

Ozone (O3): An Out-of-Order Flash Memory Controller Architecture

Eyee Hyun Nam, Bryan Suk Joon Kim,
Hyeonsang Eom, *Member, IEEE*, and Sang Lyul Min, *Member, IEEE*

Abstract—Ozone (O3) is a flash memory controller that increases the performance of a flash storage system by executing multiple flash operations out of order. In the O3 flash controller, data dependencies are the only ordering constraints on the execution of multiple flash operations. This allows O3 to exploit the multichip parallelism inherent in flash memory much more effectively than interleaving. The O3 controller also provides a prioritized handling of flash operations, equipping flash management software, such as the FTL (flash translation layer), with control knobs for managing flash operations of different time criticalities. Running a range of workloads on an FPGA implementation showed that the O3 flash controller achieves 3 to 100 percent more throughput than interleaving, with 46 to 88 percent lower response times.

Index Terms—Flash memory, flash translation layer (FTL), storage system, solid-state disk (SSD).



1 INTRODUCTION

FLASH memory is increasingly being used as a storage medium, not only in mobile devices but also in PCs and server systems, because of its fast random access, low power consumption, small size, and high resistance to shock and vibration. The density of NAND flash memory, the type of flash memory used for bulk storage applications, has been doubling every year [1]. This trend, which exceeds Moore's Law [2], is due to a combination of shrinking process geometries and multilevel cell (MLC) technology, which allows multiple bits to be stored in a single transistor. These developments have made NAND flash memory competitive with hard disk drives (HDDs), so that solid-state disks (SSDs) based on flash memory are now being used in server systems as well as PCs.

NAND flash memory, however, has unusual characteristics that prevent its direct use as a storage device. A typical NAND flash chip is organized into blocks, each of which contains a set of pages that are accessed individually by read and program operations [3]. The architecture of NAND flash memory does not allow in-place update of data, and all the pages in a block must be erased at once before any of them can be programmed.

To present the same storage interface as an HDD and to overcome the limitations of flash memory, a software layer called a flash translation layer (FTL) [4] is used in flash storage devices, such as an SSD. A simple FTL that

generates flash requests one at a time can use a sequential flash controller. However, a more practical FTL needs to exploit multiple flash chips efficiently by generating multiple concurrent flash requests, and it necessitates a flash controller that can service these multiple requests in parallel. Thus, the flash controller is as important as the FTL in determining the performance of a flash storage system. Parallel servicing of flash requests becomes even more important if the storage device interface allows the host computer to have multiple read and write requests outstanding.

For effective exploitation of the multichip parallelism in flash memory, we looked to the out-of-order execution technique, which dates back as early as the 1960s, and applied it to the design of a high-performance flash controller. The result is a flash controller called Ozone (O3), which has the following features:

1. The O3 controller exploits the multichip parallelism inherent in flash memory in a much more effective way than interleaving. In the O3 flash controller, the only ordering constraints on the execution of multiple flash operations are data dependencies between them.
2. O3 is completely decoupled from the FTL using a packet-based interface, which greatly enhances modularity and extensibility.
3. O3 is equipped with prioritized handling of flash operations that allows different levels of service to be provided to flash operations with high and low priorities. This mechanism gives the FTL the facility to control the rates at which different streams of flash operations are serviced.

We have implemented a prototype of the O3 flash controller using an FPGA-based development platform. Our evaluation of the performance of this prototype shows a 3 to 100 percent gain in throughput and a 46 to 88 percent improvement in response time compared to interleaving.

The rest of this paper is organized as follows: In the next section, we explain the basics of flash memory and review

- E.H. Nam, H. Eom, and S.L. Min are with the School of Computer Science and Engineering, College of Engineering, Seoul National University, 599 Kwanak-ro, Gwanak-gu, Seoul 151-742, Korea.
E-mail: {ehnam, symin}@archi.snu.ac.kr, hseom@cse.snu.ac.kr.
- B.S.J. Kim is with the Department of Computer Science and Engineering, Jacobs School of Engineering, University of California, San Diego, 9500 Gilman Drive, Mail Code: 0404, La Jolla, California 92093-0404.
E-mail: bryanskim@cs.ucsd.edu.

Manuscript received 8 Jan. 2010; revised 24 June 2010; accepted 22 Sept. 2010; published online 21 Oct. 2010

Recommended for acceptance by E. Miller.

For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2010-01-0011.
Digital Object Identifier no. 10.1109/TC.2010.209.

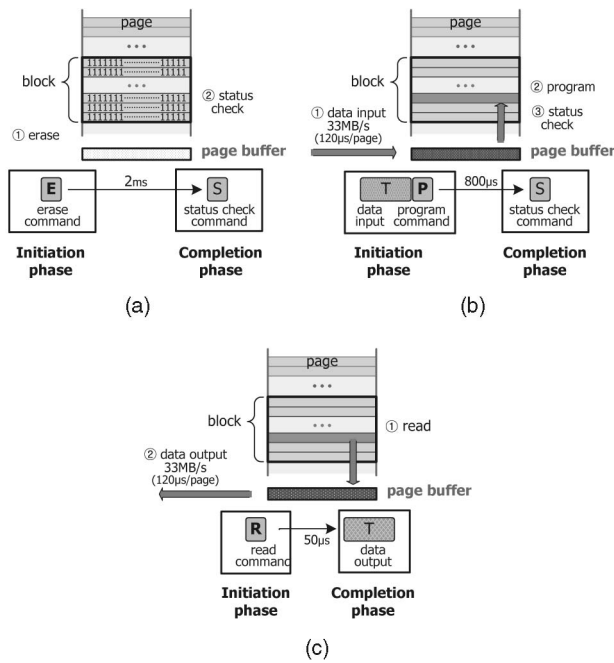


Fig. 1. Flash operations: (a) Erase block, (b) Program page, and (c) Read page.

related work on the FTL, flash memory interleaving, and out-of-order execution. In Section 3, we go on to present three execution models that correspond to different extents to which multichip parallelism can be exploited in a flash storage system. In Section 4, we describe the O3 flash controller architecture in detail. A prototype implementation of the O3 controller and the results of a performance evaluation are presented in Section 5. Finally, we conclude and discuss directions for future research in Section 6.

2 BACKGROUND

2.1 Flash Memory

Data on a NAND flash chip are organized into blocks, each of which, in turn, consists of a number of pages. Each page has a data part for user data and a spare part for metadata associated with the user data, such as mapping and ECC information. The size of the data part is a multiple of the sector size (512 bytes), and the size of the spare part is typically 16 bytes for each sector in the data part. Throughout this paper, we assume that the block size is 512 KB, and that a block consists of 128 pages of 4 KB, although our technique does not rely on this arrangement.

Fig. 1 shows the three flash operations: erase, program, and read. Each of them has an initiation phase and a completion phase, separated by a busy period. The length of the busy period is the latency of the operation. For illustrative purposes, latencies for the three operations are assumed to be 2 ms for erase, 800 μ s for program, and 50 μ s for read for the remainder of the paper, which are typical for an MLC NAND flash chip. Strictly speaking, however, the latencies are not constants and may vary depending on the manufacturer, page number, wear condition, and data being programmed [5]. Similarly, we assume that data input/output is through an 8-bit bus with a cycle time of 30 ns, a typical value in

NAND flash memory nowadays, but emerging interfaces, such as ONFI [6], Hyperlink NAND [7], and Toggle NAND [8], aim to reduce this cycle time.

The erase block operation sets all the bits in a block to one. As shown in Fig. 1a, its initiation phase is invoked by an erase command, and then the chip becomes busy. When this busy period, which is the latency of the operation, is over, the chip becomes ready again. Then, the completion phase begins and a status-check command is sent to the chip, which reports any errors that might have occurred.

The program page operation, shown in Fig. 1b, writes the data supplied to a page that have been erased in advance. In its initiation phase, the data to be written to the target page are transferred over the flash bus to an internal page buffer in the flash chip, and then a program command is sent, together with the address of the target page. Next, the chip becomes busy for the period of the program latency. During the subsequent completion phase, a status-check command is issued in the same way as it is in the erase operation.

The read page operation reads a page from flash memory, as shown in Fig. 1c. During the initiation phase, a read command is sent, together with the address of the page. When the read latency is over and the chip becomes ready again, data are read out from the flash chip during the completion phase.

NAND flash memory is subject to bit-flipping errors, in which one or more bits in a page are reversed between the programming and reading of a page. These can be countered up to a point by error-correction logic in hardware or software. The manufacturers of NAND flash memory also allow chips to have a limited number of bad blocks in order to improve the yield. These bad blocks are identified by a special mark at a designated location in each block. Even good blocks have a limited lifetime, which necessitates a technique called wear-leveling [4] that tries to even out the number of erasures between the blocks in a chip.

2.2 Flash Translation Layer

The FTL hides the idiosyncrasies of flash memory from the host computer and provides a storage device interface like that used by HDDs. The most important role of the FTL is to maintain a mapping between the logical sector addresses used by the host computer and the physical block and page addresses used in flash memory. This mapping can either be at the page level [9], [10] or at the block level [11], [12], [13], [14], [15]. Page-level mapping is more flexible, because it allows a logical page to be mapped to any page in flash memory. In a page-level mapping, the physical blocks in the flash memory are organized into the same sort of log that we find in a log-structured file system [16]. As data arrive, they are written into the next free page at the end of the log. When the number of free pages in the log drops below a given threshold, garbage collection is triggered—a physical block is selected as a victim and all the valid pages (i.e., those whose corresponding logical pages have not been overwritten) in that block are copied to the end of the log. After the copy operation, the victim block is erased and added to the list of free space. The choice of block to be garbage-collected is based on some form of cost-benefit analysis [9], [10], [16], [17].

Page-level mapping is flexible, but suffers from a number of problems. First, it requires a large amount of memory for the mapping table. For example, a 16 GB flash storage system requires a 16 MB mapping table, if each entry takes 4 bytes. However, designs such as DFTL [18] reduce this overhead by maintaining the mapping table in NAND flash while caching frequently accessed entries in memory. Second, the overheads of garbage collection increase sharply as the utilization (i.e., the proportion of valid pages) increases, an effect which is well known in log-structured file systems [10]. This effect can be mitigated by over-provisioning the capacity such that the maximum utilization can be bounded.

In a block-level mapping, each logical sector address is divided into a logical block address and a sector address within that logical block, and only the logical block address is translated into a physical block address. Although block-level mapping is free from the problems of page-level mapping explained above, it requires extra flash operations when only a few pages in a logical block are modified. For example, when there is a request to write to a subset of the pages in a logical block, an operation called a block-merge [11] needs to be performed. During a block-merge operation, the logical block is remapped to a free physical block; program operations are performed to the new physical block for the pages involved in the write request; and all the other pages in that block are copied from the old physical block to the new one. After a block-merge operation, the old physical block is erased and becomes a free block.

2.3 Exploitation of Multichip Parallelism in Flash Memory

Various interleaving techniques have been used in the FTL to exploit multichip parallelism in flash memory. Chang and Kuo proposed chip-level interleaving to improve the write performance in a flash storage system [19]. Although it is focused on program operations, their technique can easily be generalized to read and erase operations. Chip-level interleaving increases the effective bandwidth of flash operations by allowing several flash chips to operate in parallel. However, the maximum bandwidth that can be achieved by chip-level interleaving is still limited by the flash bus bandwidth.

This bandwidth limitation can be overcome by extending the interleaving across multiple flash buses [20], [21], [22]. Bus-level interleaving makes parallel flash operations possible across chips on different flash buses. A set of flash chips involved in both chip-level and bus-level interleaving can be regarded as a single logical chip, sometimes called a super-chip [23]. Concurrent operations on multiple super-chips have been used in a block-mapping FTL [23] and also in a page-mapping FTL [24]. In a block-mapping FTL, concurrent super-chip operations can be used to service host read/write requests and to perform multiple block-merge operations in parallel [23]. In addition, different super-chip operations can be assigned different priorities. For example, the super-chip operations that service host read requests can be given a higher priority than those required to perform background block-merge operations [23]. Concurrent super-chip operations in a page-mapping FTL [24] can be used in an analogous manner, for instance, to spread over several

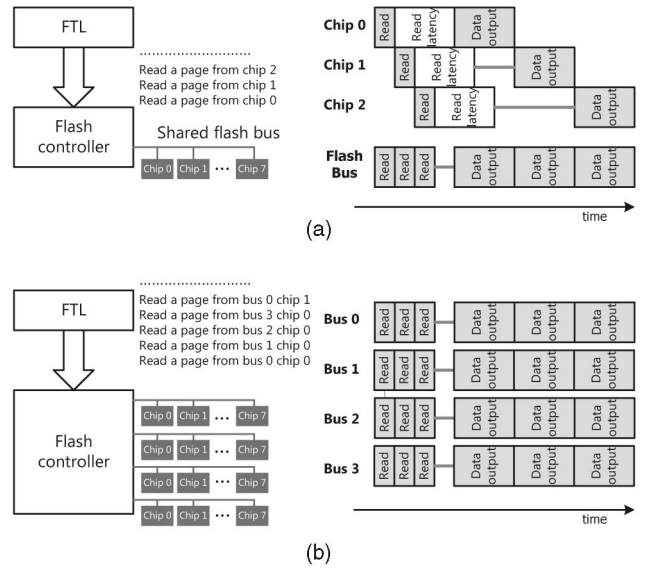


Fig. 2. Chip-level and bus-level interleaving: (a) chip-level interleaving, and (b) bus-level interleaving.

super-chips flash read and program requests arising from host read/write requests and garbage collection.

As Fig. 2 illustrates, super-chip operations are very effective in servicing homogeneous flash operations with a regular access pattern. However, they are less efficient in servicing heterogeneous flash operations with an irregular pattern, as we will see in Section 3.

Agrawal et al. [25] discussed architectural issues that arise in designing flash storage systems, including mapping granularities and the relative merits of interleaved and parallel flash operations, and analyzed their effects on performance using trace-driven simulations. Dirik and Jacob [26] extended this study and performed a comprehensive evaluation of various factors that affect the performance of a flash storage system using traces of typical PC workloads. The factors that they discussed include flash bus organization (e.g., number of buses, bus speed, and bus width), chip-level interleaving, and bus-level interleaving. A key finding of these studies is that the high latencies of flash operations are more critical to the performance of a storage system than the speed of flash buses. This emphasizes the importance of techniques that try to mitigate, or work around, flash latencies, such as those used in the O3 flash controller described in this paper.

There are commercially available SSDs and flash controllers that are believed to exploit multichip parallelism. For example, there is an SSD that provides in excess of 700 MB/s for sequential I/O and 110,000 IOPS (input/output operations per second) for random I/O, with as many as 25 flash buses [27]. There are also flash controllers available in form of IP that support multiple flash chips and buses [28]. However, the architecture and implementation details of these commercial SSDs and flash controllers remain unknown; only their interface specifications are publicly available.

2.4 Out-of-Order Execution and Consistency in Storage Systems

Out-of-order execution is a well-known technique in the computer systems area, which dates back to the 1960s. It

involves techniques for reordering requests, so as to improve performance while guaranteeing that the effect is the same as it would be if the requests were executed sequentially.

A species of out-of-order execution idea called command queuing is used in modern HDDs [29]. With command queuing, read and write requests from the host computer are reordered to minimize the seek time and rotational latency. For example, SATA (Serial ATA) drives use a technique called native command queuing (NCQ) that allows the reordering of up to 32 outstanding requests [30]. SCSI drives employ a similar technique called tagged command queuing (TCQ) [31].

Command queuing is also beneficial to SSDs. By exposing multiple concurrent host requests, it provides more opportunities for SSDs to exploit multichip parallelism in flash memory. This makes especially significant the ability to service flash requests in parallel, as our O3 flash controller is designed to do.

Command queuing in HDDs and SSDs also simplifies the design of redundant array of independent disk (RAID) [32] systems, which have some structural similarities with flash controllers, in that multiple disk drives are controlled by a single RAID controller and multiple flash chips are controlled by a single flash controller. Since each HDD or SSD in a RAID is itself capable of handling multiple requests efficiently through reordering, the RAID controller can concentrate on spreading incoming requests across as many disks as possible, while each disk tries to process the requests that it receives as fast as possible. In the flash context, the FTL concentrates on extracting as many concurrent flash requests as possible while the flash controller tries to process them as fast as possible.

Request reordering, as well as write buffering in disk drives [33] and also in the buffer cache of the operating system [34], causes a file system consistency problem when a system crash occurs. There have been two main approaches to solving this problem, journaling [35] and soft updates [36], [37], [38]. In the journaling approach, a technique called write-ahead logging [35] is used, in which modifications to the file system are recorded to a log before they are made on the disk. This log is replayed during recovery after a system crash to restore the file system consistency. The correct recovery in write-ahead logging requires ordering between write requests arising from the recording to the log and those from the actual modifications. In the soft update approach, write requests from the file system are subject to ordering constraints so as to avoid serious file system inconsistencies after a system crash [36], [37]. Like journaling, soft updates are intended to avoid the need for a costly file system scan, such as a Unix *fsck* [39], when mounting a file system; a background scan at a later time is sufficient to cure minor inconsistencies such as leaked data blocks and inodes.

Both of the two approaches above (i.e., journaling and soft updates) require a mechanism: 1) to specify ordering among write requests generated by the file system, and 2) to preserve that ordering up to the point when the write requests are made durable in the disk drive. Recently, a framework called Featherstitch [40] has been proposed to provide such a mechanism in a modular manner. The Featherstitch framework is general enough to implement both journaling and soft updates; and yet it is practical

enough to be applied to SATA and SCSI drives using techniques such as a write-through mode or a forced unit access (FUA) bit, both of which guarantee the durability of a host write from the time at which the disk drive reports the host write's completion.

3 EXECUTION MODELS

In an HDD, parallelism is limited because it has only one head assembly. On the other hand, a flash storage system offers a lot of potential parallelism, since each flash chip in the system can operate independently of the others. We will now classify the approaches to exploiting multichip parallelism in flash storage systems and define the sequential, decoupled, and out-of-order execution models. These three models impose different constraints on the execution of flash operations by multiple chips and represent different extents to which multichip parallelism is exploited.

In the sequential execution model, shown in Fig. 3a, flash operations are executed in the order in which they arrive. Thus, there is no overlap between flash operations, even when they are directed to different chips. This model is simple and resource-efficient, but its failure to make any use of parallelism limits its application to FTLs that do not generate any concurrent flash requests.

The decoupled execution model allows a partial overlap between flash operations by decoupling the two phases (i.e., the initiation and completion phases) of each operation [41]. This execution model is a generalization of the various interleaving techniques discussed in the previous section. In decoupled execution, the initiation phase of one flash operation can be followed immediately by the initiation of the next operation, unless the latter is directed to a chip that is already busy with an earlier operation. In that case, the flash operation cannot be initiated until the target chip becomes idle, preventing the operations queuing behind it from being initiated. For example, the erase operation addressed to chip 0 in Fig. 3b cannot begin because the initiation of a preceding program operation to chip 2 is blocked due to a chip conflict. Decoupled execution also requires that the completion of operations be strictly ordered. A flash operation cannot complete if any of the flash operations ahead of it have been initiated but are not yet complete. For example, the read operation addressed to chip 1 in Fig. 3b cannot complete because an earlier operation (in this case, the erase operation to chip 0) is still active. Even though the decoupled execution model allows only limited overlapping of flash operations, it has the advantage of naturally preserving the in-order completion semantics.

The drawbacks of decoupled execution motivate the out-of-order execution model, in which there is no requirement for the initiation and completion phases of flash operations to be ordered. The only ordering constraints that remain are due to data dependencies between flash operations directed to the same flash chip. This relaxation of ordering constraints greatly reduces the completion time of a typical set of flash operations, as Fig. 3c illustrates.

Data dependencies between flash operations can easily be determined by considering the set of pages that they read or write. For example, Erase (b), an erase operation on block b , is data-dependent on Erase (b') if and only if $b = b'$. Similarly, Erase (b) is data-dependent on Read (p), a

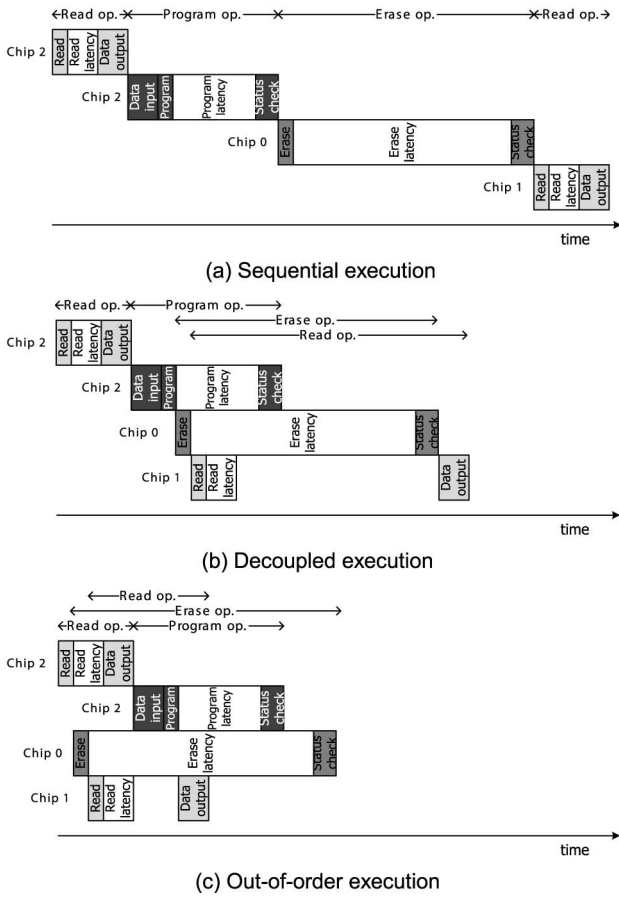


Fig. 3. Models of sequential, decoupled, and out-of-order execution: (a) sequential execution, (b) decoupled execution, and (c) out-of-order execution.

read operation on page p , if and only if p is a page in block b . Other data dependencies are determined similarly, except for that between two flash program operations. We would expect that the only data dependency between Program (p) and Program (p') to occur when $p = p'$. However, since most flash chips require the pages within a block be programmed in an ascending order, we create a data dependency if $p < p'$.

4 O3 FLASH CONTROLLER ARCHITECTURE

Fig. 4 shows the overall architecture of the O3 flash controller within the context of a flash storage system that interacts with a host computer. The host computer makes read and write requests specified in terms of logical sectors to the flash storage system. In flash storage systems, the FTL translates host requests into flash requests. In addition to this translation, the FTL needs to perform management tasks, such as garbage collection and wear-leveling at any point in time. Flash requests that originate from the same FTL task, whether for host request servicing or internal management, are identified as a stream throughout this paper.

The O3 flash controller services flash requests without any intervention by the FTL. We take this hardware-oriented approach because the flash latencies are relatively short (in the range of a few tens or hundreds of microseconds). If we involved the FTL in scheduling flash operations, Amdahl's law [42] dictates that we would be introducing a serial bottleneck that would limit the maximum speed-up from multiple chips. This contrasts with the situation with HDDs, in which media access latencies are very long (in the range of tens of milliseconds), and thus scheduling requests in software is a sensible option. The two different choices are analogous to different choices used in optimizing different levels in a memory hierarchy where the CPU cache is managed by hardware and the virtual memory by software for a similar reason.

The FTL interacts with the O3 flash controller using a pair of FIFO queues, called the inbound and outbound FIFOs. The two FIFO queues provide buffering for requests and responses, both formatted as packets, between the FTL and the O3 controller. All the operations sent to the O3 flash controller by the FTL, including configuration, reset, and flash erase/program/read operations, use this packet-based interface. Although this decoupling misses opportunities for more intelligent scheduling of flash operations by the FTL, we believe its benefits in terms of modularity and extensibility outweigh the missed opportunities. For example, programmers of the FTL can treat the controller as a black box, relieving them of the burden of handling the low-level details of flash operations. The decoupling also makes it

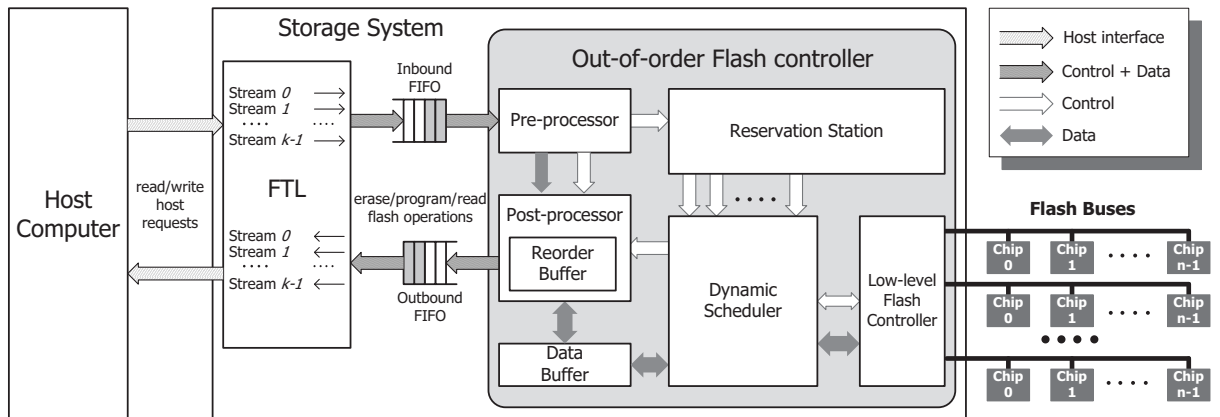


Fig. 4. Architecture of the O3 flash controller.

feasible for the FTL to be executed remotely in the host computer or even over a network. The packet-based interface enhances not only modularity but also extensibility. For example, an ECC encoder/decoder pair can be plugged between the FTL and the flash controller in a manner transparent to both.

When the O3 controller receives a request packet from the inbound FIFO, a preprocessor decodes the request and places it in a reservation station [43], where the request is buffered while awaiting dispatch to the target flash chip by a dynamic scheduler. In the case of a flash program request, the associated data are temporarily stored in a data buffer in the O3 controller for transfer to the target flash chip when the program request is dispatched. This data buffer is also used to store the data obtained by a flash read request before it is placed in a response packet and delivered to the FTL.

The dynamic scheduler in the O3 controller dispatches the flash requests buffered in the reservation station separately to each chip, so that the dispatch of operations to one chip is independent of the operations sent to other chips. Each of the two phases of a flash operation is also dispatched separately. When the target chip becomes ready, the dynamic scheduler makes the initiation request for the next flash operation addressed to that chip in the reservation station (I-REQ). After the flash latency is over and the chip becomes ready again, it sends the completion request (C-REQ) for that flash operation. In the period between the I-REQ and the C-REQ, flash operations to other chips can be handled concurrently by the dynamic scheduler.

Each phase (I-REQ or C-REQ) of a flash operation involves the delivery of a command and/or the transfer of data to or from the target chip over the flash bus. The low-level flash controller in Fig. 4 encapsulates all the details of low-level signaling needed for command delivery and data transfer for a given flash bus, which improves the portability of the O3 flash controller. This low-level controller is also responsible for relaying the ready and busy status of each chip to the dynamic scheduler. This information is needed for correct and timely dispatch of flash requests in the reservation station.

The dynamic scheduler also has a two-level prioritized request handling mechanism to provide appropriate levels of service to flash requests belonging to different request streams. High-priority request streams are mainly used for flash requests corresponding to pending host requests. Low-priority streams contain less urgent flash requests, such as those for garbage collection.

The postprocessor shown in Fig. 4 creates the illusion that flash operations belonging to the same stream have been completed in order by the use of a reorder buffer [43]. The reorder buffer temporarily stores flash requests that have been completed by the dynamic scheduler, and enforces in-order completion among those belonging to the same request stream. The postprocessor is also responsible for generating the response packet for each flash request as it leaves the reorder buffer. Outgoing response packets are placed into the outbound FIFO for delivery to the FTL.

It should be noted that the out-of-order execution of program operations directed to different chips can cause a consistency problem if a system crash occurs. For this reason, if a host write is in a write-through mode or the

FUA bit is set, we assume that the FTL will not report the completion until it receives the response packet for the final program operation involved in the processing of the host write. This enables safe recovery from a crash regardless of whether the journaling approach or the soft update approach is used.

In the following sections, we provide more details about the key features of the O3 flash controller, which are its packet-based interface, dynamic scheduler, and prioritized request handling.

4.1 Packet-Based Interface

As we have already outlined, the O3 flash controller provides a packet-based interface using the inbound and outbound FIFOs. The FTL uses this interface to send multiple streams of flash requests encoded in packets. There are two types of packet used in this communication: control packets are used for commands and status reports that have no associated data, and data packets are used for data transfers during the initiation phase of a program operation or the completion phase of a read operation. The data transferred include not only the data part of the requested page, but also the spare part that contains metadata for the page, such as mapping and ECC information.

In addition to the usual fields, such as opcode, priority, and packet size, together with the bus, chip, block, and page numbers, the header of both types of packet includes three fields for interrupt processing: "IR" (Interrupt Request), "IC" (Interrupt Condition), and "IP" (Interrupt Pending). The IR field, which is set by the FTL, requests an interrupt, and the IC field specifies the conditions attached to it. The FTL uses an unconditional interrupt so that it can be informed of the completion of a set of flash operations, which can be detected economically because the completion of a flash operation implies that all the operations ahead of it in the same stream are also complete. Conditional interrupts are used to notify the FTL of unusual events, such as errors during erase or program operations. These arrangements relieve the FTL of the burden of checking the status of every flash operation. The IP field, which is set by the O3 controller, indicates that an interrupt is pending in response to a (conditional or unconditional) interrupt requested by the FTL. The setting of the IP field triggers an interrupt that leads to the execution of the corresponding interrupt service routine within the FTL.

As an example, consider the sequence of packet exchanges between the FTL and the O3 flash controller for an erase operation. First, the FTL sends to the O3 flash controller a sequence of two control packets, one containing an erase command that will initiate the erase operation and the other a status-check command to be executed on completion. Second, the two packets are translated by the preprocessor into the I-REQ and C-REQ of the erase operation, respectively, and placed into the reservation station. Finally, after completion of the erase operation, the O3 controller sends a control packet to the FTL setting the IP field if there is an error (provided that the FTL has been asked to be interrupted when this occurs). The scenario for a program operation is the same, except that the FTL punctuates the two control packets with a data packet containing the data for the target page. In this case, the first

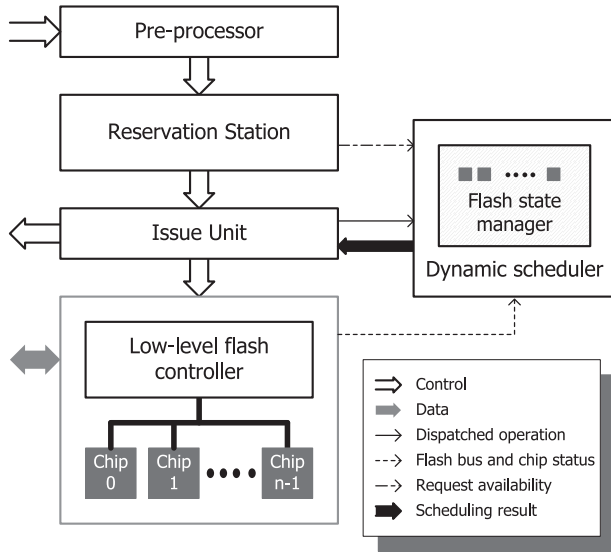


Fig. 5. Dynamic scheduling in the O3 flash controller.

control packet and the data packet correspond to the I-REQ, and the second control packet to the C-REQ.

To perform a read operation, the FTL sends a control packet containing a read command for initiation (I-REQ) and a data packet for completion (C-REQ). In response, the O3 flash controller sends a data packet that is the same as the data packet from the FTL except that it now contains the data read from flash memory.

A read operation has no intrinsic way of determining whether the data that are read from flash memory are correct, and so read operations do not have the facilities to report errors that are included in the erase and program operations. Thus, bit-flipping errors are handled separately. An ECC encoder is placed before the inbound FIFO and an ECC decoder after the outbound FIFO, making error correction transparent to both the FTL and the O3 flash controller. In the rare event, where bit-flipping errors are too extensive to be corrected, the ECC decoder triggers an interrupt by setting the IP field of the data packet involved so that the FTL can take an appropriate action.

4.2 Dynamic Scheduler

We have seen how the FTL and the O3 flash controller interact with each other using a packet-based interface. Flash requests received over this packet-based interface are decoded by the preprocessor and the corresponding I-REQs and C-REQs are placed into the reservation station. Fig. 5 shows the dynamic scheduling of flash operations for the out-of-order execution model. The O3 scheduler is general enough to emulate the other two models if the structure of the reservation station is changed, as shown in Fig. 6. The single request queue shown in Fig. 6a suffices for sequential execution but the decoupled model needs separate initiation and completion queues, as shown in Fig. 6b. The initiation queue ensures the ordering of I-REQs and the completion queue ensures the ordering of C-REQs. For out-of-order execution, a separate queue is needed for each chip, as shown in Fig. 6c.

Scheduling for the sequential execution model is trivial: when the target chip for the I-REQ or C-REQ at the head of

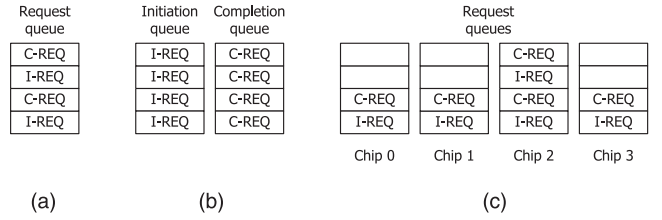


Fig. 6. Reservation stations for the three execution models: (a) sequential, (b) decoupled, and (c) out-of-order.

the (single) request queue is ready, the scheduler instructs the issue unit to dispatch the request to the chip. The issue unit fetches the control information about the request from the reservation station and issues it to the low-level flash controller responsible for the flash bus to which the target chip is connected.

Decoupled execution is also straightforward. The scheduler dispatches a request using the issue unit when the target chip becomes ready; this request is either the I-REQ at the head of the initiation queue or the C-REQ at the head of the completion queue. If both chips are ready at the same time, then priority is given to the request that does not involve data transfer so that the dispatch of the other request is not delayed by a long data transfer time.

In the out-of-order execution, each chip is handled separately in two stages. In the first stage, all pending I-REQs and C-REQs that are addressed to idle chips and do not involve data transfers are scheduled. In the second stage, for each flash bus, the dynamic scheduler scans the chips, starting where it left off during its last invocation for this bus, to search for an idle chip with a pending I-REQ or C-REQ with a data transfer. If there is such a request, the dynamic scheduler schedules it and moves to the next flash bus. When all the flash buses have been processed, the dynamic scheduler suspends itself until it is invoked again, either by completion of an I-REQ or a C-REQ, or by the arrival of a new flash operation addressed to an idle chip.

This scheduling algorithm for out-of-order execution has two aims. First, it tries to keep as many flash chips active as possible by prioritizing requests without data transfer. This helps prevent these faster requests from being blocked by long data transfers. The requests that involve data transfers are scheduled in a round-robin manner to make fair use of the bus bandwidth among the chips on the bus, and simplify implementation.

To maintain the ready/busy status of each flash chip that it manages, the dynamic scheduler has a flash state manager, which implements the finite-state machine shown in Fig. 7. States can be changed by two types of event: synchronous events triggered by the issue unit in response to an I-REQ or a C-REQ dispatched by the dynamic scheduler, and asynchronous events which occur when a completed request causes a flash chip to become ready.

As an example, consider the sequence of transitions in the flash state manager during a read operation. When the chip is in the IDLE state, the scheduler dispatches the I-REQ. This synchronous event triggers a transition to the BUSY state. When the flash read latency is over, the chip signals that it is ready again, which is an asynchronous event. The state manager transitions to the DATA OUTPUT state, and the

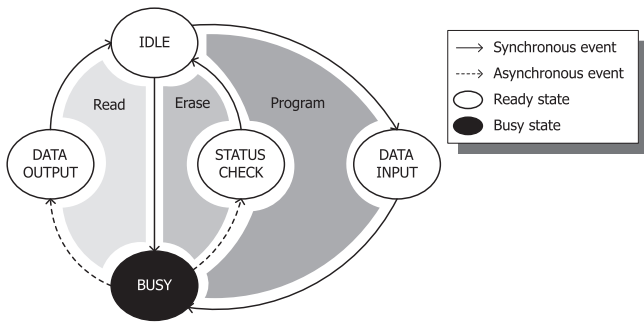


Fig. 7. Flash chip state transitions.

scheduler dispatches the C-REQ to the issue unit to transfer data from the chip to complete the read operation. When the data transfer is over, the finite-state machine returns to the IDLE state. Note that this example explains the transitions undergone by the target chip of a single flash operation, and the remaining chips are likely to be undergoing transitions by other concurrent flash operations.

Fig. 8 shows how a sequence of flash operations is scheduled by the three different flash controllers corresponding to the three execution models, leading to very different completion times. In the figure, R_j^i denotes a flash read operation to chip i with $request_id = j$ and P_j^i a flash program operation defined similarly.

4.3 Prioritized Request Handling

The O3 flash controller can prioritize the handling of different streams of flash operations. Prioritized handling is performed both at the chip level and at the bus level. At the chip level, there are separate queues for high-priority requests and low-priority requests, and requests from the high-priority queue are always selected first.

Low-priority requests are further constrained at the bus level by a mechanism that we call the flash reservation protocol (F-RSVP), which provides different levels of service to requests with high and low priorities. Reservation for prioritization is achieved by limiting the number of chips that can be occupied by low-priority requests at any given time; we denote this number by $F-RSVP(\ell)$. This mechanism only permits a low-priority request to be dispatched to an idle chip when the number of chips currently occupied by low-priority requests is less than $F-RSVP(\ell)$. As an anonymous reviewer of this paper noted, a similar effect could be obtained by a randomized algorithm, where low-priority requests are delayed for a random amount time in a probabilistic manner.

5 PROTOTYPE IMPLEMENTATION AND PERFORMANCE EVALUATION

5.1 Experimental Setup

We implemented the sequential, decoupled, and O3 flash controllers using an in-house development board, which has a Xilinx Virtex 5 FPGA (XC5VFX130T) [44] with two embedded PowerPC440 processors. The development board also has five NVRAM slots, each supporting two NAND flash buses. The flash bus is 8 bits wide and operates at 32 MHz, and thus the maximum bandwidth of the

flash bus is 32 MB/s. Table 1 gives the characteristics of the flash chip we used in the experiments. The board is also equipped with 128 MB mobile DDR memory, PCI-e host/device interfaces, SATA interfaces, an Ethernet interface, and various types of extension slot for future developments. More details about the flash memory development board can be found elsewhere [45].

We compared the performances of the sequential, decoupled, and O3 controllers using the experimental setup shown in Fig. 9. The processor subsystem in the figure consists of one PowerPC440 processor in the FPGA and various peripherals that we implemented using the Xilinx ISE and EDK tools [46]. Flash request packets were prepared offline, and then fetched from the mobile DDR memory by the request forwarder, which forwards them to the inbound FIFO for processing by the flash controller. The packet monitor connected to the request forwarder records the timestamps of all the inbound and outbound packets in real-time using a timer circuit and stores the timestamps in its local SRAM. These timestamps are later used to determine throughput and response time. In the experiments, we ran the PowerPC440 processor at 200 MHz and the DDR memory at 133 MHz. Table 2 summarizes the utilizations of different types of FPGA resource by the three flash controllers. The O3 controller uses about 25 percent more resource than the decoupled controller, which, in turn, uses about twice as much resource as the sequential controller.

5.2 Performance Evaluation (Synthetic Workloads)

We built a synthetic workload generator that can generate arbitrary mixes of erase, program, and read operations, and used it to assess the performance of the three flash controllers with various configurations and a wide range of workloads. Results using real workload traces will be given in Section 5.3.

5.2.1 Effects of Workload Variation

Fig. 10a compares the performance of the three controllers with different mixes of erase (E), program (P), and read (R) operations. In this experiment, we used eight flash chips on a single bus. The throughput of the O3 controller is 3 to 100 percent greater than that of the decoupled controller, and 23 to 443 percent greater than that of the sequential controller. Moreover, the performance of the O3 controller is largely insensitive to the workload type and its throughput approaches the maximum flash bus bandwidth of 32 MB/s.

The performance of the other two controllers is more dependent on the composition of the workload. For example, when there are a large number of low-latency read operations, even the sequential controller performs reasonably well. However, introducing more erase and program operations widens the performance gap between the O3 controller and the other two controllers. In the extreme case, where the workload does not contain any read operation and consists only of erase and program operations, the O3 controller performs almost five times better than the sequential controller, and twice as well as the decoupled controller. These results illustrate the effectiveness of relaxing the ordering constraints between flash operations in the O3 controller to hide long latencies.

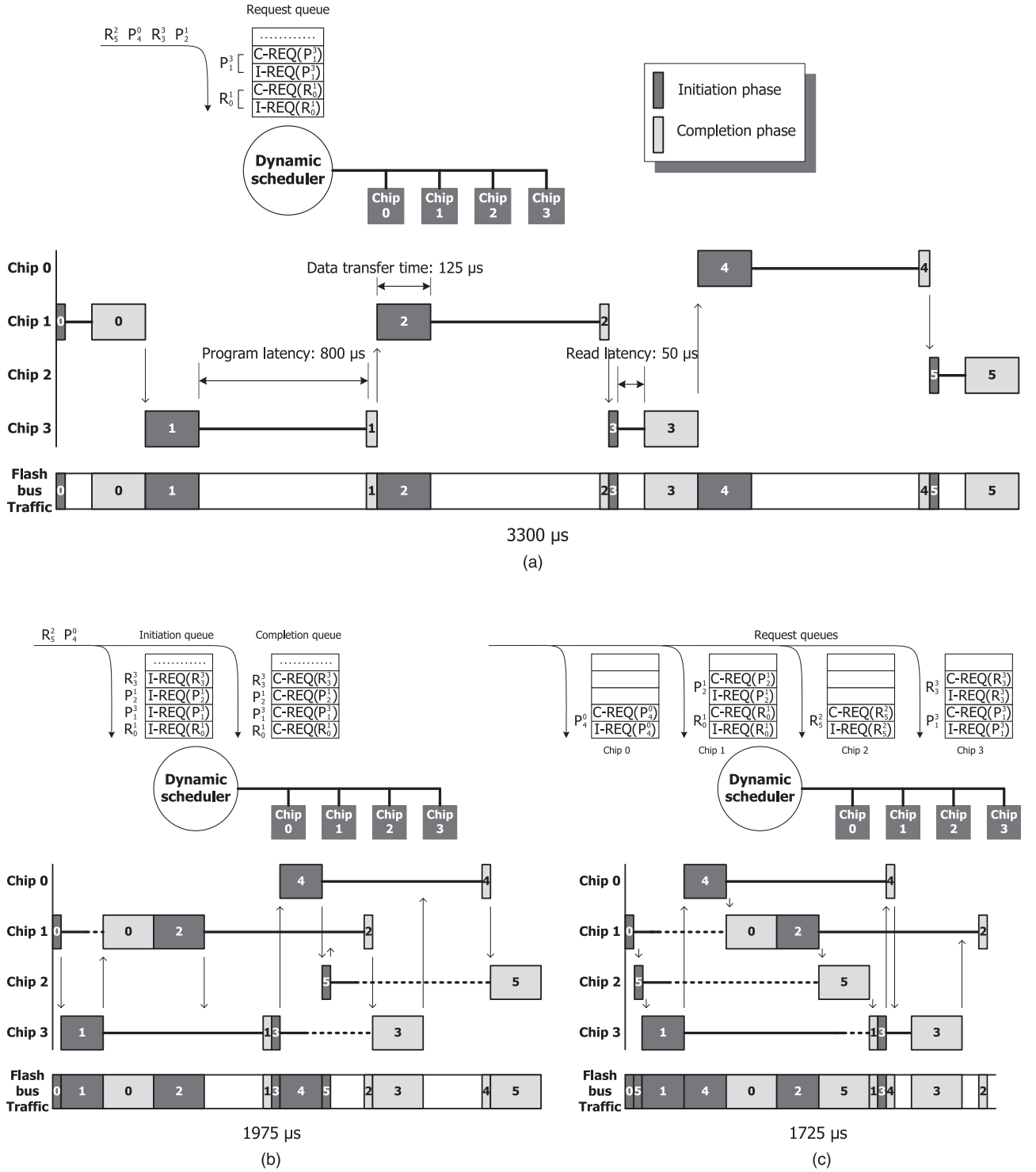


Fig. 8. Timing diagrams for a sequence of flash operations with three execution models: (a) sequential execution, (b) decoupled execution, and (c) out-of-order execution.

Fig. 10b shows the distribution of response times when the ratio of erase, program, and read operations is 1:128:128, modeling a scenario in which each page in a block is programmed and read once, on average, after the block is erased. The average response time of the O3 controller for the different types of operation is 46 to 88 percent better than that of the decoupled controller, and 70 to 94 percent better than that of the sequential controller.

5.2.2 Effects of the Number of Flash Buses and the Number of Chips per Bus

To assess the scalability of the three controllers, we performed experiments in which we varied the number of flash buses and also the number of chips on each bus. The results are shown in Fig. 11a for Erase:Program:Read = 1 : 128 : 128 and in Fig. 11b for Erase : Program : Read = 1 : 128 : 512. As expected, the throughput of the sequential controller does

TABLE 1
Flash Memory Parameters

Parameters	Value
Page read	56 μ s - 62 μ s
Page program	524 μ s - 1062 μ s
Block erase	2 ms
I/O cycle time	30 ns (min)
Page size	4KB + spare area
Block size	128 pages

not increase with either the number of buses or the number of chips on each bus.

The gap in performance between the O3 controller and the decoupled controller increases as we use more flash buses and chips. The O3 controller achieves throughputs of over 90 percent of the maximum, regardless of the number of buses when there are eight chips or more on each bus.

5.2.3 Effects of Prioritized Handling

To assess the effects of prioritized request handling on performance in a realistic setting, we prepared four different streams of flash requests. The first is a high-priority stream that contains only read requests emulating the processing of host read requests by the FTL. The remaining three low-priority streams contain a mixture of erase, program, and read requests emulating host write processing and garbage collection inside the FTL.

We also varied the value of $F\text{-RSVP}(\ell)$, which is the maximum number of chips that can be occupied in servicing low-priority requests at the same time. Fig. 12 shows that, as $F\text{-RSVP}(\ell)$ decreases, responses to high-priority requests gradually become faster and responses to low-priority requests become slower, as we would expect.

Fig. 13 shows the throughputs of the three controllers for different values of $F\text{-RSVP}(\ell)$. As expected, the throughput of the sequential controller is the same regardless of $F\text{-RSVP}(\ell)$. For the decoupled and O3 controllers, the total throughput by both high-priority and low-priority requests naturally decreases as we decrease $F\text{-RSVP}(\ell)$ (remember that a smaller $F\text{-RSVP}(\ell)$ value increases the probability that a low-priority request is prevented from being dispatched to a chip even in the case the chip is idle).

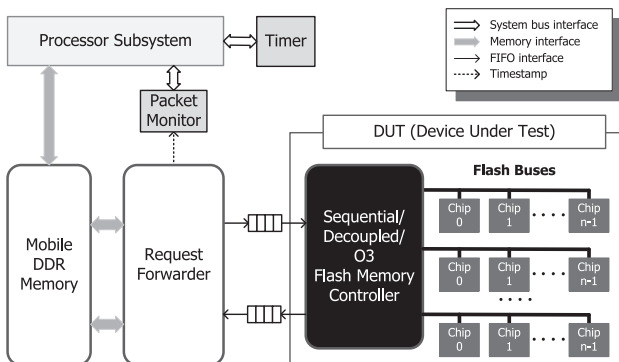


Fig. 9. Experimental setup.

TABLE 2
FPGA Resource Utilization by the Three Flash Controllers

Name	Sequential	Decoupled	O3
Slice-registers	2,833(3%)	5,021(6%)	6,253(7%)
Slice-LUTs	3,897(4%)	8,288(8%)	11,453(13%)
Occupied slices	1,428(6%)	3,067(14%)	4,292(20%)
Bonded I/O	311(37%)	311(37%)	311(37%)
Block RAM	40(13%)	40(13%)	59(19%)

When low-priority requests can only use one chip at a time, both the decoupled and O3 controllers perform as badly as the sequential controller, because the low-priority requests have to be executed sequentially. This increases the backlog of low-priority requests, filling buffers in the controller, including those in the reservation station. This build-up of full buffers can even reach the inbound FIFO, which blocks high-priority requests before they have even entered the controller. This suggests the need for a flow control mechanism in the FTL to limit the number of outstanding low-priority requests so that they do not block the processing of high-priority requests.

5.3 Performance Evaluation (PCMark05)

We also conducted experiments using traces obtained from running the PCMark05 benchmark [47], which emulates

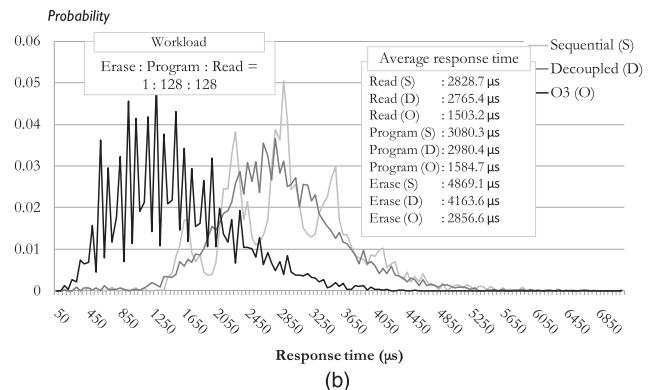
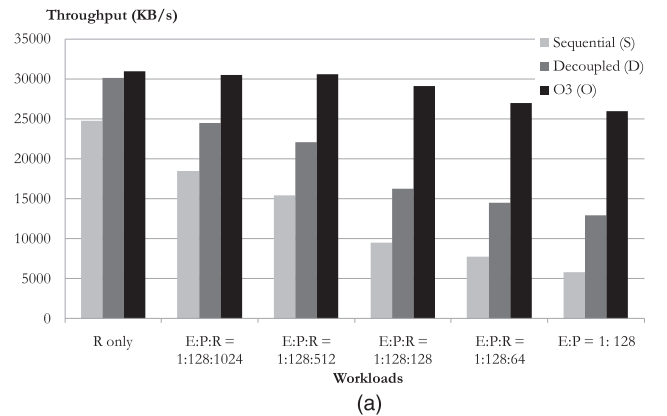


Fig. 10. Performance of the sequential, decoupled, and O3 controllers with different workloads: (a) throughput, and (b) response times.

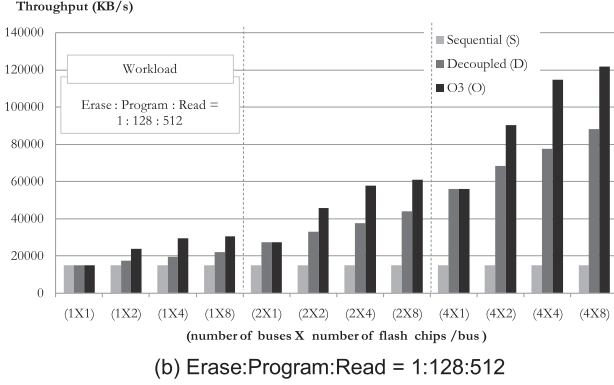
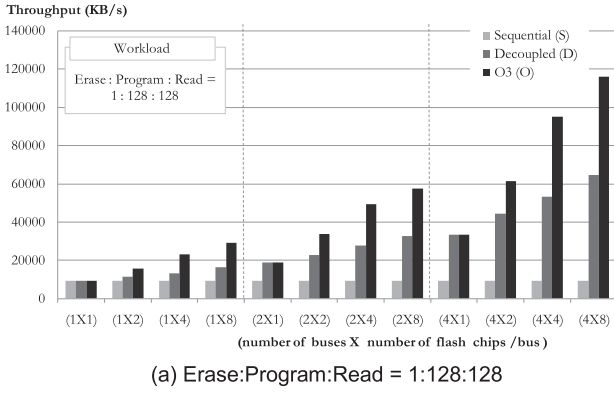


Fig. 11. Effects of number of buses and number of chips per bus.

workloads from a typical PC environment. This benchmark has five components: *XP Start-up*, *Application Loading*, *General Usage*, *Virus Scan*, and *File Write*. *XP Start-up* replays read and write requests made during a Windows XP boot-up. About 90 percent of its requests are for reading and 10 percent for writing. *Application Loading* contains host requests made when application programs, such as Microsoft Word, Adobe Acrobat Reader, and Mozilla Internet Browser, are launched and terminated. It consists of 83 percent reads and 17 percent writes. *General Usage* contains host requests from application programs, such as Microsoft Word, Winzip, Winamp, Microsoft Internet Explorer, and Windows Media Player. It has 60 percent reads and 40 percent writes. *Virus Scan* contains host requests for scanning 600 MB of files for viruses. As expected, its requests are mostly (99.5 percent)

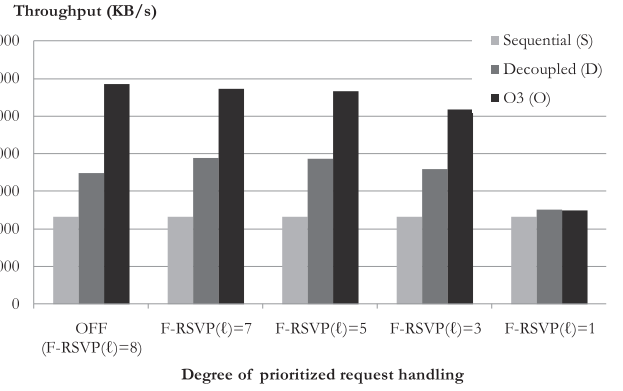


Fig. 13. Effects of prioritized request handling on throughput.

reads. Finally, *File Write* contains the host requests for writing 680 MB of files, and does not contain any read requests.

To obtain traces of the flash requests generated during the execution of the PCMark05 benchmark, we implemented a parallelized page-mapping FTL similar to the one explained in [24]. For a host write request, the FTL determines the number of pages to be programmed from the size of the host request and the flash page size. It then randomly selects the chips to be programmed and generates program requests to these chips after updating the mapping table. For a host read request, the FTL looks up the mapping table and generates a set of flash read requests.

Fig. 14a gives the throughput for the five component benchmarks along with the ratio between erase, program, and read operations in each. These results are similar to those in Fig. 10a. For example, the result for *File Write*, which contains only erase and program operations is almost identical to that in Fig. 10a for $E : P = 1 : 128$. Similarly, the result for *Virus Scan*, in which most requests are reads, is very similar to that for the “R only” case in Fig. 10a.

The distribution of response times is also very similar. For example, the response times shown in Fig. 14b for *General Usage* with Erase:Program:Read = 1 : 126.3 : 110.75 are almost the same as those in Fig. 10b with Erase:Program:Read = 1 : 128 : 128. Overall, the results in Fig. 14 confirm that the performance of the three controllers is most affected by the ratio between the number of erase, program,

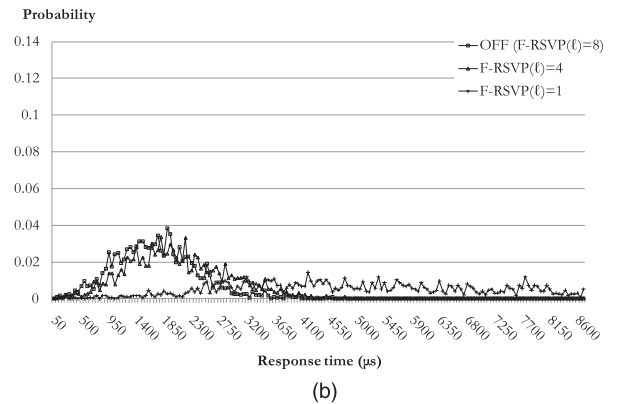
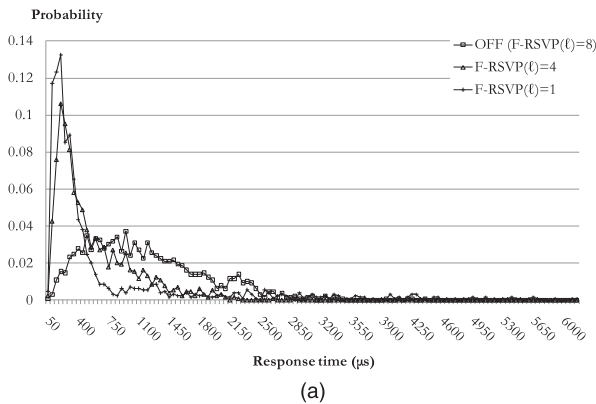


Fig. 12. Effects of prioritized request handling on response times: (a) response times of high-priority requests, and (b) response times of low-priority requests.

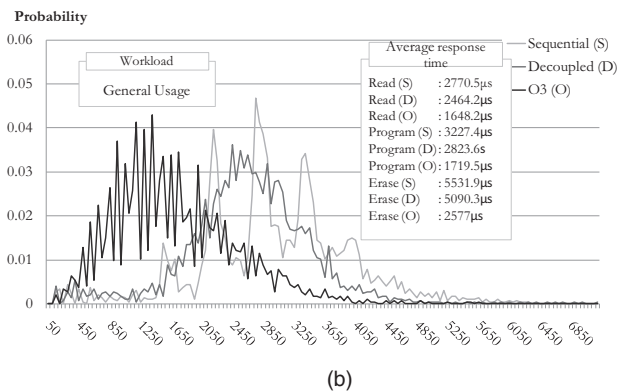
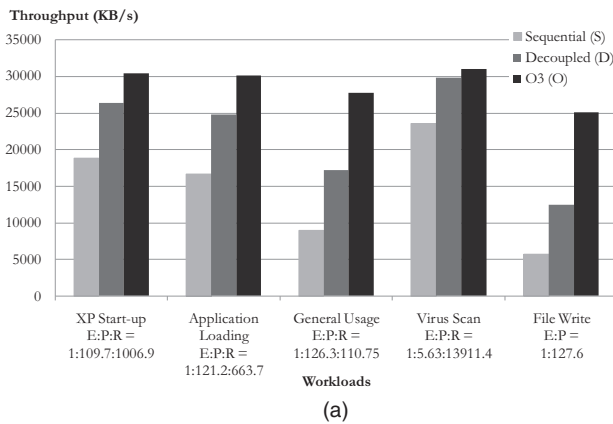


Fig. 14. Performance results from PCMark05 traces: (a) throughput, and (b) response time.

and read operations in the workload. They also confirm our earlier observation that the performance difference between the O3 controller and the other two controllers is more noticeable when there are more long-latency erase and program operations than short-latency read operations.

To assess the performance impact of the degree of randomness in selecting target chips for flash operations, we varied the degree of randomness with Erase:Program: Read = 1 : 128 : 128. When there is no randomness (0 percent), the target chip is selected in a round-robin manner. Then, we progressively expanded the set of candidate chips for each flash operation, choosing at random within that set. The size of the set, as a proportion of the total number of chips, is taken to be the degree of randomness. When this reaches 100 percent, we are selecting chips in the same way as we did for results in Fig. 14. This experiment with different degrees of randomness was designed to model the separate streams of flash operations that are generated by the FTL in processing host reads and writes, in garbage-collection, and in wear-leveling. We note that the merger of streams of operations from several processes will exhibit a degree of randomness, even though each stream is actually selecting its target chips in a round-robin manner.

The results in Fig. 15 show that the O3 controller always performs better than the other two controllers, and its performance is largely insensitive to the degree of randomness. On the other hand, the performance of the decoupled controller is adversely affected by randomness; even if the degree of randomness is only 20 percent, its performance is less than 75 percent of that of the O3 controller.

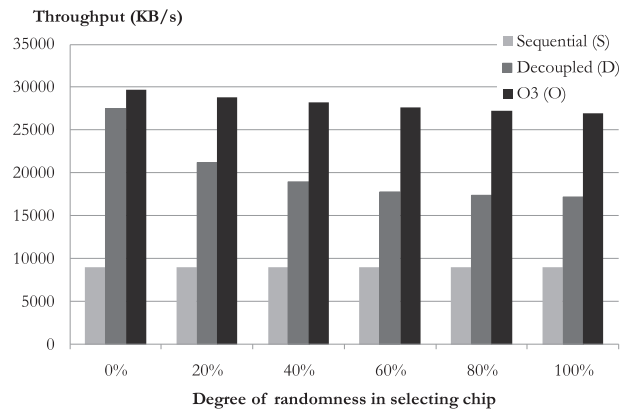


Fig. 15. Effects of randomness in selecting the target chips on throughput.

6 CONCLUSIONS

We have presented a flash controller called Ozone (O3) that executes multiple flash operations out of order. The O3 flash controller has the following properties: First, data dependencies between flash operations are the only ordering constraints on their concurrent execution, enabling much better exploitation of multichip parallelism than interleaving. Second, it presents a packet-based interface to flash management software, such as the FTL (flash translation layer), enhancing its modularity and extensibility. Third, it provides prioritized handling of flash operations, allowing appropriate levels of service to be given to different streams of flash operations.

Evaluation using an FPGA implementation of the O3 flash controller showed that its performance is between three percent and 100 percent better in terms of throughput and between 46 percent and 88 percent better in terms of response time than interleaving, for the workloads that we considered.

The O3 controller can free programmers of flash management software from the burden of handling low-level details of flash operations to concentrate on extracting as many parallel streams of flash operations as possible. We are currently exploring different approaches to parallelizing flash management software to realize the full potential of the O3 controller.

ACKNOWLEDGMENTS

The authors thank associate editor Professor Ethan L. Miller and the anonymous reviewers for their very detailed and helpful feedback. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MEST) (No. 2010-0015149). ICT at Seoul National University provided research facilities for this study. Hyeonsang Eom is the corresponding author for this paper.

REFERENCES

- [1] C.-G. Hwang, "Nanotechnology Enables a New Memory Growth Model," *Proc. IEEE*, vol. 91, no. 11, pp. 1765-1771, Nov. 2003.

- [2] G.E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics*, vol. 38, no. 8, pp. 114-117, Apr. 1965.
- [3] Samsung Electronics, Datasheet: 1G x 8 Bit/2G x 8 Bit NAND flash memory, http://www.datasheetcatalog.org/datasheets2/12/1244179_1.pdf, Mar. 2005.
- [4] E. Gal and S. Toledo, "Algorithms and Data Structures for Flash Memories," *ACM Computing Surveys*, vol. 37, no. 2, pp. 138-163, 2005.
- [5] L.M. Grupp, A.M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P.H. Siegel, and J.K. Wolf, "Characterizing Flash Memory: Anomalies, Observations, and Applications," *Proc. 42nd Ann. IEEE/ACM Int'l Symp. Microarchitecture*, 2009.
- [6] ONFI, ONFI 2.3 Specification, <http://onfi.org/specifications/>, Aug. 2010.
- [7] R. Schuetz, O. HakJune, K. Jin-Ki, P. Hong-Beom, S.A. Przybylski, and P. Gillingham, "HyperLink NAND Flash Architecture for Mass Storage Applications," *Proc. 22nd IEEE Non-Volatile Semiconductor Memory Workshop*, pp. 3-4, 2007.
- [8] Samsung Electronics, Toggle DDR NAND Flash, http://www.samsung.com/global/business/semiconductor/products/flash/Products_Toggle_DDR_NANDFlash.html, 2010.
- [9] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory Based File System," *Proc. USENIX Ann. Technical Conf.*, 1995.
- [10] M. Wu and W. Zwaenepoel, "ENVy: A Non-Volatile, Main Memory Storage System," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 1994.
- [11] J. Kim, J.M. Kim, S.H. Noh, S.L. Min, and Y. Cho, "A Space-Efficient Flash Translation Layer for Compactflash Systems," *IEEE Trans. Consumer Electronics*, vol. 48, no. 2, pp. 366-375, May 2002.
- [12] A. Ban and R. Hasharon, "Flash File System Optimized for Page-Mode Flash Technologies," United States Patent, no. 5,637,425, Aug. 1999.
- [13] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, "A Superblock-Based Flash Translation Layer for NAND Flash Memory," *Proc. Sixth ACM and IEEE Int'l Conf. Embedded Software*, 2006.
- [14] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation," *ACM Trans. Embedded Computing Systems*, vol. 6, no. 3, July 2007.
- [15] C.-H. Wu and T.-W. Kuo, "An Adaptive Two-Level Management for the Flash Translation Layer in Embedded Systems," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design*, 2006.
- [16] M. Rosenblum and J.K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Trans. Computer Systems*, vol. 10, no. 1, pp. 26-52, 1992.
- [17] M.-L. Chiang, P.C.H. Lee, and R.-C. Chang, "Using Data Clustering to Improve Cleaning Performance for Flash Memory," *Software Practice and Experience*, vol. 29, no. 3, pp. 267-290, 1999.
- [18] A. Gupta, Y. Kim, and B. Ugaonkar, "DFTL: A Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings," *Proc. 14th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, 2009.
- [19] L.-P. Chang and T.W. Kuo, "An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems," *Proc. Eighth Real-Time and Embedded Technology and Applications Symp.*, 2002.
- [20] C. Park, P. Talawar, D. Won, M. Jung, J. Im, S. Kim, and Y. Choi, "A High Performance Controller for NAND Flash-Based Solid State Disk (NSSD)," *Proc. 21st IEEE Non-Volatile Semiconductor Memory Workshop*, 2006.
- [21] J.-U. Kang, J.-S. Kim, C. Park, H. Park, and J. Lee, "A Multi-Channel Architecture for High-Performance NAND Flash-Based Storage System," *J. Systems Architecture*, vol. 53, no. 9, pp. 644-658, 2007.
- [22] J.H. Yoon, E.H. Nam, Y.J. Seong, H. Kim, B.S. Kim, S.L. Min, and Y. Cho, "Chameleon: A High Performance Flash/FRAM Hybrid Solid State Disk Architecture," *IEEE Computer Architecture Letters*, vol. 7, no. 1, pp. 17-20, Jan.-June 2008.
- [23] Y.J. Seong, E.H. Nam, J.H. Yoon, H. Kim, J.-Y. Choi, S. Lee, Y.H. Bae, J. Lee, Y. Cho, and S.L. Min, "Hydra: A Block-Mapped Parallel Flash Memory Solid-State Disk Architecture," *IEEE Trans. Computers*, vol. 59, no. 7, pp. 905-921, July 2010.
- [24] A.M. Caulfield, L.M. Grupp, and S. Swanson, "Gordon: Using Flash Memory to Build Fast, Power-Efficient Clusters for Data-Intensive Applications," *Proc. Architectural Support for Programming Languages and Operating Systems*, 2009.
- [25] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," *Proc. USENIX Ann. Technical Conf.*, 2008.
- [26] C. Dirik and B. Jacob, "The Performance of PC Solid-State Disks (SSDs) as a Function of Bandwidth, Concurrency, Device Architecture, and System Organization," *Proc. 36th Int'l Symp. Computer Architecture*, 2009.
- [27] FUSION IO, ioDrive datasheet, http://www.fusionio.com/images/data-sheets/iodevice_data_sheet.pdf, 2008.
- [28] Denali, NAND Flash Controller IP, http://www.denali.com/en/products/databahn_flash.jsp, 2010.
- [29] Intel, Serial ATA II Native Command Queuing Overview, <http://www.intel.com/assets/pdf/whitepaper/252664.pdf>, 2003.
- [30] K. Grimsrud and H. Smith, *Serial ATA Storage Architecture and Applications*. Intel Corporation, 2007.
- [31] Int'l Committee for Information Technology Standards (INCITS), *SCSI Architecture Model-3 (SAM-3) T10 Project 1561-D, Revision 14*. pp. 402-2005, T10 Technical Committee, Sept. 2004.
- [32] D.A. Patterson, G. Gibson, and R.H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, 1988.
- [33] C. Ruemmler and J. Wilkes, "An Introduction to Disk Drive Modeling," *Computer*, vol. 27, no. 3, pp. 17-28, Mar. 1994.
- [34] M.J. Bach, *The Design of the UNIX Operating System*. Prentice-Hall, 1986.
- [35] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," *ACM Trans. Database Systems*, vol. 17, no. 1, pp. 94-162, 1992.
- [36] G.R. Ganger, M.K. McKusick, C.A.N. Soules, and Y.N. Patt, "Soft Updates: A Solution to the Metadata Update Problem in File Systems," *ACM Trans. Computer Systems*, vol. 18, no. 2, pp. 127-153, 2000.
- [37] M.K. McKusick and G.R. Ganger, "Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem," *Proc. USENIX Ann. Technical Conf.*, 1999.
- [38] M.I. Seltzer, G.R. Ganger, M.K. McKusick, K.A. Smith, C.A.N. Soules, and C.A. Stein, "Journaling versus Soft Updates: Asynchronous Meta-Data Protection in File Systems," *Proc. USENIX Ann. Technical Conf.*, 2000.
- [39] M.K. McKusick, W.N. Joy, S.J. Leffler, and R.S. Fabry, "Fsc—The UNIX File System Check Program," *Unix System Manager's Manual-4.3 BSD Virtual VAX-11 Version*, 1986.
- [40] C. Frost, M. Mammarella, E. Kohler, A.D.I. Reyes, S. Hovsepian, A. Matsuoka, and L. Zhang, "Generalized File System Dependencies," *Proc. ACM Symp. Operating Systems Principles*, 2007.
- [41] B. Kim, E.H. Nam, Y.J. Seong, H.J. Min, and S.L. Min, "Efficient Flash Memory Read Request Handling Based on Split Transactions," *Proc. Int'l Workshop Software Support for Portable Storage*, 2009.
- [42] G.M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," *Proc. AFIPS Spring Joint Computer Conf.*, 1967.
- [43] D.A. Patterson and J.L. Hennessy, *Computer Architecture: A Quantitative Approach*, fourth ed. Morgan Kaufmann, 2007.
- [44] Xilinx, Virtex-5 FPGA family, <http://www.xilinx.com/products/virtex5/index.htm>, 2010.
- [45] E.H. Nam, K.S. Choi, J.-y. Choi, H.J. Min, and S.L. Min, "Hardware Platforms for Flash Memory/NVRAM Software Development," *J. Computing Science and Eng.*, vol. 3, no. 3, pp. 181-194, 2009.
- [46] Xilinx, Design tools, <http://www.xilinx.com/tools/designtools.htm>, 2010.
- [47] Futuremark Corporation, PCMark05 whitepaper, <http://www.futuremark.com/>, 2010.



Eyee Hyun Nam received the BS degree in electrical engineering from Seoul National University, Korea, in 1998, where he is working toward the PhD degree. He worked at Comtec System, Seoul, where he developed network devices from 1999 to 2002. He was a senior engineer in the R & D Center at Future System, Seoul, from 2003 to 2004. His research interests include computer architecture, embedded systems, file systems, and storage systems.



Bryan Suk Joon Kim received the BS degree in electrical engineering and computer science from the University of California, Berkeley, in 2006, and the MS degree in computer science and engineering from Seoul National University, Korea, in 2009. He is working toward the PhD degree from the University of California, San Diego. He was an application engineer at n & k Technology from 2006 to 2007. His research interests include computer architecture, memory systems, and storage systems.



Hyeonsang Eom received the BS degree in computer science and statistics from Seoul National University (SNU), Korea, in 1992, and the MS and PhD degrees in computer science from the University of Maryland at College Park, in 1996 and 2003, respectively. Since 2005, he has been an assistant professor in the School of Computer Science and Engineering at SNU, where he has been a faculty member. He was a senior engineer in the Telecommunication R & D

Center at Samsung Electronics, Korea, from 2003 to 2004. His research interests include high performance storage systems, distributed systems, cloud computing, energy efficient systems, fault tolerant systems, digital rights management, and information dynamics. He is a member of the IEEE and the ACM.



Sang Lyul Min received the BS and MS degrees in computer engineering, both from Seoul National University, Seoul, in 1983 and 1985, respectively, and the PhD degree in computer science from the University of Washington, Seattle, in 1989. He is currently a professor in the School of Computer Science and Engineering, Seoul National University, Korea. He has served on a number of program committees of technical conferences and work-

shops, including the International Conference on Embedded Software (EMSOFT), the Real-Time Systems Symposium (RTSS), and the Real-Time Technology and Applications Symposium (RTAS). He was also a member of the editorial board of the *IEEE Transactions on Computers*. His research interests include embedded systems, computer architecture, real-time computing, and parallel processing. He is a member of the IEEE and the ACM.

► For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.