modules also plays an important role in flash memory based storage systems [4]. Flash memory operations such as erase-block, program-page, and read-page have operation latencies during which the chip is busy and other operations cannot be accepted. These latencies typically range from a few tens of microseconds to a few milliseconds. In order to hide the latency of flash memory operations, multiple chips are used concurrently by flash memory based storage systems through request scheduling. Because the efficiency of request scheduling directly impacts the overall performance of the system, many research groups have explored and studied this area [2, 5, 9, 10, 11]. However, most previous scheduling techniques focus on optimizing a single (large sequential) host request, and are not very effective in handling series of multiple (small random) host requests, although this type of workload is common in multi-tasking environments, such as desktops and servers.
This paper presents a flash memory request scheduling technique based on split transactions that allows a stream of multiple requests to be reordered and optimized. By decoupling the first phase and second phase of a single flash memory operation, this technique allows overlapping of multiple requests over multiple chips. The proposed technique exploits not only intra (host) request parallelism but also inter (host) request parallelism and thus, it cures the inefficiency of previous techniques in handling multiple small random host read requests.

We implemented a prototype flash storage system that incorporates the request handling technique based on split transactions. Performance evaluation using this prototype shows that for multiple small random read requests, this technique improves read performance by up to 46%.

The remainder of the paper is organized as follows. Section 2 gives backgrounds on flash memory organization and supported operations. Section 3 explains the conventional architecture of flash memory based storage systems and how host read requests are handled. Section 4 explores flash memory read request handling based on split transactions. Section 5 describes the implementation and evaluation environment, and presents performance measurements. Finally, Section 6 concludes and suggests future research directions.

2. BACKGROUND

2.1 NAND flash memory internal organization

NAND flash memory chip is internally organized as follows: (1) I/O and control logic that accept commands and addresses, (2) data buffer that temporarily stores data read or to be written, (3) the NAND flash memory array, the non-volatile memory inside the chip. The NAND flash memory array is further composed of blocks, the chip.

The NAND flash memory array is further composed of blocks, each of which in turn is composed of pages [4]. Typically 64 pages or 128 pages make up a single block. A page is conceptually divided into a data area and a spare area. The size of a data area is multiples of 512 bytes, typically 2KB or 4KB. The spare area is typically 16B per 512B of data area, but some recent chips have spare area of 24B or 27.5B per 512B of data area. Although there is no difference between the data area and spare area at the flash memory cell-level, flash memory based storage systems typically use the spare area for ECC or meta-data. More importantly, manufacture-time bad blocks are marked in the spare area, so it is imperative to read this area for constructing a bad block management structure upon system initialization.

2.2 NAND flash memory operations

This section will briefly explain basic flash memory operations that access non-volatile flash memory data: erase-block, program-page, and read-page. For specific timing values, Samsung K9WBG08U1M [12] chip is assumed.

Erase-block operation sets all the flash cell values to 1 in the selected block. As shown in Figure 2.1(a), flash memory block erase requires no data transfer, but has a long latency, typically 1.5ms. The first phase of erasing a block requires the erase command and the physical address of the block. After the erase latency, a status check on the chip is required to check the pass or failure of the erase operation. A failure in erasing a block is an indication of a bad block, and thus it is imperative to check the status.

Program-page operation writes the data transferred to the data buffer to the selected page. As shown in Figure 2.1(b), flash memory page program requires a data input to the chip, typically at 40MB/s (1B per 25ns), followed by the program latency, typically 200μs. The first phase of programming a page requires the physical address of the page, followed by the data transfer, and then the program command. Like the flash memory erase, the program operation may fail, and thus the status check on the chip is required after the latency. Again, a program failure is an indication of a bad block.

Read-page operation moves the data in the flash memory array to the data buffer. As shown in Figure 2.1(c), the first phase of reading a page requires the read command and the physical address of the page, followed by the read latency, typically 25μs, but up to 200μs in some recent chips. After the read latency, data in the data buffer can be transferred out, typically at 40MB/s (1B per 25ns). Unlike erase and program, read operation does not require a status check on the chip, but since data read may suffer from bit-flip errors during program or simply over time, it is imperative to have ECC (error correction code) encoding and decoding logic in the flash memory controller.

3. FLASH READ REQUEST SCHEDULING

As explored in the previous section, a single NAND flash memory chip has limited bandwidth for program and read operations. Thus, flash memory based storage systems utilize multiple NAND flash memory chips to overlap multiple operations over multiple chips [2, 4]. Because of the long program latency and the erase-before-program requirement of NAND flash memory, previous research has focused on improving the write performance – in particular, small random write performance [7, 13]. On the other hand, this paper focuses on improving the read performance because of the blocking nature of read requests from the host – while write requests may be buffered and their materialization can be delayed by the storage system, reads may not. The following sections will explore how the conventional design hides operation latencies for read requests, and will further investigate how it falls short in handling series of small read requests.

3.1 Flash memory based storage system

The basic architecture of a typical flash memory based storage system is illustrated in Figure 3.1. The processor core executes software functionalities of the FTL, and the flash memory controller enhances the effectiveness of the FTL as a hardware accelerator. SRAM provides code execution environment, and DRAM serves as a volatile buffer for read caching, pre-fetching,
and write buffering. The host interface and flash memory interfacer allow a modular design approach of the storage system by decoupling interface protocols from the rest of the storage system functionalities. More specifically, the host interface allows multiple implementations of interface protocols such as Serial ATA (SATA) and Serial Attached SCSI (SAS), and the flash interfacer supports diverse interfaces of NAND flash chips such as the traditional NAND [12], OneNAND [14], and Hyper-Link NAND [15].

Previously, the flash memory controller had not been emphasized. However, the flash memory controller plays a central part in the overall performance of the system as it is responsible for handling flash memory requests such that multiple operations can be performed in parallel over multiple chips. Moreover, the flash memory controller abstracts flash memory operations and allows the FTL to execute oblivious of the operation latencies. The details of the flash memory controller will be explored in the next section.

### 3.2 Conventional read request handling

Figure 3.2 describes the operation of a basic flash memory controller that accepts host read requests from the host interface and translates them into series of flash memory requests. For simplicity, only one flash bus traffic is depicted in the examples to emphasize the performance degradation caused by the latencies of flash memory operations, but this can be easily generalized to multiple buses. Furthermore, for clarification, this paper assumes that the mapping table look up can be accelerated by hardware components in the flash memory controller, and that host read requests are directly forwarded to the flash memory controller without software intervention.

Upon receiving a host read request, the flash memory controller looks up the mapping table and locates the physical flash memory addresses. To read data out of flash memory, the basic flash memory controller issues a read page request, then a data output request to the flash memory interfacer. Similarly, other pages to be read are accessed in the same manner – a read page request followed by a data output request. Although this sequential access of each page is simple, it results in poor read performance as the unscheduled stream of read requests incur operation latencies between the read request and the data output request.

In order to improve the read performance, many research groups have studied interleaving techniques that hide operation latencies by overlapping operations to multiple chips [2, 5, 9, 10, 11, 16]. Figure 3.3 is an example of such an intelligent flash memory controller that utilizes interleaving techniques to translate host read requests into a stream of scheduled flash memory read requests. Instead of sequentially accessing chips, flash memory controller that interleaves requests issues read page requests to more than one chip, and overlaps the data output traffic of one chip to hide the latency of another.

However, the efficiency of conventional interleaving techniques heavily relies on the size and sequentiality of data to be read because they usually adopts static data layout scheme such as super-block [11, 16], in which large sequential data is placed over multiple flash memory chips. If data to be read is too small to take advantage of the super-block scheme, or if the data to be read for a host request is clustered to only one or a few chips, the interleaving technique becomes less effective. Furthermore, the conventional interleaving techniques only consider parallelism within host requests and do not exploit parallelism across multiple host requests.

Figure 3.4 exemplifies the limitations of the conventional flash memory controllers. Upon receiving the first host read request, the flash memory controller issues a stream of scheduled flash memory read requests based on the conventional technique. However, because it is unknown to the flash memory controller (and even to the flash memory based storage system) when the subsequent host read request will arrive, the flash memory controller is unable to optimize between multiple host read requests. This naturally induces the controller to exhibit a one-at-a-time behavior in servicing host requests, failing to hide latencies between multiple read requests. This performance vulnerability is amplified if each host read request size is small – in such a case, the performance of a flash memory controller using an
interleaving technique may be degraded to that of a basic controller that does not schedule requests at all.

4. REQUEST HANDLING BASED ON SPLIT TRANSACTIONS

This section introduces a request handling technique based on split transactions and a hardware module that implements this technique. The design goals are two-folds: first, to solve the performance problem of multiple small read requests, and second, to make the technique compatible with the existing flash memory based storage system architecture.

The basic principle behind the request handling based on split transactions is decoupling the two phases in flash memory read operations. More specifically for a flash memory page read, it loosens the coupling between the first read page request and the subsequent data output request so that they can be serviced under different policies. The details on how the two phases are handled will be covered in the next sub-section.

Figure 4.1 shows a hardware module called the “request reordering unit” between the flash memory controller and the flash memory interfacer. The request reordering unit implements the request handling technique based on split transactions to improve the read performance of the system while maintaining the same interface semantics between the flash memory controller and the flash memory interfacer. Because the interface semantics are maintained, the request reordering unit is pluggable, and can be easily integrated into existing flash memory based storage systems.

4.1 Design of request reordering unit

The design and implementation of the request reordering unit is simplified by two reasons: straightforward correctness constraint and architectural assumption. There is only one dependency in ordering flash memory requests: the read request and the following data output request. Data output request may not be issued from the request reordering unit without the matching read request being issued first. A read request may not be issued without the data output request of the previous read request to the same chip being issued. This constraint must be met in order to guarantee correctness of the flash memory based storage system. In the flash memory based storage system in which the request reordering unit is incorporated, the architectural assumption constrains the total ordering of data outputs to be in-order.

Because of this restriction, the request reordering unit would only need one first-in, first-out buffer to delay the issue of data output requests.

Figure 4.2 explains the operation details of the request reordering unit. Figure 4.2(a) shows the initial state, with requests queued at the request reordering unit. The queued requests are, in the order of items placed, read to chip 0, data output to chip 0, read to chip 1, data output to chip 1, read to chip 0, and data output to chip 0. The read request to chip 0 is passed to the flash memory interfacer upon receiving. The data output request, on the other hand, is queued, as shown in Figure 4.2(b). By delaying the data output request the request reordering unit allows subsequent read requests, in this case, the read request to chip 1, to be issued. This effectively reduces the latency penalty, and provides a window of parallelism. When requests are queued, an internal data structure keeps track of how many requests have been queued for each chip.
Figure 4.2(c) shows the state of the request reordering unit, in which read requests to chip 0 and chip 1 have been issued, and data output requests to chip 0 and chip 1 are queued. Upon receiving the read request to chip 0, all data output requests to chip 0 queued must be issued before the second read request can be issued. The internal data structure to keep track of how many requests are queued is used to determine if all the data output requests have been sent or not.

As shown in Figure 4.2(d), once the data output request to chip 0 has been sent, the second read request to chip 0 can be issued, and the subsequent data output to chip 0 is queued.

In order to issue queued data output requests, an internal timer is used to specify the size of the window for exploiting parallelism. Once there are no more requests to the request reordering unit, and there exist requests queued in the internal FIFO, the timer increments until the specified window size. Once the timer expires, all the queued requests are issued, as shown in Figure 4.2(e).

The reordered stream of requests effectively hides the read latency by delaying data output requests until necessary for correctness, or until the time window for exploiting parallelism expires. As shown in Figure 4.3(a) and Figure 4.3(b), the reordered stream of requests has a shorter completion time than the original stream of requests.

5. EVALUATION

In order to evaluate the performance of the request handling based on split transactions, the request reordering unit described in the previous section has been incorporated into a flash memory controller and prototyped onto an FPGA based development platform.

5.1 Implementation and test environment

Synthetic workloads are used for performance evaluation to eliminate the effects of the host interface processing overhead and to isolate the effect of the proposed technique. The synthetic workload is comprised of two independent dimensions – randomness and request size. Although workloads such as “large random” and “small sequential” are rarely found in real workloads, they are included nonetheless in the synthetic workload. The evaluation framework using synthetic workload is shown in Figure 5.1.

The experimental set-up is as follows. One block is randomly selected for each chip, and the block is programmed with a random data stored in the volatile buffer. Then up to 32 host read requests are generated and also stored in the volatile buffer. These requests will later be fetched and decoded by the flash memory controller. The timer for performance measurement starts as the flash memory controller begins fetching host requests. When all data is read from flash memory to the volatile buffer, the timer stops, and the programmed data and read data are compared for correctness.

The flash memory controller with the request reordering unit has been prototyped using an in-house development platform. Figure 5.2 shows the major components of the development platform: SDRAM, FPGA, and DIMM sockets. The 64MB SDRAM on the prototype board serves as data and request storage for the flash memory controller. The Xilinx Virtex4 FPGA implements all the hardware components of the storage system, including the flash memory controller and the request reordering unit. In addition, the PowerPC405 processor embedded in the FPGA executes the test firmware that emulates the host interface and runs the FTL. The DIMM sockets allow diverse NAND flash chip modules to be plugged – for this specific evaluation, 64 Samsung K9WBG08U1M [12] chips (8 chips/bus x 8 buses) were used.

5.2 Performance measurement

5.2.1 Random workload

Figure 5.3 shows the measured bandwidth of the flash memory based storage system for synthetic random workload. The x-axis represents the size of each request, ranging from 512B (1 sector) to 64KB (128 sectors). For all request sizes, the storage system with the request reordering unit enabled (represented by light bars) outperforms that with it disabled (represented by dark bars).
In particular, the measurements for 4KB random read requests show approximately 46% increase in bandwidth. Even for 64KB random read requests, the performance improves by almost 10%. The relative performance gap between the request reordering unit disabled and enabled is the greatest when the request size is 4KB (one sector from each flash bus). As the request size increases beyond 4KB, the chip-level interleaving technique employed by the flash memory controller allows the performance of the request reordering unit disabled to catch up the enabled one. On the other hand, as the request size decreases below 4KB, idle flash buses allow overlapping of host requests, reducing the performance gap between the two.

Figure 5.4(a) and Figure 5.4(b) show the distribution of response times for 4KB and 64KB request sizes, respectively. For random workload with 4KB requests, nearly half of the measurements fall between 720μs and 780μs when the request reordering unit is enabled; when it is disabled, more than half of the measurements are between 1060μs and 1100μs. These response time measurements are consistent with the bandwidth measurements in Figure 5.3. Similarly, response time measurements for 64KB request size in Figure 5.4(b) are consistent with the bandwidth measurements: the average response time when enabled is 9034μs, while when disabled is 9596μs.

5.2.2 Sequential workload

Figure 5.5 shows the measured bandwidth for synthetic sequential workload. The results show that although the performance improvement for sequential workload is much less than that for random workload, the request reordering unit does not degrade the performance nevertheless.

6. CONCLUSION

This paper explores the limitation of conventional request scheduling in flash memory based storage systems, and presents a request handling technique based on split transactions. This technique decouples the first phase and second phase of a flash memory read operation to facilitate parallel processing on multiple chips. The proposed technique is particularly effective for multiple small random requests as compared against the conventional techniques.

The proposed request handling technique based on split transactions has been incorporated into a hardware module that handles host read requests to improve the read performance of a flash memory based storage system. Performance evaluation on an FPGA based development board shows that the proposed technique improves the read performance by up to 46% for a series of small random read requests.

For future research, we plan to investigate a request handling technique based on more general out-of-order execution, a concept used by most modern processors. We expect that this relaxation of constraints using the out-of-order execution model will significantly improve the performance of flash memory based storage systems by allowing more overlapping of flash operations to multiple chips.
7. REFERENCES


