

Hydra: A Block-Mapped Parallel Flash Memory Solid-State Disk Architecture

Yoon Jae Seong, Eyee Hyun Nam, Jin Hyuk Yoon, Hongseok Kim, Jin-Yong Choi, Sookwan Lee, Young Hyun Bae, Jaejin Lee, *Member, IEEE*, Yookun Cho, *Member, IEEE*, and Sang Lyul Min, *Member, IEEE*

Abstract—Flash memory solid-state disks (SSDs) are replacing hard disk drives (HDDs) in mobile computing systems because of their lower power consumption, faster random access, and greater shock resistance. We describe Hydra, a high-performance flash memory SSD architecture that translates the parallelism inherent in multiple flash memory chips into improved performance, by means of both bus-level and chip-level interleaving. Hydra has a prioritized structure of memory controllers, consisting of a single high-priority foreground unit, to deal with read requests, and multiple background units, all capable of autonomous execution of sequences of high-level flash memory operations. Hydra also employs an aggressive write buffering mechanism based on block mapping to ensure that multiple flash memory chips are used effectively, and also to expedite the processing of write requests. Performance evaluation of an FPGA implementation of the Hydra SSD architecture shows that its performance is more than 80 percent better than the best of the comparable HDDs and SSDs that we considered.

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

1 INTRODUCTION

FLASH memory is increasingly being used as a storage medium in mobile devices because of its low power consumption, fast random access, and high shock resistance. Moreover, the density of flash memory chips has doubled every year for the past 10 years and this trend is expected to continue until 2012 [11]. Flash memory solid-state disks (SSDs) provide an interface identical to hard disk drives (HDDs), and are currently replacing HDDs in mobile computing systems, such as ultramobile PCs (UMPCs) and notebook PCs.

The type of flash memory used for bulk storage applications is NAND flash, which is organized into physical blocks, each of which contains a set of pages that are accessed by read and program operations [29]. HDDs allow data to be overwritten directly, but flash memory cannot perform in-place updating. 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. Moreover, there is an asymmetry in read and program speeds—the read operation is much faster than the program operation (20 μ s versus 200 μ s).

Operating systems use storage devices to provide file systems and virtual memory, and it is usually assumed that these devices have an HDD-like interface. In order for flash

memory to achieve wide acceptance as a storage device, it has to emulate the functionality of HDDs. The software layer that provides this emulation is called the flash translation layer (FTL) [10]. It hides the peculiarities of flash memory and gives the illusion of an HDD.

In this paper, we describe a flash memory SSD architecture, called Hydra, that exploits the parallelism of multiple NAND flash memory chips to enhance storage system performance. The Hydra SSD architecture uses various techniques to achieve this goal:

1. The disparity between the slow flash memory bus (<40 MB/s) and the fast host interface (>150 MB/s) is overcome by interleaving enough flash memory buses so that their collective bandwidth meets or exceeds that of the host interface. In addition to this bus-level interleaving, chip-level interleaving hides the flash read latency.
2. Multiple high-level flash memory controllers execute sequences of high-level flash memory operations without any intervention by the FTL. One of these controllers is designated as the foreground unit and has priority over the remaining controllers, called background units. The foreground unit is used to expedite the processing of host read requests for which processes in the host system are waiting.
3. Aggressive write buffering expedites the processing of host write requests. More importantly, it also allows the parallelism in multiple flash memory chips to be exploited by multiple background units that perform materialization to flash memory in parallel on different interleaved units.

The rest of this paper is organized as follows: The basics of NAND flash memory and related work on FTL and solid-state disks are reviewed in the next section. We then present

• The authors are with the School of Computer Science and Engineering, Seoul National University, Seoul 151-742, Korea.
E-mail: {yjsung, ehnam, hskim, jychoi, symin}@archi.snu.ac.kr, {jhyoon, sklee}@ssrnet.snu.ac.kr, yhbac@mttron.net, jlee@cse.snu.ac.kr, ykcho@snu.ac.kr.

Manuscript received 12 Feb. 2008; revised 11 July 2009; accepted 21 Aug. 2009; published online 2 Mar. 2010.

Recommended for acceptance by F. Lombardi.

For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2008-02-0067.

Digital Object Identifier no. 10.1109/TC.2010.63.

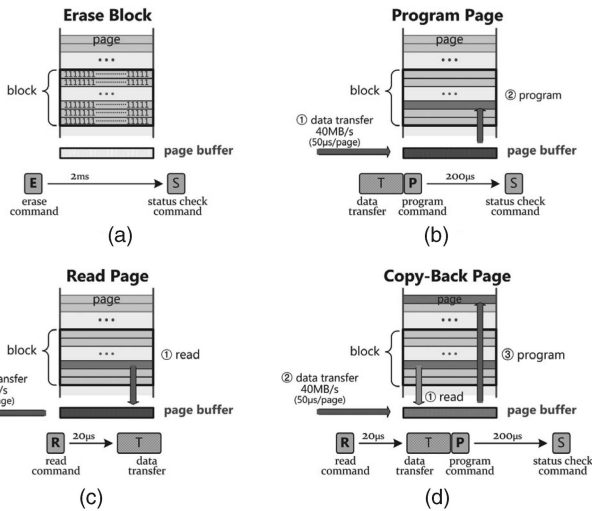


Fig. 1. Low-level NAND flash memory operations.

the Hydra SSD architecture in Section 3. A prototype implementation in FPGA and its performance evaluation results are presented in Section 4. Finally, we conclude in Section 5, and suggest some directions for future research.

2 BACKGROUND AND RELATED WORK

2.1 NAND Flash Memory

A NAND flash memory chip consists of a set of blocks, each of which consists of a set of pages. Each page has a data part that stores the user data and a spare part that stores metadata associated with the user data. 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. Currently, the most popular block size is 128 KB, consisting of 64 pages, each of 2 KB (a data part of four sectors and a spare part of 64 bytes) [29], and this configuration will be considered throughout this paper although our technique does not rely on it.

Fig. 1 shows the four major operations provided by a typical low-level flash memory controller. We will explain these operations in detail using the timings of the Samsung K9K8G08U0M 8 Gb NAND flash memory chip [29].

The erase block operation sets all the bits in a block to 1 and takes about 2 ms. As shown in Fig. 1, this operation is initiated by an erase command that includes the address of the block to be erased. After the erase is complete, a status check command is issued to detect any errors that might have occurred during the operation.

The program page operation writes the data supplied to a page that has been previously erased. It consists of three phases. In the data transfer phase, the data to be written to the target page is transferred at 40 MB/s over the flash memory bus to the internal page buffer in the NAND flash memory chip. Then, a program command is issued along with the address of the target page. Programming takes about 200 μ s. When it is complete, a status check command is issued (as in the erase block operation) to check for errors.

The read page operation reads a page from flash memory. First, a read command is issued. This loads a page into the memory's internal page buffer, which takes

about 20 μ s. Then, the data in the internal page buffer is read out at 40 MB/s over the flash memory bus.

The copy-back page operation transfers data from one page to another inside the chip while allowing a portion of the data to be modified with externally supplied data. This operation is much more efficient than moving the data out of the chip and back in again, using a read page operation followed by a program page operation. The copy-back page operation is initiated by a read command that moves the data to the internal page buffer. Then, the data corresponding to the portion of the page to be modified is transferred to the internal page buffer and a program command is issued. Finally, the usual status check is performed.

NAND flash memory is subject to bit-flipping errors, causing one or more bits in a page to be reversed. This can be accommodated up to a point by external error-correction logic. NAND flash memory can also tolerate a limited number of bad physical blocks, which increases both the yield and the lifetime of the chips. These bad blocks are identified by a special mark at a designated location in each block. Even the good blocks have a limited lifetime, which necessitates a technique called wear-leveling [10] that tries to even out the number of times that each block is erased.

2.2 Flash Translation Layer (FTL)

The flash translation layer (FTL) hides the peculiarities of flash memory and emulates the functionality of an HDD. The most important role of the FTL is to maintain a mapping between the logical sector address used by the host system and the physical flash memory address. This mapping can either be at the page level [17], [33] or at the block level [2], [16], [21], [22], [32]. In a page-level mapping, a logical page can be mapped to any page in flash memory, making it more flexible than block-level mapping, which will be explained shortly. In a page-level mapping, the physical blocks in flash memory form the same sort of log that we find in a log-structured file system [27]. As data is written into the memory, it is simply appended to the end of the log. When the amount of free space in the log drops below a given threshold, garbage collection is triggered: a physical block is selected 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 whole block is erased and added to the list of free space. The choice of block to be garbage-collected is based on a cost-benefit analysis [8], [17], [27], [33].

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 memory storage device requires a 32 MB mapping table, assuming that each entry takes 4 bytes. Second, the overheads of garbage collection increase sharply as the proportion of valid pages increases, an effect that is well known in log-structured file systems [33]. Finally, the performance of page-level mapping on sequential reads is relatively poor since logically sequential sectors are physically scattered over the whole memory.

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 memory

operations when any pages in a logical block are modified. For example, when there is a write request to a single page in a logical block, that block must be remapped to a free physical block; a program operation must then be performed on the target page in the new physical block; and all the other pages in that block have to be copied from the old physical block to the new one.

The need for expensive copy operations in block-level mapping can be reduced by various nonvolatile write buffering schemes [2], [16], [21], [22], [32]. These buffering schemes temporarily store the data from the host in physical blocks called write buffer blocks before block remapping is performed. Write buffering schemes can be classified as block-level [2] or page-level [16], [21], [22], [32] depending on where the newly written page is placed. In a block-level write buffering scheme (e.g., [2]), a newly written page can only be placed in the same page in a write buffer block. On the other hand, in a page-level write buffering scheme, a new page can be placed in any page in a write buffer block, regardless of whether the association between logical blocks and write buffer blocks is direct mapped [21], [32], set associative [16], or fully associative [22].

2.3 Solid-State Disks (SSDs)

In the early days of flash memory, Wu and Zwaenepoel proposed a mass storage system architecture called eNVy [33]. This used page-level mapping and was equipped with battery-backed SRAM for efficient write buffering. It was designed to act as a main memory, whereas more recent SSDs have been designed to replace HDDs, as their name suggests.

In hybrid HDDs [24], a small amount of flash memory is added to an otherwise standard HDD, and it is used to buffer write requests from the host so that the HDD can be spun down for extended periods of time, improving both energy efficiency and reliability. In addition, the flash memory can cache frequently accessed sectors to improve the read performance, and it can also be used to speed up boot-up and resume by pinning the sectors required during these operations.

Another hybrid approach is to combine flash memory with nonvolatile RAM (NVRAM). In the Chameleon SSD architecture [34], a small amount of NVRAM, such as ferroelectric RAM [30], is combined with the flash memory in an SSD. This allows the efficient handling of small random writes which are otherwise slow because flash memory does not allow in-place updates. In Chameleon, the bulk data is stored in flash memory and the NVRAM contains the nonvolatile data structures in the FTL, such as the mapping table, that are subject to frequent small random writes.

Kgil et al. [18], [19] proposed a disk cache architecture based on flash memory, in which the disk cache is partitioned into a read cache and a write cache. This partitioning improves the read performance of the disk cache as a whole since the sectors cached in the read disk cache are much less involved in garbage collection. Another interesting feature of this disk cache architecture is a programmable error-correction code (ECC) which is of adjustable strength so as to allow a trade-off between performance and reliability.

Kim and Ahn [20] proposed an efficient buffer management scheme called BPLRU for block mapping SSDs with three significant features: block-level LRU that takes the size

of flash memory blocks into account; page padding to convert scattered writes within a flash memory block into one sequential write; and LRU compensation to optimize the handling of sequential writes.

Prabhakaran et al. [26] proposed an extended, transactional interface to SSDs with two alternative commit protocols, simple cyclic commit and back pointer cyclic commit, both of which are optimized to the unique characteristics of flash memory such as the lack of in-place updates and the spare part in each page.

An SSD architecture based on page-level mapping was suggested by Birrel et al. [4], who described both volatile and nonvolatile data structures, and the reconstruction of the former from the latter during power-on. This reconstruction process is relatively fast because the last page in each block is reserved for summary information about the other pages in that block; this obviates the need to scan all the pages in the flash memory during power-on.

Agrawal et al. [1] discussed the many design compromises required in building an SSD based on a page-mapping FTL and evaluated those trade-offs using trace-driven simulations. The trade-offs discussed include the choice of mapping granularity, interleaved or parallel flash memory operations, free block managements, and wear-leveling. They also analyzed the effects of design parameters such as cleaning thresholds and overprovisioning in a page-level mapping FTL.

This paper presents Hydra, a new SSD architecture based on block-level mapping. It extensively exploits the parallelism inherent in multiple flash memory chips using various design techniques such as interleaving, prioritized handling, and volatile write buffering. They are well-known techniques in many areas in computing, but Hydra adopts them in the context of block-mapped flash memory SSD. The evaluation based on a prototype implementation provides comprehensive analysis on the effect of those techniques.

3 HYDRA SOLID-STATE DISK ARCHITECTURE

The Hydra SSD is based on block-level mapping and its overall architecture is shown in Fig. 2. The SSD is connected to a host system (e.g., a desktop PC or notebook PC) through a device-side storage system protocol such as serial ATA (S-ATA) [13] or serial attached SCSI (SAS) [15]. The embedded processor together with the SRAM and code storage provides the execution environment for the FTL.

The host system requires not only a high bandwidth that meets or exceeds the maximum speed of its interface, but also fast response times for both reads and writes. On the other hand, the NAND flash memory is characterized by a slow (<40 MB/s) bus. Hydra reconciles this mismatch by the use of multiple flash memory buses whose collective bandwidth meets or exceeds the maximum bandwidth of the host interface. It also uses chip-level interleaving to hide the flash read latency. Both bus-level and chip-level interleaving are implemented by the MUX/DEMUX unit shown in Fig. 2.

In Hydra, the set of flash memory chips that are related to each other by the bus-level and chip-level interleaving is called a superchip. Fig. 3 shows an example in which the degrees of bus-level and chip-level interleaving are 4 and 2,

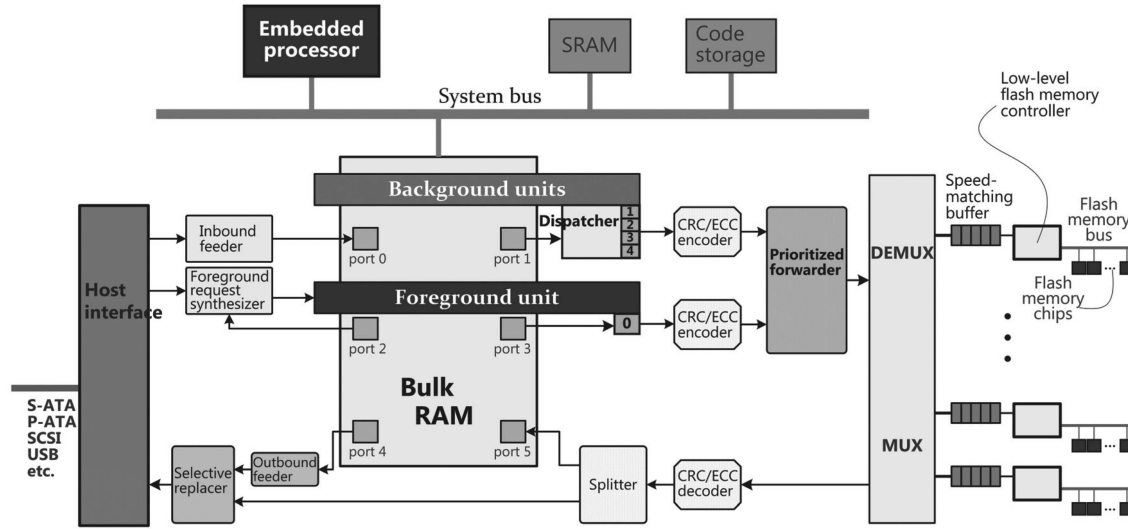


Fig. 2. Overall architecture of the Hydra SSD.

respectively. Like a (physical) block in a flash memory chip, we can define a (physical) superblock in a superchip as a set of physical blocks, one from each chip in the superchip. (This is not to be confused with the term “superblock” used to refer to the metadata describing a file system.) Similarly, a superpage i of a superblock is formed from the i th pages of all the constituent physical blocks in the superblock. As the result of these definitions, the size of a superblock is the same as that of physical block (128 KB), multiplied by the combined bus-level and chip-level interleaving, and the size of a superpage is determined in the same way. For example, in the configuration shown in Fig. 3, the sizes of the superblock and superpage are 1,024 KB and 16 KB, respectively.

To meet the maximum bandwidth requirement of host write requests, Hydra employs volatile write buffering using bulk RAM, which is typically implemented as high-bandwidth DRAM. This write buffering reduces the response time of host write requests, since a write request received from the host by the inbound feeder in Fig. 2 can be acknowledged as soon as all the data from the host has been buffered to the bulk RAM. The data is later materialized to flash memory by the multiple background units shown in

Fig. 2. Each of these units is capable of performing high-level flash memory operations on superchips without any intervention by the FTL.

To achieve the fast response time required for host read requests, Hydra uses the foreground unit shown in Fig. 2. This has a higher priority than the background units to avoid the read requests being delayed by non-time-critical materialization tasks performed by the background units. This prioritized access to the bus is implemented by the prioritized forwarder unit, also shown in Fig. 2.

To further reduce the response time of host read requests, Hydra uses a foreground request synthesizer unit. This contains hardware which automatically generates requests to the foreground unit in response to a read request by the host, using a mapping stored in a table in the bulk RAM.

Some of the sectors requested by a host read request may already be write-buffered in the bulk RAM. Therefore, Hydra uses the outbound feeder unit to read the write-buffered data from the bulk RAM and the selective replacer unit to make the required replacements to the sectors read from flash memory.

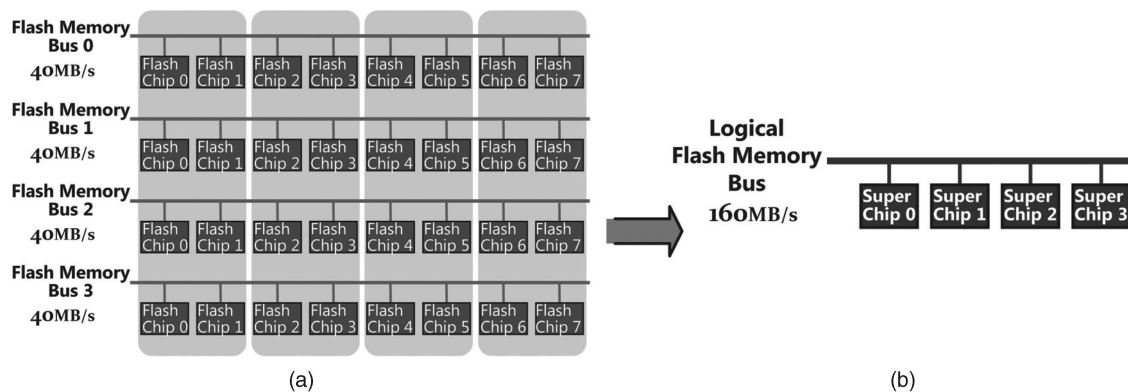


Fig. 3. Example of bus-level and chip-level interleaving. (a) Block size = 128 KB, page size = 2 KB. (b) Superblock size = 1,024 KB, superpage size = 16 KB.

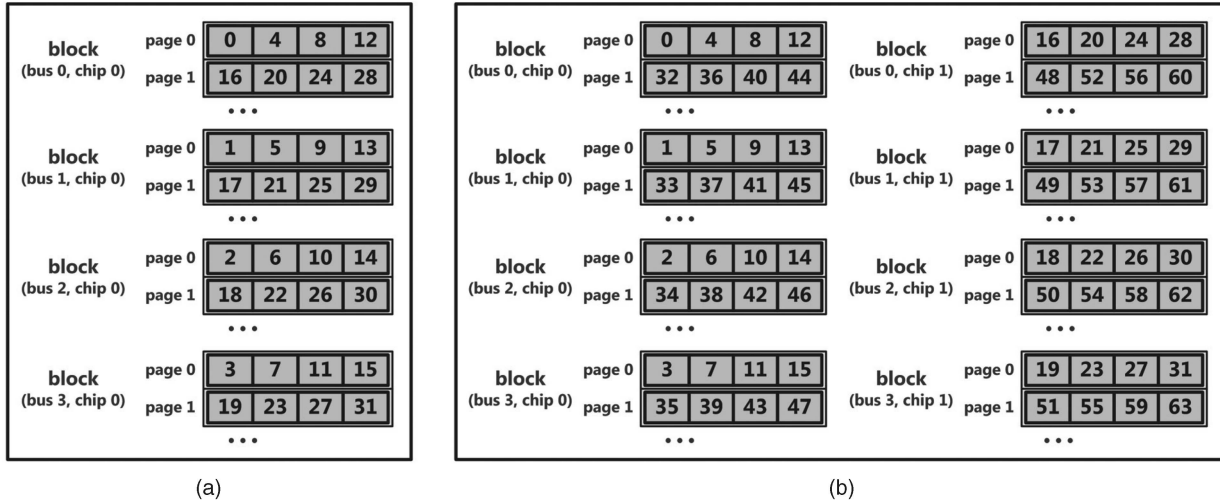


Fig. 4. Distribution of sectors within a superblock. (a) Without chip-level interleaving. (b) With two-way chip-level interleaving.

The CRC/ECC encoder and decoder units in Fig. 2 are used to detect and correct possible bit-flipping errors in flash memory.

The final component of note is the splitter unit, which is used to forward data from flash memory either to the host interface or to the bulk RAM.

Prefetching and read caching are not used in Hydra because a preliminary study showed that the read speed provided by flash memory is sufficiently high so that the overhead of the software needed to implement these two techniques would outweigh their benefits.

In the following sections, we detail the key features of the Hydra SSD architecture, which are the bus-level and chip-level interleaving mechanism, the write buffering technique, the foreground and background units with their interactions, and the wear-leveling technique.

3.1 Bus-Level and Chip-Level Interleaving

The Hydra SSD uses interleaving over multiple flash memory buses to overcome the bandwidth limitation of the flash memory bus. In the bus-level interleaving, sectors within a superblock are distributed in a round-robin manner. This is illustrated in Fig. 4a, without any chip-level interleaving.

In the context of this organization, Fig. 5a shows the timing of sequential read accesses from a superblock. In this figure, the dependencies between flash memory operations are indicated by arrows. For example, the arrow that starts at the first flash read command addressed to chip 0 on bus 0, and goes to the first data transfer from the same chip, indicates that this data transfer is only possible after the flash read latency of 20 μ s.

In this example, flash read commands are initially issued to all the chips belonging to the same superchip (i.e., chip 0 on all buses). After the flash read latency, data transfers are made over the flash memory buses. With the sector distribution shown in Fig. 4a, sectors are fetched from the flash memory buses in a round-robin manner. For example, sectors 0, 4, 8, and 12 are fetched from flash memory bus 0, sectors 1, 5, 9, and 13 from flash memory bus 1, and so on. The speed-matching buffer between the MUX/DEMUX unit

and each flash memory bus allows concurrent data transfers from the flash memory bus while the MUX/DEMUX unit is fetching data from the speed-matching buffer associated with another flash memory bus. This bus-level interleaving achieves an effective bandwidth of 160 MB/s (16 sectors in 50 μ s) within a superpage by the use of four 40 MB/s flash memory buses, as shown in Fig. 5a. However, there is an unavoidable time interval between accesses to different superpages, during which the flash memory buses are idle (indicated by a black box in the figure) because the flash read latency is not fully hidden by the data transfer time.

This idle time can be eliminated if we introduce chip-level interleaving, as shown in Fig. 5b; this assumes that the distribution of sectors within a superblock follows Fig. 4b. In this example, a chip-level interleaving of degree two is sufficient to hide the flash read latency. But if this latency is longer, a higher degree of chip-level interleaving will be required to hide it.

With the superblock organization, the logical sector address from the host is divided into a logical superblock address and a sector address within that superblock, as shown in Fig. 6. The logical superblock is mapped to a physical superblock consisting of a set of physical blocks, one from each chip in a superchip. The required mapping is provided by the block mapping table, which is stored in both the bulk RAM and the flash memory. One restriction in the mapping between logical and physical superblocks is that a given logical superblock can only be remapped to a physical superblock in the same superchip. This allows Hydra to utilize the copy-back page operation which is allowed only between pages in the same flash memory chip, as we explained earlier. To enforce this restriction, we use the lower i bits of a logical superblock address to index the superchip, where $i = \log_2(n)$, and n is the number of superchips. This means that a logical superblock is always remapped to a physical superblock in the same superchip. Fig. 6 shows a decomposition of the sector address within a superblock for arbitrary bus-level and chip-level interleaving, under the assumption that sectors within a superblock are distributed across the constituent physical blocks in the manner shown in Fig. 4.

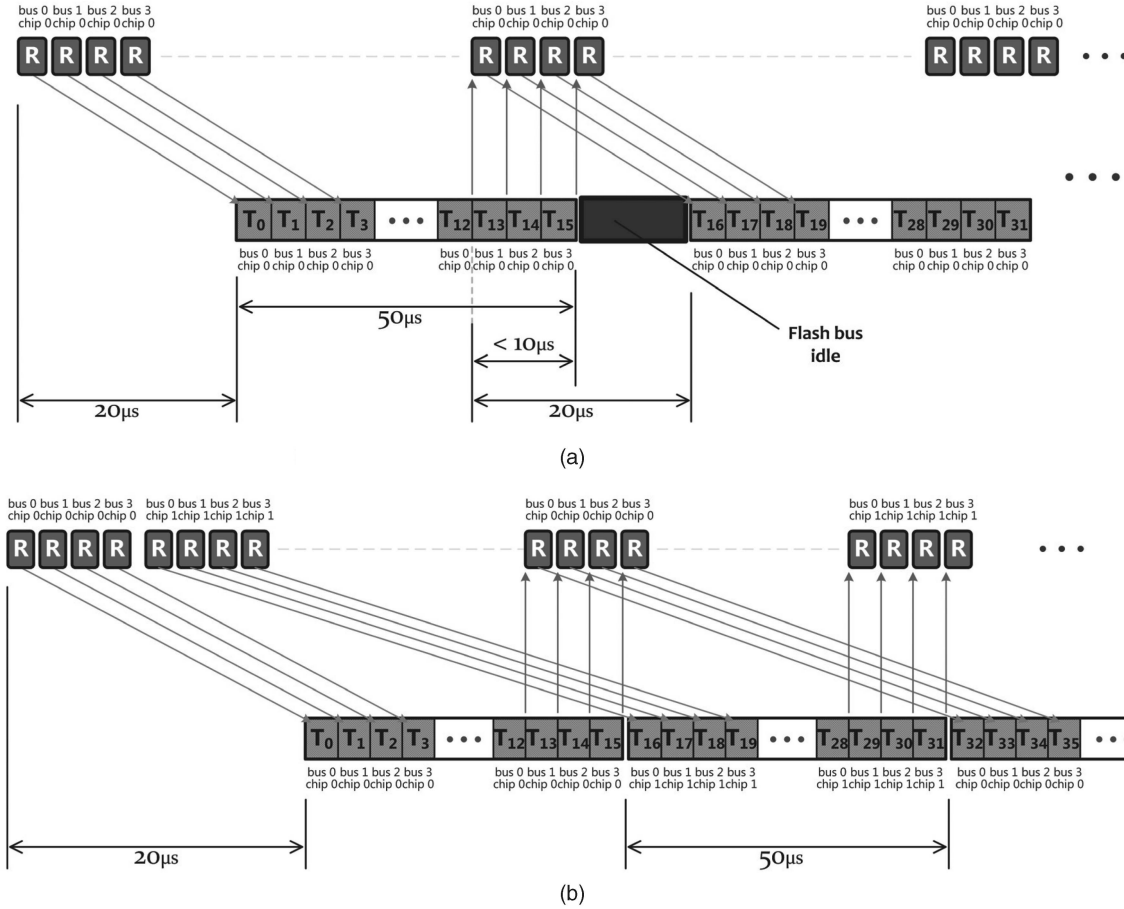


Fig. 5. Timing of sequential read access with and without chip-level interleaving. (a) Sequential read access without chip-level interleaving. (b) Sequential read access with two-way chip-level interleaving.

3.2 Write Buffering

Hydra uses volatile write buffering to decouple the materialization to flash memory from the processing of host write requests. This requires a large portion of the bulk RAM to be reserved as a sector buffer. The sector buffer operates as a circular buffer, to which the sectors written by the host are simply appended by the inbound feeder using one of the ports provided by a multiport memory controller.

The information about the sectors in the sector buffer is stored in the bulk RAM and is maintained for each logical superblock by the FTL. This information is used when the

sectors are later materialized to flash memory. This materialization is performed by the background units, and it will be explained in detail in the next section. The background materialization process is invoked in three circumstances: when the free space in the sector buffer is below a given threshold (called the flush high-watermark); when the host sends a flush-cache request that requires sectors write-buffered in volatile storage to be materialized in nonvolatile storage; or when the number of superblocks in the sector buffer rises above a given threshold.

3.3 Multiple High-Level Flash Memory Controllers

The Hydra SSD architecture uses multiple high-level flash memory controllers, consisting of one foreground unit and several background units. Each high-level controller is capable of executing a sequence of high-level flash memory operations, specified as a linked list of operation descriptors. In Hydra, a high-level flash memory operation is directed to a superchip, and for this reason, we refer to it as a superchip operation. A linked list containing a sequence of superchip operations is prepared by the FTL and the address of the first descriptor is given to the high-level flash memory controller through a command register.

A superchip operation operates on a physical superblock and can span multiple superpages. For each superchip operation, a high-level flash memory controller generates a sequence of low-level flash memory operations, such as those explained in Section 2.1, to perform the requested task

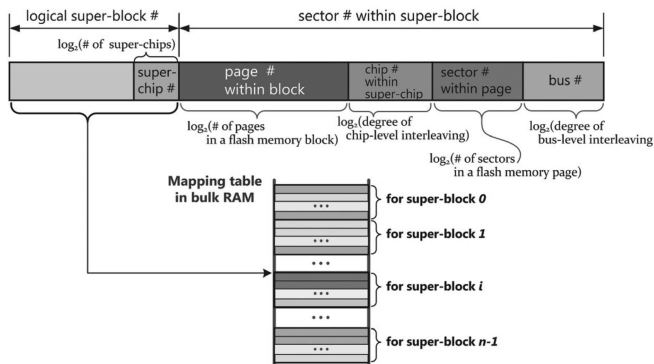


Fig. 6. Mapping from a logical sector address to a physical sector address.

without any intervention by the FTL. We will now explain the main superchip operations that can be performed by a high-level flash memory controller.

3.3.1 High-Level Flash Memory Operations

Erase superchip. This operation erases all the physical blocks in a physical superblock. It accepts the superchip number and the physical block addresses of the physical superblock to be erased. It then uses a sequence of low-level erase block operations, as explained in Section 2.1, to perform the erasure. If there is an error during one of the low-level erase block operations, an interrupt is generated in the FTL, which is also supplied with the identity of the physical block which caused the error. This bad block is mapped out and replaced by one of the spare physical blocks which were reserved when the SSD was initialized. After this replacement, the FTL retries the erase superchip operation.

Program superchip. This operation programs an arbitrary number of sectors in a physical superblock. This superchip operation takes five arguments: the superchip number, the physical block addresses of the physical superblock, the number of the start sector within the physical superblock, the number of sectors to be programmed, and the start sector buffer index from which the sectors to be programmed are sequentially read. An error during a program superchip operation is handled in a similar manner to an error during an erase superchip operation, except that valid sectors in the remapped superblock are copied to the new superblock before the program superchip operation is retried.

Read superchip. This operation reads an arbitrary number of sectors from a physical superblock. Like the program superchip operation, it requires five arguments: the superchip number, the physical block addresses of the physical superblock, the number of the start sector within the physical superblock, the number of sectors to be read, and the start sector buffer index to which the requested sectors are sequentially stored. The data returned from flash memory can either be stored in the sector buffer or forwarded to the host interface. For this purpose, a special sector buffer index is reserved to designate the host interface, and the splitter unit in Fig. 2 handles the routing required.

Copy-back superchip. This is a merge operation between the physical superblock currently mapped to a logical superblock and the set of sectors that are write-buffered in the sector buffer and belong to the logical superblock. The operation has four arguments: the superchip number of the source and destination physical superblocks, the physical block addresses of the source physical superblock (i.e., the physical superblock mapped to the logical superblock before the merge), the address of the data structure that contains information about the sectors write-buffered in the sector buffer for the logical superblock, the physical block addresses of the destination physical superblock (i.e., the physical superblock mapped to the logical superblock after the merge). Note that the superchip numbers for both the source and destination physical superblocks are the same, since a given logical superblock is always mapped to the same superchip.

After a merge operation for a logical superblock, using a copy-back superchip operation, the block mapping table in the bulk RAM is updated for that logical superblock and a

log is created in a reserved area of flash memory. Periodically, the block mapping table in the bulk RAM is flushed to another reserved area of flash memory. During a power-on recovery, the block mapping table in flash memory is loaded into the bulk RAM and the logs are replayed to reconstruct the up-to-date block mapping table.

3.3.2 Multiple Background Units

In the Hydra SSD architecture, there are multiple background units, each of which is a high-level flash memory controller. This allows more than one superchip operation involving different logical superblocks to be performed in parallel. However, if the flash memory buses are reserved by a background unit during the whole period of a low-level flash memory operation, their utilization will be severely impaired, as Fig. 7a illustrates. In this example, since the flash memory buses are reserved for a low-level erase operation issued by background unit 1, those from background units 2 through 4 cannot be performed, even though they are directed to different superchips. Note that each **E**, **P**, **R**, and **S** command in the figure denotes multiple low-level flash memory commands of the same type directed to the constituent chips in the superchip. Similarly, **T** indicates parallel data transfer to/from the chips in the superchip over multiple flash memory buses.

To rectify the problem resulting from the coarse-grained interleaving explained above, operations from background units are more finely interleaved. If a background unit issues a long-latency command such as an erase, program, or read command, as part of a low-level flash operation, that unit is suspended and another eligible background unit is resumed. Afterwards, when the long-latency command has been completed does the associated background unit again become eligible for scheduling. The dispatcher unit shown in Fig. 2 is responsible for this scheduling between multiple background units. Fig. 7b shows how this fine-grained interleaving corrects the problem that occurred in the example of Fig. 7a. Background unit 1 is now suspended after issuing an erase command, allowing other background units to issue commands directed to other superchips. After the erase command is complete, background unit 1 resumes and it issues the required status check command.

3.4 Prioritized Handling of Foreground and Background Requests

To reduce the response time of host read requests, one of the high-level flash memory controllers is designated as the foreground unit and is dedicated to servicing read requests. The prioritized forwarder unit in Fig. 2 gives this foreground unit priority over the background units. When a new foreground request arrives, the prioritized forwarder unit preempts the background processing in progress, at the earliest possible time at which this can be done without violating the correctness.

More specifically, when the foreground request targets a superchip which is not being accessed by any of the background units, the preemption occurs at the end of the current primitive operation, as shown in Fig. 8a. In the example given in this figure, the read operation by the foreground unit is accessing super-chip 5, which is not being used by any of the background units. This allows preemption almost immediately after the current primitive operation is complete.

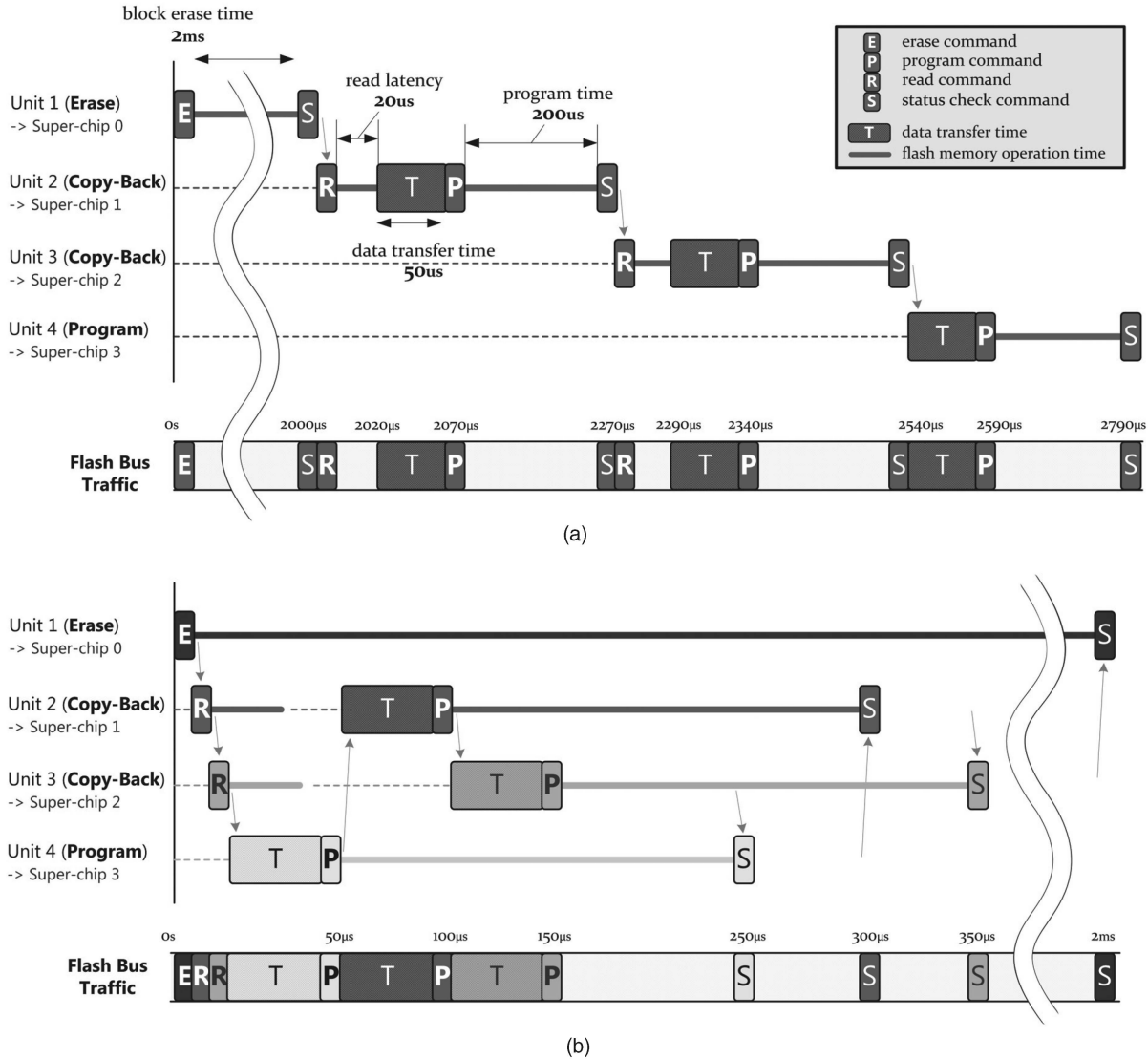


Fig. 7. Two alternative interleaving techniques between multiple background units. (a) Coarse-grained interleaving between multiple background units. (b) Fine-grained interleaving between multiple background units.

However, if the superchip which is the target of the foreground request is currently being used by one of the background units, preemption is delayed until the end of the current low-level flash memory operation, as shown in Fig. 8b. In this example, the foreground unit waits until the current low-level flash operation from background unit 4 is complete, since both units need to access the same superchip (superchip 3). This delayed preemption is necessary since an operation in a NAND flash memory chip cannot be suspended by another operation addressed to the same chip.

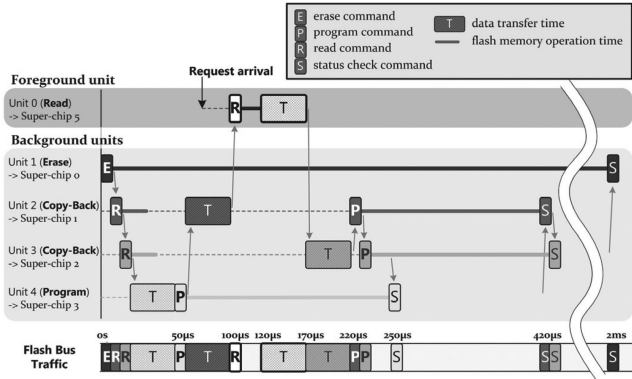
3.5 Wear-Leveling

The use of block-level mapping in Hydra greatly simplifies wear-leveling since block-level mapping, unlike page-level mapping, does not suffer from the complications that arise if wear-leveling is coupled to garbage collection [8], [10], [17]. Instead, Hydra uses two simple techniques borrowed from wear-leveling in page-mapping FTLs: one is implicit and the other explicit [3], [10]. In implicit wear-leveling, when a merge operation is performed, the free physical superblock with the smallest erase count is used as the

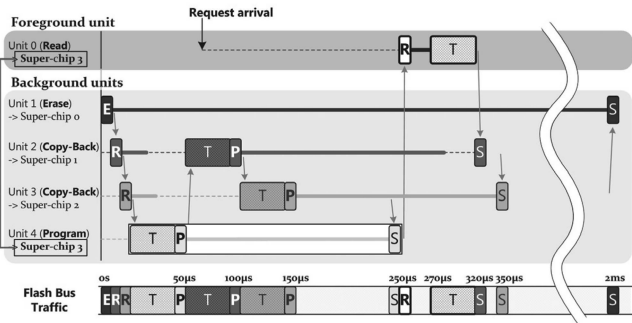
destination of the copy-back superchip operation. In explicit wear-leveling, when the SSD is idle, the physical superblock with the smallest erase count (among those mapped to logical superblocks) is swapped with the free physical superblock with the largest erase count, provided that the difference between the two counts is above a certain threshold. To facilitate both implicit and explicit wear-leveling, Hydra keeps erase counts of all the physical superblocks in the bulk RAM, and flushes this information to flash memory when the block mapping table is flushed.

4 PROTOTYPE IMPLEMENTATION AND PERFORMANCE EVALUATION

We constructed a prototype implementation of the Hydra SSD architecture and evaluated its performance. After describing the experimental setup, we will present the results that we obtained with the PCMark05 [9] and TPC-C [31] benchmarks in Sections 4.2 and 4.3, respectively. Finally, we will discuss issues of energy consumption and wear-leveling in Section 4.4.



(a)



(b)

Fig. 8. Prioritized handling of foreground and background requests. (a) When there is no superchip conflict. (b) When there is a superchip conflict.

4.1 Experimental Setup

We implemented a prototype of the Hydra SSD architecture using the in-house development board shown in Fig. 9, which has a Xilinx Virtex 4 FPGA (XC4VFX100-10FF1517) with two embedded PowerPC405 processors, of which we use only one. Most of the functionality of Hydra is implemented in the FPGA, except for bulk RAM and NAND flash memory.

The bulk RAM consists of a 64 MB SDRAM, of which only 16 MB is used by the prototype implementation. The NAND flash memory consists of four NAND flash memory modules, each of which fits into a socket in the development board and connects to a separate flash memory bus. The socket can accept various NAND flash memory modules, as shown in Fig. 9. The module that we used contains eight 1 GB NAND flash memory chips, giving a total capacity of 32 GB (1 GB/chip \times 8 chips/module \times 4 modules).

The host interface of the development board has both serial ATA (S-ATA) [13] and parallel ATA (P-ATA) [12]

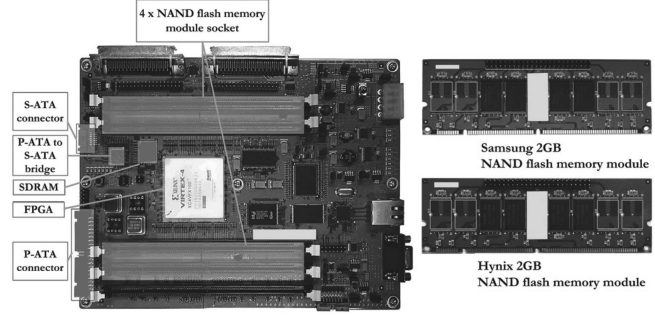


Fig. 9. Hydra SSD prototype.

connectors. A device-side P-ATA controller is implemented in the FPGA, and the S-ATA interface is achieved by a P-ATA to S-ATA bridge chip. The board also has an extension slot, a DDR SDRAM, an Ethernet interface, a synchronous SRAM, and a NOR flash memory to support future developments. More details about the development board can be found in [23].

In the default configuration, the embedded PowerPC405 processor in the FPGA operates at 200 MHz, and the system bus runs at 100 MHz. The code, data, and stack sections required by the FTL are provided by the embedded RAM in the FPGA. All the Hydra-specific units and data paths in the prototype are 32 bits wide and operate at 80 MHz, except for the flash memory subsystem, in which the data-path is 8 bits wide and operates at 40 MHz, as required to interface to the NAND flash memory buses. The default interleaving configuration is 4-way bus-level interleaving and 2-way chip-level interleaving, and sectors are distributed within a superblock, as shown in Fig. 4b. The code and data size of the FTL implemented using embedded RAM in the FPGA is 62.3 KB and 11.9 KB, respectively. Table 1 summarizes the utilization of the FPGA by the Hydra SSD prototype in this default configuration, which shows that Hydra can be comfortably implemented in a single ASIC chip.

Out of the 16 MB of bulk RAM accessible in the prototype, 1 MB is used for storing various Hydra metadata including 256 KB for the block mapping table and the remaining memory (i.e., 15 MB) for the sector buffer. In the prototype implementation, background materialization is invoked if the free space in the sector buffer drops below 30 percent or there are more than eight superblocks in the sector buffer. In both cases, background materialization continues until only the superblock most recently written by the host remains in the sector buffer. But if a flush-cache request is received from the host, all the data in the sector buffer is materialized to flash memory.

TABLE 1
Hydra SSD Prototype FPGA Utilization

Name	Used	Available	Utilization
Flip-Flops	24,581	84,352	29%
Occupied Slices	25,159	42,176	59%
Total 4-input LUTs	36,299	84,352	43%
Bonded IOBs	282	768	36%
Block RAMs	139	376	36%

TABLE 2
Specification of HDDs and SSDs

	Seagate 2.5-inch HDD	Seagate 3.5-inch HDD	Adtron 3.5-inch SSD	M-Systems 2.5-inch SSD	Samsung 2.5-inch SSD	Hydra SSD
Manufacturer	Seagate	Seagate	Adtron	M-Systems	Samsung	—
Model number	ST9120821A	ST3500630A	I35FB-16GC10	FFD-25-UATA-16384-B	MCAQE32G5-APP-0XA	—
Capacity	120GB	500GB	16GB	16GB	32GB	32GB
Interface	P-ATA	P-ATA	P-ATA	P-ATA	P-ATA	P-ATA
DRAM buffer	8MB	16MB	Unknown	Unknown	Unknown	16MB
Spindle speed	5400RPM	7200RPM	N/A	N/A	N/A	N/A

For comparison purposes, we also evaluated the performance of two HDDs and three SSDs, whose specifications are given in Table 2. All of them use the P-ATA host interface, allowing a fair comparison. The host system is a desktop PC with a 2.8 GHz Intel Pentium 4 processor and 1 GB of main memory, running Windows XP Professional (Version 5.1.2600). The main-board chipset is the Intel i965P and the host-side ATA controller is the Intel ICH7R.

4.2 Performance Evaluation (PCMark05)

This section reports results from our performance evaluation using the PCMark05 HDD benchmark program (build 1.2.0) [9] that emulates the workload of a typical PC environment. PCMark05 has five components: *XP Startup*, *Application Loading*, *General Usage*, *Virus Scan*, and *File Write*. Each replays actual disk accesses recorded in the corresponding context and contains not only regular read and write requests but also flush-cache requests that require sectors previously buffered to volatile storage to be materialized to nonvolatile storage. *XP Startup* replays read and write requests made by the host during a Windows XP boot-up. About 90 percent of its requests are for reading and 10 percent for writing. *Application Loading* contains the host requests made during the launch

and termination of application programs such as Microsoft Word, Adobe Acrobat Reader, and the Mozilla Internet Browser. It consists of 83 percent reads and 17 percent writes. *General Usage* contains host requests made during the execution of 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 made while scanning 600 MB of files for viruses. As might be expected, its requests are 99.5 percent reads. Finally, *File Write* contains host requests for writing 680 MB of files, and does not include any read requests.

The result of running each component benchmark is expressed as a transfer rate in megabytes per second and the PCMark05 benchmark reports an HDD score [9] which is the geometric mean of the five-component benchmark results multiplied by 300.

4.2.1 Comparison of Hydra with HDDs and Other SSDs

Fig. 10 shows PCMark05 HDD benchmark results for the Hydra SSD prototype and the other five storage systems in Table 2. The resulting scores show that Hydra performs 80 percent better (6080 versus 11045) than the best of the other disks, which was the Samsung 2.5-inch SSD.

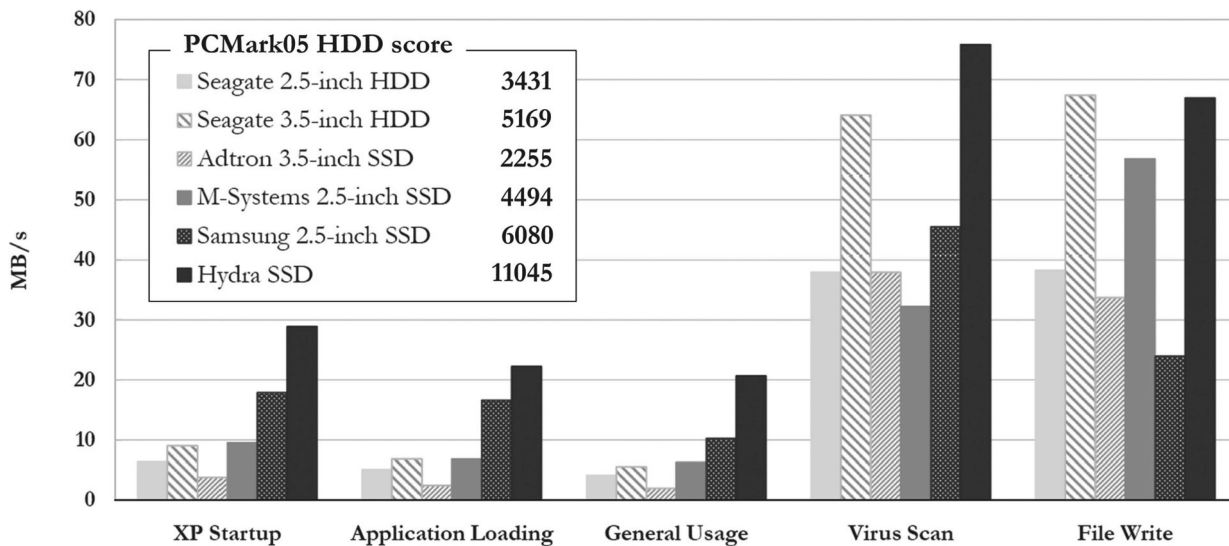


Fig. 10. PCMark05 HDD benchmark results.

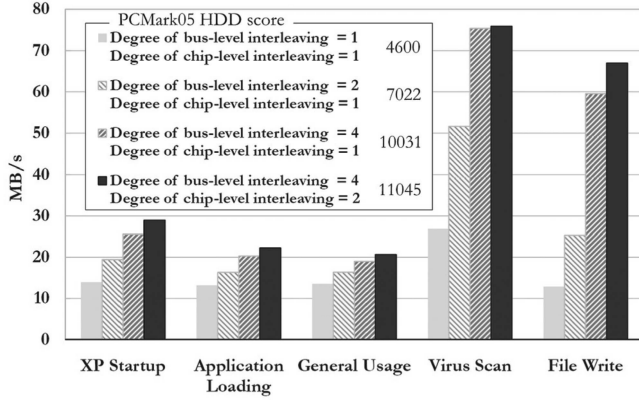


Fig. 11. Effects of bus-level and chip-level interleaving.

In general, the results suggest that SSDs perform better than HDDs for nonsequential workloads, such as *XP Startup*, *Application Loading*, and *General Usage*, because there is no seek time or rotational latency. However, for sequential workloads (*Virus Scan* and *File Write*), all the storage systems perform better, as we would expect, and there is little difference between the performance of the SSDs and HDDs.

4.2.2 Sensitivity Analysis

Effects of bus-level and chip-level interleaving. Fig. 11 shows the PCMark05 scores for the Hydra prototype for various degrees of bus-level and chip-level interleaving. These results show a substantial improvement in performance on all the component benchmarks as the degree of either type of interleaving is increased. This can be attributed to faster servicing of both read requests from the host and also background materialization to flash memory.

One interesting result is that *Virus Scan* hardly benefits from increasing the degree of chip-level interleaving when the degree of bus-level interleaving is four. It appears that the desktop PC used as the host system becomes the bottleneck at about 75 MB/s when the P-ATA interface is used. To investigate this further, we slowed down the flash memory bus clock from 40 to 30 MHz, while leaving the 4-way bus-level and 2-way chip-level interleaving unchanged. The result of *Virus Scan* stayed at about 75 MB/s, reinforcing our diagnosis of a bottleneck in the host system.

Effects of write buffering. Fig. 12a gives the PCMark05 scores for various sizes of the sector buffer used for write buffering. We can see that performance is severely limited when there is no write buffering for all the component benchmarks except for *Virus Scan*, which consists mostly of read accesses. The results also show that increasing the sector buffer size beyond 2 MB does not improve the performance very much, except in the case of *File Write*. That benchmark mainly consists of bulk writes, and so the performance improves up to the maximum sector buffer size of 15 MB.

We then fixed the buffer size at 15 MB, and looked at the effect of changing the flush high-watermark (Fig. 12b). On all the benchmarks except *File Write*, the result is similar to that of changing the sector buffer size, which is plausible because reducing the flush high-watermark has the indirect

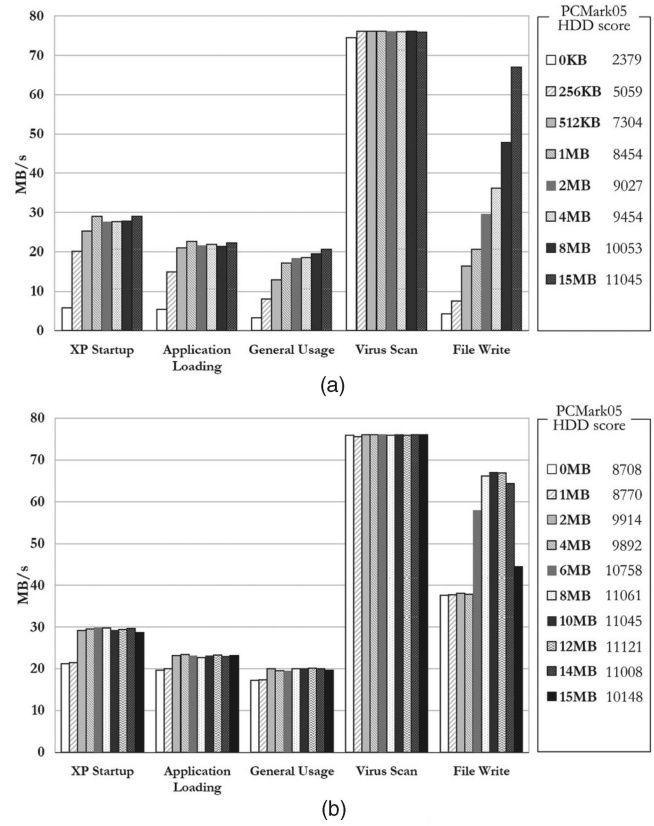


Fig. 12. Effects of write buffering. (a) Effects of write buffer size. (b) Effects of flush high-watermark.

effect of limiting the size of the sector buffer available for write buffering. For *File Write*, however, the performance improves until the flush high-watermark reaches 10 MB, and then deteriorates. It would seem that this anomalous behavior occurs because materialization is delayed as the high-watermark increases, since the materialization is only triggered when the high-watermark is reached.

Effects of multiple background units. Fig. 13 shows that the performance of Hydra improves gradually as more background units are used for writing to flash. The exception is *Virus Scan*, which consists mostly of reads. The performance improvement is most noticeable for *File Write*, which

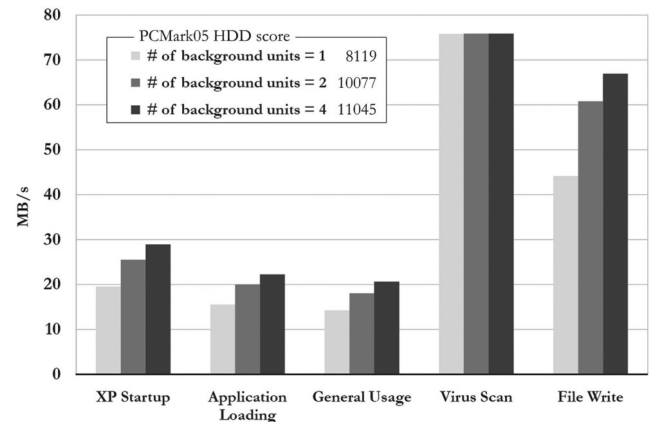


Fig. 13. Effects of multiple background units.

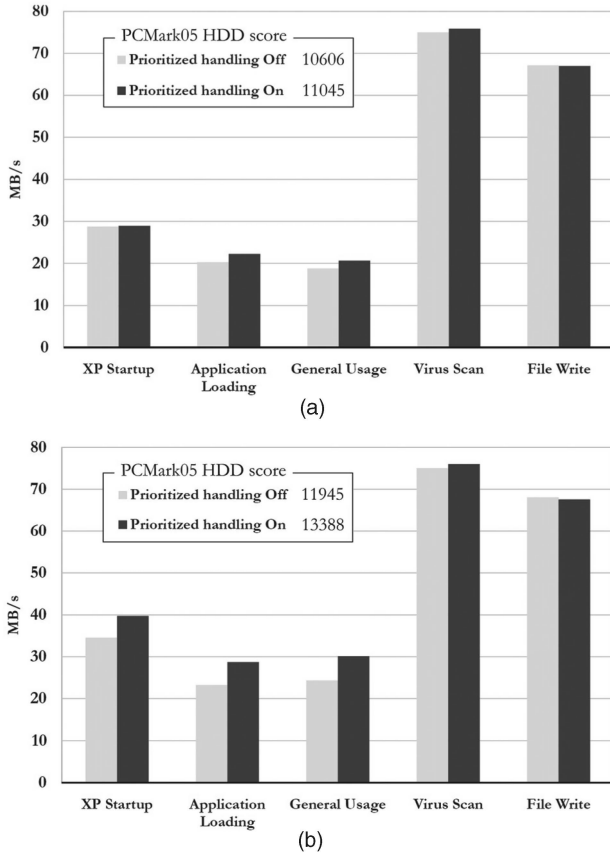


Fig. 14. Effect of prioritized handling. (a) Default setting. (b) No flush-cache handling.

consists entirely of writes, providing many opportunities for parallel materialization by multiple background units.

Effect of prioritized handling of foreground and background requests. To evaluate the effect of prioritizing foreground requests over background ones, we disabled this feature. The results are shown in Fig. 14a. As we would expect, *Virus Scan* and *File Write* are largely unaffected, as they are either mostly reads or mostly writes. Much more surprising is the marginal effect on the other three benchmarks. These rather unexpected results seem to be due to flush-cache requests from the host requiring sectors write-buffered in the sector buffer to be flushed to flash memory to maintain file system consistency. This decreases the probability of reads conflicting with the materialization to flash memory of previously written data.

To investigate further, we performed an experiment in which we ignored all the flush-cache requests from the host. The results in Fig. 14b show that in this new setting there is a significant performance improvement of more than 15 percent for *XP Startup*, *Application Loading*, and *General Usage*.

We then performed a further experiment with synthetic workloads using the IOMeter tool [14]. We generated a series of 4 KB read requests intermixed with a series of 4 KB write requests. We varied the proportion of read and write requests and measured the average response time for the read requests. The results are shown in Fig. 15. Without prioritized handling, the average response time for read requests

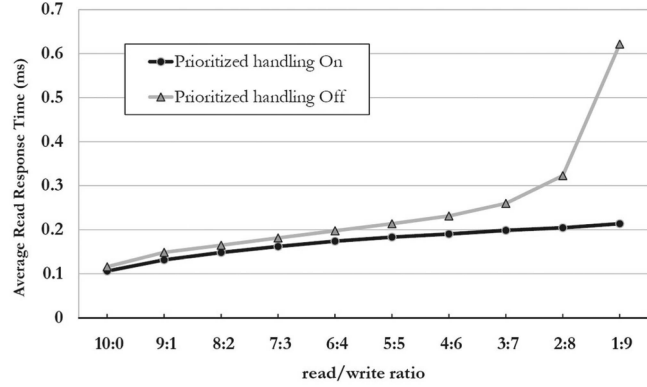


Fig. 15. Effects of prioritized handling (IOMeter).

increases more rapidly as we inject more write requests. As a result, when the read/write ratio reaches 1:9, the average response time without prioritized handling is almost triple the average response time with prioritized handling.

Effects of optimized handling of flush-cache requests.

The results above show that flush-cache requests from the host have a significant effect on performance. In the default setting, when it receives a flush-cache request, the Hydra SSD materializes all the superblocks that are write-buffered in the sector buffer before signaling completion. This sometimes creates a large delay when there are a large number of superblocks in the sector buffer. One way of reducing this delay is to flush temporarily the write-buffered data to a known location in flash memory without performing merge operations. The results in Fig. 16 show that this optimization yields a significant performance improvement of around 20 percent for *XP Startup*, *Application Loading*, and *General Usage*, in which most of the write requests are small. But this change has little effect on *Virus Scan*, in which most requests are reads, or on *File Write*, in which whole superblocks are written in most cases.

Effects of processor speed. In the default setting, multiple background units are implemented in hardware and execute a sequence of high-level flash memory operations without any intervention by the FTL. This hardware automation improves overall performance by

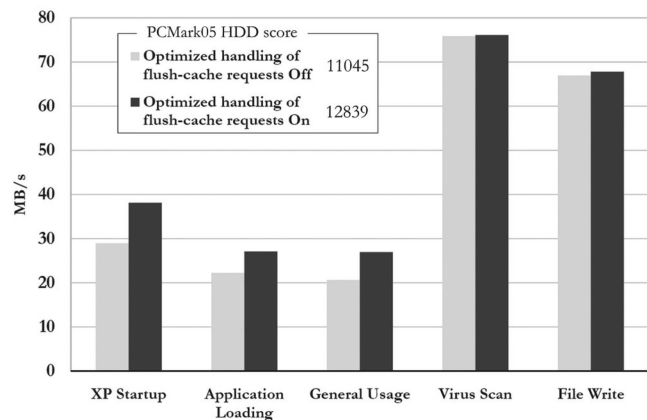


Fig. 16. Effects of optimized handling of flush-cache requests.

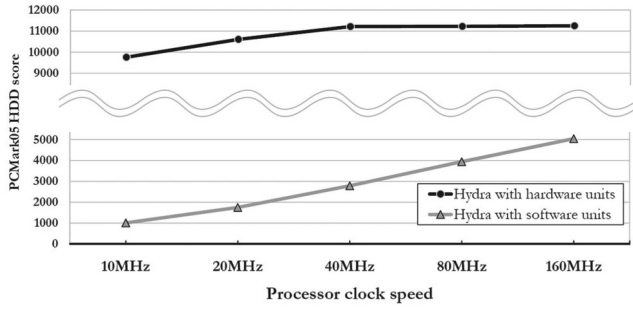


Fig. 17. Effects of processor speed.

reducing the overhead of management work in the FTL. To assess the effectiveness of this, we also implemented a software version of the background units and compared the performance of the two versions, while varying the speed of the processor from 10 to 160 MHz. The results, shown in Fig. 17, demonstrate that the performance with the hardware background units is far better than that with the software units, and unaffected by the speed of the processor beyond 40 MHz. On the other hand, the performance with the software units continuously increases with the speed of the processor, and the processor still remains as a bottleneck even at 160 MHz.

Effect of the foreground request synthesizer. The foreground request synthesizer tries to reduce the response time of host read requests by generating the corresponding foreground requests entirely in hardware. To assess the effectiveness of this, we tried turning this feature off, and the results are given in Fig. 18. To our disappointment, the hardware foreground request synthesizer seemed largely ineffectual. The expense of this part of the hardware cannot be justified if the same functionality runs just as efficiently when implemented in software.

Effect of mapping table replication. The block mapping table is the central data structure in Hydra, and is critical to the integrity of the system. One way to improve the reliability of Hydra would be to replicate this table in flash memory for added protection. The results in Fig. 19 show that this added protection can be obtained without seriously affecting the performance (<5 percent).

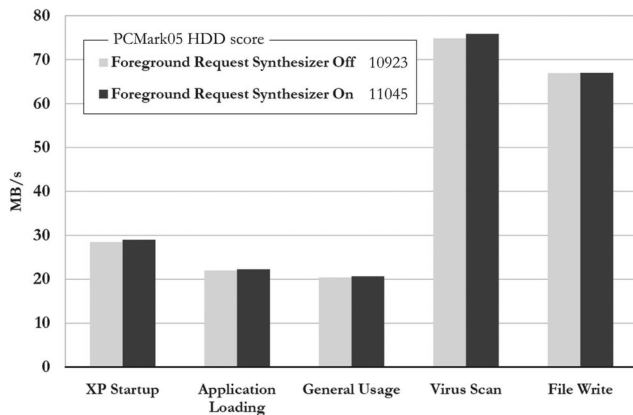


Fig. 18. Effect of the foreground request synthesizer.

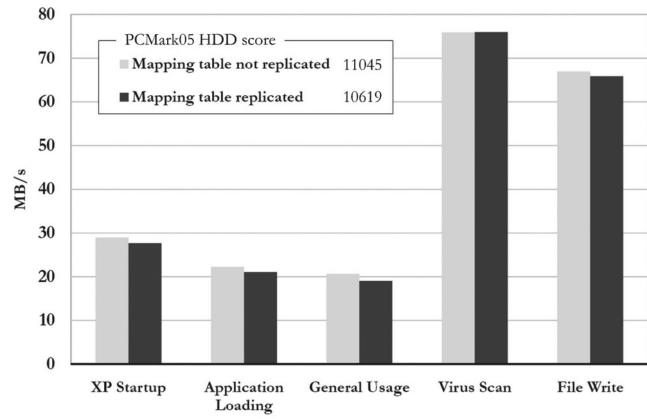


Fig. 19. Effect of mapping table replication.

4.3 Performance Evaluation (TPC-C)

To complement the performance evaluation based on PCMark05, which emulates a typical PC workload, we also performed experiments using traces obtained from running the TPC-C benchmark [31], which emulates an online transaction processing (OLTP) system. The TPC-C benchmark traces were collected from an Oracle database server running on Linux. The benchmark ran for an hour and disk-level I/O traces were collected using the Blktrace tool [5].

The traces contained 2,701,720 sectors (~1.3 GB) and the ratio of reads to writes was 3.36:1. The I/O requests were mostly for 2 KB of data, and spanned over 9 GB of logical sector address space. The traces were replayed using the Btrecord [6] and Btreplay [7] tools and we used I/Os per second (IOPS) as the performance metric.

The results are shown in Fig. 20 for the various versions of Hydra and the set of HDDs and SSDs discussed previously. For Hydra, we tested software background units and hardware background units. In the latter case, we varied the number of units, and also turned off the foreground request synthesizer, and then turned off prioritized handling as well.

The results, shown in Fig. 20, show similar trends to those obtained with PCMark05. We see that all the variations of Hydra, including the one with software background units, outperform the other storage systems, and the effect of implementing background units in hardware dwarfs the other modifications to Hydra. The number of background units also has a considerable impact, while prioritized handling has a modest effect and the foreground request synthesizer makes no noticeable difference. These phenomena have already been explained.

4.4 Related Issues

4.4.1 Energy Consumption

Table 3 presents the energy consumption (in microjoules per sector read or written) of the Hydra SSD prototype and the other five storage systems, for four different types of workload (sequential read, sequential write, random read, and random write). These results were obtained by measuring the current drawn from the 5 V/12 V power supply to the various storage systems while the workloads were applied by the host. To measure the currents, we

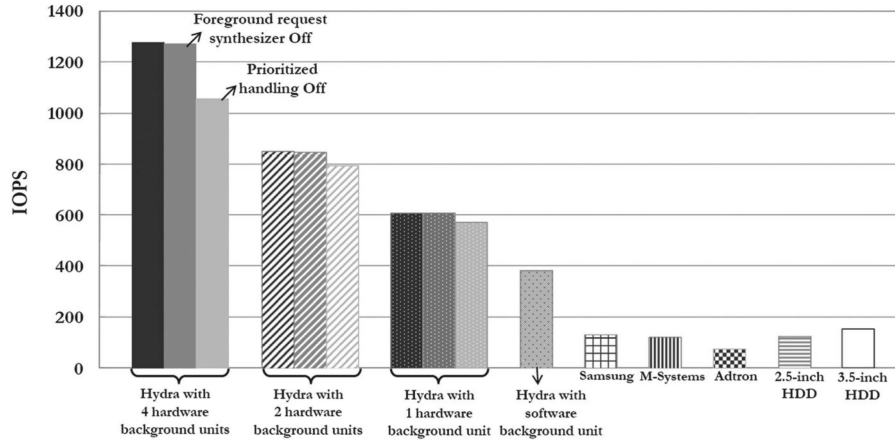


Fig. 20. TPC-C benchmark results.

inserted a small resistor in each power line and gauged the voltage drop across the resistor using a data acquisition (DAQ) board (National Instrument PCI-6259) with a sampling rate of 1.25 million samples per second. In the case of Hydra, we also measured the individual power consumption of the NAND flash memory and the SDRAM, using separate DAQ connections in the development board.

The results in Table 3 show that, for sequential reads, sequential writes, and random reads, the SSDs including the Hydra prototype consume less energy than the HDDs, and that for these three types of workload, the Samsung 2.5-inch SSD consumes the least amount of energy. For random writes, Hydra consumes less energy than the other SSDs, but more than the Seagate 2.5-inch HDD. The relatively large amount of energy required by SSDs to perform random writes is a result of the inherent inability of flash memory to overwrite data directly, so that many page copies are required during garbage collection or merge operation depending on the mapping scheme used.

The energy consumption of the Hydra prototype is bound to be considerably larger because it is implemented in FPGA. We would expect the energy consumption of an ASIC implementation to be somewhere between that of the prototype and that of the NAND flash memory and SDRAM components of the prototype (both shown in the table).

4.4.2 Wear-Leveling

We assessed the effectiveness of the wear-leveling technique used in Hydra by means of simulation, since it would

take too long to observe actual patterns of wear in the Hydra prototype. In this simulation, we divided the storage space into two regions: a read-only region (30 percent) and a read/write region (70 percent). The latter is, in turn, divided into a hot region (20 percent) and a cold region (80 percent). The hot region is subject to 80 percent of the total write operations and the cold region to 20 percent.

Fig. 21a shows the maximum difference in erase counts among the physical superblocks over time for different threshold values used in Hydra's explicit wear-leveling technique. The results show that when this explicit method of wear-leveling is used, there is an upper bound on the variation in erase counts.

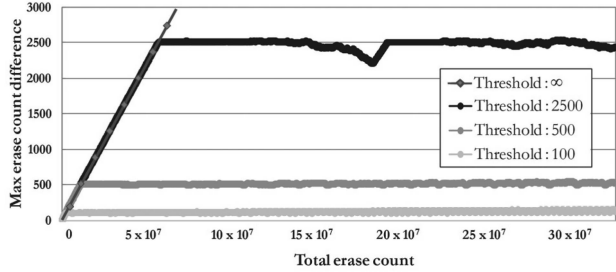
The results also show that a smaller threshold value gives a tighter upper bound and more effective wear-leveling, but this is at the expense of an increase in the number of extra erase operations, as Fig. 21b illustrates. This indicates that there is a trade-off between the effectiveness of wear-leveling and the amount of the overhead involved.

Fig. 21 also shows the results obtained without explicit wear-leveling (i.e., when the threshold value = ∞). In this case, the variation in wear increases linearly over time, because some superblocks only contain read-only data. The value of explicit wear-leveling is clear.

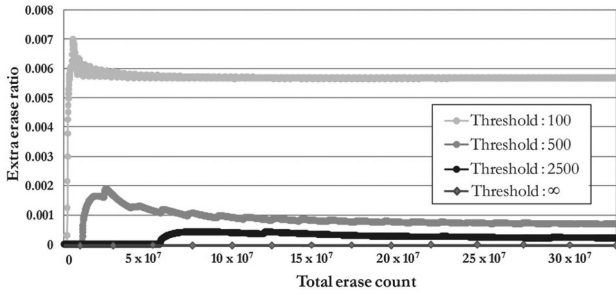
If there is no implicit wear-leveling, then the maximum difference between the erase counts of physical superblocks is still bounded, as we can see in Fig. 22a; but Fig. 22b shows that more explicit wear-leveling is required to bring the difference in erase counts back below the required

TABLE 3
Energy Consumption Measurement Results

	Seagate 2.5-inch HDD	Seagate 3.5-inch HDD	Adtron 3.5-inch SSD	M-Systems 2.5-inch SSD	Samsung 2.5-inch SSD	Hydra SSD	
						Total	SDRAM +Flash
Sequential read	51.926	110.578	35.042	49.322	6.045	34.491	1.790
Sequential write	45.436	144.605	38.829	44.857	15.905	42.641	4.188
Random read	271.693	850.650	37.396	68.193	9.910	45.665	2.052
Random write	151.764	561.752	723.151	743.144	273.342	260.144	34.495



(a)



(b)

Fig. 21. Effectiveness of the Hydra wear-leveling technique. (a) Variation of the maximum erase count difference with different threshold values. (b) Variation of extra erase ratio with different threshold values.

threshold. This shows that explicit wear-leveling does not make implicit wear-leveling redundant.

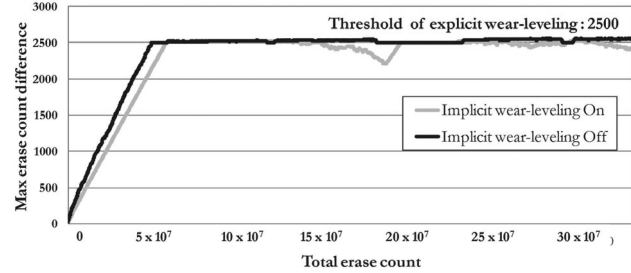
5 CONCLUSIONS

We have presented a new flash memory SSD architecture called Hydra that exploits multichip parallelism effectively.

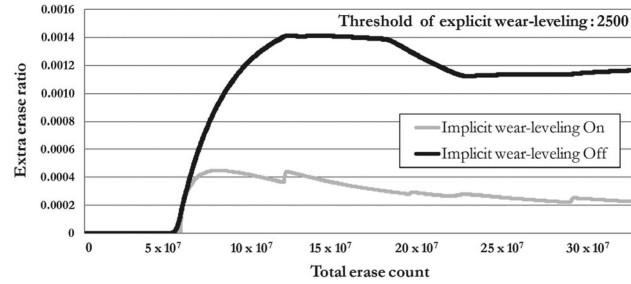
1. It uses both bus-level and chip-level interleaving to overcome the bandwidth limitation of the flash memory bus/chip.
2. It has a prioritized structure of memory controllers, consisting of a single high-priority foreground unit and multiple low-priority background units, all capable of executing sequences of high-level flash memory operations without any software intervention. The foreground unit is dedicated to the processing of read requests from the host to minimize the response time.
3. It employs an aggressive write buffering scheme, which ensures that the background units are utilized effectively, and also reduces the response time of write requests.

Evaluation of an FPGA implementation of the Hydra SSD architecture showed that its performance is over 80 percent better than the best of the HDDs and other SSDs that we considered.

Currently, we are building an SSD which is similar to Hydra, except that it is based on page-level mapping. Although we have already mentioned the problems inherent in page-level mapping, such as the need for a large mapping table and the overhead of garbage collection that sharply increases with utilization, this approach has great potential to improve the performance of small random writes, which cannot be dealt with efficiently by an SSD based on block-level mapping. We plan to compare



(a)



(b)

Fig. 22. Effect of implicit wear-leveling. (a) Variation of the maximum erase count difference with and without implicit wear-leveling. (b) Variation of extra erase ratio with and without implicit wear-leveling.

the two types of SSD architecture using workloads that are sufficient to trigger the utilization-dependent effects of garbage collection in the case of page-level mapping. We hope that this performance characterization will allow us to design an efficient RAID [25] architecture, in which both SSD architectures are combined to achieve an improved level of overall performance.

ACKNOWLEDGMENTS

The authors would like to thank Sang-Won Lee for providing us with TPC-C traces and useful information about them. The authors also would like to thank the Associate Editor and the anonymous reviewers for their constructive comments. This work was supported in part by Creative Research Initiatives (Center for Manycore Programming, 2009-0081569) of MEST/KOSEF.

REFERENCES

- [1] N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," *Proc. USENIX 2008 Technical Conf.*, 2008.
- [2] A. Ban, *Flash File System Optimized for Page-Mode Flash Technologies*, US Patent no. 5,937,425, Aug. 1999.
- [3] A. Ben-Aroya and S. Toledo, "Competitive Analysis of Flash-Memory Algorithms," *Proc. 14th Ann. European Symp. Algorithms*, pp. 100-111, Sept. 2006.
- [4] A. Birrell, M. Isard, C. Thacker, and T. Wobber, "A Design for High-Performance Flash Disks," *SIGOPS Operating Systems Rev.*, vol. 41, no. 2, Apr. 2007.
- [5] Blktrace Manual Page, <http://manpages.ubuntu.com/manpages/intrepid/en/man8/blktrace.html/>, 2010.
- [6] Btrecord Manual Page, <http://manpages.ubuntu.com/manpages/intrepid/en/man8/btrecord.html/>, 2010.
- [7] Btreplay Manual Page, <http://manpages.ubuntu.com/manpages/intrepid/en/man8/btreplay.html/>, 2010.
- [8] 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, Mar. 1999.

- [9] Futuremark Corporation, "PCMark05 Whitepaper," <http://www.futuremark.com/>, 2010.
- [10] E. Gal and S. Toledo, "Algorithms and Data Structures for Flash Memories," *ACM Computing Surveys*, vol. 37, no. 2, pp. 138-163, June 2005.
- [11] C. Hwang, "Nanotechnology Enables a New Memory Growth Model," *Proc. IEEE*, vol. 91, no. 11, pp. 1765-1771, Nov. 2003.
- [12] INCITS, "AT Attachment with Packet Interface—7, Volume 2—Parallel Transport Protocols and Physical Interconnect (ATA/ATAPI-7 V2)," Working Draft, Apr. 2004.
- [13] INCITS, "AT Attachment with Packet Interface—7, Volume 3—Serial Transport Protocols and Physical Interconnect (ATA/ATAPI-7 V3)," Working Draft, Apr. 2004.
- [14] Iometer Project, <http://www.iometer.org/>, 2010.
- [15] M. Jackson, *SAS Storage Architecture*. MindShare Press, 2005.
- [16] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee, "A Superblock-Based Flash Translation Layer for NAND Flash Memory," *Proc. Sixth ACM Conf. Embedded Systems Software (EMSOFT '06)*, pp. 161-170, 2006.
- [17] A. Kawaguchi, S. Nishioka, and H. Motoda, "A Flash-Memory Based File System," *Proc. USENIX 1995 Winter Technical Conf.*, pp. 155-164, 1995.
- [18] T. Kgil and T. Mudge, "FlashCache: A NAND Flash Memory File Cache for Low Power Web Servers," *Proc. 2006 Int'l Conf. Compilers, Architecture and Synthesis for Embedded Systems (CASES '06)*, Oct. 2006.
- [19] T. Kgil, D. Roberts, and T. Mudge, "Improving NAND Flash Based Disk Caches," *Proc. 35th Int'l Symp. Computer Architecture (ISCA '08)*, pp. 327-338, June 2008.
- [20] H. Kim and S. Ahn, "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," *Proc. Sixth USENIX Conf. File and Storage Technologies (FAST '08)*, 2008.
- [21] 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.
- [22] 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, Jul. 2007.
- [23] E.H. Nam, K.S. Choi, J. 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, Sept. 2009.
- [24] R. Panabaker, "Hybrid Hard Disk & ReadyDrive Technology: Improving Performance and Power for Windows Vista Mobile PCs," <http://www.microsoft.com/whdc/winhec/pres06.msp, 2010>.
- [25] D.A. Patterson, G. Gibson, and R.H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proc. 1988 ACM SIGMOD*, pp. 109-116, 1988.
- [26] V. Prabhakaran, T.L. Rodeheffer, and L. Zhou, "Transactional Flash," *Proc. Eighth USENIX Symp. Operating Systems Design and Implementation (OSDI '08)*, 2008.
- [27] M. Rosenblum and J. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Trans. Computer Systems*, vol. 10, no. 1, pp. 26-52, Feb. 1992.
- [28] C. Ruemmler and J. Wilkes, "An Introduction to Disk Drive Modeling," *Computer*, vol. 27, no. 3, pp. 17-28, Mar. 1994.
- [29] Samsung Electronics, NAND Flash Memory Data Sheets, <http://www.samsung.com/>, 2010.
- [30] A. Shekholeslami and P.G. Gulak, "A Survey of Circuit Innovations in Ferroelectric Random-Access Memories," *Proc. IEEE*, vol. 88, no. 5, pp. 667-689, May 2000.
- [31] Transaction Processing Performance Council (TPC), TPC Benchmark C, <http://www.tpc.org/>, 2010.
- [32] 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 (ICCAD '06)*, pp. 601-606, 2006.
- [33] 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 (ASPLOS-6)*, pp. 86-97, 1994.
- [34] 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. 2008.



Yoon Jae Seong received the BS and MS degrees in computer science and engineering both from Seoul National University, in 2005 and 2007, respectively. He is currently working toward the PhD degree at Seoul National University. His research interests include computer architecture, virtualization technology, embedded systems, and storage systems. He is also interested in parallel computing and distributed systems.



Eyee Hyun Nam received the BS degree in electrical engineering from Seoul National University in 1998. He was with Comtec Systems from 1999 to 2002 and Future Systems from 2003 to 2004. He is currently working toward the PhD degree at Seoul National University. His research interests include computer architecture, operating systems, embedded systems, and flash-memory-based storage systems.



Jin Hyuk Yoon received the BS, MS, and PhD degrees in computer science and engineering from Seoul National University, in 1999, 2001, and 2008, respectively. He was a software engineer at Mtron Storage Technology from 2008 to 2009. He is currently a postdoctoral researcher in the Institute of Computer Technology, Seoul National University. His research interests include operating systems, embedded systems, and flash-memory-based storage systems.



Hongseok Kim received the BS degree in electrical engineering from Seoul National University in 2005. He is currently working toward the PhD degree in computer science and engineering at Seoul National University. His research interests include computer architecture, embedded systems, and storage systems.



Jin-Yong Choi received the BS degree in electrical engineering from Seoul National University in 2005. He is currently working toward the PhD degree in computer science and engineering at Seoul National University. His research interests include flash-based storage systems, embedded systems, computer architecture, and host interface systems.



Sookwan Lee received the BS degree in computer science and engineering from Seoul National University in 2000. He is currently working toward the PhD degree at Seoul National University. His research interests include computer architecture, operating systems, embedded systems, file systems, and storage systems.



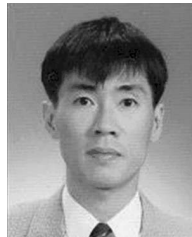
Young Hyun Bae received the BS, MS, and PhD degrees in computer science and engineering from Seoul National University, in 1993, 1995, and 2006, respectively. He is currently the CTO at Mtron Storage Technology. He has been engaged in research in flash memory application area since 2001. His research interests include embedded systems and storage systems.



Jaejin Lee received the BS degree in physics from Seoul National University in 1991, the MS degree in computer science from Stanford University in 1995, and the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 1999. He is an associate professor in the School of Computer Science and Engineering at Seoul National University, Korea, where he has been a faculty member since September 2002. Before joining Seoul National University, he was an assistant professor in the Computer Science and Engineering Department at Michigan State University. His research interests include compilers, computer architecture, and embedded computer systems. He is a member of the IEEE and the ACM. More information can be found at <http://aces.snu.ac.kr/jlee>.



Yookun Cho received the BE degree from Seoul National University, Korea, in 1971, and the PhD degree in computer science from the University of Minnesota at Minneapolis, in 1978. Since 1979, he has been with the School of Computer Science and Engineering, Seoul National University, where he is currently a professor. He was a visiting assistant professor at the University of Minnesota during 1985 and a director of the Educational and Research Computing Center at Seoul National University from 1993 to 1995. He was president of the Korea Information Science Society during 2001. He was a member of the program committee of the IPPS/SPDP'98 in 1997 and the International Conference on High Performance Computing from 1995 to 1997. His research interests include operating systems, algorithms, system security, and fault-tolerant computing systems. He is a member of the IEEE.



Sang Lyul Min received the BS and MS degrees in computer engineering, both from Seoul National University, Seoul, Korea, in 1983 and 1985, respectively. In 1985, he was awarded a Fulbright scholarship to pursue further graduate studies at the University of Washington. He received 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, Seoul, Korea. Previously, he was an assistant professor in the Department of Computer Engineering, Pusan National University, Pusan, Korea, from 1989 to 1992 and a visiting scientist at the IBM T. J. Watson Research Center, Yorktown Heights, New York, from 1989 to 1990. He has served on a number of program committees of technical conferences and workshops, 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.

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