Computing and Informatics, Vol. 36, 2017, 140–168, doi: 10.4149/cai_2017_1_140

EXPLOITING FINE-GRAINED SPATIAL OPTIMIZATION FOR HYBRID FILE SYSTEM SPACE

Jaechun No*

College of Electronics and Information Engineering
Sejong University, 98 Gunja-dong, Gwangjin-gu, Seoul, Korea
e-mail: jano@sejong.ac.kr

Sung-Soon Park

Department of Computer Engineering
Anyang University and Gluesys Co. LTD, Manan-gu, Korea

Cheol-Su Lim

Department of Computer Engineering
Seokyeong University, 16-1 Jungneung-dong, Sungbuk-gu, Seoul, Korea

Abstract. Over decades, I/O optimizations implemented in legacy file systems have been concentrated on reducing HDD disk overhead, such as seek time. As SSD (Solid-State Device) is becoming the main storage medium in I/O storage subsystems, file systems integrated with SSD should take a different approach in designing I/O optimizations. This is because SSD deploys the peculiar device characteristics that do not take place in HDD, such as erasure overhead on flash blocks and absence of seek time to positioning data. In this paper, we present HP-hybrid (High Performance-hybrid) file system that provides a single hybrid file system space, by combining HDD and SSD partitions. HP-hybrid targets for optimizing I/O while considering the strength and weakness of two different partitions, to store large-scale amounts of data in a cost-effective way. Especially, HP-hybrid proposes

* corresponding author

spatial optimizations that are executed in a hierarchical, fine-grained I/O unit, to address the limited SSD storage resources. We conducted several performance experiments to verify the effectiveness of HP-hybrid while comparing to ext2, ext4 and xfs mounted on both SSD and HDD.

Keywords: Spatial optimization, hybrid file system, SSD, positive inclusion, hierarchical extent layout

Mathematics Subject Classification 2010: 68-N25

1 INTRODUCTION

Providing high I/O bandwidth is an essential objective of file system implementation. As data sizes being generated from applications, such as cloud applications [33, 34], become large, the role of file systems as a means of data management has been emphasized to achieve fast I/O performance. As a result, a wide range of optimization schemes have been implemented in file systems and their efficiency and correctness have been proved in many applications. However, many of them targeted for minimizing disk overhead in HDD, which might be no longer required in I/O storage subsystems built with SSD.

SSD is a device medium that is considered as the next-generation storage device due to its advantages such as non-volatility, fast random I/O speed and low-power consumption [1, 11, 26]. As the technology of NAND flash memory is becoming improved, the usage of SSD is also widening from small-size mobile devices to large-scale I/O storage subsystems. However, file system development to utilize SSD's promising potentials in I/O does not keep pace with such a technical improvement of flash memory.

For example, there are several flash memory-related file systems, such as JFFS and YAFFS [2, 7, 16, 29], which have addressed the peculiar semiconductor characteristics of flash memory. However, since they have been developed for small-size devices, those file systems are not appropriate for managing large-scale data storages. One way of managing large-scale amounts of data is to use legacy file systems, such as ext2 and ext4, on top of I/O storage subsystems built with SSDs. However, those file systems do not consider SSD's physical device characteristics, thus they could not effectively address the inherent SSD-related properties, such as wear-leveling [4, 9, 14, 20, 27] and write amplification [1, 22]. Furthermore, the ratio of cost to capacity of SSD is high, compared to HDD [21, 25], thereby building large-scale I/O storage subsystems with only SSDs can incur considerable expenses.

To address those disadvantages, we implement a hybrid file system, called HP-hybrid (High Performance-hybrid), which integrates SSD with HDD in a cost-effective way. HP-hybrid provides multiple, logical data sections in SSD partition that are organized with the different size of I/O units, called extents. On top of

those data sections, files can selectively be mapped to the appropriate data section based on file size, access pattern and usage, in order to reduce file allocation cost and to increase the utilization of SSD storage resources. HP-hybrid has the following advantages:

- File allocations in SSD partition are performed in a fine-grained way, to address SSD space constraints. Although I/O on SSD partition is executed per extent, the remaining space of extents after file allocations can be used further to alleviate extent fragmentation. Utilizing the remaining space of extents is performed by using the in-memory allocation table, thereby not deteriorating overall I/O bandwidth.
- Given that the flash block size is known to HP-hybrid, the extent size can be aligned with flash block boundaries, by collecting data in VFS layer. It is noted that matching the logical data size with flash block boundaries can reduce the erasure overhead in FTL [1]. Furthermore, gathering data in VFS layer does not require the access to SSD internal structures. HP-hybrid attempts to take such an advantage while providing the hierarchical extent layout to avoid fragmentation problem.
- HP-hybrid provides the transparent file mapping where main objectives are to overcome SSD's space restriction and to provide better I/O bandwidth by reducing file allocation cost. In the transparent file mapping, the files that do not need high I/O response time, such as backup files, can bypass SSD partition, while storing them only in HDD partition. Also, data sections can be constructed with different extent sizes. As a result, the files that have a large-size, sequential access pattern, for instance multimedia files, can be stored in the logical SSD data section that is composed of large-size extents, reducing file I/O cost.

The rest of this paper is organized as follows: In Section 2, we discuss related studies. Section 3 describes an overview and optimizations implemented in HP-hybrid. In Section 4, we present several performance experiments of HP-hybrid while comparing them to those of ext2, ext4 and xfs. In Section 5, we discuss the summary of the performance experiments and conclude our paper.

2 BACKGROUND WORK

The storage component of NAND-based SSD is flash memory [8, 17, 24]. One of the critical components affecting SSD performance is FTL (Flash Translation Layer), which emulates the block device driver of HDD. Even though flash memory cell is programmed per page, the erasure operation due to data modification should be executed in terms of flash blocks. Because the life time of flash memory cell is limited, the wear-leveling process in FTL is necessary to evenly distribute erase/program cycles over memory cells. Furthermore, the wear-leveling process can be more effective than ever by reordering flash blocks or pages prior to passing them to FTL.

The wear-leveling algorithm proposed by Chang et al. [4] provides two block pools: hot and cold. When a block is erased, the algorithm compares the erasure count of the old block in the hot pool to that of the younger block in the cold pool. If the difference between two blocks is larger than a threshold value then two blocks are swapped to prevent the old block from being involved in the block reclamation. Also, there is a wear-leveling algorithm using log blocks [14, 20] in which small writes to blocks are collected in log blocks as long as free pages are available in them. The pages in the log block are merged with the corresponding data blocks to be written to flash memory.

A disadvantage of using log blocks is that it can incur low space utilization because each page is associated to a dedicated log block and the space for reserving log blocks is very limited. FAST [19] tried to solve this problem in a way that a log block can be used by write operations directed to multiple data blocks, thus postponing the erase procedure for log and data blocks. However, when pages of a log block are originated from several different data blocks, erasing the log block can also cause a number of erase operations of data blocks that are merged with the pages in the log block.

The performance of FTL can be enhanced by combining with the reordering scheme before data are passed to FTL. For example, CFLRU [18] maintains the clean-first region where clean pages are selected as victim over dirty pages because clean pages can be evicted from the region without additional flash operations. However, the size of clean-first region is defined by a window size that is difficult to find the appropriate value for various applications.

Another flash-aware algorithm is LRU-WSR [12] where each page is associated to "cold-flag". The idea is to select cold-dirty pages as victim to retain hot-dirty pages in the buffer. When a candidate for buffer eviction is needed, the algorithm examines the LRU list from its end. If a dirty page is chosen as a victim and its cold-flag is not set, then the page is moved to the head of the list with the cold-flag being marked and another page is examined. If the candidate is a clean page, then it is selected as a victim regardless of the value of cold-flag. The disadvantage of LRU-WSR is that, with applications where a large number of hot and clean pages are used, the buffer hit ratio can be reduced. BPLRU [15] uses a block-level LRU list where a victim block is flushed to flash memory, along with pages belonging to the block. Likewise, when a page is re-referenced, all pages in the same block are moved to the head of LRU list. The LRU list is served for only write requests. The read requests are simply redirected to FTL.

Although the reordering schemes mentioned can contribute to reducing the number of write and erase operations in flash memory, they require to access the internal structure of flash memory or SSD, to acquire the block and page numbers to be examined. Such information cannot be available unless commercial SSD products expose their internal structure, which happens rarely. One of our objectives in implementing HP-hybrid is to collect data in VFS layer prior to passing them to SSD partition, what does not require an access to SSD internals.

Given that the size of flash block is known to HP-hybrid, it can coalesce data in VFS layer as many data as flash block size, and therefore the erasure overhead taking place in FTL can be optimized by reducing the number of small write operations. Also, Rajimwale et al. [22] show that aligning data with SSD stripe size produces high I/O performance. If the stripe size of SSD partition is given to HP-hybrid, then the file system can even align I/O unit (extent) with stripe size in VFS layer prior to write operations to SSD partition.

The other objective of HP-hybrid is to use the hybrid file system structure to facilitate SSD performance. There are several file systems providing the hybrid file system structure. One of those file systems is Conquest [28], which tried to minimize disk accesses by using the persistent RAM with the battery backup. It stores all small files and file system metadata in RAM and stores the remaining large files in HDD.

The main differences between Conquest and HP-hybrid come from the distinct storage component integrated with HDD. Unlike SSD, the persistent RAM allows in-place data updates and does not deploy semiconductor overhead. On the other hand, using SSD needs to consider a flash memory-related overhead. HP-hybrid attempts to mitigate such overhead through data alignment in VFS layer. Also, Conquest does not support multiple, logical data sections and all file allocations should be performed in a single unit size. On the other hand, HP-hybrid is capable of mapping files to the appropriate data section, by considering file access characteristics, such as file size, access pattern and usage.

Another example is hFS [30], which combines the advantages of LFS (Log-structured File System) and FFS (Fast File System). LFS [13, 23] supports update-out-of-place in which file updates take place without seeking back to their original location. Although such an update behavior is appropriate for flash memory, the sequential log structure of LFS can produce a significant I/O overhead in random environments. In HP-hybrid, every I/O is performed in-place per extent. Thus, there is no need to organize sequential log structures. Like Conquest, hFS does not support file mapping that considers file access characteristics.

Also, several file systems have been implemented by using flash memory. ELF [5] is a flash-based file system targeting at micro sensor platforms. Although ELF is a log-structured file system like JFFS [29], it uses a different log management, according to write behaviors. For example, in write-append operations, it does not create a new log entry to append data. ELF uses the existing log entry, by caching file data to RAM. In write-modify operations, ELF follows the traditional log management, by creating a new log and writing each modification to the new page. In this way, ELF can reduce the number of logs on the resource-constrained platform.

Another flash-related file system is FlexFS [17]. FlexFS was designed to combine the advantages of MLC (multi-level cell) and SLC (single-level cell). The layout of the file system is divided into MLC region and SLC region, and each region is associated to its own write buffer. When a write operation takes place, the data are first stored into the appropriate write buffer until the page size of data is collected

in the buffer and the data are flushed to flash memory. When there is not enough free space available, data are migrated from SLC region to MLC region to make more space for the incoming data.

The difference between HP-hybrid and flash file systems is that HP-hybrid is not a log-structured file system. File updates in HP-hybrid occur in-place. Also, unlike most flash file systems, HP-hybrid has been developed for the large-scale data storage by integrating its file system space with HDD.

ZFS [31] has a similarity to HP-hybrid in a sense that both file systems provide the tiered storage space using SSD and HDD. ZFS is the transactional file system where multiple instances of file system are grouped into a common storage pool. In ZFS, two types of SSD cache are provided: L2ARC (Layer 2 Adaptive Replacement Cache) working as a read cache and ZIL (ZFS Intent Log) working as a write cache. L2ARC deploys its cache by reading data periodically from the tail of DRAM cache (ARC). L2ARC does not contain dirty data to eliminate the time for flushing data to disk. ZIL was implemented to minimize the overhead of synchronous writes. It buffers the logs of system calls to replay write operations to disks later.

The difference between HP-hybrid and ZFS is that HP-hybrid supports several optimizations to increase SSD storage utilization. First of all, it enables to configure the different size of I/O units for each logical data section, to map files based on file attributes, such as file size and access pattern. Furthermore, to alleviate the fragmentation overhead, segment partitioning is performed to the lower levels to reuse the remaining space of extents. Dm-cache [32] differs from HP-hybrid because the cache operation in dm-cache is performed in the block I/O layer. It checks whether the concerned sectors are already stored in the cache device to save the block I/O overhead.

3 IMPLEMENTATION DETAIL

3.1 HP-Hybrid Overview

HP-hybrid provides a hybrid structure in which a single file system space is constructed on top of two physical partitions: SSD partition and HDD partition. In HP-hybrid, SSD partition stores hot files recognized by file access time and file system metadata to be used for SSD file allocation including map table and extent bitmap. On the other hand, HDD partition is used as the permanent data storage while retaining file system metadata and real data.

Figure 1 shows an overview of HP-hybrid disk layout. The beginning of SSD partition stores the configuration parameters that are used at file system mount. Those parameters include the number of logical data sections, section and extent sizes of each data section, and flash block and SSD stripe sizes, if available. In HP-hybrid, SSD partition can be divided into several data sections, while each of them being organized into a different extent size. Figure 1 illustrates that SSD partition is divided into three logical data sections, D_0, D_1 and D_2 where extent sizes are δ,

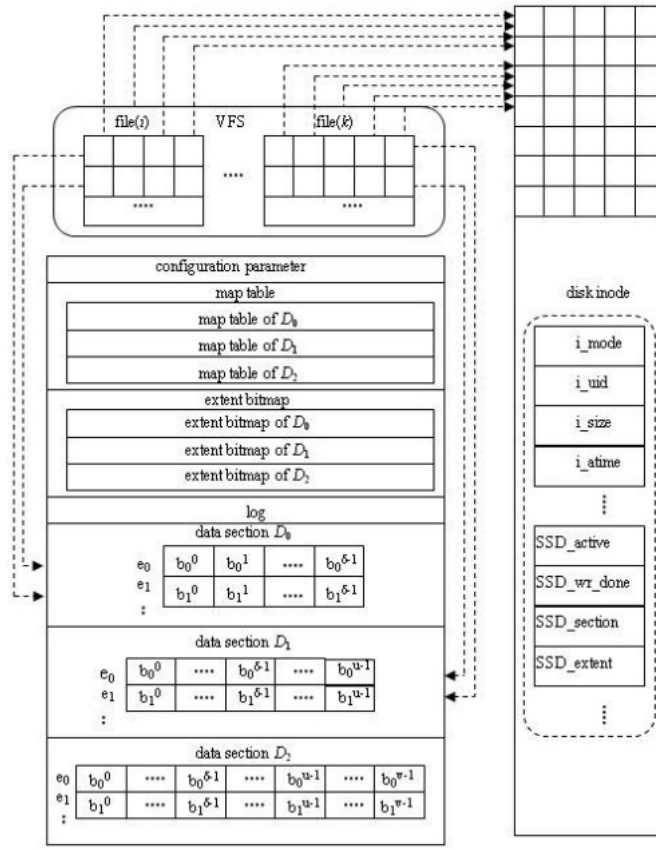


Figure 1. HP-hybrid disk layout

u and v in blocks, respectively, where $\delta < u < v$. The default is the first data section D_0.

Next to the configuration parameters, the map table being used for the transparent file mapping is stored. The map table consists of the directory path and data section where the path is mapped, and the mapping flag that shows what kind of mapping is defined for each pair of directory path and data section, including *SSD bypass*. The map table is followed by the extent bitmap that shows the allocation status of extents.

The inode stored in HDD partition contains several attributes to access files from SSD partition. *SSD_wr_done* denotes I/O status of the corresponding file. The two bits of *SSD_wr_done* describe four states of write operations: 00 for initialization, 10 and 01 for the write completion in SSD partition and HDD partition, respectively, and 11 for the write completion on both partitions. *SSD_active* indicates if the associated file is available in SSD partition. This flag is turned off in case that the

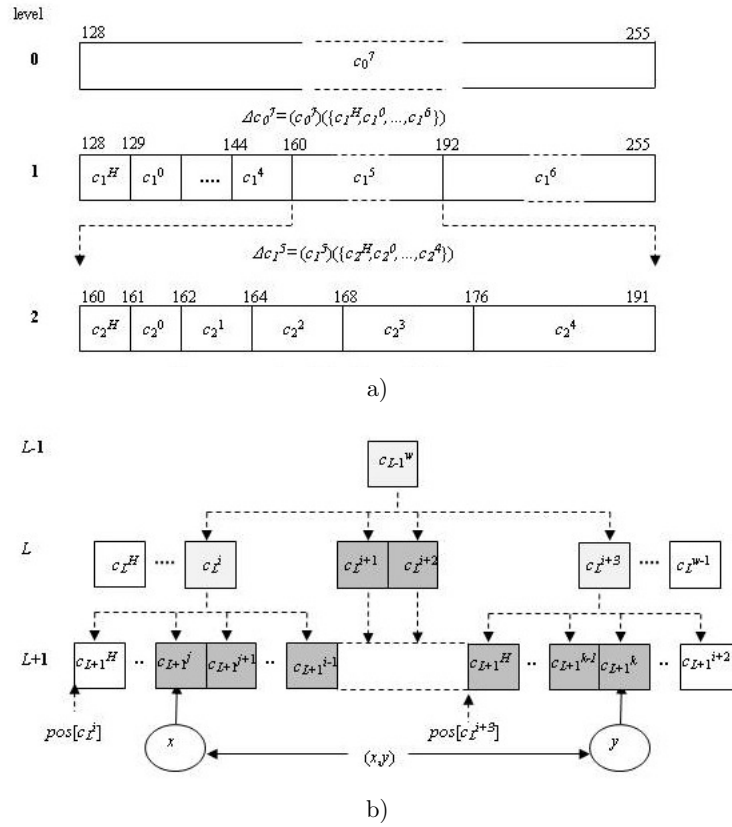


Figure 2. Hierarchical extent layout. a) An example of the hierarchical segment partition. b) Segment partition for an extent point (x, y)

file is evicted from the partition. By referring to these two flags, HP-hybrid serves file requests from either of two partitions.

The logs are used for data recovery, together with *SSD_wr_done* stored in inode. When the logs are read for recovery, HP-hybrid checks *SSD_wr_done* to see if and in which partition I/O operation indicated by each log is completed. If *SSD_wr_done* is set to 10, then the file stored in SSD partition is duplicated to HDD partition as a background process. Otherwise, no recovery for the file has taken place in SSD partition because when the file is accessed from HDD partition it would also be replicated to SSD partition. The logs are also stored in HDD partition to be protected against SSD crash.

Besides, inode includes *SSD_section* and *SSD_extent* to make a connection between two partitions. *SSD_section* shows the identification of data section where the associated file is mapped and *SSD_extent* includes an array of SSD extent addresses – each address consisting of the starting block number and block count in the extent.

3.2 Hierarchical Extent Layout

HP-hybrid attempts to reduce the fragmentation overhead by reusing the free extent portion as much as possible. The extent size of each logical data section is defined at file system creation. If no extent size of a data section is specified, then the default size δ in blocks is assigned to the data section.

HP-hybrid divides each extent into a set of segments and file allocations on extents are executed based on segments. An extent with a size s in blocks is composed of $(\log_2 s) + 1$ segments. For segments where size is larger than or equal to δ (segment index in an extent is $\log_2 \delta$), HP-hybrid recursively partitions the segments until each segment size becomes less than δ. This operation is called the segment partition and each level in which segments are divided into the smaller ones is called the segment level. Hereafter, we briefly call the segment level as 'level'.

Definition 1. Let $E = \{e_0, e_1, \ldots\}$ be a set of extents to be stored in SSD partition. An extent $e_p \in E$ is defined as (s, H, C, Δ) where s is the size of e_p in blocks and $H(= -1)$ is the index of head segment of e_p. C is a set of segments consisting of e_p, with each being indexed from H to $(\log_2 s) - 1$.

Let c_L^i be segment i at level L. Assume that c_L^i is partitioned from c_0^a through c_{L-1}^w of the upper level $L-1$. Then, c_L^i can either be denoted as $(c_{L-1}^w)(\{c_L^i\})$, in terms of its parent, or be denoted as $(c_0^a(\ldots(c_{L-1}^w)\ldots))(\{c_L^i\})$, in terms of its all predecessors. Furthermore, for $\ni c_L^j$, if the predecessors of c_L^i and c_L^j are the same, then two segments can be expressed together as $(c_{L-1}^w)(\{c_L^i, c_L^j\})$ or $(c_0^a(\ldots(c_{L-1}^w)\ldots))$ $(\{c_L^i, c_L^j\})$.

The $pos[c_L^i]$ is the starting block position of c_L^i and $size[c_L^i]$ is the size of c_L^i in blocks. If $L = 0$, then c_L^i is denoted as it is, because it does not have any predecessors. Finally, Δ is the hierarchical segment partition taking place at each level and it satisfies the following:

- $\Delta e_p = \{c_0^k | k \in \{H, 0, 1, \ldots, (\log_2 s) - 1\}\}$.
- $\ni c_L^i, \Delta c_L^i = (c_L^i)(\{c_{L+1}^k | k \in \{H, 0, 1, \ldots, i-1\}\})$ where $i \geq \log_2 \delta$.
- $\text{pos}[c_L^i] = \text{pos}[c_{L-1}^w] + 2^i$ if $i > H$. Otherwise, $\text{pos}[c_L^i] = pos[c_{L-1}^w]$.
- $\text{size}[c_L^i] = 2^i$ if $i > H$. Otherwise, $\text{size}[c_L^i] = 1$.

Example 1. Figure 2 a) shows an overview of the segment partition for c_0^7 located at between block position 128 and 255. The default size δ is 32 blocks. The hierarchical segment partition Δc_0^7 is $(c_0^7)(\{c_1^H, c_1^0, \ldots, c_1^6\})$ that is a set of segments at level one split from c_0^7. Because the sizes of two segments at level one, c_1^5 and c_1^6, are larger than or equal to δ, they are partitioned further to the lower level, by performing the hierarchical segment partition Δc_1^5 and Δc_1^6. Also, $\text{pos}[c_1^5] = \text{pos}[c_0^7] + 2^5 = 128 + 32 = 160$ and $\text{size}[c_1^5] = 2^5 = 32$.

3.3 Segment Partition Functions

In the following definitions, an extent portion (x, y) expresses the part of an extent e_p where a new file is allocated. The x denotes the starting block position and y denotes the ending block position of the extent portion.

3.3.1 Positive and Negative Inclusions

In HP-hybrid, the primitive functions for the segment partition are positive inclusion and negative inclusion. Both functions are used to calculate the segments of an extent belonging to a given extent portion.

Definition 2. The positive inclusion $\psi(+, \Delta c_L^i, j)$ contains the segments that are partitioned from c_L^i and whose index is larger than j. $\psi^*(+, \Delta c_L^i, j)$ is the same as $\psi(+, \Delta c_L^i, j)$, except it contains the segment with index j. The negative inclusion $\psi(-, \Delta c_L^i, j)$ includes the segments split from c_L^i but whose indexes are smaller than j. On the other hand, $\psi^*(-, \Delta c_L^i, j)$ includes the segment with index j, along with the segments calculated from $\psi(-, \Delta c_L^i, j)$. If $i = j$ at the top level, then the segment partition to the lower level does not take place.

$$\psi(+, \Delta c_L^i, j) = \begin{cases} \left\{c_L^k | k \in \{j+1, \ldots, (\log_2 s) - 1\}\right\}, & \text{if } L = 0 \text{ and } i = j, \\ (c_L^i)\left(\left\{c_{L+1}^k | k \in \{j+1, \ldots, i-1\}\right\}\right), & \text{otherwise,} \end{cases}$$

$$\psi(-, \Delta c_L^i, j) = \begin{cases} \left\{c_L^k | k \in \{H, 0, 1, \ldots, j-1\}\right\}, & \text{if } L = 0 \text{ and } i = j, \\ (c_L^i)\left(\left\{c_{L+1}^k | k \in \{H, 0, 1, \ldots, j-1\}\right\}\right), & \text{otherwise.} \end{cases}$$

Example 2. In Figure 2 b), the positive and negative inclusions are computed as follows:

$$\begin{aligned} \psi\left(+, \Delta c_L^i, j\right) &= (c_L^i)\left(\left\{c_{L+1}^{j+1}, c_{L+1}^{j+2}, \ldots, c_{L+1}^{i-1}\right\}\right), \\ \psi^*\left(+, \Delta c_L^i, j\right) &= (c_L^i)\left(\left\{c_{L+1}^{j}, c_{L+1}^{j+1}, c_{L+1}^{j+2}, \ldots, c_{L+1}^{i-1}\right\}\right), \\ \psi\left(-, \Delta c_L^{i+3}, k\right) &= (c_L^{i+3})\left(\left\{c_{L+1}^{H}, c_{L+1}^{0}, \ldots, c_{L+1}^{k-1}\right\}\right), \\ \psi^*\left(-, \Delta c_L^{i+3}, k\right) &= (c_L^{i+3})\left(\left\{c_{L+1}^{H}, c_{L+1}^{0}, \ldots, c_{L+1}^{k-1}, c_{L+1}^{k}\right\}\right). \end{aligned}$$

3.3.2 MOVE Function

Definition 3. MOVE$(c_L^i, x, +)$ and MOVE$(c_L^j, y, -)$ are executed in case that the further partition to the lower level is needed because c_L^i and c_L^j where x and y are mapped still have the segment sizes of no smaller than δ. Without the further segment partition, the fragmentation overhead might be severe by throwing away available blocks in SSD storage space. If the segment index of the child is smaller

than $\log_2 \delta$, which means the segment size is less than δ, then no more segment partition to the lower level takes place. MOVEs are computed as follows:

1. Calculate the child segment of c_L^i where x is mapped. Let k be the index of such a child segment:
$$\lfloor 2^k \rfloor \leq x - \text{pos}[c_L^i] < 2^{k+1}.$$

2. Calculate the child segment of c_L^j where y is mapped. Let l be the index of such a child segment:
$$\lfloor 2^l \rfloor \leq y - \text{pos}[c_L^j] < 2^{l+1}.$$

3. Calculate $MOVE$ functions:
$$\text{MOVE}\left(c_L^i, x, +\right) = \begin{cases} \psi\left(+, \Delta c_L^i, k\right) \cup \text{MOVE}\left(c_{L+1}^k, x, +\right), & \text{if } k \geq \log_2 \delta, \\ \psi^*\left(+, \Delta c_L^i, k\right), & \text{otherwise,} \end{cases}$$
$$\text{MOVE}\left(c_L^j, y, -\right) = \begin{cases} \psi\left(-, \Delta c_L^j, l\right) \cup \text{MOVE}\left(c_{L+1}^l, y, -\right), & \text{if } l \geq \log_2 \delta, \\ \psi^*(-, \Delta c_L^j, l), & \text{otherwise.} \end{cases}$$

Example 3. In Figure 2 b), MOVE$(c_{L-1}^w, x, +)$ and MOVE$(c_{L-1}^w, y, -)$ are given by

$$\text{MOVE}\left(c_{L-1}^w, x, +\right) = \psi\left(+, \Delta c_{L-1}^w, i\right) \cup \text{MOVE}\left(c_L^i, x, +\right)$$
$$\text{where } 2^i \leq x - \text{pos}[c_{L-1}^w] < 2^{i+1} \text{ and } i \geq \log_2 \delta,$$
$$\text{MOVE}\left(c_{L-1}^w, y, -\right) = \psi\left(-, \Delta c_{L-1}^w, i+3\right) \cup \text{MOVE}\left(c_L^{i+3}, y, -\right)$$
$$\text{where } 2^{i+3} \leq y - \text{pos}[c_{L-1}^w] < 2^{i+4}.$$

3.3.3 MAP Function

Definition 4. Given an extent portion (x, y), the segment partition of an extent begins by calling MAP$(x, +)$ and MAP$(y, -)$. MAP$(x, +)$ is used to find the segment of the top level to be mapped to x and MAP$(y, -)$ to y. As with MOVE function, no segment partition takes place if the size of a segment is smaller than δ. MAPs are computed as follows:
for i and j such that $\lfloor 2^i \rfloor \leq x < 2^{i+1}$ and $\lfloor 2^j \rfloor \leq y < 2^{j+1}$,

$$\text{MAP}(x, +) = \begin{cases} \psi\left(+, \Delta c_0^i, i\right) \cup \text{MOVE}\left(c_0^i, x, +\right), & \text{if } i \geq \log_2 \delta, \\ \psi^*\left(+, \Delta c_0^i, i\right), & \text{otherwise,} \end{cases}$$
$$\text{MAP}(y, -) = \begin{cases} \psi\left(-, \Delta c_0^j, j\right) \cup \text{MOVE}\left(c_0^j, y, -\right), & \text{if } j \geq \log_2 \delta, \\ \psi^*\left(-, \Delta c_0^j, j\right), & \text{otherwise.} \end{cases}$$

Example 4. In Figure 2 b), with $L = 1$ and $w \geq \log_2 \delta$, MAP$(x, +)$ and MAP$(y, -)$ are given by

$$\begin{aligned}
\text{MAP}(x, +) &= \psi\left(+, \Delta c_0^w, w\right) \cup \text{MOVE}\left(c_0^w, x, +\right), \\
\text{MAP}(y, -) &= \psi\left(-, \Delta c_0^w, w\right) \cup \text{MOVE}\left(c_0^w, y, -\right) \\
&\qquad \text{where } 2^w \leq x,\ y < 2^{w+1}, \\
\text{MAP}(32, +) &= \psi^*\left(+, \Delta c_0^5, 5\right), \qquad (1) \\
\text{MAP}(106, -) &= \psi\left(-, \Delta c_0^6, 6\right) \cup \text{MOVE}\left(c_0^6, 106, -\right). \qquad (2)
\end{aligned}$$

From Equations (1) and (2), the predecessors of two segment sets are the same. Therefore,

$$\psi^*\left(+, \Delta c_0^5, 5\right) \cap \psi\left(-, \Delta c_0^6, 6\right) = \left\{c_0^5, c_0^6, c_0^7\right\} \cap \left\{c_0^H, c_0^0, \ldots, c_0^5\right\} = \left\{c_0^5\right\}.$$

In Equation (2), since $2^5 \leq 106 - \text{pos}[c_0^6] < 2^6$, 106 is mapped to segment five at level one (c_1^5)

$$\begin{aligned}
\text{MOVE}(c_0^6, 106, -) &= \psi\left(-, \Delta c_0^6, 5\right) \cup \text{MOVE}\left(c_1^5, 106, -\right), \\
\psi\left(-, \Delta c_0^6, 5\right) &= \left(c_0^6\right)\left(\left\{c_1^H, c_1^0, c_1^1, c_1^2, c_1^3, c_1^4\right\}\right).
\end{aligned}$$

c_1^5 is split further to level two. In Equation (4), since $2^3 \leq 106 - (\text{pos}[c_1^5]) < 2^4$, which is $2^3 \leq 106 - (2^6 + 2^5) < 2^4$, 106 is mapped to c_2^3.

$$\text{MOVE}\left(c_1^5, 106, -\right) = \psi^*\left(-, \Delta c_1^5, 3\right) = \left(c_1^5\right)\left(\left\{c_2^H, c_2^0, c_2^1, c_2^2, c_2^3\right\}\right).$$

According to Equations (3), (5) and (6), the segment set T including $(32, 107)$ is

$$T = \left\{c_0^5, \left(c_0^6\right)\left(\left\{c_1^H, c_1^0, c_1^1, c_1^2, c_1^3, c_1^4\right\}\right), \left(c_1^5\right)\left(\left\{c_2^H, c_2^0, c_2^1, c_2^2, c_2^3\right\}\right)\right\}.$$

Figure 3 shows an example of allocating files on an extent that consists of 256 blocks. Initially, there are nine segments to be created on the extent, including the head segment. The size and starting block position of each segment are both $2^i (0 \leq i < 8)$ in blocks, except for the head segment that starts from block zero and its size is of a single block. In this example, $\delta = 32$ blocks, thus the segments from zero to four do not invoke the segment partition.

The request for 30 blocks begins by performing MAP$(0, +)$ and MAP$(29, +)$. The request is mapped to segments between zero and four whose size is smaller than δ each. Two MAP functions are executed by invoking $\psi^*\left(+, \Delta c_0^H, H\right) \cap \psi^*\left(-, \Delta c_0^4, 4\right)$, which results in the segment set $\left\{c_0^H, c_0^0, \ldots, c_0^4\right\}$. The next allocation request on the extent starts from $\text{pos}[c_0^5] = 32$.

In the second request for 75 blocks, as pictured in Figure 3 b), the corresponding allocation process is performed in the extent portion (32,106), starting from the

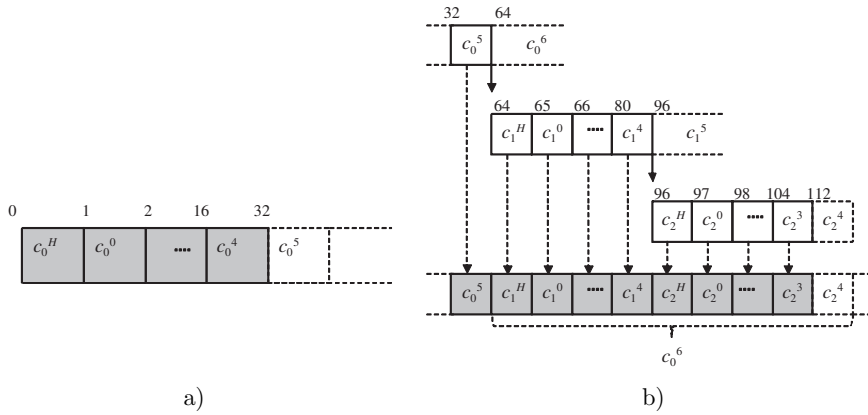


Figure 3. An example of segment partition; a) Allocate 30 blocks, b) Allocate 75 blocks

segment c_0^5. In the process, because the entire blocks of segment five (c_0^5) are included in the extent portion, no segment partition to the lower level takes place on c_0^5: The next file allocation on the extent begins from pos $[(c_1^5)\,(\{c_2^4\})] = 2^6 + 2^5 + 2^4 = 112$.

3.4 Allocation Table

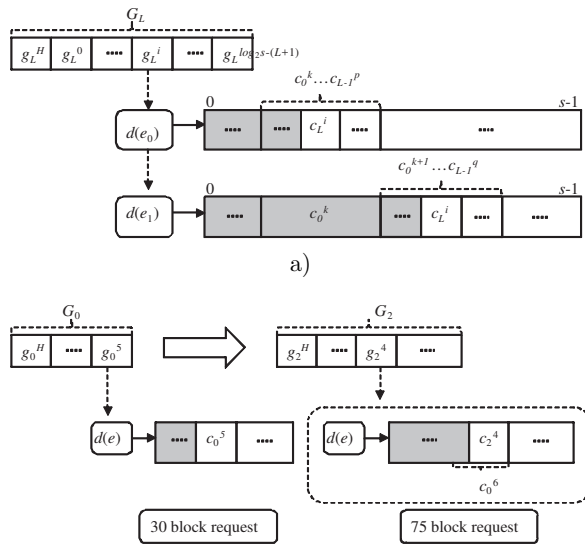


Figure 4. In-memory allocation table, a) Table entry g_L^i of level L, b) Extent movement

HP-hybrid uses in-memory allocation table to assign extents to files. There are three reasons for using the allocation table. First, by filling with data based on segments of extents, HP-hybrid can mitigate the fragmentation overhead. Second, the allocation process can be performed in-memory, thus resulting in better I/O bandwidth. Third, supporting the user-defined extent size might help to reduce SSD-related semiconductor overhead, such as erasure cost, by aligning the extent size with flash block boundaries.

The allocation table is organized for each segment level. At a level, extents are connected to the table entry according to the index of the segment at which the largest free space of the extent begins. Each table entry maintains its own linked list of extent descriptors. The extent descriptor contains the information about the associated extent, such as extent address, total data size being mapped, a sequence of segments created in the extent, and pointer to the callback function to be invoked when the corresponding extent is moved to the other table entry. It also contains the information about the files being mapped to the extent, including inode number, file and extent block positions, and mapping length.

Let $G_L = \{g_L^i | H \leq i \leq (\log_2 s) - (L+1)\}$ be the allocation table to be created at level L where g_L^i is table entry i in G_L. Also, for an extent e_k, let $d(e_k)$be the extent descriptor of e_k, e_k(Large) be the largest hole of e_k and size[e_k(Large)] be the size of e_k(Large) in blocks. Finally, let pos[e_k(Large)], ind[e_k(Large)] and level[e_k(Large)] be the starting block position, starting segment index and segment level of e_k(Large), respectively.

Definition 5. Given the allocation table G_L at level L, the table entries of G_L are defined as follows:

$$g_L^i = \{e_0(\text{Large}), e_1(\text{Large}), \ldots | \text{size}[e_0(\text{Large})] \geq \text{size}[e_1(\text{Large})] \geq \ldots \text{ and}$$
$$\forall e_k(\text{Large}), \text{level}[e_k(\text{Large})] = L, \text{ind}[e_k(\text{Large})] = i\}.$$

Example 5. In Figure 4 a), two extent descriptors are linked at g_L^i, in the decreasing order of the largest hole of two extents. The Largest free space of e_0 starts at c_L^i originated from $c_0^k, \ldots, c_{L-1}^p$ and the one of e_1 starts at c_L^i originated from $c_0^{k+1}, \ldots, c_{L-1}^q$. Also, for e_0 and e_1,

$$\text{pos}[e_0(\text{Large})] = \text{pos}[(c_{L-1}^p)(\{c_L^i\})], \qquad \text{pos}[e_1(\text{Large})] = \text{pos}[(c_{L-1}^q)(\{c_L^i\})]$$
$$\text{where } p < q,$$
$$\text{level}[e_0(\text{Large})] = L, \qquad \text{level}[e_1(\text{Large})] = L,$$
$$\text{ind}[e_0(\text{Large})] = i, \qquad \text{ind}[e_1(\text{Large})] = i.$$

At HP-hybrid mount, the allocation table is created by reading the extent bitmap stored in SSD partition and the extent descriptors of clean extents are linked at the allocation table of the top level. When a new file attempts to be written to SSD partition and its size is larger than or equal to the extent size, HP-hybrid uses clean

Algorithm 1 segSet(T, x, y)

```
1.  calculate MAP(x, +) and MAP(y, −);
2.  include the intersection of two segment sets to T;
3.      /* let c_L^i and c_N^k be the two segments partitioned from MAP(x, +)
    and MAP(y, −) */
4.  if(there exists MOVE to be defined on segments)
5.    do
6.      calculate MOVEs on c_L^i and c_N^k;
7.      if(the predecessors are the same)
8.        include the intersection of child segments of c_L^i and c_N^k to T;
9.      else
10.       include the union of child segments of c_L^i and c_N^k to T;
11.     end if
12.     assign child segments containing x and y to c_L^i and c_N^k;
13.   while(the size of either of segments is at least δ);
14. end if
15. return T
```

Algorithm 2 allocate(f, s, G_L)

```
1.  if(size[f] >= s)
2.    assign ⌈size[f]/s⌉ number of clean extents to f;
3.    /* let e be the last extent allocated to f */
4.    /* let bp be the last block position allocated to f on e */
5.    if(less than δ/2 free blocks is available)
6.      return;
7.    end if
8.    x ← 0; y ← bp;
9.  else
10.   S ← φ;
11.   for all e linked at g_L^i ∈ G_L do
12.     if(size[e(Large)] ≥ size[f])
13.       S ← S ∪ e(Large);
14.     end if
15.   end for
16.   select e such that min{dist(e)/time(e)};
17.   x ← pos[e(Large)]; y ← x + size[f] − 1;
18. end if
19. call segSet(T, x, y);
20. /* let e(h_k) be the hole on e */
21. choose the new e(Large) such that size[e(Large)] = max{size[e(h_k)]|k ≥ 0};
22. N = level[e(Large)]; k = ind[e(Large)];
23. insert e to g_N^k, in the descending order of size[e(Large)];
24. return
```

Figure 5. Algorithm for executing allocation requests

extents from the clean extent table entry. After file allocation is completed, if there is a remaining free space in the last extent where size is not smaller than $\delta/2$, then its extent descriptor is linked to the appropriate table entry, according to the starting segment index of the largest unused space.

In case that a small file whose size is less than extent size attempts to be written to SSD partition, HP-hybrid first checks the extents linked at the table entries. Among the available extents, an extent is chosen for the new file based on two criteria: First, the distance of the directory hierarchy between the new file and the previously allocated files on the extent must be kept small. Second, the extent that has been staying at the allocation table the longest is chosen for file allocation.

Figure 4 b) illustrates how the allocation table works, using the example illustrated in Figure 3. In Figure 3 a), after executing MAP operations, the index of the starting segment of free space is five and thus the extent descriptor is connected to the table entry $g_0^5 \in G_0$. The next request for 75 blocks, pictured in Figure 3 b), leaves free space from c_2^4, resulting in the extent movement to $g_2^4 \in G_2$.

Figure 5 shows two algorithms needed for allocating files using the allocation table. Algorithm 1 shows the steps for calculating the segment set to be used for allocation requests. In the algorithm, x and y are the starting and ending block positions of the request. The algorithm calls MAP to obtain the segment set at the top level. The MOVE is repeatedly performed until the size of child segments at the lower level becomes less than δ.

Algorithm 2 shows the steps for allocating a new file f of size$[f]$ in blocks. In the algorithm, time(e) is the insertion time to the allocation table and dist(e) is the distance between f and the previously allocated files on e. In case that size$[f]$ is not smaller than s, $\lceil \text{size}[f]/s \rceil$ number of clean extents is assigned to f. Otherwise, the extent that has been in the allocation table and where the largest free space is big enough to allocate f is selected in the step 10 to 16. The algorithm calls segSet(T, x, y) to perform the segment partition and inserts the extent to the appropriate table entry in the decreasing order of the next largest hole.

3.5 Transparent File Mapping

HP-hybrid provides the transparent file mapping in which files can be mapped to the appropriate logical data section, according to file size, access pattern and usage. There are two reasons in supporting such a transparent file mapping: 1) to address the limitation of SSD storage resources; 2) to provide better I/O bandwidth by reducing file allocation cost.

In order to address SSD space restriction, HP-hybrid supports *SSD bypass* in which the files that do not need fast I/O speed, such as backup files, would bypass SSD partition, thereby being stored only in HDD partition. Consequently, a large portion of SSD partition can be available for only files requiring high I/O response time. On the other hand, providing better I/O bandwidth is performed by mapping files to the data section that is composed of extents whose size is appropriate to the

files in reducing allocation cost. For example, the files with large, sequential access pattern, such as multimedia files, can have I/O benefit by being stored in the data section composed of large-size extents.

The transparent file mapping is performed in two ways. In the static mapping where directory paths to be mapped to a data section are statically defined at file system mount. Once a directory path is mapped to the data section, the files to be created under the path are all mapped to the same data section. On the contrary, in the dynamic file mapping, file mapping to the data section is dynamically performed at file creation time based on file size and available storage space of the data section. For instance, files with unpredictable sizes, such as emails, can dynamically be mapped to data sections. Both file mappings are specified in the map table, which is submitted at file system mount.

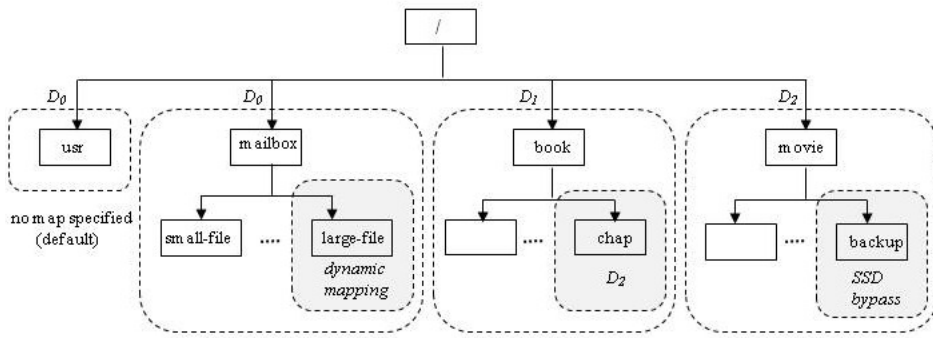


Figure 6. Transparent file mapping

Figure 6 shows an example of the transparent file mapping using the data sections illustrated in Figure 1: δ for the data section D_0, u for the data section D_1 and v for the data section D_2 where $\delta < u < v$. Directories `/mailbox`, `/book` and `/movie` are mapped to D_0, D_1 and D_2, respectively. However, the static file mapping to a data section can be changed to another, by modifying the map table at file system mount. For example, `/book/chap` is changed from D_1 to D_2 and `/movie/backup` from D_2 to SSD bypass. The mapping change is needed because the access characteristics of either files or subdirectories can be varied, such as `/movie/backup` being created to store backup files. In the static file mapping, every time a new file is created, the map descriptor associated to its parent directory is attached to the file, storing the new file to the same data section.

Figure 6 also shows an example of the dynamic mapping in which files are mapped to the appropriate data section, according to file size. For example, the files being created in `/mailbox/large-file` are dynamically mapped to data sections based on their sizes. If no file mapping is specified in the map table, such as `/usr`, then the files to be created under the directory path are stored in the default data section D_0.

HP-hybrid provides the extent replacement algorithm using the multilevel, circular queue. The queue is constructed for each data section. When a file is accessed, the corresponding inode is inserted into the queue. The queue at the highest level contains the most-recent-referenced files and the queue at the bottom contains the files that would be the immediate candidate for SSD eviction. If a file is re-referenced, then the associated inode is moved to the front of the highest level.

When the capacity of the data section drops below the threshold θ, the extent replacement process starts to flush out files linked at the bottom queue and also releases the extents allocated to those files. Since the file data is already stored in HDD partition, there is no need for the data replication for backup. The necessary step for SSD eviction turns off *SSD_active* and modifies the bits of *SSD_wr_done* to 01, to notify that the file no longer exists in SSD partition.

4 PERFORMANCE EVALUATION

4.1 Experimental Platform

Name	Extent Size	Map Dir.	File Size	Num. of Calls		
				MAP	*MOVE*	inc.
HP-hybrid(16)	16 KB	`/hphybrid/ds16`	4 KB	2	0	2
			16 KB	2	0	2
			64 KB	2	0	2
			1 MB ∼ 512 MB	2	0	2
HP-hybrid(64)	64 KB	`/hphybrid/ds64`	4 KB	2	2	4
			16 KB	2	2	4
			64 KB	2	0	2
			1 MB ∼ 512 MB	2	0	2
HP-hybrid(256)	256 KB	`/hphybrid/ds256`	4 KB	2	6	8
			16 KB	2	6	8
			64 KB	2	5	7
			1 MB ∼ 512 MB	2	0	2

Table 1. Overhead of the segment partition for writing a single file

We performed all experiments on a PC system equipped with a 2.3 GHz AMD Phenom triple-core processor, 4 GB of main memory and 320 GB of Seagate Barracuda 7 200 RPM disk. For SSD partition, we installed a 80 GB of fusion-io SSD ioDrive [6] on the system. The operating system was CentOS release 5.6 with a 2.6.32 kernel.

We measured I/O performance of HP-hybrid while comparing to that of ext2, ext4 and xfs installed on HDD and SSD. For the evaluation, we used two popular I/O benchmarks, IOzone [10] and Bonnie++ [3].

With HP-hybrid, we divided SSD partition into three data sections of about 16 GB each. Those data sections were composed of 16 KB, 64 KB and 256 KB of

extent sizes while named as HP-hybrid(16), HP-hybrid(64) and HP-hybrid(256), respectively. To observe the effect of the transparent file mapping, we mapped them to `/hphybrid/ds16`, `/hphybrid/ds64` and `/hphybrid/ds256` and measured I/O performance with a different file size. The default value for the segment partition was set to 32 and the block size was 1 KB.

The write bandwidth of HP-hybrid includes the cost for performing MAP, MOVE and positive/negative inclusions of the segment partition. For example, when the first 4 KB of file is allocated to a 16 KB of extent size, the segment partition at the top level takes place as follows:

$$\begin{aligned} \mathrm{MAP}(0,+) &= \psi^* \left(+, \Delta c_0^H, H\right), \\ \mathrm{MAP}(3,-) &= \psi^* \left(-, \Delta c_0^1, 1\right), \\ \psi^* \left(+, \Delta c_0^H, H\right) \cap \psi^* \left(-, \Delta c_0^1, 1\right) &= \left\{c_0^H, c_0^0, c_0^1\right\}. \end{aligned}$$

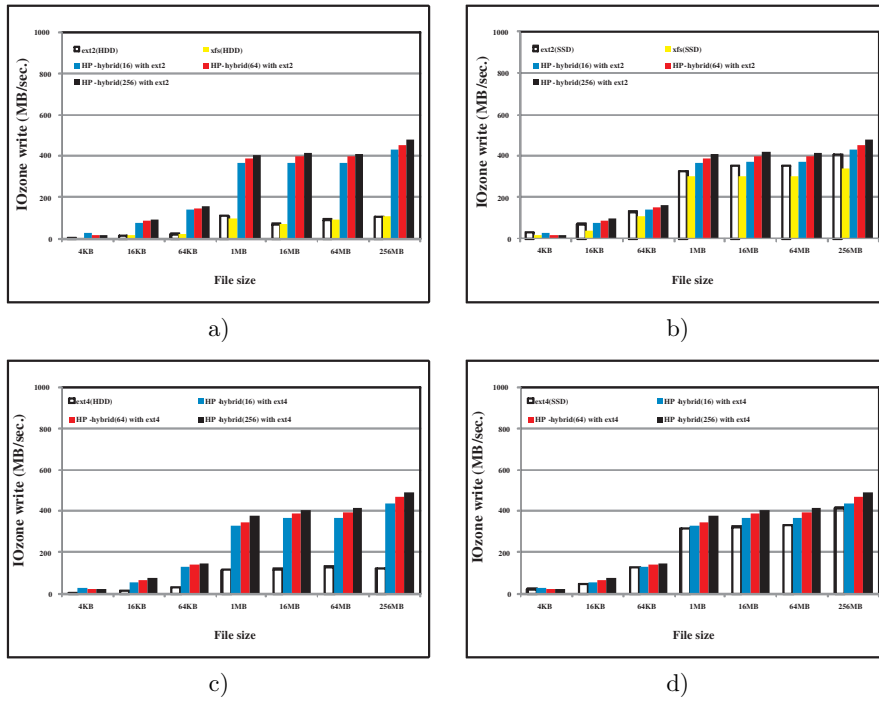
Table 1 shows the maximum number of MAP, MOVE and positive/negative inclusions to write a single file from 4 KB to 256 MB, using the extent sizes between 16 KB and 256 KB and $\delta = 32$ in blocks.

If the file size is equal to or a multiple of extent sizes, then only the segment partition at the top level takes place to include all segments of the top level. This is because there is no remaining space left after file allocations. In this case, only two MAPs and two inclusions are needed to assign the entire space of an extent to the file. If the file size is less than the extent size and the free space after the file allocation is no smaller than δ, then the segment partition to the lower level involving MOVEs takes place to use the remaining free space on the extent. For example, on top of 256 KB of extent size, writing a 4 KB of file needs to perform at maximum two MAPs, six MOVEs and eight inclusions for the segment partition to the lower level.

We chose ext2, ext4 and xfs for the performance comparison. The reason for choosing ext2 and ext4 is because most I/O modules mounted on HDD partition of HP-hybrid are borrowed from ext2 and ext4. In the performance comparison, we will observe I/O benefit of SSD partition in HP-hybrid. Also, xfs was chosen for the comparison because of its B+ tree-based extent allocation. In the evaluation, we want to observe the effectiveness of HP-hybrid allocation scheme, by comparing to xfs.

4.2 IOzone Benchmark

The first evaluation was performed by using IOzone-3-396. We modified IOzone to iterate I/O operations 64 times on each file size and took the average as the final bandwidth. Also, we used $-e$ option to invoke fsync() for write and rewrite operations. If HP-hybrid(16) uses ext2 I/O modules for its HDD partition, then it is named as *HP-hybrid(16) with ext2*. Similarly, *HP-hybrid(16) with ext4* uses ext4



a) b)

c) d)

I/O modules for the HDD partition. The same notation is applied for HP-hybrid(64) and HP-hybrid(256).

Figures 7 a) and 7 c) show the write bandwidth of HP-hybrid where its HDD partition is combined with ext2 in Figure 7 a) and with ext4 in Figure 7 c), while comparing to three other file systems installed on HDDs. As can be seen in the figures, the write throughput of HP-hybrid(16) is higher than that of ext2, ext4 and xfs. In HP-hybrid, file write operations are simultaneously performed on both SSD and HDD partitions. If either of write operations is completed, then control returns to user. Figures 7 a) and 7 c) tell us that such a synchronized write operation of HP-hybrid does not affect much the write performance.

When we compare the write throughput of HP-hybrid(16) to that of ext2 and ext4 installed on SSD partition (ext2 in Figure 7 b) and ext4 in Figure 7 d)), with file size less than 1 MB, HP-hybrid(16) reveals the similar bandwidth to that of ext2 and ext4. However, with file size equal to or larger than 1 MB, HP-hybrid(16) produces the better write throughput. This is because HP-hybrid(16) performs I/O in the larger granularity than ext2 and ext4 and the delay in the allocation table to collect data is very small.

With file sizes larger than 64 KB, the performance speedup to be obtained by using a large I/O granularity is also observed when the extent size of HP-hybrid increases to 64 KB and 256 KB in HP-hybrid(64) and HP-hybrid(256). In Figure 7 f),

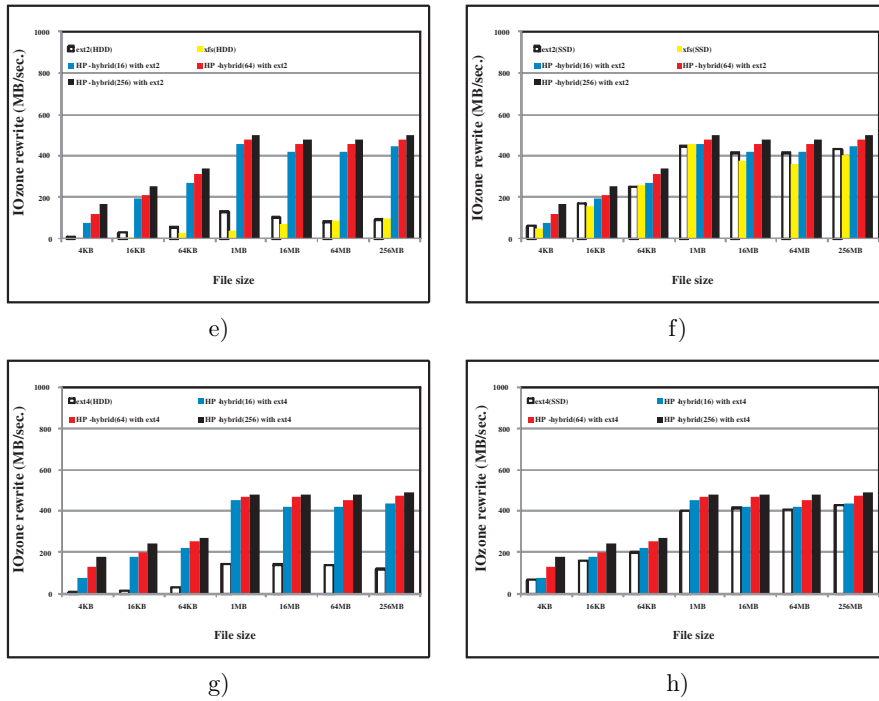


Figure 7. IOzone bandwidth comparison with HP-hybrid whose HDD partition is integrated with ext2 and ext4: a) Write with ext2 and xfs on HDD, b) Write with ext2 and xfs on SSD, c) Write with ext4 on HDD, d) Write with ext4 on SSD, e) Rewrite with ext2 and xfs on HDD, f) Rewrite with ext2 and xfs on SSD, g) Rewrite with ext4 on HDD, h) Rewrite with ext4 on SSD.

when we change the extent size of HP-hybrid to 64 KB and 256 KB in 256 MB of file writes, there are about 11 % and 18 % of performance improvement, respectively, compared to that of ext2. The same I/O behavior can also be observed in Figure 7 h) where HP-hybrid(64) and HP-hybrid(256) generate 12 % and 18 % of speedup as compared to that of ext4 installed on SSD. However, with small-size files such as 4 KB of files, using the large extent size does not generate the noticeable performance advantage, due to the overhead of data collection in the allocation table and the segment partition to the lower level. According to this experiment, we can see that writing files using the appropriate extent size has a critical impact in I/O performance of HP-hybrid.

Figures 7 e) to 7 h) show the performance comparison of file rewrite operations where ext2, ext4 and xfs are installed on HDD and SSD. Similar to write operations, in Figures 7 e) and 7 f), the HDD partition of HP-hybrid is combined with ext2 and in Figures 7 g) and 7 h), it is combined with ext4.

In rewrite operations, the access frequency of file system metadata operations is not high. Figure 7 e) shows that in small files such as 16 KB the extent structure of xfs does not work well on HDD. On the contrary, when installed on SSD, the performance difference between xfs and HP-hybrid(16) using ext2 is about 20 %. Because HP-hybrid maintains the hybrid structure using SSD partition, its extent structure outperforms that of xfs installed on both devices. The performance comparison with ext2 and ext4 illustrates that the usage of the large I/O granularity based on extent size also works well in the rewrite operation, due to the less file access cost. For example, in Figure 7 f), with 256 MB of file size, HP-hybrid(64) and HP-hybrid(256) show about 11 % and 14 % of performance speedup, respectively, compared to ext2. Also, in Figure 7 h), with 256 MB of file size, the same I/O pattern can be observed while using 64 KB and 256 KB of extent sizes in HP-hybrid combined with ext4 improves 10 % and 14 % of I/O bandwidth as compared to that of ext4 installed on SSD. Although we could not measure the overhead of SSD's semiconductor behavior because of inaccessibility to FTL embedded in fusion-io, we guess that the part of such performance improvement might also come from the data alignment in VFS layer.

4.3 Bonnie++ Benchmark

We used Bonnie++ to evaluate HP-hybrid using 256 MB of file size. Since the file size is a multiple of extent sizes, only the segment partition at the top level takes place while executing two MAPs and two inclusions. We used 16 KB of chunks and $-b$ option to invoke fsync() for every write and rewrite operations.

Figures 8 a) to 8 d) illustrate I/O performance of Bonnie++ benchmark. Among the figures, Figures 8 a) and 8 b) are I/O bandwidth comparisons with HP-hybrid whose HDD partition uses ext2 I/O modules. On the other hand, Figures 8 c) and 8 d) are the performance comparisons with HP-hybrid using ext4 I/O modules for its HDD partition. In the figures, ext2, ext4 and xfs produce the large performance difference between two devices in three I/O operations, meaning that SSD deploys higher I/O performance over HDD even with sequential I/O access pattern.

In Figure 8 b), when the write and rewrite throughputs of xfs installed on SSD are compared to those of HP-hybrid, we could notice a large performance difference between two file systems. We guess that as file write and rewrite operations continue on xfs, the extents of the data chunk might not be aligned with flash block boundaries, resulting in the large erasure overhead in SSD. Such performance difference does not appear in the read operation where the erasure overhead of SSD does not take place.

In Figure 8 b), HP-hybrid(16) produces the similar I/O bandwidth to that of ext2 installed on SSD. Also, increasing the extent size to 64 KB and 256 KB does not generate high performance speedup, compared to ext2 and ext4. For example, in Figure 8 b), HP-hybrid(256) shows only 4 % of performance improvement in write and read operations, compared to ext2, due to the sequential I/O pattern of Bonnie++.

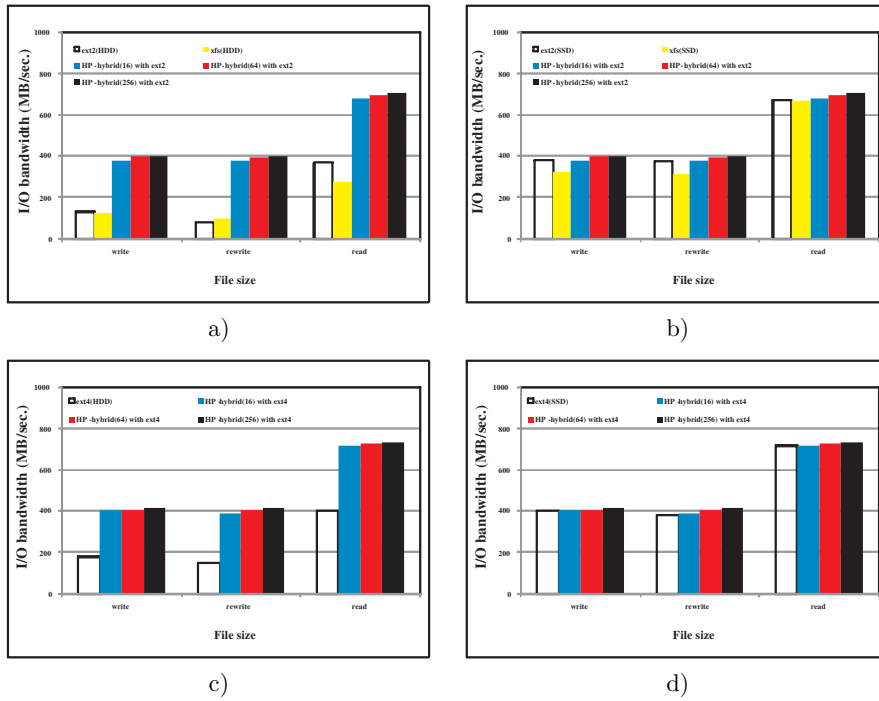
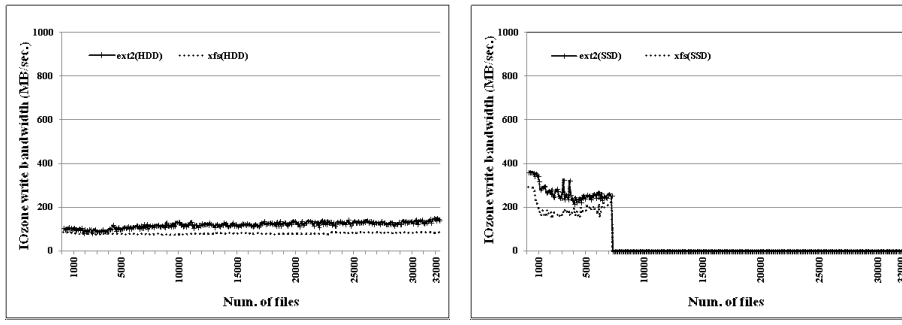


Figure 8. Bonnie++ I/O bandwidth comparison with HP-hybrid where HDD partition is integrated with ext2 and ext4. The file size is 256 MB. a) I/O bandwidth with ext2 and xfs on HDD, b) I/O bandwidth with ext2 and xfs on SSD, c) I/O bandwidth with ext4 on HDD, d) I/O bandwidth with ext4 on SSD.

4.4 Cost-Effectiveness Experiment

One of the main objectives of HP-hybrid is to build the large-scale, hybrid structure to utilize SSD's performance potentials and HDD's low-cost and vast storage availability. To verify such cost-effectiveness, we first provided two partitions, 128 GB of HDD partition and 32 GB of SSD partition, and ran HP-hybrid write operations using IOzone to observe the space usage and write bandwidth. In IOzone, we used the same configuration described in Section 4.2 and continuously wrote 4 MB of files until almost all spaces of HDD and SSD partitions were exhausted. Also, we compared HP-hybrid write bandwidth and space utilization to those of ext2 and xfs installed on HDD and SSD. Since ext4 shows the similar I/O behavior to ext2, we omit the ext4 experiment in this subsection. In HP-hybrid, we varied the extent size from 16 KB to 256 KB. Since the file size to be allocated is larger than the extent size, only the segment partition at the top level takes place, while generating two MAPs and two inclusions.



a) b)

Figure 9. Space utilization of ext2 and xfs on HDD and SSD. a) Ext2 and xfs on HDD, b) Ext2 and xfs on SSD.

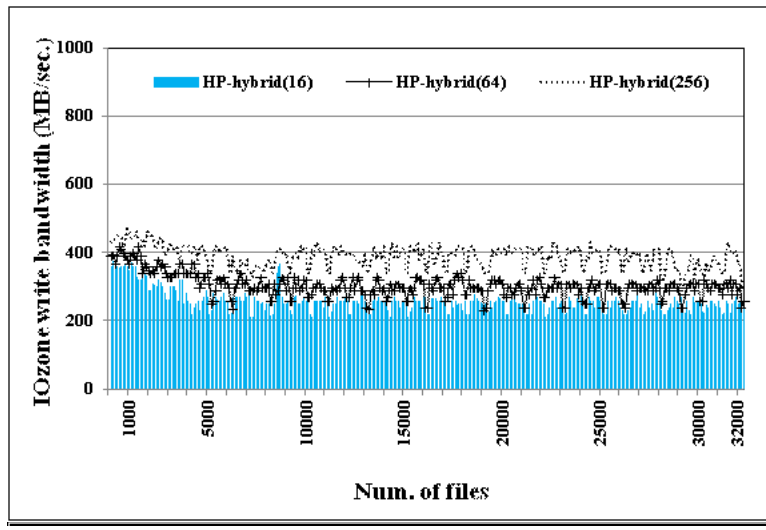


Figure 10. HP-hybrid space utilization

Figure 9 a) shows the write bandwidth and space usage of ext2 and xfs installed on a 128 GB of HDD partition. Each file system can allocate about 32 300 4 MB of files with additional space for storing file system metadata. Also, in Figure 9 b) where two file systems are installed on a 32 GB of SSD partition, about 7 200 4 MB of files can be stored in the partition with the extra space for file system metadata. We would see if HP-hybrid can use the entire space of HDD partition while generating the higher write bandwidth than that of ext2 and xfs on HDD partition.

Figure 10 shows the write throughput and space utilization of HP-hybrid configured with extent sizes from 16 KB to 256 KB. In the evaluation, there are three

aspects to be observed. First, we can notice that, unlike ext2 and xfs installed on SSD partition, the file system space of HP-hybrid is restricted by HDD partition rather than SSD partition, while providing the larger storage space than those file systems. Also, the write throughput of HP-hybrid is similar or even higher than that of ext2 on SSD. As a result, the hybrid structure of HP-hybrid integrated with a small portion of SSD partition can contribute to generating the performance improvement, while providing the larger storage space than SSD partition.

Second, in HP-hybrid, we can observe that the periodic performance degradation takes place. Since the storage space of SSD partition is restricted, whenever about 30 % of total space remains free, the extent replacement de-allocates 1 K files linked at the bottom queue, while modifying the bits of the associated extents in the extent bitmap and disabling *SSD_active*, *SSD_wr_done* and SSD addresses stored in inode. Such a procedure causes the periodic performance turbulence. Third, the effect of the large I/O granularity is still available in this experiment. For example, after writing 32 000 4 MB of files that consumes about 125 GB of space, writing files with 256 KB of extents produces 29 % and 21 % of performance speedup, compared to 16 KB and 64 KB of extent sizes, respectively.

According to the experiment, we can conclude that building the hybrid structure combined with the small portion of SSD partition can be an alternative to produce I/O performance improvement.

5 SUMMARY AND CONCLUSION

Despite its superior characteristics, such as non-volatility, fast random I/O throughput and data reliability file system development to utilize SSD's benefit in I/O does not keep pace with the technical improvement of flash memory. For example, several flash memory-related file systems have been developed for small-size devices, but they are not appropriate for managing large-scale data storages. Also, using legacy file systems, such as xfs, ext2 and ext4, does not maximize SSD usage because they do not address SSD's peculiar device characteristics. One of our primary objectives in developing HP-hybrid is to exploit SSD's performance advantage as much as possible while providing a large storage resource in a low cost. This is performed by constructing the hybrid file system structure which uses the integration of a small portion of SSD partition with the large HDD partition. To address the limited SSD storage capacity, file allocations in SSD partition are performed in a fine-grained way, while utilizing the remaining space of I/O unit (extent). Furthermore, HP-hybrid provides the flexible disk layout where multiple, logical data sections can be configured with the different extent sizes. In the layout, files that have a large-size, sequential access pattern can be stored in SSD data section that is composed of large-size extents, to reduce file allocation cost. Also, files that do not need high I/O response time can bypass SSD partition. Section 5.1 describes the experimental results of HP-hybrid using several I/O benchmarks.

5.1 Summary of Experimental Findings and Lessons Learned

We evaluated I/O performance of HP-hybrid using IOzone and Bonnie++, while comparing to that of xfs, ext2 and ext4 installed on HDD and SSD. The following is what we learned from the evaluation:

- First of all, we divided SSD partition into three data sections where each data section is composed of 16 KB, 64 KB and 256 KB of extent sizes and named as HP-hybrid(16), HP-hybrid(64) and HP-hybrid(256), respectively. If the file size to be allocated to a data section is less than the extent size the segment partition to the lower level involving $MOVE$ operations takes place to use the remaining free space. Otherwise, the segment partition occurs only at the top level while executing two MAPs and two inclusions.
- In IOzone, there is a large performance difference between HP-hybrid(16) and three other file systems installed on HDD because of the hybrid structure of HP-hybrid(16). Also, increasing the extent size of HP-hybrid to 64 KB and 256 KB shows the performance speedup due to the large I/O granularity. However, writing small-size files using the large extent size generates the overhead of data collection in the allocation table and segment partition to the lower level, in order to use the remaining free space after file allocations. Therefore, it is important to write small-size files using the appropriate extent size.
- In the evaluation of using Bonnie++, only the segment partition at the top level takes place in HP-hybrid because of the use of large-size files. In Bonnie++, we can observe that SSD deploys a better I/O bandwidth over HDD even with the sequential access pattern. Also, the performance comparison between xfs installed on SSD and HP-hybrid shows a large difference in write and rewrite operations. We guess that the part of the reason is due to the data alignment with flash block boundaries. The read operation does not show such a performance difference.
- We tried to verify the cost-effectiveness of HP-hybrid, by using a 128 GB of HDD partition and a 32 GB of SSD partition. In those partitions, we continuously wrote 4 MB of files using IOzone until almost all storage spaces of both partitions were consumed by files. In this experiment, we noticed that, unlike ext2 and xfs installed on SSD partition, the file system space of HP-hybrid was restricted by a larger HDD storage space, while generating the similar or even higher write bandwidth than that of ext2 and xfs on SSD partition. Also, the effect of the large I/O granularity can also be observed in the experiment. However, the periodic performance turbulence took place due to the extent replacement.

As a future work, we plan to port HP-hybrid to a web server where several, long-term applications will generate a large number of files with various sizes. In such environment, we can prove more precisely the effectiveness of the hybrid structure and the fine-grained, hierarchical extent layout implemented in HP-hybrid.

Acknowledgement

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (No. 2014R1A2A2A01002614). Also, this work was supported by a grant (13AUDP-C067836-01) from the Architecture and Urban Development Research Program funded by the Ministry of Land, Infrastructure and Transport of the Korean government.

REFERENCES

[1] Agrawal, N.—Prabhakaran, V.—Wobber, T.—Davis, J. D.—Manasse, M.—Panigrahy, R.: Design Tradeoffs for SSD Performance. Proceedings of USENIX 2008 Annual Technical Conference (ATC '08), San Diego, CA, USA, 2008, pp. 57–70.

[2] Aleph1 Company: YAFFS: Yet Another Flash File System. Available on: `http://www.yaffs.net`, 2008.

[3] Bonnie++1.03a. Available on: `www.coker.com.au/bonnie++`.

[4] Chang, L. P.—Du, C. D.: Design and Implementation of an Efficient Wear-Leveling Algorithm for Solid-State-Disk Microcontrollers. ACM Transactions on Design Automation of Electronic Systems, Vol. 15, 2009, No. 1, Art. No. 6.

[5] Dai, H.—Neufeld, M.—Han, R.: ELF: An Efficient Log-Structured Flash File System For Micro Sensor Nodes. Proceedings of SenSys '04, Baltimore, MD, USA, 2004, doi: 10.1145/1031495.1031516.

[6] Fusion-io: ioDrive User Guide for Linux. 2009.

[7] Gal, E.—Toledo, S.: Algorithms and Data Structures for Flash Memories. ACM Computing Surveys, Vol. 37, 2005, No. 2, pp. 138–163, doi: 10.1145/1089733.1089735.

[8] Imation Corporation: Solid State Drives: Data Reliability and Lifetime. White paper, 2008.

[9] Intel Corporation: Understanding the Flash Translation Layer (FTL) Specification. Technical paper, 2008.

[10] IOzone Filesystem Benchmark. Available on: `www.iozone.org`.

[11] Josephson, W. K.—Bongo, L. A.—Flynn, D.—Li, K.: DFS: A File System for Virtualized Flash Storage. Proceedings of 8th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, 2010, doi: 10.1145/1837915.1837922.

[12] Jung, H.—Shim, H.—Park, S.—Kang, S.—Cha, J.: LRU-WSR: Integration of LRU and Writes Sequence Reordering for Flash Memory. IEEE Transactions on Consumer Electronics, Vol. 54, 2008, No. 3, pp. 1215–1223.

[13] Jung, J.—Won, Y.—Kim, E.—Shin, H.—Jeon, B.: FRASH: Exploiting Storage Class Memory in Hybrid File System for Hierarchical Storage. ACM Transactions on Storage, Vol. 6, 2010, No. 1, Art. No. 3.

[14] Kim, J.—Kim, J. M.—Noh, S. H.—Min, S. L.—Cho, Y.: A Space-Efficient Flash Translation Layer for Compactflash Systems. IEEE Transactions on Consumer Electronics, Vol. 48, 2002, No. 2, pp. 366–375.

[15] Kim, H.—Ahn, S.: BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. Proceedings of 6th USENIX Symposium on File and Storage Technologies, San Jose, CA, USA, 2008.

[16] Lee, C.—Baek, S. H.—Park, K. H.: A Hybrid Flash File System Based on NOR and NAND Flash Memories for Embedded Devices. IEEE Transactions on Computers, Vol. 57, 2008, No. 7, pp. 1002–1008.

[17] Lee, S.—Ha, K.—Zhang, K.—Kim, J.—Kim, J.: FlexFS: A Flexible Flash File System for MLC NAND Flash Memory. Proceedings of USENIX Annual Technical Conference, San Diego, CA, USA, 2009.

[18] Park, S.—Jung, D.—Kang, J.—Kim, J.—Lee, J.: CFLRU: A Replacement Algorithm for Flash Memory. Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, Seoul, Korea, 2006, doi: 10.1145/1176760.1176789.

[19] Lee, S.—Park, D.—Chung, T.—Lee, D.—Park, S.—Song, H.: A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation. ACM Transactions on Embedded Computing Systems, Vol. 6, 2007, No. 3, Art. No. 18.

[20] Park, C.—Cheon, W.—Kang, J.—Roh, K.—Cho, W.: A Reconfigurable FTL (Flash Translation Layer) Architecture for NAND Flash-Based Applications. ACM Transactions on Embedded Computing Systems, Vol. 7, 2008, No. 4, Art. No. 38.

[21] Polte, M.—Simsa, J.—Gibson, G.: Comparing Performance of Solid State Devices and Mechanical Disks. Proceedings of the 3rd Petascale Data Storage Workshop held in conjunction with Supercomputing '08, Austin, TX, USA, 2008, doi: 10.1109/pdsw.2008.4811886.

[22] Rajimwale, A.—Prabhakaran, V.—Davis, J. D.: Block Management in Solid-State Devices. Proceedings of USENIX Annual Technical Conference, San Diego, CA, USA, 2009.

[23] Rosenblum, M.—Ousterhout, J. K.: The Design and Implementation of a Log-Structured File System. ACM Transactions on Computer Systems, Vol. 10, 1992, No. 1, pp. 26-52, doi: 10.1145/146941.146943.

[24] Samsung Electronics: K9XXG08XXM. Technical paper, 2007.

[25] Saxena, M.—Swift, M.: FlashVM: Virtual Memory Management on Flash. Proceedings of USENIX Annual Technical Conference, Boston, MA, USA, 2010.

[26] Soundararajan, G.—Prabhakaran, V.—Balakrishnan, M.—Wobber, T.: Extending SSD Lifetimes with Disk-Based Write Caches. Proceedings of USENIX Annual Technical Conference, San Diego, CA, USA, 2008.

[27] Texas Memory Systems: Increase Application Performance with Solid State Disks. White paper, 2010.

[28] Wang, A.—Kuenning, G.—Reiher, P.—Popek, G.: The Conquest File System: Better Performance Through a Disk/Persistent-RAM Hybrid Design. ACM Transactions on Storage, Vol. 2, 2006, No. 3, pp. 309–348.

[29] Woodhouse, D.: JFFS: The Journalling Flash File System. Proceedings of Ottawa Linux Symposium, Ottawa, Canada, 2001.

[30] Zhang, Z.—Ghose, K.: hFS: A Hybrid File System Prototype for Improving Small File and Metadata Performance. Proceedings of EuroSys '07, Lisboa, Portugal, 2007, doi: 10.1145/1272996.1273016.

[31] Nelson, C.: NexentaStor: An Introduction to ZFS's Hybrid Storage Pool. White paper, Nexenta Systems, 2012.

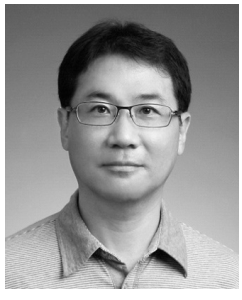
[32] Ali, A.—Rose, C.: bcache and dm-Cache. White paper, Dell Inc., 2013.

[33] Nguyen, B. M.—Tran, V.—Hluchý, L.: A Generic Development and Deployment Framework for Cloud Computing and Distributed Applications. Computing and Informatics, Vol. 32, 2013, pp. 461–485.

[34] Hung, S.-H.—Shieh, J.-P.—Lee, C.-P.: Virtualizing Smartphone Applications to the Cloud. Computing and Informatics, Vol. 30, 2011, pp. 1083–1097.

Jaechun No received her Ph.D. degree in computer science from Syracuse University in 1999. She worked as a postdoctoral researcher at Argonne National Laboratory from 1999 to 2001. She also worked at Hewlett-Packard from 2001 to 2003. She is Professor at the College of Electronics and Information Engineering at Sejong University. Her research areas include file systems, large-scale storage system and cloud computing.

Sung-Soon Park received his Ph.D. degree in computer science from Korea University in 1994. He worked as a fulltime lecturer at Korea Air Force Academy from 1988 to 1990. He also worked as a postdoctoral researcher at Northwestern University from 1997 to 1998. He is Professor at the Department of Computer Science and Engineering at Anyang University and also CEO of Gluesys Co. Ltd. His research areas include network storage system and cloud computing.



Cheol-Su Lim received his Master's degree from Indiana University and his Ph.D. degree in computer engineering from Sogang University. He worked as a senior researcher at SK Telecomm from 1994 to 1997. He also worked as National Research Program Director from 2009 to 2010. He is Professor at the Department of Computer Engineering at Seokyeong University. His research areas include multimedia systems and cloud computing.